Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart

Diplomarbeit Nr. 2372

Dynamische semantikbasierte Suche nach Web Services in parametrisierten BPEL-Prozessen

Branimir Wetzstein

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Frank Leymann
Betreuerin:	M. Sc. Dimka Karastoyanova
begonnen am:	18. Juli 2005
beendet am:	17. Januar 2006
CR-Klassifikation	D.2.12, D.2.13, H.3.3, H.3.4, H.4.1, I.2.4

Zusammenfassung

Die Business Process Execution Language (BPEL) ist ein Standard zur Beschreibung von Prozessen auf Basis von Web Services. In BPEL werden Web-Service-Aufrufe durch WSDL-Schnittstellen modelliert. Dabei werden die PortType- und Operation-Namen im Prozessmodell hartkodiert. Eine Auswahl von möglicherweise passenderen Services zur Laufzeit ist dadurch eingeschränkt. Im Rahmen von parametrisierten Prozessen wird durch eine Erweiterung von BPEL die Namensbindung aufgehoben. Aufbauend auf parametrisierten Prozessen, wird in dieser Arbeit untersucht, wie BPEL erweitert werden kann, um eine automatische Suche nach Web Services zur Laufzeit zu ermöglichen.

Als eine Einschränkung für die automatische Suche wird zunächst die fehlende Semantik in WSDL-Beschreibungen identifiziert. Daraufhin werden bestehende Ansätze im Bereich der Semantic Web Services (SWS) untersucht. Es wird eine Suchanfrage spezifiziert, die eine Suche anhand der semantischen funktionalen Beschreibung und nichtfunktionalen Eigenschaften unterstützt. Insbesondere kann die Suchanfrage auch BPEL-Übergangsbedingungen und WS-Policy-Ausdrücke enthalten. Für die Umsetzung der dynamischen Suche wird eine Broker-basierte Architektur vorgestellt. Dabei führt ein Broker die semantische Suche und den Aufruf stellvertretend für die BPEL-Engine durch. Die Suche und der Aufruf können aus Sicht der BPEL-Engine in einem Schritt oder auch getrennt durchgeführt werden.

Im Rahmen der Arbeit wurden die Konzepte in einer Erweiterung der ActiveBPEL-Engine unter Verwendung der SWS-Technologie OWL-S umgesetzt. Die Broker-Implementierung umfasst einen OWL-S-Matchmaker, der die semantische Suche durchführt, und eine Aufruf-Komponente, die einen Service unter Einbeziehung möglicherweise notwendiger Datentransformationen aufrufen kann. Es wurde weiterhin eine einfache Form der automatischen Service-Komposition implementiert. Dabei kann eine Sequenz von mehreren Services aufgefunden und ausgeführt werden, die zusammengenommen die Anforderungen der Suchanfrage erfüllen.

Meinen Eltern

zum Dank für ihre liebevolle Unterstützung

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

Verzeichnis der Listings **viii**

1 Einleitung **1**
1.1 Hintergrund . . . 1
1.1.1 Einführung . . . 1
1.1.2 Serviceorientierte Architektur . . . 2
1.1.3 Web Services . . . 4
1.1.4 Business Process Execution Language . . . 10
1.2 Motivation . . . 12
1.2.1 Dynamische Bindung an Partner Services . . . 12
1.2.2 Parametrisierung von Web Service Flows . . . 13
1.2.3 Aspekte der Parametrisierung . . . 16
1.3 Aufgabenstellung . . . 18
1.4 Aufbau der Arbeit . . . 19

2 Verwandte Ansätze **20**
2.1 Semantic Web . . . 20
2.1.1 Einführung . . . 20
2.1.2 RDF und RDFS . . . 21
2.1.3 OWL . . . 21
2.1.4 Ontologien . . . 22
2.2 Semantic Web Services . . . 23
2.2.1 Einführung . . . 23
2.2.2 WSDL-S . . . 24
2.2.3 OWL-S . . . 27
2.2.4 METEOR-S . . . 33
2.2.5 WSMO . . . 35

3 Lösungsansatz **39**
3.1 Spezifizierung der Suchanfrage . . . 39
3.1.1 Anforderungen . . . 39
3.1.2 Vorüberlegungen . . . 40
3.1.3 Inhalt der Suchanfrage . . . 43

3.2 Erweiterung der BPEL-Engine . . . 44
3.2.1 Anforderungen . . . 44
3.2.2 Architektur . . . 44
3.2.3 Vergleich mit anderen Ansätzen . . . 48

4 Umsetzung **50**
4.1 Verwendete Software . . . 50
4.2 Kreditantragsprozess . . . 51
4.2.1 Modellierung der domänenspezifischen Ontologie . . . 52
4.2.2 Erstellen der OWL-S-Servicebeschreibungen . . . 53
4.3 Erweiterung der OWL-S-Service-Ontologie . . . 55
4.4 Suchanfrage . . . 57
4.4.1 OWL-S-Anfragebeschreibung . . . 57
4.4.2 Service-Kategorien . . . 58
4.4.3 Service-Properties . . . 58
4.4.4 WS-Policy . . . 60
4.4.5 Bedingungen in der Suchanfrage . . . 60
4.5 Erweiterung der BPEL-Engine . . . 61
4.5.1 Überblick . . . 61
4.5.2 Adapter . . . 62
4.5.3 Broker . . . 63
4.5.4 OWL-S Matchmaker . . . 63
4.5.5 Serviceaufruf . . . 67

5 Zusammenfassung **69**

A Suchanfrage-Beispiel **71**
A.1 Suchanfrage . . . 71
A.2 OWL-S-Anfragebeschreibung . . . 73

Literaturverzeichnis **76**

Abbildungsverzeichnis

1.1 SOA-Dreieck . . . 3
1.2 Service Bus . . . 4
1.3 Web Services Technologien . . . 5
1.4 Kontrollfluss ohne/mit Parametrisierung . . . 18

2.1 Semantic Web Services . . . 23
2.2 Annotation in WSDL-S . . . 25
2.3 Basisklassen der OWL-S-Ontologie . . . 27
2.4 OWL-S/UDDI-Matchmaker . . . 30
2.5 OWL-S/UDDI Mapping . . . 31
2.6 Semantic Discovery Service . . . 32
2.7 Dynamische Komposition in METEOR-S . . . 34
2.8 WSMX-Architektur . . . 37

3.1 Annotation von WSDL-Operationen . . . 40
3.2 Mapping zwischen WSDL-Messages und Ontologie-Konzepten . . . 41
3.3 Mapping-Problematik . . . 41
3.4 Architektur der BPEL-Engine-Erweiterung . . . 45

4.1 Kreditantragsprozess ohne Suchanfrage . . . 51
4.2 Kreditantragsprozess mit Suchanfrage . . . 52
4.3 Ein Teil der OWL-Ontologie des Domänenmodells . . . 53
4.4 WSDL2OWL-S Converter . . . 54
4.5 Zusätzliche Service-Eigenschaften . . . 55
4.6 Architektur der ActiveBPEL-Erweiterung . . . 61
4.7 Matchmaking-Prozess . . . 64
4.8 Serviceaufruf . . . 67

Verzeichnis der Listings

1.1 SOAP-Nachricht 6
1.2 WS-Policy-Expression 9
1.3 Evaluate-Konstrukt 15
2.1 Annotation des WSDL-Interfaces in WSDL-S 26
2.2 Annotation der Datentypen in WSDL-S 26
4.1 Format der Suchanfrage 57
4.2 Service-Property mit numerischem Wert 59
4.3 Service-Property mit einem OWL-Typ als Wert 59
4.4 Service-Property zur Sortierung 59
4.5 Switch-Element in der Suchanfrage 60
4.6 Verwendung des Switch-Elements 60
4.7 Das Broker-Interface 63
4.8 RDQL-Anfrage 64

1 Einleitung

1.1 Hintergrund

1.1.1 Einführung

Die meisten Unternehmen verkaufen Produkte an ihre Kunden, um ihre Geschäftsziele zu erreichen. Geschäftsprozesse definieren dabei wie diese Produkte entworfen, produziert, vermarktet und verkauft werden. Von einer gewissen Abstraktionsstufe her gesehen, sind die Geschäftsprozesse eines Unternehmens eine direkte Abbildung der Produkte, die es anbietet. Daraus ergibt sich, dass die Flexibilität bezüglich der Änderung der bestehenden oder der Erstellung neuer Prozesse sich direkt auf die Geschwindigkeit auswirkt, wie bestehende Produkte angepasst oder neue Produkte erschaffen werden können. Aufgrund der immer kürzeren Produktlebenszyklen und steigendem Konkurrenzdruck im Zuge der Globalisierung wird es immer wichtiger, dass ein Unternehmen schneller als die Konkurrenz seine Geschäftsprozesse anpassen kann, um z.B. auf neue Kundenanforderungen zu reagieren. [WCL+05]

Ein Geschäftsprozess modelliert in einem sogenannten Kontrollfluss die Reihenfolge, in der bestimmte Aktivitäten unter bestimmten Bedingungen durchzuführen sind. Die Durchführung der Aktivitäten kann dabei manuell durch Menschen oder automatisch durch Softwaresysteme erfolgen. Die Durchführung der einzelnen Aktivitäten kostet Zeit und Geld.

Bei einer Prozessoptimierung versucht man nun typischerweise die Laufzeit und/oder die Kosten zu senken. Eine Möglichkeit besteht darin, die durchführenden Akteure von Aktivitäten zu ändern. So wird beim Outsourcing das Durchführen von Aktivitäten, die typischerweise nicht zum Kerngeschäft des Unternehmens gehören, an externe Dienstleister ausgelagert, da diese die Aktivitäten schneller und/oder günstiger durchführen können. Durch die Auslagerung von Aktivitäten an Partner führt das Unternehmen seine Geschäftsprozesse nicht mehr alleine durch. Übersteigt diese Auslagerung ein gewisses Maß, so kann man von virtuellen Unternehmen sprechen [WCL+05]. Aus Sicht des Kunden, der die Produkte kauft, hat sich nichts verändert, er ist auch nur an dem Service des Unternehmens interessiert.

Auch wenn die Diensterbringung bzw. Durchführung einer Aktivität seitens des Partners durch Software geschieht, wird die Partnerauswahl aus Sicht des Unternehmens heute noch weitgehend manuell durchgeführt. Es wäre aber wünschenswert auch in diesem Fall eine gewisse Virtualisierung zu erreichen, indem beispielsweise der günstigste Dienstleister zur

Prozesslaufzeit automatisch ausgewählt und aufgerufen wird. Aus Sicht des Geschäftsprozesses ist alleine die Durchführung der Aktivität bzw. Erbringung eines Dienstes wichtig, natürlich unter der Einbeziehung von gewissen Dienstgüte-Kriterien wie Sicherheit, Vertrauenswürdigkeit, Preis usw. Welcher Dienstleister den Dienst erbracht hat, ist dann nicht mehr wichtig. Diese Arbeit beschäftigt sich mit dieser Art der Flexibilisierung von Geschäftsprozessen, d.h. der automatischen Suche nach geeigneten Partnern anhand bestimmter Auswahlkriterien.

1.1.2 Serviceorientierte Architektur

Serviceorientierte Architektur gehört seit einiger Zeit zu den viel diskutierten Themen der IT-Branche. Sie versteht sich als ein neues Architekturparadigma zur Erstellung und Integration von Anwendungen. Die Hauptziele von SOA sind lose Kopplung von Softwaresystemen und Erhöhung der Flexibilität und Wiederverwendung von Geschäftsprozessen.

Services im Kontext von SOA können aus fachlicher und technologischer Sicht betrachtet werden. Fachlich gesehen ist ein Service ein Dienst, den ein Unternehmen seinen Kunden, Partnern, Mitarbeitern anbietet. In SOA werden Services in den Mittelpunkt der Betrachtungen gestellt und es wird versucht die IT-Infrastruktur an diese anzupassen und nicht umgekehrt. Im Gegensatz dazu tendierten frühere Ansätze dazu, die fachlichen Probleme, die durch IT gelöst werden sollen, an die Möglichkeiten bestimmter Implementierungstechnologien wie CICS, CORBA, J2EE, COM/DCOM, .NET oder Ansätzen wie prozeduraler Programmierung, Objektorientierung und Message-basierter Kommunikation auszurichten.

Wichtige Aspekte von Services sind: Erstellen neuer Services durch rekursive Komposition, Granularität, Service-Metadaten und lose Kopplung. Diese Aspekte sollen im Folgenden am Beispiel eines Bankunternehmens erläutert werden. Eine Bank bietet als Services für ihre Kunden beispielsweise Kontoführung, Kreditvergabe und Geldtransaktionen an. Ein Service bietet also immer einen Nutzen aus fachlicher Sicht.

Bei einem Kreditantrag z.B. lässt der Bankmitarbeiter die Kreditwürdigkeit des Kunden feststellen, prüft seinen Kontostand, seine Einnahmen und Ausgaben und berechnet die Zinsrate. Diese Tätigkeiten stellen wiederum Services dar, die in diesem Fall der Bankangestellte verrichtet oder auch in Anspruch nimmt. Typischerweise würde er die Bonitätsprüfung als Service eines anderen Unternehmens nutzen. Zusammengenommen wird durch die Komposition dieser Services der Geschäftsprozess „Kreditvergabe“ implementiert, aus Sicht des Antragstellers wurde aber nur ein einzelner Service in Anspruch genommen. Es interessiert den Kunden auch nicht, welche Tätigkeiten der Bankangestellte im Einzelnen durchführen muss, um den Service zu erbringen. Ein Geschäftsprozess, der andere Services nutzt, wird nach außen wiederum als Service angeboten und kann in anderen Geschäftsprozessen verwendet werden. Dies bezeichnet man als *rekursive Komposition* [WCL+05].

Werden Services auf Softwarekomponenten direkt abgebildet, so haben diese Softwarekomponenten genau die *Granularität* der fachlichen Services und können dadurch leicht in verschiedene Geschäftsprozesse integriert werden. So kann ein Service zur Prüfung der Kreditwürdigkeit sowohl bei der Kreditvergabe als auch bei der Kontoeröffnung verwendet werden. Wäre die Granularität der Softwarekomponente geringer, so müssten die Geschäftsprozesse immer mehrere „technische“ Komponenten aufrufen, wodurch die Komplexität unnö-

tig steigt. Bei einer größeren Granularität leidet wiederum die Wiederverwendung. Ist ein Service nicht profitabel kann dieser leicht in ein anderes Unternehmen ausgelagert werden bzw. von diesem bezogen werden. Da die bestehenden Systeme an Services ausgerichtet sind, können sie relativ leicht so angepasst werden, dass sie Services von externen Unternehmen verwenden.

Insbesondere dann, wenn mehrere Services dieselbe Funktionalität anbieten, ist es unabdingbar, dass der Service durch *Metadaten* beschrieben wird, um aufgefunden werden zu können. Diese enthalten Informationen zur Funktionalität des Services, aber auch zu nichtfunktionalen Eigenschaften wie Dienstqualität, Preis für die Nutzung usw. Dadurch kann der Service erst aufgefunden und ausgewählt werden.

Ein weiteres wichtiges Ziel von SOA ist die *lose Kopplung* der Softwarekomponenten. Eine Voraussetzung dafür ist die Unabhängigkeit von der Implementierungstechnologie. Softwaresysteme innerhalb eines Unternehmens verwenden typischerweise eine Vielzahl von verschiedenen Technologien. Wichtig ist nun, dass die Implementierungstechnologie hinter der Service-Schnittstelle verborgen wird. Die Schnittstellen werden dabei durch eine standardisierte, plattformunabhängige Sprache beschrieben. Technologisch gesehen ist ein Service also ein Softwaresystem, dessen Schnittstelle durch eine standardisierte Sprache beschrieben wird, die unabhängig von Implementierungstechnologien ist.

SOA-Dreieck

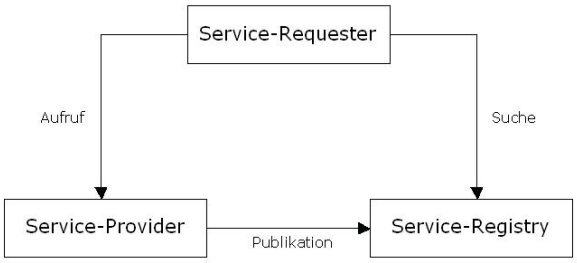

Abbildung 1.1: SOA-Dreieck

Die SOA-Architektur wird häufig durch die in Abbildung 1.1 dargestellte Dreiecksbeziehung mit den Akteuren Service-Requester, Service-Provider und Service-Registry charakterisiert:

- Der *Service-Provider* bietet Services an. Er publiziert die Servicebeschreibung in der Service-Registry, damit sie von Interessenten aufgefunden werden können. Die Metadaten enthalten Informationen zur Funktionalität des Services, wie auf ihn zugegriffen werden kann und u.U. seine nichtfunktionalen Eigenschaften, wie Dienstgüte, Preis, usw.
- Der *Service-Requester* sucht zunächst in der Service-Registry nach einem geeigneten Service. Die Service-Registry bietet eine entsprechende Schnittstelle an, über die der Service-Requester eine Suchanfrage stellen kann. Die Service-Registry liefert eine Liste von Services zurück, aus der der Requester den für ihn passendsten auswählt. Um den

Service nutzen zu können, muss der Service-Requester eine Bindung herstellen, dies bedeutet die Adresse, Transportprotokoll und Nachrichtenformate bestimmen, um mit diesem kommunizieren zu können. Diese Information hat er entweder schon von der Service-Registry erhalten oder er muss sie vom Service-Provider in einem separaten Schritt anfordern.

- Die *Service-Registry* verwaltet Service-Metadaten und bietet Schnittstellen für Service-Requester und Service-Provider an, um diese abzufragen bzw. zu veröffentlichen.

Eine Voraussetzung für das Funktionieren dieses Ansatzes, ist das Vorhandensein von Standards für die einzelnen Aktionen bzw. Schnittstellen der drei Akteure.

Aus Sicht des Service-Requesters ist die Interaktion relativ aufwändig. Er muss das Suchen, die Auswahl und das Binden selbst implementieren bzw. durchführen. Dieser Prozess kann für den Requester vereinfacht bzw. transparent gemacht werden, indem die Tätigkeiten an eine Middleware delegiert werden. Diese wird im Kontext von SOA als Service Bus bezeichnet [WCL+05].

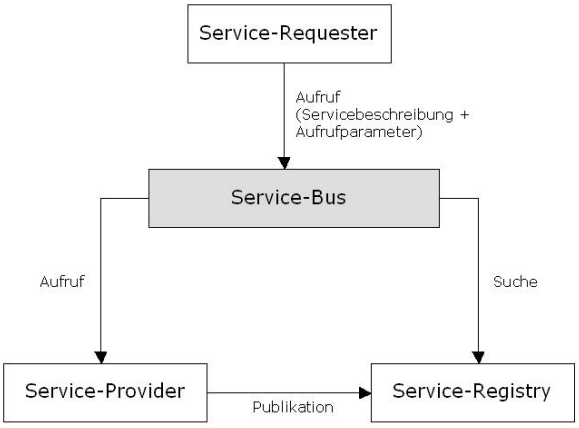

Abbildung 1.2: Service Bus

Wie in Abbildung 1.2 dargestellt, übergibt der Requester die gewünschte Servicebeschreibung und zusätzlich die Eingabedaten für den Serviceaufruf an den Service-Bus. Dieser sucht den passendsten Service aus, indem er eine entsprechende Anfrage an die Service-Registry stellt, führt dann eine Bindung durch, ruft den Service auf und übergibt die Ausgabedaten an den Service-Requester zurück. Bei einem solchen Aufruf weiß der Requester nicht und will auch nicht wissen, welcher konkrete Service eigentlich aufgerufen wurde, er ist nur an dem Ergebnis bzw. an der Erbringung eines Service interessiert. Der aufgerufene Service ist aus Sicht des Service-Requesters virtuell.

1.1.3 Web Services

Web Services stellen eine konkrete XML-basierte Realisierung des abstrakten SOA-Architekturparadigmas dar. Es gibt zahlreiche Definitionen von Web Services, die sich in

Sichtweise und Detailliertheit unterscheiden. Das World Wide Web Consortium (W3C) gibt die folgende Definition [Haa04]:

> 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.

Ein Web Service ist also ein Softwaresystem, das über das Netzwerk aufgerufen werden kann. Er besitzt eine Schnittstelle, die in WSDL beschrieben und über SOAP-Nachrichten erreichbar ist, wobei typischerweise webbasierte Standards wie HTTP und XML verwendet werden.

Die Technologien SOAP und WSDL werden als Web Services-Basistechnologien bezeichnet. Sie sind bereits in Form von Spezifikationen standardisiert und stehen in vielen Produkten als Implementierung zur Verfügung. Um aber Unternehmensanwedungen plattformunabhängig integrieren zu können, müssen auch „fortgeschrittene“ Aspekte wie Transaktionen, Sicherheit und Geschäftsprozesse unterstützt werden.

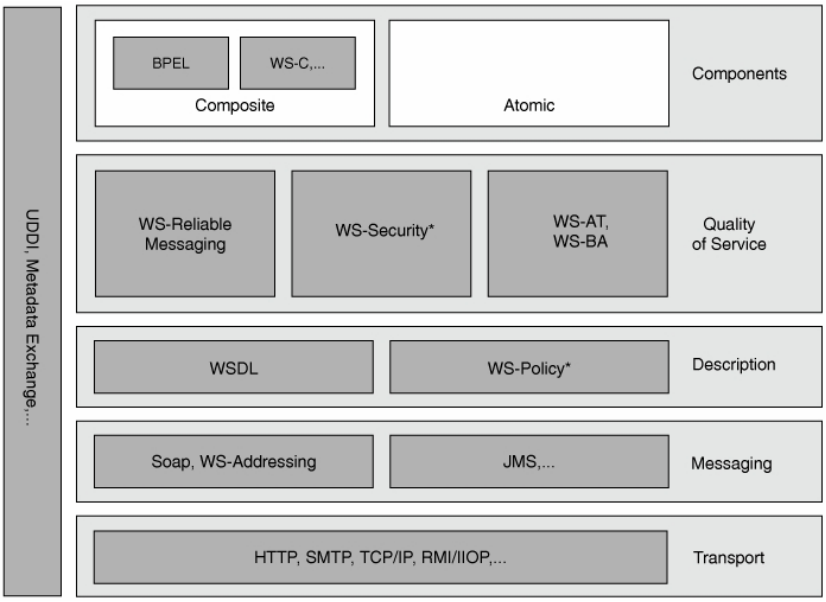

Abbildung 1.3: Web Services Technologien

In Abbildung 1.3 [WCL+05] werden Web-Services-Technologien in einem Schichtenmodell veranschaulicht. Dieses zeigt, dass neben SOAP und WSDL noch weitere WS-Technologien notwendig sind, um sinnvolle Anwendungen bauen zu können. Zu beachten ist, dass dies noch lange nicht alle WS-Spezifikationen sind, sondern eine Auswahl der im Moment relevantesten aus Sicht von [WCL+05]. Es gibt viele weitere, teilweise konkurrierende Vorschläge, da WS-Spezifikationen von verschiedenen Organisationen standardisiert werden, z.B. W3C, OASIS, bzw. von verschiedenen Unternehmen zur Standardisierung eingereicht werden. Einige der dargestellten Spezifikationen sind noch nicht endgültig standardisiert.

Im Folgenden werden nur diejenigen WS-Technologien näher vorgestellt, die für diese Arbeit von Bedeutung sind. Einen umfassenden Überblick über derzeitige Technologien, Standards, Sprachen und Entwicklungsplattformen für Web Services geben z.B. [ACKM03] und [WCL+05].

Interaktion mit Web Services

Web Services kommunizieren über Nachrichten. Wie diese Nachrichten über das Netzwerk transportiert werden, wird durch Protokolle in der Transportschicht definiert, wie in Abbildung 1.3 dargestellt. Die Basistechnologie SOAP [GHM+03] (W3C) liegt in der darüber liegenden Schicht und definiert wie Nachrichten zwischen Web Services übertragen werden. SOAP definiert ein XML-basiertes Nachrichtenformat und legt fest, wie SOAP-Nachrichten zu verarbeiten sind. SOAP abstrahiert von den darunter liegenden Transportprotokollen.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
    xmlns:wssec="http://schemas.xmlsoap.org/ws/2002/04/secext">
  <env:Header>
    <wsa:To>http://www.example.com/stq</wsa:To>
    <wssec:Security>
      <wssec:BinarySecurityToken
        ValueType="wssec:X509v3"
        EncodingType="wssec:Base64Binary">
        dWJzY3JpYmVyLVBic.....eFw0wMD
      </wssec:BinarySecurityToken
    </wssec:Security>
  </env:Header>
  <env:Body>
    <app:TrafficStatus xmlns:app="http://highwaymon.org/payloads">
      <road>520W</road>
      <speed>3MPH</speed>
    </app:TrafficStatus>
  </env:Body>
</env:Envelope>
```

Listing 1.1: SOAP-Nachricht

Wie in der Abbildung dargestellt ist eine SOAP-Nachricht ein XML-Dokument bestehend aus einem SOAP-Envelope-Element. Dieses besteht wiederum aus einem optionalen SOAP-Header und einem SOAP-Body. Der SOAP-Body enthält den Payload, also die eigentliche Nachricht für den Empfänger. Der SOAP-Header dagegen enthält technische Informationen, die mit der Zustellung der Nachricht zu tun haben und für die entsprechende Middleware bestimmt sind. Typische Informationen sind z.B. Security-Token und Transaktions-Ids. Die Elemente im SOAP-Header sind nicht Teil der SOAP-Spezifikation sondern domänenspezifisch. Im Beispiel oben wird beispielsweise ein Element aus der WS-Security-Spezifikation verwendet.

SOAP-Nachrichten können über verschiedene Protokolle der Transportschicht an den Empfänger transportiert werden. Ein typisches Transportprotokoll wäre z.B. HTTP. Wenn eine

SOAP-Nachricht über HTTP an den Empfänger geschickt wird, so sind die Adressierungsinformationen des Empfängers bzw. des Web Services als auch des Senders auf HTTP-Ebene zu finden. Die SOAP-Nachricht könnte aber auch über andere bzw. mehrere Protokolle auf dem Weg zum Empfänger transportiert werden. Deshalb ist ein Addresierungsschema für Web Services notwendig, das unabhängig von Protokollen auf der Transportschicht ist. Dieses wird durch WS-Addressing spezifiziert.

WS-Addressing [BCC+04] beschreibt das Konzept der Endpoint Reference (EPR), die Addressierungsinformationen eines Web Service Endpoints modelliert und definiert zusätzlich wie diese EPR auf SOAP-Header abzubilden ist.

Eine EPR ist eine Datenstrukur bestehend aus folgenden Elementen:

- *Address:* Gibt die Adresse des Endpoints als URI an und ist das einzige Muss-Feld.
- *Reference Properties:* Domänenspezifische XML-Elemente, die zur Identifikation des Endpoints dienen bzw. die Nachrichtenzustellung erst ermöglichen, da unter einer Adresse mehrere Endpoints erreichbar sein können. Ein Beispiel dafür wäre ein Online-Einkauf. Alle Kunden rufen dieselbe Adresse auf. Um aber den Endpunkt, der den Zustand des kundenspezifischen Einkaufskorbs hält, zu referenzieren wird eine „Kunden-Id“ mitgeliefert.
- *Reference Parameters:* Domänenspezifische XML-Elemente, die nicht zur Identifikation des Endpoints dienen, aber trotzdem für die Kommunikation mit dem Endpoint benötigt werden.
- *PortType:* Der qualifizierte Name des WSDL-PortTypes des Endpunkts. Enthält ein Endpoint eine WSDL-Schnittstelle, so werden durch das PortType die Nachrichtentypen definiert, die mit dem Endpoint ausgetauscht werden.
- *Port/Service:* Der Name des WSDL-Ports und des WSDL-Services des Endpoints. Ein WSDL-Port enthält ein Binding, das definiert, wie die Nachrichten, die mit dem Endpoint ausgetauscht werden, serialisiert werden.
- *Policy:* Eine WS-Policy-Expression definiert die technischen Aspekte der Kommunikation, wie Transaktionen und Sicherheit.

Der Aufrufer bzw. Nutzer eines Web Service benötigt eine EPR, diese bekommt er z.B. von der Service-Registry. Ist die EPR vorhanden, dann kann eine SOAP-Nachricht an den Endpunkt geschickt werden. Dabei definiert WS-Addressing wie die EPR-Elemente auf SOAP-Header abgebildet werden. Beispielsweise wird das „Address“-Feld aus der EPR auf das „wsa:To“-Element abgebildet, wie im obigen Beispiel gezeigt.

Servicebeschreibung

Eine Servicebeschreibung besteht aus Metadaten, die benötigt werden, um den Service auffinden bzw. aufrufen zu können. Um einen Service aufrufen zu können, muss der Service-Requester wissen, welche Nachrichten in welchem Format über welches Transportprotokoll er an welche Adresse schicken muss. Diese Informationen gehören zur funktionalen Beschreibung eines Services und werden in WSDL formuliert.

Zu nichtfunktionalen Eigenschaften gehören Dienstgüte-Eigenschaften (QOS) wie Sicherheit, transaktionales Verhalten, Reliable Messaging. Diese Eigenschaften können sich zusätzlich zu den funktionalen darauf auswirken, wie die Anfragen formuliert werden, beispielsweise kann ein Service Authentifizierung durch den Requester verlangen. Für diese Art von Service-Eigenschaften wird WS-Policy verwendet.
Metadaten wie Name des Anbieters oder Kategorisierungen innerhalb von Taxonomien werden im Zusammenhang mit der Publikation und dem Auffinden von Services weiter unten betrachtet.

Web Services Description Language

Die Web Services Description Language (WSDL) ist eine vom W3C standardisierte XML-basierte Sprache zur funktionalen Beschreibung von Web Services. Im Moment ist die Version 1.1 [CCMW01] eine W3C Recommendation, während der Nachfolger WSDL 2.0 noch Working-Draft-Status hat. WSDL 2.0 wird im Folgenden nicht betrachtet, da es noch nicht endgültig fertiggestellt ist und BPEL ausschließlich WSDL 1.1 verwendet.

WSDL beschreibt die Schnittstelle eines Services, der in einer beliebigen Technologie implementiert sein kann. Dadurch, dass WSDL bei W3C standardisiert ist und als Standard in der Industrie auch anerkannt ist und verwendet wird, können Web Services als Integrationstechnologie zwischen proprietären Technologien wie J2EE und .NET verwendet werden.

Eine WSDL 1.1-Beschreibung eines Services enthält folgende Elemente:

- *Types:* Dieser Abschnitt definiert die Datentypen, die in Nachrichten (Messages) verwendet werden. Für die Definition wird typischerweise XML-Schema verwendet, es können aber auch andere Typsysteme wie Java eingesetzt werden.
- *Message:* Ein Message-Element definiert eine Nachricht, die entweder als Eingabe, Ausgabe oder Fehlerausgabe einer Operation fungiert. Es gliedert sich weiter in Part-Elemente, die als Typ eines der unter „Types“ definierten Datentypen erhalten.
- *Operation:* Eine Operation kapselt eine Funktionalität, die der Service anbietet. Sie enthält optional Eingabe-, Ausgabe und Fehler-Messages.
- *PortType:* Ein PortType-Element bündelt mehrere Operation-Elemente und stellt die abstrakte Schnittstelle eines Services dar.
- *Binding:* Ein Binding-Element bindet die abstrakte PortType-Schnittstelle an konkrete Protokolle. Ein SOAP/HTTP-Binding definiert beispielsweise wie ein HTTP-Request bzw. die darin enthaltene SOAP-Nachricht auszusehen hat, um eine Operation aufzurufen.
- *Port:* Ein Port-Element beschreibt einen konkreten Web-Service-Endpoint, indem er das zugehörige Binding angibt und die Adresse, unter welcher dieser erreichbar ist.
- *Service:* Ein Service bündelt mehrere Port-Elemente.

Ein WSDL-Dokument besteht also aus zwei Teilen: einem abstrakten PortType, der die Schnittstelle anhand von eingehenden und ausgehenden Nachrichten beschreibt und einem konkreten Teil, der beschreibt *wie* (Binding) und *wo* (Port) auf den Service zugegriffen werden kann.

WS-Policy

Das Web Services Policy Framework [BBC+04a] (WS-Policy) ist eine XML-basierte Sprache zur Spezifikation von nichtfunktionalen Beschreibungen und Anforderungen an Services mit dem Schwerpunkt auf Dienstgüte-Eigenschaften. WS-Policy ist noch kein WS-Standard.

```
<wsp:Policy>
  <wsp:ExactlyOne>
    <wsp:All>
      <wsse:SecurityToken>
        <wsse:TokenType>wsse:Kerberosv5TGT</wsse:TokenType>
      </wsse:SecurityToken>
    </wsp:All>
    <wsp:All>
      <wsse:SecurityToken>
        <wsse:TokenType>wsse:X509v3</wsse:TokenType>
      </wsse:SecurityToken>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

Listing 1.2: WS-Policy-Expression

Im aufgeführten Beispiel wird eine WS-Policy-Expression in der Normalform gezeigt. Eine Policy-Expression besteht aus mehreren Alternativen. Jede Alternative besteht wiederum aus einer Kombination von Assertions. Die Assertions sind domänenspezifisch und nicht Teil der WS-Policy-Spezifikation, z.B. spezifiziert WS-RM Policy [BF03] die Assertions für Reliable Messaging, WS-SecurityPolicy [DLGHB+05] dagegen Sicherheit-spezifische Assertions. Das Beispiel zeigt zwei Assertions aus WS-SecurityPolicy.
WS-Policy macht keine Unterschiede zwischen Constraints und Capabilities, d.h. zwischen Anforderungen an den Kommunikationspartner und eigenen Fähigkeiten. Wenn ein Requester und ein Provider miteinander kommunizieren wollen, müssen sie zunächst ihre Policies abgleichen und die kompatiblen Alternativen herausfinden. Dieser Abgleich kann auf Basis von WS-Policy nur zum Teil erfolgen, da die Assertions außerhalb der WS-Policy-Spezifikation definiert werden. WS-Policy kann lediglich die Struktur und die Namen der verwendeten Assertions als Vergleichsgrundlage nehmen und so wenigstens die Alternativen aussortieren, die nicht in Frage kommen. Auf Basis einer der kompatiblen Alternativen, wenn vorhanden, findet dann die Kommunikation statt.

WS-PolicyAttachment [BBC+04b] definiert wie WS-Policy-Expressions mit Subjekten, die sie beschreiben sollen, verbunden werden. Als Subjekte kommen z.B. in Frage: beliebiges XML-Dokument, WSDL-Message, WSDL-PortType, WSDL-Port. Es werden zwei Mechanismen zur Verfügung gestellt:

- Annotation von Subjekten, die beliebige XML-Elemente sein können. Dazu können wahlweise das Attribut „PolicyURIs“ oder „PolicyReference“-Elemente verwendet werden, die auf die Policy-Definitionen durch URIs verweisen.
- Verweis auf Subjekte von außen. Dazu werden innerhalb eines PolicyAttachment-Elements die Subjekte und die zugehörigen Policies mit Hilfe von URIs identifiziert.

Zusätzlich definiert WS-PolicyAttachment wie Policies in WSDL und UDDI eingebettet werden.

Publikation und Auffinden von Services

Die Publikation und das Auffinden von Services in einer zentralen Service-Registry wird durch die UDDI-Spezifikation ermöglicht [TC02]. Diese definiert die Datenstrukturen, die notwendig sind, um die relevanten Informationen in der Registry abzuspeichern und jeweils eine API für den Service-Requester und Service-Provider um auf die Registry zuzugreifen.

Eine UDDI-Registry unterstützt drei Arten von Informationen, aus welchen sich unterschiedliche Möglichkeiten zur Suche nach geeigneten Web Services ableiten lassen:

- Über *White-Pages*-Informationen können Web Services anhand von Provider-Daten wie Name und Adresse wie in einem normalen Telefonbuch gesucht werden.
- *Yellow-Pages*-Informationen ermöglichen eine auf Kategorien basierende Suche ähnlich den „Gelben Seiten“. Standardisierte und typisch verwendete Kategorien sind NAICS, UNSPSC, DUNS und ISO-3166.
- *Green-Pages*-Informationen umfassen schließlich technische Details der publizierten Web Services, wie z.B. zugehörige WSDL-und WS-Policy-Dokumente. Man kann also beispielsweise nach Services suchen, die einen bestimmten PortType implementieren.

1.1.4 Business Process Execution Language

Wie bereits im Abschnitt 1.1.2 beschrieben, sind Geschäftsprozesse ein zentrales Thema von SOA. Im Kontext von Web Services spricht man bei einer prozessbasierten Komposition von Web Services von sogenannten Web Service Flows (WS-Flows). Diese sind, wie der Name erkennen lässt, als das Web Service-Pendant von traditionellen Workflow-Systemen [LR00] zu sehen. Ein WS-Flow implementiert also einen Geschäftsprozess, indem er andere Web Services, im Folgenden als Partner Services bezeichnet, aufruft.

Die Business Process Execution Language (BPEL) ist eine XML-basierte Sprache zur Modellierung von WS-Flows [ACD+03]. BPEL ist der Nachfolger zweier zuvor konkurrierender WS-Flow-Sprachen: der Web Services Flow Language (WSFL) [Ley01] von IBM und der Sprache XLANG [Tha01] von Microsoft. BPEL kombiniert Konzepte aus beiden Sprachen und ist im Moment der De-Facto-Standard für die Definition von WS-Flows. Die Begriffe WS-Flow und BPEL-Prozess werden im Folgenden im Kontext von BPEL synonym verwendet.

Ein BPEL-Prozess ruft Partner Services auf, die durch WSDL beschrieben sind, und ist nach außen selbst auch wieder ein Web Service, besitzt also eine WSDL-Schnittstelle. Die Definition eines BPEL-Prozesses, ein BPEL-Prozessmodell, besteht aus folgenden Entitäten:

- *Partner Links:* Der Prozess definiert durch Partner Links die Services, mit denen er interagieren will. Ein Partner Link kapselt eine WSDL 1.1-PortType-Beschreibung.

- *Variablen:* Variablen werden typischerweise dazu verwendet um die Eingabe- und Ausgabenachrichten von Partner Services zu verarbeiten, zwischenzuspeichern usw. Die Variablen besitzen als Typ entweder eine WSDL-Message oder einen XML-Schema-Datentyp.

- *Aktivitäten:* Die „Abfolge“ in der die einzelnen Partner Services aufgerufen werden, der Kontrollfluss, wird mit Hilfe von Aktivitäten definiert. BPEL unterscheidet zwischen Basisaktivitäten und strukturierten Aktivitäten. Strukturierte Aktivitäten ermöglichen die Schachtelung von anderen Aktivitäten. Eine Sequence-Aktivität definiert z.B., dass die darin enthaltenden Aktivitäten nacheinander ausgeführt werden sollen, während eine Flow-Aktivität eine parallele Abarbeitung erlaubt. Switch- und While-Aktivitäten entsprechen den Konstrukten aus den gängigen Programmiersprachen. Die Aktivitäten sind durch Links verbunden. Diese können Übergangsbedingungen definieren. Nur wenn die Bedingung erfüllt ist, wird die Zielaktivität des Links ausgeführt. Die Bedingungen werden typischerweise in XPath oder XQuery formuliert. Für die Kommunikation mit Partner Services werden die Basisaktivitäten Invoke, Receive und Reply verwendet. Durch eine Invoke-Aktivität wird eine WSDL-Operation eines Partners Services aufgerufen. Eine Receive-Aktivität erhält eine WSDL-Message von einem Partner Service, der Prozess bietet dazu eine WSDL-Schnittstelle an. In Request-Reply-Operationen gibt die zugehörige Reply-Aktivität ein Ergebnis an den Partner Service zurück.

- *„Fortgeschrittene“ Konstrukte:* BPEL bietet auch Unterstützung für Fehlerbehandlung, Kompensation, Timer und Korrelation, die für diese Arbeit aber keine Rolle spielen.

Durch BPEL können sowohl ausführbare als auch nicht ausführbare, sogenannte abstrakte Prozessmodelle definiert werden. Für diese Arbeit sind aber nur ausführbare Prozessmodelle relevant. Ein ausführbares Prozessmodell wird in einer Ablaufumgebung, im Folgenden als BPEL-Engine bezeichnet, installiert und darin ausgeführt. Nach außen bietet ein BPEL-Prozess eine WSDL-Schnittstelle. Die Ausführung des Prozesses beginnt mit dem Aufruf einer WSDL-Operation dieser Schnittstelle von außen. Dabei erstellt die BPEL-Engine eine neue Prozessinstanz und führt diese aus, indem sie durch den Kontrollfluss navigiert.

Das Prozessmodell referenziert Partner Services innerhalb von Partner Links durch WSDL-PortType-Beschreibungen, also den abstrakten Teil der WSDL-Definition. Für die Bindung konkreter WSDL-Ports bzw. Endpoints über EPRs gibt es mehrere Möglichkeiten. Die Bindung kann aus Sicht des Prozessmodells transparent oder nicht transparent geschehen. Transparent bedeutet, dass nicht das Prozessmodell selbst, sondern die BPEL-Engine die Bindung durchführt. Im Gegensatz dazu kann auch im Prozessmodell selbst die Zuweisung durchgeführt werden. BPEL erlaubt nämlich die Zuweisung von EPRs an Partner Links mit Hilfe der Assign-Aktivität. Daraus ergeben sich vier Möglichkeiten für die Bindung [WCL+05]:

- *Statische Zuweisung von EPRs:* Ist die EPR des Partner Services zur Design-Zeit bekannt, kann diese im Prozessmodell hartkodiert und an den Partner Link mit Hilfe der Assign-Aktivität zugewiesen werden. Dies macht allerdings nur dann Sinn, wenn die EPR Teil der Geschäftslogik ist, also im BPEL-Prozess in irgendeiner Form verarbeitet wird, sonst ist die folgende Strategie sinnvoller.

- *Statisch zur Deployment-Zeit:* Im Vergleich zur ersten Strategie referenziert das Prozessmodell hier die Endpoints nicht direkt, die Bindung ist also transparent. Die Bindung an einen bestimmten Endpoint wird zur Deployment-Zeit in der BPEL-Engine konfiguriert und gilt dann für alle Prozessinstanzen.
- *Dynamische Bindung anhand von Auswahlkriterien:* Es werden Auswahlkriterien spezifiziert anhand derer passende Endpoints gesucht werden. Als Kriterien kommen z.B. Dienstgüte, Preis usw. in Frage. Die Spezifikation dieser Kriterien und die unterstützende Infrastruktur ist nicht Teil der BPEL-Sprache und müsste als Erweiterung einer BPEL-Engine entwickelt werden.
- *Dynamische Zuweisung von EPRs:* Im Vergleich zur ersten Strategie werden auch hier die EPRs über Assign-Aktivitäten an die Partner Links zugewiesen, allerdings wurden sie von einem Partner Service als Ergebnis eines Aufrufs geliefert.

Diese Arbeit beschäftigt sich im Folgenden mit der dritten Möglichkeit der Bindung, die anderen sind nicht relevant.

1.2 Motivation

1.2.1 Dynamische Bindung an Partner Services

Ein WS-Flow ruft externe Funktionalität der Partner Services über WSDL-Schnittstellen auf. Das Prozessmodell referenziert dabei im Fall von transparenter Bindung (siehe Abschnitt 1.1.4) nur die WSDL-PortTypes, d.h. den abstrakten Teil der WSDL-Schnittstelle. Die konkrete Bindung an einen WSDL-Port geschieht typischerweise zur Deployment-Zeit des Prozesses. Diese Bindung an einen konkreten Port gilt dann entsprechend für alle Prozessinstanzen.

Es wäre zwar durchaus möglich die Bindung auch zur *Laufzeit* bei Instanziierung des Prozessmodells für jede Prozessinstanz einzeln oder auch unmittelbar vor dem ersten Aufruf des Partners durchzuführen. Dieses „späte Binden“ würde aber nur dann Sinn machen, wenn der Partner aus mehreren Alternativen ausgewählt werden könnte. In diesem Fall wäre es nämlich durchaus in manchen Fällen wünschenswert die Auswahl möglichst spät auszuführen, um alle zum Zeitpunkt der Prozessausführung in Frage kommenden Alternativen bei der Auswahl berücksichtigen zu können. Die Bindung zur Laufzeit könnte durch einen Prozessadministrator durchgeführt werden, dazu müsste aber dis Ausführung der Prozessinstanz entsprechend längere Zeit angehalten werden, oder aber die Auswahl findet automatisch statt. Dazu müssten entsprechende Such- und Auswahlkriterien spezifiziert werden.

Soll eine Aktivität beispielsweise immer den preiswertesten Service aufrufen, so kann der Prozessmodellierer zur Modellierungs- oder Deployment-Zeit einen solchen auswählen und an die Aktivität binden. Nun kann es aber sein, dass später zur Laufzeit weitere noch preiswertere Partner Services zur Verfügung stehen. In diesem Fall müsste das Prozessmodell gestoppt, geändert und neu deployt werden, was v.a. bei langlaufenden Prozessen aufwändig und unflexibel ist. Besser wäre es, wenn man bereits zur Modellierungszeit deklarativ

die Auswahl-Strategie „günstigster Service“ spezifizieren könnte, so dass eine entsprechende Auswahl anhand dieses Kriteriums zur Laufzeit *automatisch* erfolgen kann. So würde also jede Prozessinstanz, den für sie im Moment des Aufrufs günstigsten Service finden, z.B. über eine Suche in einer Service-Registry, und automatisch an diesen gebunden werden.

Die WS-Flow-Sprache WSFL [Ley01] unterstützt die Bindung von Ports zur Laufzeit in Form des „locator“-Konstrukts. Dieses wird als Kindelement der Partner-PortType-Definition (in WSFL „serviceProvider“) spezifiziert und bestimmt welcher Port gesucht und gebunden werden soll. Als Locator-Strategie wird u.a. eine Suche in der UDDI-Registry unterstützt. Dabei wird eine UDDI-Anfrage mit Hilfe der UDDI-Inquiry-API spezifiziert. Über das Attribut „bindTime“ kann gesteuert werden, wann die Suche stattfinden soll: beim Deployment, während der Prozessinstanziierung oder vor dem ersten Aufruf einer Operation. Das Attribut „selectionPolicy“ spezifiziert, welcher Port auszuwählen ist, wenn mehrere Alternativen gefunden wurden: erster Port, zufällige Auswahl oder Angabe eines Programms, der die Auswahl durchführt.

Im Gegensatz zu WSFL hat BPEL (und auch XLANG) keine direkte Unterstützung auf Sprachebene für die dynamische Bindung von Ports durch vorherige Suche und Auswahl aus einer Liste von Alternativen. Wann die Bindung an einen Port stattfindet, ist Teil der BPEL-Engine-Implementierung und wird von der BPEL-Sprache nicht vorgegeben oder eingeschränkt.

Ähnlich wie WSFL durch das Locator-Konstrukt, kann auch BPEL entsprechend erweitert werden. In [KHC+05] wird vorgeschlagen ein „Find-Bind“-Element unter die Aktivitäten zu hängen, die dynamische Suche und Auswahl unterstützen sollen. Dieses Erweiterungselement enthält dann die Auswahlkriterien, anhand derer nach einer Suche in der UDDI-Registry die Auswahl stattfinden soll. Die Auswahl findet dann anhand von Policies statt, die beispielsweise in WS-Policy formuliert werden könnten. Es wird aber in [KHC+05] kein konkreter Vorschlag bezüglich des Formats oder Inhalts einer Such- oder Auswahlanfrage gemacht. Neben der dynamischen Bindung anhand von Policies erlaubt das Find-Bind-Konstrukt den Wechsel auf einen alternativen Partner Service zur Laufzeit, im Fall, dass der im Moment gebundene nicht erreichbar ist. Dadurch wird ein Fehler, der u.U. zum Abbruch der Prozessinstanz führen würde, verhindert.

Zusammenfassend lässt sich festhalten, dass die dynamische Bindung von Ports anhand von Auswahlkriterien die Flexibilität von Prozessen erhöht, denn die Prozessinstanz kann zur Laufzeit automatisch den im Moment passendsten Service auswählen. Anhand des vorhandenen PortTypes kann eine Suche in einer UDDI-Registry eine Auswahl von implementierenden Ports liefern. Die Schwierigkeit besteht in der Formulierung der Auswahlkriterien, da es noch keine Standards bezüglich der Beschreibung von nichtfunktionalen Eigenschaften von Web Services gibt.

1.2.2 Parametrisierung von Web Service Flows

Wie im letzten Kapitel dargestellt, wird durch die Möglichkeit WSDL-Ports zur Laufzeit anhand von Auswahlkriterien zu suchen und zu binden die Flexibilität von WS-Flows erhöht. In [KLB05] wird argumentiert, dass die Suche nach geeigneten Partner Services immer noch

dadurch eingeschränkt ist, dass nur diejenigen Services für die dynamische Suche bzw. Bindung in Frage kommen, die genau die gleiche WSDL-PortType-Beschreibung besitzen wie zur Modellierungszeit des Prozessmodells spezifiziert. Die Gleichheit bezieht sich dabei auf die *Namen* des PortTypes und der zugehörigen Operationen. Ein Name besteht dabei aus dem Namespace und dem lokalen Namen im WSDL-Dokument.

Implementieren also zwei Services mit unterschiedlichen PortType-Namen dieselbe Funktionalität und haben auch die gleiche Schnittstelle, was die übertragenen Nachrichtenformate betrifft, sind sie aus Sicht des BPEL-Prozesses unterschiedliche Services, weil das Prozessmodell die PortTypes und Operations über Namen referenziert. Das führt dazu, dass Services, die dieselbe Funktionalität erfüllen aber lediglich die Namen der WSDL-Einheiten anders vergeben haben, nicht alternativ verwendet werden können.

In [KLB05] wird weiter argumentiert, dass man sich in der Praxis durchaus auf die Nachrichtentypen, also die ausgetauschten Nachrichtenformate einigen kann, z.B. in Form von domänenspezifischen XML-basierten Standards wie RosettaNet PIPs [Ros], aber kaum auf die Namen der PortTypes und Operations bzw. die Gruppierung von Operations innerhalb von PortTypes. Da also WSDL-Schnittstellen bezüglich der Namensgebung nicht standardisiert und häufig automatisch aus dem Programmcode generiert werden, wobei dadurch die Namen der WSDL-Einheiten automatisch anhand der Programmcode-Benennung festgelegt sind, haben in der Praxis Services mit der gleichen Semantik verschiedene WSDL-Schnittstellen. Ein BPEL-Prozess ist dadurch bei der automatischen Suche zur Laufzeit nach alternativen Services stark eingeschränkt. Es werden nur die Services gefunden, die exakt die gleiche WSDL-PortType-Beschreibung haben.

Die starke Kopplung an Namen ist nicht im Sinne von SOA, denn statt zu sagen, welcher Service bzw. Funktionalität benötigt wird, müssen die genauen Namen der syntaktischen Einheiten bekannt sein. Die *lose Kopplung*, eines der Hauptziele von SOA, ist dadurch eingeschränkt. Diese wird nur im Bezug auf die WSDL-Ports erreicht.

Parametrisierung von PortType- und Operation-Namen

Als Lösung für dieses Problem wird in [KLB05] vorgeschlagen die PortType- und Operation-Namen der Partner Services, die erst zur Laufzeit bestimmt werden sollen, in den entsprechenden Aktivitäten durch Parameter zu ersetzen. Dadurch wird die Bindung an feste Namen aufgehoben. Die Parameter werden dann zur Laufzeit durch konkrete Werte, nämlich die PortType- und Operation-Namen des ausgewählten Services, substituiert.

Für die Substitution werden in [KLB05] mehrere alternative Strategien vorgeschlagen. Diese sind Teil des Prozessmetamodells und werden über Erweiterungselemente modelliert. Die Strategien spezifizieren wie ein passendes PortType/Operation-Paar bestimmt werden soll. Als nächster Schritt folgt dann die Auswahl implementierender Ports.

Die vier alternativen Strategien lauten:

- *static*: Parameter bekommen von vornherein feste Werte zugewiesen, d.h. der Effekt ist gleich wie ohne Parameter.

- *prompt*: Ein Prozessadministrator soll die Angaben zur Laufzeit machen, typischerweise über eine entsprechende GUI-Oberfläche. Die Eingabe kann einmal bei Instanziierung des Prozesses für alle parametrisierten Aktivitäten erfolgen, oder, v.a. bei langlaufenden Prozessen sinnvoll, unmittelbar vor Aufruf einer bestimmten Aktivität.
- *query*: Es wird eine Suchanfrage spezifiziert, anhand derer die passenden PortTypes gefunden werden können. Diese werden anhand von Kriterien wie funktionalen und nichtfunktionalen Eigenschaften, Übergangsbedingungen oder Geschäftsregeln ausgewählt. Die Suchanfragen können zur Modellierungszeit innerhalb der Prozessdefinition spezifiziert werden oder aus dieser heraus referenziert werden und daher flexibler auch zur Deployment-Zeit definiert werden können. Wann die Suchanfrage ausgewertet wird, ist eine Frage der Performance-Anforderungen des Prozesses. Wird die Suchanfrage kurz vor dem Aufruf der Aktivität ausgewertet, so gewinnt man an Flexibilität. Die Ausführung der Aktivität und damit des gesamten Prozesses wird aber um diese Zeit blockiert.
- *from variable*: Die Parameter werden aus einer Variable, die im Prozessmodell definiert ist, geliefert. Die entsprechenden Werte, die in dieser Variable gespeichert sind, können beispielsweise von einem vorher aufgerufenen Partner Service stammen.

Konkrete Verwendung in BPEL

In BPEL werden Partner Services durch die Invoke-Aktivität aufgerufen. Sie enthält als Attribute u.a. „portType“ und „operation“, deren Werte Namen der entsprechenden WSDL-Definition referenzieren. Um diese zu parametrisieren, wird in [KLB05] vorgeschlagen, das BPEL-Metamodell so zu erweitern, dass die Invoke-Aktivität ein Unterelement „evaluate“ erhält. Die erweiterte Invoke-Aktivität sieht dann folgendermaßen aus:

```
<invoke
  name="<activity-name>"
  partnerLink="<partnerLink-name>"
  portType="<portType-name>"
  operation="<operation-name>"
  inputVariable="<variable-name>"
  outputVariable="<variable-name>">

    <evaluate
      activated="yes"|"no"
      changeType="static"|
                 "prompt"|
                 "fromVariable"|
                 "query"
      substitute="<value>"
    />
</invoke>
```

Listing 1.3: Evaluate-Konstrukt

Das Evaluate-Element enthält folgende Attribute:

- *activated:* Gibt an, ob die Evaluation überhaupt stattfinden soll.

- *changeType:* Spezifiziert, welche der vier Evaluationsstrategien verwendet werden soll.
- *substitute:* Definiert im Zusammenhang mit der gewählten Strategie, welche Werte substituiert werden sollen. Im Falle von „query“ enthält er beispielsweise die Suchanfrage oder eine Referenz auf die Suchanfrage.

Das Evaluate-Element ist in einem eigenen Namensraum definiert. Das Hinzufügen von Unterelementen zur Invoke-Aktivität wird durch die BPEL-Sprache unterstützt, die Erweiterung ist also BPEL-konform. Sie benötigt aber eine entsprechende Erweiterung der BPEL-Engine-Implementierung, die das Evaluate-Element erkennt und ausliest. Eine Standard-BPEL-Engine-Implementierung wird also das Evaluate-Element einfach übergehen und den vorkonfigurierten PortType benutzen. In den Attributen „portType“ und „operation“ stehen Namen aus dem während der Modellierung verwendeten vorkonfigurierten PortType. Durch das Evaluate-Element wird festgelegt, wie zur Laufzeit ein PortType/Operation-Paar bestimmt werden soll.

1.2.3 Aspekte der Parametrisierung

Höhere Flexibilität und lose Kopplung

Parametrisierte Prozesse, also Prozesse, die mindestens einen parametrisierten Partner Service haben, erhöhen die *Flexibilität*, denn es können nun zur Laufzeit Services gefunden und gebunden werden, die zur Modellierungszeit noch nicht existierten und nicht dieselbe WSDL-Schnittstelle im Bezug auf PortType- und Operation-Namen haben. Dies ist wesentlich flexibler als die bloße dynamische Bindung an Ports, die im Abschnitt 1.2.1 vorgestellt wurde, da jetzt eine größere Anzahl von Services für die Auswahl in Frage kommt.

Semantik-Beschreibungen notwendig

Im Vergleich zur Suche nach Ports wird jetzt der Such- und Auswahlprozess schwieriger, da jetzt außer nichtfunktionalen Auswahlkriterien auch die Semantik der aufgerufenen Operation berücksichtigt werden muss. Die eigentliche Ursache des Problems liegt darin, dass die WSDL-Schnittstelle keine Semantikinformationen enthält. Sie spezifiziert wie die Operationen und ihre Parameter heißen und welche Datentypen, typischerweise in XML-Schema definiert, sie besitzen, aber nicht welche Funktionalität der Service eigentlich erbringt. Die Semantik ist nur implizit vorhanden. Der Prozessmodellierer kennt sie, indem er die textuelle Beschreibungen der WSDL-Schnittstelle liest oder er kann sie sich u.U. aus den Namen der WSDL-Einheiten herleiten, wenn diese sorgfältig gewählt wurden. Leider ist diese aber nicht maschinenverwertbar und dies ist notwendig, um eine automatische Suche zu ermöglichen. Semantik im Bezug auf Web Services wird im Kapitel 2 behandelt.

Format der ausgetauschten Nachrichten

Eine WSDL-Operation enthält u.U. eine Input- und/oder Output-Message. Diese werden in BPEL auf Variablen abgebildet. Vor und nach dem Aufruf der Invoke-Aktivität werden sie durch Assign-Aktivitäten vorbelegt und ausgelesen und in der Invoke-Aktivität über die Attribute „inputVariable“ und „outputVariable“ referenziert. Um einen Service aufzurufen, der Input- und/oder Output-Messages besitzt, reicht eine Parametrisierung des PortType-

und der Operation-Namen nur dann aus, wenn man annimmt, dass die Input- und Output-Messages genau die gleiche Struktur haben, wie die im Prozessmodell verwendeten.

Zustandsbehaftete Konversation

Wenn der Prozess mehrere Operationen eines Partners aufruft, die in Kombination verwendet werden, d.h. wenn eine zustandsbehaftete Konversation mit dem Partner stattfindet, so müsste der beim ersten Aufruf, d.h. beim Ausführen der ersten parametrisierten Invoke-Aktivität, gebundene Partner auch in den folgenden Invoke-Aktivitäten verwendet werden. Beim Binden des Partners im ersten Aufruf müsste sichergestellt werden, dass dieser auch die folgenden Operationen, die im Prozessmodell festkodiert sind, unterstützt. Nach dem ersten Aufruf würde der aufgerufene Partner an den Partner Link gebunden und in den folgenden Operationen dann wieder verwendet werden. Die zustandsbehaftete Konversation wie hier skizziert, wird nicht in [KLB05] behandelt und ist nicht Teil der Aufgabenstellung. Diese beschränkt sich auf zustandslose Interaktionen mit Partnern.

Vereinfachung des Kontrollflusses

In [KLB05] wird aufgezeigt, wie durch Parametrisierung in manchen Fällen der Kontrollfluss des Prozessmodells vereinfacht werden kann. Wird im Prozessmodell ein Serviceaufruf durch mehrere alternative Zweige modelliert, in BPEL beispielsweise durch eine Switch-Aktivität, so können die Bedingungen der Switch-Aktivität in die Suchanfrage eines parametrisierten Serviceaufrufs verlagert werden. D.h. aus mehreren Invoke-Aktivitäten entsteht eine parametrisierte Invoke-Aktivität, wobei als Evaluationsstrategie eine Suchanfrage verwendet wird. In Abbildung 1.4 wird dies an dem Beispiel eines Einkaufsprozesses aufgezeigt.

Man geht hier davon aus, dass die alternativen Zweige auf einer gewissen Abstraktionsstufe dieselbe Funktionalität bzw. Semantik haben. Im o.g. Beispiel ist es die Funktion „Produkt bestellen“. Die alternativen Produkte und der Aufruf entsprechender Zulieferer werden dann über die Suchanfrage modelliert.

Dieser Ansatz hat folgende Vorteile [KLB05]:

- Die Anzahl der Aktivitäten im Prozessmodell und damit auch seine Komplexität werden dadurch reduziert.

- Später hinzukommende Produktarten und Anbieter führen nicht dazu, dass das Prozessmodell durch Hinzufügen neuer Zweige verändert werden muss. Es muss höchstens die Suchanfrage angepasst werden, was deutlich einfacher ist.

- Das Prozessmodell hat einen höheren Wiederverwendungsgrad, da er abstrakter formuliert ist. Man spricht hier auch von generischen Prozessmodellen. Es werden keine konkreten Services hartkodiert. So könnte beispielsweise ein generisches Prozessmodell für einen Einkaufsprozess erstellt werden, wobei diejenigen Services parametrisiert werden, die die spezifischen Produkte betreffen, während andere wie „Bestellung entgegennehmen“ immer gleich sind und deshalb statisch definiert werden können.

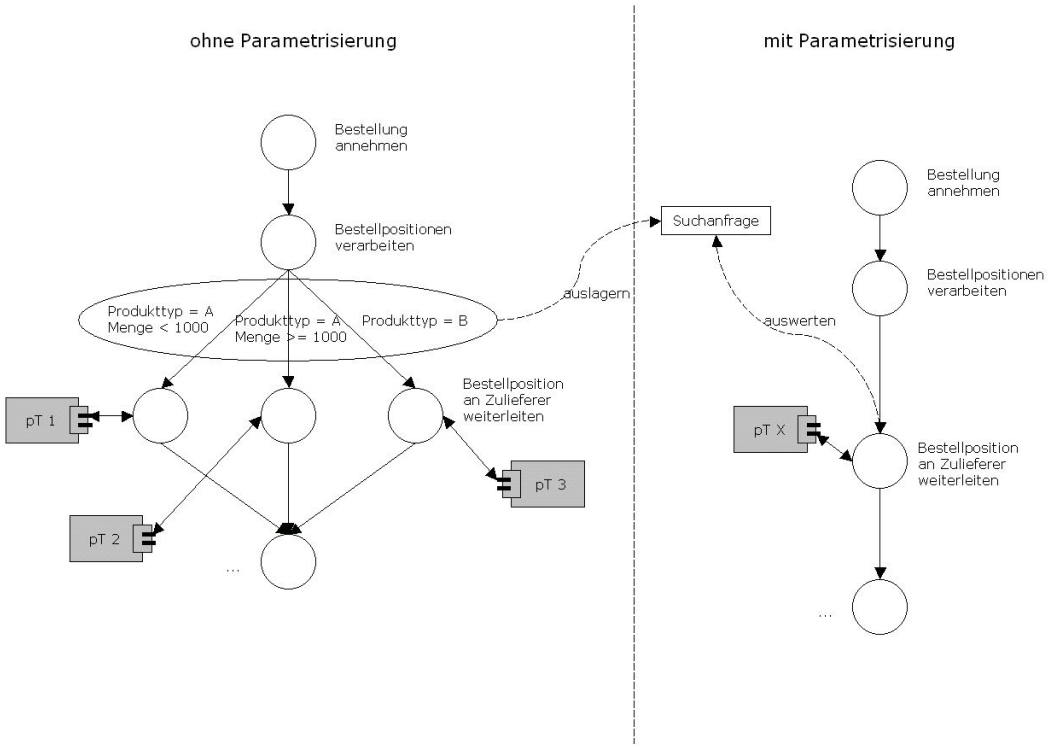

Abbildung 1.4: Kontrollfluss ohne/mit Parametrisierung

1.3 Aufgabenstellung

Im Abschnitt 1.2 wurde dargestellt wie durch dynamische Suche und Binden von Partner Services die Flexibilität und lose Kopplung in WS-Flows verbessert werden kann. Es wurden die beiden Fälle unterschieden, ob lediglich die WSDL-Ports oder auch die WSDL-PortTypes der Partner Services dynamisch bestimmt und gebunden werden sollen.

In beiden Fällen ist es wünschenswert deklarativ Such- und Auswahlkriterien spezifizieren zu können, so dass die Suche und Bindung zur Laufzeit automatisch erfolgen kann. Als mögliche Kriterien wurden nichtfunktionale Eigenschaften wie Dienstgüte und zusätzlich im Fall, dass PortTypes ausgetauscht werden, auch die semantische Beschreibung der Funktionalität identifiziert. Für diese Art von Beschreibungen existieren zwar mehrere Ansätze, die in Kapitel 2 vorgestellt werden, aber noch kein Standard.

Die Aufgabenstellung basiert auf [KLB05], in der das Konzept der Parametrisierung von WS-Flows beschrieben wird, wie auch im Abschnitt 1.2.2 erläutert. Darin wurde als eine mögliche Strategie zur Evaluierung der Parameter die Suchanfrage angedacht aber nicht konkret behandelt. In BPEL wird die Suchanfrage innerhalb des Evaluate-Elements spezifiziert, das eine Invoke-Aktivität erweitert. Im Folgenden wird nur noch auf BPEL Bezug genommen, obwohl die Konzepte grundsätzlich auch für andere WS-Flow-Sprachen gelten.

Die Aufgabe besteht darin, aufzuzeigen, wie eine Suchanfrage innerhalb des Evaluate-Elements zu spezifizieren ist, um eine automatische Suche nach Partner Services zu ermöglichen, die unabhängig von den im Prozessmodell vorkonfigurierten PortType- und

Operation-Namen ist und auch nichtfunktionale Eigenschaften berücksichtigt. Weiterhin ist zu untersuchen, wie die Erweiterung der BPEL-Engine zu konzipieren ist, um eine dynamische Suche anhand der Suchanfrage zu unterstützen. Die Konzepte sollen in einer entsprechenden Erweiterung der BPEL-Engine ActiveBPEL [Act] implementiert werden.

1.4 Aufbau der Arbeit

Der Rest der Arbeit ist wie folgt aufgebaut. Das nächste Kapitel beschreibt, welche Ansätze existieren, um Web Services semantisch zu beschreiben. Im Kapitel 3 wird dann ein Lösungsansatz vorgestellt, der die in der Aufgabenstellung gestellten Anforderungen erfüllt. Kapitel 4 beschreibt eine konkrete Umsetzung der gewählten Lösung. Im letzten Kapitel werden die Ergebnisse der Arbeit zusammengefasst und ein Ausblick auf mögliche Erweiterungen gegeben.

2 Verwandte Ansätze

Es wurde bereits in der Motivation im Abschnitt 1.2 festgestellt, dass semantische Beschreibungen notwendig sind, um die dynamische Suche nach Partner Services unabhängig von WSDL-PortTypes zu unterstützen. Dieses Kapitel stellt deshalb Ansätze vor, wie WS-Technologien und dabei insbesondere WSDL und BPEL um Semantik angereichert werden können. Dazu werden zunächst die Grundlagen des Semantic Web vorgestellt, da die darin enthaltenen Konzepte und Technologien die Grundlage für Semantic Web Services darstellen.

2.1 Semantic Web

2.1.1 Einführung

Der Begriff „Semantic Web“ wird in [BLHL01] geprägt und fasst die Bestrebungen zusammen das derzeitige World Wide Web um Semantik anzureichern.
Die Informationen, die das World Wide Web heute zur Verfügung stellt, werden auf HTML-Seiten dargestellt. Diese enthalten die Informationen als Textfragmente, die für den Menschen zum Lesen bestimmt sind, und Markup-Tags, die für die graphische Darstellung der Textfragmente in einem Web Browser verwendet werden. Das Wissen bzw. die Bedeutung der dargestellten Information ist dabei im Text nur implizit und in natürlicher Sprache kodiert. Dies ist für den Menschen zwar völlig ausreichend, nicht aber für die maschinelle Verarbeitung.
Die Idee ist nun, das im World Wide Web befindliche Wissen so zu formalisieren, dass es maschinell eindeutig interpretiert und verarbeitet werden kann. Das Semantic Web bedient sich hierzu der im Gebiet der Künstlichen Intelligenz erforschter Techniken zur Wissensrepräsentation und -verarbeitung. Das nötige Wissen wird dabei in Form von Ontologien (siehe Abschnitt 2.1.4) erfasst. Diese stellen ein formal spezifiziertes Vokabular zur Verfügung, welches von Softwareagenten gemeinsam verwendet wird. Die Erstellung der Ontologien geschieht konkret in RDFS (siehe Abschnitt 2.1.2) oder OWL (siehe Abschnitt 2.1.3). Das so formalisierte Wissen wird in Form von semantischem Markup in HTML-Seiten eingebettet. Dieser Markup kann dann von einem Softwareagenten interpretiert und verarbeitet werden. Zu beachten ist dabei, dass das Erstellen von Ontologien und die Annotation von HTML-Seiten bzw. -Elementen mit Konzepten aus diesen Ontologien, also die Bedeutungszuweisung, immer noch durch den Menschen geschieht, es ist also keine „künstliche Intelligenz“ dafür notwendig. Das Ziel dieser Erweiterungen ist, eine maschinengestützte Interaktion

zwischen Menschen zu ermöglichen, so dass Softwareagenten die Inhalte des World Wide Web interpretieren und im Auftrag des Menschen Aufgaben vollautomatisiert erledigen können. Ein Beispiel dafür wäre die Buchung einer Reise. Will man z.B. die günstigste Reise anhand bestimmter Kriterien im Internet buchen, so muss man zur Zeit noch von Hand die Angebote verschiedener Anbieter vergleichen. Die Suchmaschinen des World Wide Web helfen lediglich beim Auffinden dieser Anbieter. Softwareagenten, die diese Aufgabe automatisch erledigen sollen, versagen hier, da sie die auf den Web Seiten der Reiseanbieter enthaltene Information nicht interpretieren können.

2.1.2 RDF und RDFS

Das Resource Description Framework (RDF) ist ein vom W3C entwickeltes Datenmodell zur Beschreibung von Aussagen über jegliche Art von Internet-Ressourcen [W3C04]. RDF besitzt eine formale Semantik und bildet die Basis für die Definition von ausdrucksstarken Wissensrepräsentationssprachen wie OWL. Die Ressourcen werden durch Uniform Resource Identifiers (URIs) referenziert. Eine Aussage besteht aus einem Tripel aus Subjekt, Prädikat und Objekt. Alternativ kann man die Aussage auch als ein Tripel aus Ressource, Property(-Name) und Property-Wert betrachten. Objekte können auch beliebige Zeichenketten sein, müssen also nicht unbedingt durch URIs identifiziert werden. RDF erlaubt weiterhin auch Aussagen über Aussagen zu formulieren, was als Reifikation bezeichnet wird.
RDF-Dokumente können als gerichteter Graph dargestellt werden, wobei die Knoten Subjekten und Objekten, die Kanten den Prädikaten entsprechen. Für RDF-Aussagen gibt es verschiendene Serialiserungsmöglichkeiten, insbesondere auch eine XML-basierte Syntax.

RDF Vocabulary Description Language (RDFS) definiert auf der Basis von RDF ein Typsystem, mit dessen Hilfe man Klassen und Properties definieren kann. Das Typsystem ist ähnlich wie dem in objektorientierten Sprachen mit dem Unterschied, dass die Klassen- und Property-Vererbungshierarchie in RDFS getrennt ist. RDFS kann dazu verwendet werden, um domänenspezifische Vokabulare (Domänenmodelle) zu definieren. RDFS definiert z.B. die Klassen „rdfs:Class“ und „rdfs:Property“. Von diesen Klassen kann man mit Hilfe des vordefinierten Propertys „subClassOf“ bzw. „subPropertyOf“ ableiten und eigene Klassen definieren. Das Property „rdf:type“ wird verwendet, um auszudrücken, dass eine Ressource Instanz einer Klasse ist. Eine Property, die ja dem Prädikat im RDF-Tripel entspricht, wird über „rdfs:domain“ und „rdfs:range“ mit dem Subjekt bzw. Objekt verbunden. Mit RDFS lassen sich bereits einfache Inferenzmechanismen realisieren, wobei insbesondere die Vererbungsbeziehung interessant ist. So kann beispielsweise inferiert, d.h. hergeleitet, werden, dass ein Attribut in der Kindklasse (implizit) vorhanden sein muss, da es in der Vaterklasse definiert wurde.

2.1.3 OWL

Die Web Ontology Language (OWL) ist eine von W3C standardisierte Repräsentationssprache zur Definition von Ontologien [BvHH+04]. OWL ist ausdrucksstärker als RDFS. Während in RDFS neue Klassen nur durch die „subClassOf“-Beziehung erstellt werden können,

unterstützt OWL zusätzlich beispielsweise die Mengenoperatoren „intersection“, „union“ und „complement“, um neue Klassen zu erstellen. Ein RDFS-Reasoner kann aus den beiden Aussagen „A ist links von B“ und „B ist links von C“ nicht schließen, dass daraus „A ist links von C“ folgt. Dies kann in OWL mit Hilfe von transitiven Properties erreicht werden. Verwenden Applikationen verschiedene Ontologien und das ist im Web häufig der Fall, so müssen zur Datenintergation die Gemeinsamkeiten und Unterschide der Ontologien festgestellt werden. Diese können mit Hilfe von Operatoren wie „owl:equivalentClass“ oder „owl:disjointWith“ ausgedrückt werden. Diese stellen also eine Verbindung zwischen zwei „Domänenmodellen“, die beispielsweise in RDFS definiert sind, dar. OWL umfasst drei Teilsprachen mit ansteigender Ausdrucksmächtigkeit: Lite, DL und Full. So kann je nach Bedürfnissen die geeignetste Teilsprache ausgewählt werden. OWL Full ist die komplette Sprache, hat aber den Nachteil, dass in ihr definierte Ontologien unentscheidbar sein können. OWL DL bietet die maximale Ausdrucksstärke bei gleichzeitiger Entscheidbarkeit. Manche Sprachkonstrukte können dafür nur mit bestimmten Einschränkungen verwendet werden.

2.1.4 Ontologien

Ontologien sind ein weiterer Grundbaustein des Semantic Web [BLHL01]. Eine häufig zitierte Definititon ist die folgende [SBF98]:

> An ontology is a formal, explicit specification of a shared conceptualization.

Ontologien beschreiben demnach auf formale, also maschinenverwertbare, und explizite Weise Konzepte und deren Beziehungen in einer bestimmten Anwendungsdomäne. Sie stellen somit eine gemeinsame Wissensbasis und ein Vokabular zur Verfügung, das zum Austausch von Informationen verwendet werden kann.

Verwenden also zwei verschiedene Webseiten Konzepte einer gemeinsamen Ontologie um ihre Informationen zu beschreiben, können Softwareagenten diese Informationen extrahieren, zueinander in Beziehung bringen und verarbeiten, da die Semantik der Informationen eindeutig in der Ontologie festgelegt ist. Da die Ontologiesprache eine formale Semantik besitzt, kann in Ontologien beschriebenes Wissen auf Konsistenz untersucht werden. Dies ist besonders wichtig bei der Erweiterung von Ontologien. Der Softwareagent muss diese Art von Operationen nicht selbst durchführen, dies wird durch Ontologiesprachen-spezifische Inferenz-Maschinen durchgeführt.

Die Komplexität des mit Ontologien formalisierbaren Wissens hängt von der Ausdrucksstärke der verwendeten Repräsentationssprache ab, d.h. welche sprachlichen Mittel zur Definition von Konzepten und deren Beziehungen zur Verfügung stehen. Typischerweise spricht man erst dann von Ontologiesprachen, wenn sie mehr als „nur“ Vererbungshierarchien, auch bekannt als Taxonomien, definieren können. So wird beispielsweise RDFS (im Gegensatz zu OWL) nicht als Ontologiesprache bezeichnet, obwohl sie die obige Definition durchaus erfüllt. Ontologiesprachen basieren in der Regel auf Logikformalismen, OWL z.B. auf Beschreibungslogiken.

2.2 Semantic Web Services

2.2.1 Einführung

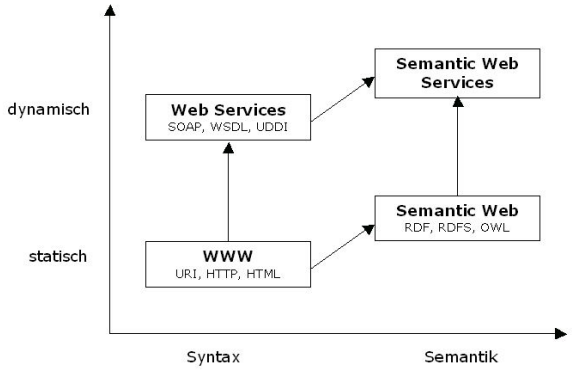

Abbildung 2.1: Semantic Web Services

In Abbildung 2.1 wird dargestellt, wie Semantic Web Services (SWS) konzeptionell einzuordnen sind. Während das World Wide Web statische Informationen für den Menschen anbietet, stellen Web Services eine dynamische Maschine-zu-Maschine-Kommunikation dar. So wie das Semantic Web das klassische WWW um maschinen-verständliche Beschreibungen erweitert, werden Web Services durch Verwendung der Semantic-Web-Technologien um Semantik angereichert.

Die bestehenden WS-Technologien haben bezüglich Semantik konzeptionelle Einschränkungen. WSDL definiert die Schnittstelle eines Web Service nur auf syntaktischer Ebene. Eine WSDL-Operation spezifiziert lediglich welche XML-Struktur bzw. -Datentypen sie als Eingabe und Ausgabe verwendet. Hierdurch wird jedoch nur eine formale Semantik auf unterster Ebene definiert, d.h. es ist z.B. feststellbar, dass eine bestimmte Zeichenfolge einen Integer-Wert darstellt, ob dieser Wert aber ein „Alter“ oder einen „Geldbetrag“ darstellt, ist nicht maschinen-verständlich spezifiziert. Der entsprechende Integer-Wert kann in der XML-Schema-Definition zwar einen in natürlicher Sprache formulierten Namen besitzen und damit für den Menschen verständlich sein, maschinenverwertbar ist er aber nicht. Da WSDL-Beschreibungen und darin verwendete Namen nicht standardisiert werden und jeder Service-Provider seine eigene WSDL-Beschreibung entwickelt und publiziert, können verschiedene Namensgebungen im WSDL-Dokument nicht vermieden werden.

Im Moment muss der Service-Requester also noch manuell vom Menschen im Bezug auf die zu verwendende WSDL-PortType-Beschreibung programmiert werden. Das Ziel von SOA ist aber eine automatisierte Dienstnutzung, indem der Service-Requester *automatisch* in der Service-Registry nach passenden Services sucht und diese dann automatisch aufruft.

Das Ziel von Semantic Web Services ist also eine Erhöhung der Automatisierung. Diese bezieht sich auf drei Aspekte [MBH+04]:

- *Automatisches Auffinden von Web Services:* Die Suche nach Web Services in einer Service-Registry soll anhand einer semantischen Beschreibung der benötigten Funktionalität

automatisch erfolgen können. Ohne SWS muss diese Tätigkeit manuell durch den Menschen durchgeführt werden.

- *Automatisches Aufrufen von Web Services:* Das automatische Aufrufen bezieht sich hier auf nicht-atomare Services, also Services, die aus mehr als einem Methodenaufruf bestehen. Beispielsweise wird man, um ein Buch zu kaufen, zunächst danach suchen, es dann auswählen und in den Einkaufskorb verschieben und schließlich bezahlen. Dazu müssen mehrere WSDL-Operationen aufgerufen werden. Ohne semantische Beschreibungen muss die „Choreographie“ im Requester hartkodiert werden.

- *Automatische Komposition von Web Services:* Kann ein einzelner Web Service nicht die Anforderung seitens des Service-Requesters erfüllen, so wird versucht automatisch eine Komposition von Services zu finden und auszuführen, die zusammengenommen diese Anforderung erfüllen. Die Komposition von Web Services in BPEL wird dagegen auch wieder manuell durch den Menschen spezifiziert.

Es existieren im Moment mehrere konkurrierende SWS-Ansätze. Sie werden in den folgenden Abschnitten kurz vorgestellt. Man kann sie daran charakterisieren, ob sie einen Top-Down- oder Bottom-Up-Ansatz bezüglich bestehender WS-Technologien, insbesondere WSDL, bestreiten. Top-Down-Ansätze modellieren den Web Service und seine Semantik zunächst unabhängig von den bestehenden WS-Technologien, in einer Ontologiesprache, mit dem Ziel dadurch eine optimale Beschreibungssprache für Web Services zu finden, um die obengenannten Ziele zu erreichen. Durch ein sogenanntes Grounding wird dann eine Beziehung zu WSDL hergestellt, indem spezifiziert wird, wie die Elemente der semantischen Servicebeschreibung auf WSDL abzubilden sind. Zu den Top-Down-Ansätzen zählen OWL-S und WSMO.

Bei einem Bottom-Up-Ansatz wird dagegen kein neues konzeptionelles Modell eines Web Service erstellt, sondern versucht die bestehenden Technologien, insbesondere WSDL und BPEL, um Semantik anzureichern. Dies ist auch der Weg des Semantic Web, das ja das bestehende World Wide Web erweitert. Analog zu Semantic Web und unter Verwendung seiner Technologien werden WSDL-Beschreibungen durch Annotation von Ontologiekonzepten erweitert. Bottom-Up-Ansätze sind WSDL-S und METEOR-S.

Drei der vorgestellten Ansätze, nämlich OWL-S, WSMO und WSDL-S wurden beim W3C als Submissions eingereicht, die Festlegung eines Standards ist aber in der nächsten Zeit nicht zu erwarten.

2.2.2 WSDL-S

WSDL-S [AFM+05] ist ein gemeinsamer Vorschlag des METEOR-S-Projektteams von der University of Georgia und IBM Research wie WSDL durch Ausnutzen seiner Erweiterungsmöglichkeiten um semantische Beschreibungen erweitert werden kann. WSDL-S ist seit November 2005 eine W3C-Submission.

In WSDL-S wird nicht versucht ein neues konzeptionelles Modell von Web Services zu erstellen wie etwa in OWL-S und WSMO, sondern es werden in einem Bottom-Up-Ansatz die

Elemente des WSDL-PortTypes durch Annotation von Konzepten eines externen Domänenmodells um Semantik angereichert. Aspekte wie Komposition und Choreographie werden in WSDL-S nicht berücksichtigt.

Die Autoren stellen zunächst Prinzipien auf, die nicht nur in WSDL-S berücksichtigt werden, sondern aus ihrer Sicht generell für das Einbringen von Semantik in Web Services gelten sollen:

- Existierende Standards sollen erweitert werden, statt komplett neue Frameworks zu konzipieren. WSDL ist in der Entwicklergemeinde etabliert und wird bereits erfolgreich verwendet. Es sind viele Tools vorhanden. Eine Erweiterung muss deshalb abwärts kompatibel sein, um bestehende Services nicht zu beeinträchtigen bzw. um leichter oder überhaupt akzeptiert zu werden.
- Die Annotation sollte unabhängig von der Ontologie-Sprache sein. Es gibt mehrere Alternativen wie OWL, WSML, UML und diesbezüglich noch keinen Standard. Die Sprachen sind unterschiedlich, was ihre Ausdrucksfähigkeit betrifft und können sich je nach Anwendnung besser oder schlechter für ein bestimmtes Problem eignen. Bereits existierende Domänenmodelle können dadurch auch leichter eingebunden werden.
- Da man sich nicht auf eine bestimmte Ontologie-Sprache festlegt, sollte es auch möglich sein mehrere gleichzeitig zu verwenden. Dadurch können Service-Provider durch mehrfache Annotation möglicherweise eine größere Anzahl von Requestern unterstützen, die jeweils nur eine bestimmte Sprache verstehen.
- Es sollte explizit die Annotation von Web Services unterstützt werden, die XML-Schema zur Definition der verwendeten Datentypen unterstützen. Obwohl WSDL auch andere Typsysteme unterstützt, ist XML-Schema in der Praxis am meisten verbreitet. Es muss möglich sein, Mappings zu definieren, die die verwendeten komplexen XML-Schema-Datentypen auf die zugehörigen Konzepte der verwendeten Ontologie-Sprache abbilden. Als Mapping-Sprachen kommen z.B. XSLT, XQuery oder auch Ontologiesprachen wie OWL in Frage.

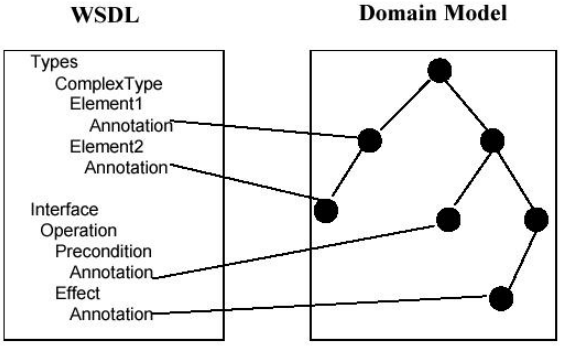

Abbildung 2.2: Annotation in WSDL-S

In WSDL-S wird der WSDL-PortType durch Semantik annotiert, der Annotationsmechanismus könnte aber auch problemlos auf die Binding-Elemente ausgeweitet werden. Es werden Erweiterungsmöglichkeiten von WSDL ausgenutzt um durch URI-Referenzen auf Konzepte

eines Domänenmodells zu verweisen, das in einer beliebigen Sprache modelliert sein kann, wie in Abbildung 2.2 [AFM+05] dargestellt. WSDL-S bezieht sich auf WSDL 2.0, die Mechanismen können aber auch mit leichten Abwandlungen auf WSDL 1.1 angewendet werden.

```
<interface name="PurchaseOrder">
  <wssem:category name="Electronics"
      taxonomyURI="http://www.naics.com/"
      taxonomyCode="443112" />
    <operation name="processPurchaseOrder"
      wssem:modelReference="Rosetta:RequestPurchaseOrder" >
    <input element="..."/>
    <output element="..."/>
    <wssem:precondition name="ExistingAcctPrecond"
      wssem:modelReference="POOntology#AccountExists"/>
    <wssem:effect name="ItemReservedEffect"
      wssem:modelReference="POOntology#ItemReserved"/>
  </operation>
</interface>
```

Listing 2.1: Annotation des WSDL-Interfaces in WSDL-S

Im Listing 2.1 wird aufgezeigt, wie ein Interface (entspricht dem PortType in WSDL 1.1) und eine Operation erweitert werden können [AFM+05]. Ein WSDL-Interface wird erweitert durch Category-Elemente, die auf Konzepte in einer Taxonomie wie z.B. NAICS oder UNSPSC verweisen. Diese Kategorisierung hilft die Funktionalität des PortTypes grob einzuordnen, enthält aber keine semantische Information. Die Funktionalität einer WSDL-Operation wird semantisch durch drei Typen von Elementen beschrieben. Durch das Attribut „wssem:modelReference“ verweist die Operation auf ein Konzept, das die Funktionalität der Operation beschreibt, im Folgenden auch als funktionale Klasse bezeichnet. Im obigen Beispiel ist dieses Konzept Teil der RosettaNet-Ontologie. Zusätzlich zur funktionalen Klasse werden über entsprechende Elemente Vorbedingungen und Effekte spezifiziert. Auch auf sie wird über eine URI zugegriffen. Sie werden typischerweise mittels logischer Ausdrücke definiert. Vorbedingungen, Effekte und die funktionale Klasse definieren die funktionale Semantik einer WSDL-Operation.

```
<complexType name="POAddress"
    wssem:schemaMapping="http://.../POAddress.xsl"
    wssem:modelReference="POOntology#Address" >
  <all>
    <element name="recipientInstName" type="string" />
    <element name="streetAddr1" type="string" />
    <element name="streetAdd2" type="string" />
    <element name="city" type="string" />
    <element name="state" type="string" />
    <element name="zipCode" type="string" />
    <element name="country" type="string" />
  </all>
</complexType>
```

Listing 2.2: Annotation der Datentypen in WSDL-S

Durch die Annotation der Eingabe- und Ausgabeparameter der WSDL-Operationen wird die Datensemantik spezifiziert. Das Listing 2.2 zeigt die Annotation eines XML-Schema-

ComplexTypes durch eine OWL-Klasse. Durch „wssem:modelReference“ wird wie gewohnt auf das zugehörige Konzept verwiesen. In diesem Fall wird zusätzlich ein Schema-Mapping als XSL-Transformation angegeben. Dieses definiert wie die XML-Elemente des ComplexTypes auf die OWL-Properties der OWL-Klasse abgebildet werden sollen.

Die Annotation von XML-Schema-Elementen mit Ontologie-Konzepten führt zu einem Schema-Mapping-Problem, das neben syntaktischer und struktureller Unterschiede auch die verschiedene Ausdrucksfähigkeit der verwendeten Sprachen beinhaltet, so ist z.B. OWL weit ausdrucksfähiger als XML-Schema. Man kann sich in vielen Fällen durch ein manuelles Mapping in einer Sprache wie XSLT behelfen, in manchen Fällen kann das auch nicht möglich sein. Eine Übersicht und Referenzen auf die Forschungsarbeiten im Bereich des Schema-Mapping werden in der WSDL-S-Spezifikation [AFM+05] gegeben.

2.2.3 OWL-S

Die Web Ontology Language for Services (OWL-S) ist eine auf OWL basierende Ontologie zur Beschreibung von Web Services [MBH+04, MPM+04]. OWL-S wurde unter dem DARPA Agent-Markup-Language-Programm (DAML) als Nachfolger von DAML-S [ABH+01] entwickelt. Die aktuellste Version 1.1 ist seit November 2004 eine W3C Submission.

Servicebeschreibung

Das Ziel von OWL-S ist es, im Wesentlichen drei Operationen für die Interaktion mit Web Services zu unterstützen: das Auffinden des Web Service anhand seiner Fähigkeiten, die Interaktion mit dem Web Service anhand eines Protokolls, das definiert, in welcher Reihenfolge welche (abstrakten) Nachrichten mit dem Web Service ausgetauscht werden müssen und schließlich wie diese Nachrichten konkret serialisiert werden bzw. wie der Web Service konkret aufzurufen ist.

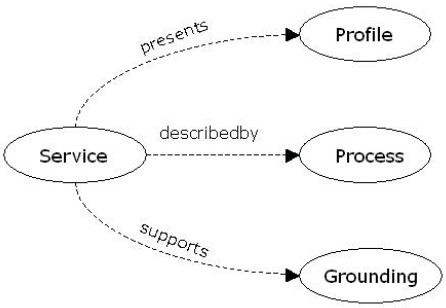

Abbildung 2.3: Basisklassen der OWL-S-Ontologie

Ein Web Service wird entsprechend modelliert als OWL-Klasse `Service` bestehend aus den Klassen `Profile`, `Process` und `Grounding`. Im Folgenden werden die drei Service-Bestandteile näher erläutert.

Profile

Ein `Profile` definiert, *was* der Web Service tut und dient zum Publizieren und Auffinden von Web Services. Dazu enthält er folgende Elemente:

- textuelle, menschenlesbare Beschreibung des Services sowie der Kontaktinformation des Service-Providers. Die dazugehörigen Properties sind `serviceName`, `textDescription` und `contactInformation`.
- die funktionale Beschreibung des Services in Form von Eingaben, Ausgaben, Vorbedingungen und Effekten (Inputs/Outputs/Preconditions/Effects). Die Eingaben und Ausgaben bezeichnen dabei die Eingabe- und Ausgabeparameter. Sie verweisen auf Konzepte von domänenspezifischen Ontologien und definieren dadurch die Datensemantik. Die Vorbedingungen und Effekte dagegen modellieren die funktionale Semantik als Zustandsveränderung, die der Service verrichtet. Die Vorbedingung muss gelten, damit der Service überhaupt ausgeführt werden kann, während ein Effekt den Zustand nach der Ausführung beschreibt. Vorbedingungen und Effekte werden typischerweise in einer Rule-Language wie SWRL [HPS+04] spezifiziert. Bei einer Kauftransaktion wäre z.B. eine der Eingaben die Kreditkartennummer, die Vorbedingung würde aussagen, dass die Kreditkarte gültig sein muss, die Ausgabe wäre eine Kaufbestätigung und ein Effekt, dass der entsprechende Geldbetrag abgebucht wurde. Die Kreditkartennummer und die Kaufbestätigung sind dabei Konzepte einer domänenspezifischen Ontologie.
- Kategorisierungen über das Property `serviceCategory`. Dadurch kann auf beliebige Kategorien verwiesen werden, z.B. UNSPSC und NAICS, die in UDDI verwendet werden.
- `serviceParameter` für beliebige weitere Informationen, die als Name-Wert-Paare definiert sind, wobei der Wert eine URI darstellt und typischerweise auf ein Konzept einer Ontologie verweist.

Process

Ein `Process` definiert wie der Service-Requester mit dem Service interagieren soll, um die im `Profile` beschriebenen Effekte zu erzielen. Ein `Process` ist nicht ausführbar, sondern ist lediglich eine Spezifikation der möglichen Interaktionen des Clients mit dem Service, d.h. er spezifiziert welche WSDL-Operationen (ein WSDL-Grounding vorausgesetzt) eines Services mit welchen Eingaben zu welchen Bedingungen in welcher Abfolge aufgerufen werden müssen, um die im `Profile` beschriebenen Effekte zu erzeugen.

Atomare Prozesse (`AtomicProcesses`) sind direkt ausführbar und enthalten keine weiteren Subprozesse mehr. Sie werden aus der Sicht des Aufrufers in einem einzigen Schritt ausgeführt und entprechen dem Aufruf einer WSDL-Operation. Für jeden atomaren Prozess muss ein `Grounding` existieren, mit dessen enthaltener Information der Aufrufer die nötigen Nachrichten zur Kommunikation mit dem Web Service konstruieren kann.

Mit Hilfe zusammengesetzter Prozesse (`CompositeProcesses`) können komplexe Prozesse durch Komposition aus atomaren und/oder weiteren zusammengesetzten Prozessen definiert werden. Sie modellieren dadurch zustandsbasierte Services. Die Reihenfolge der Ausführung wird über Kontrollkonstrukte festgelegt, dazu gehören z.B. Sequence, Split, Choice, If-Then-Else, Repeat-While usw. Die Eingaben, Ausgaben, Vorbedingungen und Effekte, die

im `Profile` den Service beschreiben, werden auch bei der Spezifikation der Prozesse verwendet.

Grounding

Ein `Grounding` legt fest, wie auf den Web Service konkret zugegriffen wird. Dazu gehört die Festlegung, wie die abstrakten Eingaben und Ausgaben eines atomaren Prozesses in konkrete Nachrichten verpackt werden, wie sie serialisiert und transportiert werden und unter welcher Adresse der Service zu finden ist.
Diese konkrete Beschreibung des Zugriffs wird bereits durch WSDL geleistet. Deshalb definiert OWL-S wie ein `Grounding` auf WSDL realisiert werden kann. Beide Technologien ergänzen einander, indem einerseits die höhere Beschreibungsfähigkeit von OWL gegenüber XML-Schema verwendet wird, auf der anderen Seite bereits vorhandene und weit verbreitete WSDL/SOAP-Infrastruktur zum Einsatz kommt. Das Konzept des Grounding in OWL-S entspricht im Wesentlichen dem Konzept des Binding in WSDL. Das WSDL-Grounding stützt sich auf drei zwischen OWL-S und WSDL korrespondierende Beschreibungselemente [MBL+]:

- Ein atomarer Prozess in OWL-S wird auf eine Operation in WSDL abgebildet.
- Die Menge der Eingaben bzw. Ausgaben eines atomaren Prozesses in OWL-S wird auf eine Input- bzw. Output-Message einer WSDL-Operation abgebildet.
- Die Typen (d.h. OWL-Klassen) der Eingaben und Ausgaben korrespondieren mit dem erweiterbaren Konzept der abstrakten Typen in WSDL. Wenn die WSDL-Typen in XML-Schema definiert sind, so kann mit Hilfe von XSLT ein Mapping von der XML-Serialisierung der OWL-Konzepte auf die entprechenden XML-Schema-Datentypen angegeben werden.

Publikation und Auffinden von Services

In OWL-S beschreibt das `Profile` die Funktionalität des Services und bietet die Grundlage für das Publizieren und Auffinden von Web Services. Die OWL-S-Spezifikation sagt aber nicht aus, wie die Suche umgesetzt werden soll.

In [SPS04] wird die Architektur eines OWL-S/UDDI-Matchmakers beschrieben. Dieser erweitert die UDDI-Registry um die semantische Suche nach OWL-S-Services.
Ein `Profile` wird dabei sowohl vom Service-Provider als auch vom Service-Requester verwendet. Service-Provider verwendet die Informationen aus dem `Profile`, um Web Services zu publizieren. Man spricht hier von Advertisements. Auf der anderen Seite definiert ein Service-Requester mit dem `Profile` seine Suchbeschreibung. Die Suchbeschreibung wird dann mit den Advertisements verglichen um passende Services zu finden. Dieser Vorgang wird als Matchmaking bezeichnet.

Das grundsätzliche Problem besteht also darin Advertisements der Service-Provider mit der Suchanfrage des Service-Requesters zu vergleichen und passende Services herauszufinden. Dabei kann man im Falle von semantischen Beschreibungen, die im Fall von OWL-S auf

Beschreibungslogik basieren, auf logische Inferenz (logisches Schließen) zurückgreifen. Leider unterstützt UDDI keine logische Inferenz, sondern nur Schlüsselwortsuche und muss deshalb erweitert werden.

UDDI unterstützt die Suche nach Services basierend auf Namen, Bindings oder T-Models. Es ist also z.B. möglich nach Services zu suchen, die eine bestimmte WSDL-Beschreibung besitzen oder als „Autoverkäufer“ kategorisiert sind. Kategorien werden über T-Models abgebildet. Dabei wird aber keine logische Inferenz unterstützt, sondern lediglich die Schlüsselwortsuche. Verwendet z.B. ein Autoverkäufer in seinem Advertisement „Neuwagenverkäufer“ als Kategorie (z.B. innerhalb von NAICS), so wird die Suche nach „Autoverkäufer“ diesen Eintrag nicht zurückliefern, obwohl „Neuwagenverkäufer“ Kindklasse von „Autoverkäufer“ ist und in diesem Fall der Service durchaus in Frage käme. Eine weitere Einschränkung von UDDI ist, dass nur nach Kategorien gesucht werden kann, nicht aber nach Eingaben und Ausgaben der Services. Diese beiden Probleme können durch semantisches Matchmaking behoben werden.

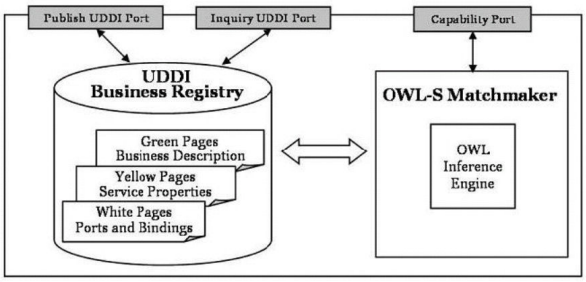

Abbildung 2.4: OWL-S/UDDI-Matchmaker

Abbildung 2.4 [SPS04] zeigt die Architektur des OWL-S/UDDI-Matchmakers. Die Schnittstellen der UDDI-Registry (Publish und Inquiry Port) werden nicht geändert, wohl aber die Implementierung des Publish Ports. Die Publikation von Services geschieht über den Publish Port. Enthält das Advertisement ein OWL-S-Profile, so muss die Implementierung des Publish Ports dies erkennen und das Advertisement an den Matchmaker weiterleiten. Zusätzlich wird das OWL-S-Profile auf UDDI-Strukturen abgebildet.

Die Abbildung des `Profiles` auf UDDI-Strukturen ist in Abbildung 2.5 [SPS04] dargestellt. Dabei werden Elemente des `Profiles`, die auch in UDDI vorhanden sind, z.B. Kontaktinformation und Servicename, direkt abgebildet. Existieren keine entsprechenden UDDI-Elemente, werden für diese T-Models angelegt, ähnlich wie im Falle eines WSDL-to-UDDI Mappings [Not04]. Es wird also beispielsweise jeweils ein T-Model für die Eingaben, Ausgaben, Vorbedingungen und Effekte angelegt. Ein Service kann dann auf diese T-Models in seiner CategoryBag über KeyedReferences verweisen, wobei als KeyValue das entsprechende Ontologie-Konzept angegeben wird.

Durch die Abbildung des `Profiles` auf UDDI-Strukturen ist die Service-Beschreibung komplett in UDDI enthalten und kann über den Inquiry-Port gesucht werden. Dabei kann aber immer nur Schlüsselwortsuche durchgeführt werden. Für semantische Suche ist der Capability Port zuständig, der auf den Matchmaker zugreift. Um die Suche zu ermöglichen, werden die Advertisements bei der Publizierung in einer Advertisement-Datenbank innerhalb des Matchmakers gespeichert und für schnellere Suche indiziert.

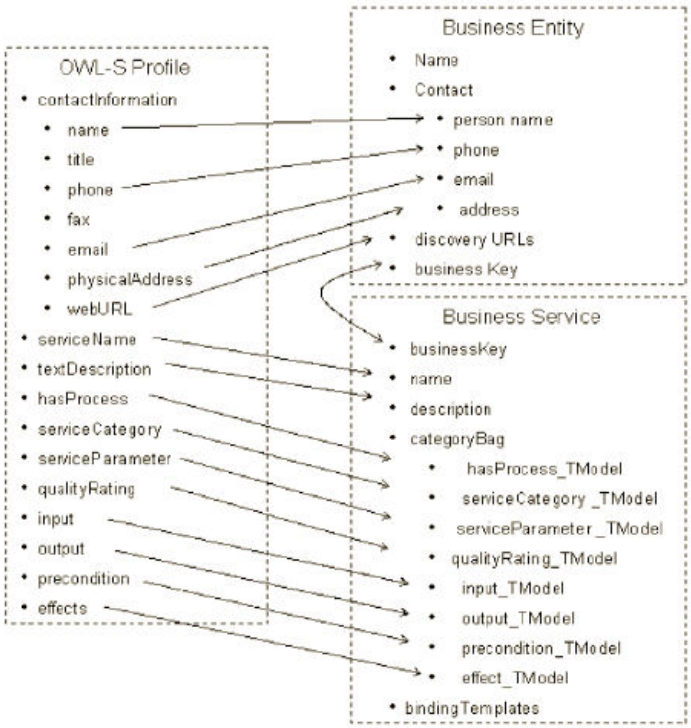

Abbildung 2.5: OWL-S/UDDI Mapping

Der Matchmaking-Algorithmus basiert auf logischer Inferenz über Vererbungsbeziehungen. Wird eine Suchanfrage über den Capability Port in Form eines OWL-S-Profiles gestellt, so werden zunächst alle Ausgaben der Anfrage mit den Ausgaben der Advertisements verglichen, anschließend an den übriggebliebenen Advertisements auch die Eingaben. Vorbedingungen und Effekte werden nicht berücksichtigt. Die Eingaben und Ausgaben werden durch Konzepte einer domänenspezifischen OWL-Ontologie repräsentiert. Beim Vergleichen der Konzepte (OWL-Klassen) werden nicht nur genaue Matches berücksichtigt, d.h. Anfrage-Konzept = Advertisement-Konzept, sondern es wird die Vererbungsbeziehung zwischen Konzepten einer Ontologie mit berücksichtigt, um auch „ähnliche“ Treffer berücksichtigen zu können. Der Ähnlichkeitsgrad basiert dabei auf der Vererbungshierarchie und ist in drei Stufen unterteilt. Im Folgenden wird der Vergleich zwischen Ausgabe-Konzepten dargestellt, für Eingabe-Konzepte wird die Reihenfolge umgekehrt, der Ähnlichkeitsgrad ist absteigend:

- *exact:* Der Match ist exakt, wenn Anfrage-Konzept = Advertisement-Konzept, wenn die beiden Konzepte also äquivalant sind. Zusätzlich wird auch dann von exaktem Match ausgegangen, wenn Anfrage-Konzept unmittelbare Kindklasse ist von Advertisement-Konzept. Dabei geht man davon aus, dass wenn ein Provider angibt, ein Advertisement-Konzept zu unterstützen, er auch die unmittelbaren Kindklassen unterstützen muss.

- *plug in:* Anfrage-Konzept ist Kindklasse aber nicht unmittelbare Kindklasse des Advertisement-Konzepts.

- *subsumes:* Advertisement-Konzept ist Kindklasse von Anfrage-Konzept, d.h. Advertisement-Konzept unterstützt möglicherweise nur einen Teil von dem was der Requester erwartet.
- *fail:* Anfrage-Konzept und Advertisement-Konzept stehen in keiner Vererbungsbeziehung.

Erweiterung von BPEL

In [MM03] wird aufgezeigt, wie DAML-S [ABH+01] verwendet werden kann, um in BPEL dynamische Suche nach Partner Services zu ermöglichen. Für diese Art der Erweiterung sind die Unterschiede zwischen DAML-S und OWL-S unerheblich, es könnte also auch OWL-S verwendet werden. Das Ziel ist es, für einzelne Serviceaufrufe durch die Invoke-Aktivität einen Service zur Laufzeit zu bestimmen und aufzurufen.

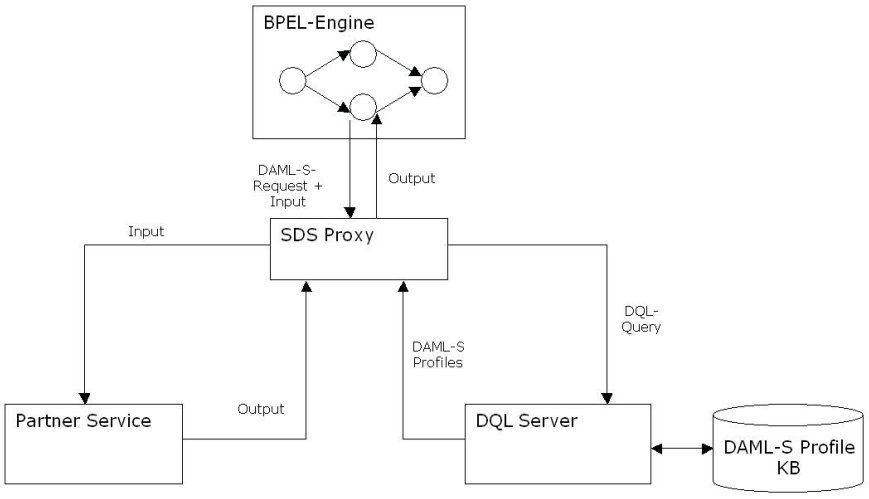

Abbildung 2.6: Semantic Discovery Service

Soll ein Partner Service zur Laufzeit dynamisch anhand einer DAML-S-Servicebeschreibung ausgewählt und aufgerufen werden, wird für diesen Service im BPEL-Prozessmodell ein Partner Link definiert, der auf den Semantic Discovery Service (SDS) verweist. Der SDS fungiert aus Sicht des BPEL-Prozesses als Proxy für den eigentlichen Partner Service, der ausgewählt und aufgerufen werden soll. Der SDS besitzt eine WSDL-Schnittstelle, die als Eingabe eine DAML-S-Beschreibung und zusätzlich den eigentlichen Input-Parameter der aufzurufenden WSDL-Operation bekommt. Die DAML-S-Beschreibung enthält die Eingaben und Ausgaben und zusätzlich ein „functionalClass“-Property, das auf eine Ontologie von Service-Funktionen verweist und damit die funktionale Semantik beschreibt, siehe dazu auch Abschnitt 2.2.2.

Zur Laufzeit ruft der BPEL-Prozess den SDS auf. Der SDS generiert anhand der DAML-S-Beschreibung eine DAML-Query-Langauge-Anfrage und schickt diese an den DQL-Server. Der DQL-Server sucht in einer Wissensbasis, die DAML-S-Profiles enthält, nach passenden

Services und liefert diese an den SDS zurück. Der SDS wählt einen davon aus und ruft den entsprechenden Partner Service auf. Das Ergebnis des Aufrufs wird an den BPEL-Prozess zurückgegeben.

Der Algorithmus zum Auffinden der benötigten Services wird durch *Service-Verkettung* verfeinert. Dabei wird zunächst nach passenden Services gesucht, die nur die korrekten Ausgaben der Anfrage enthalten. Enthält keines der Suchergebnisse die passenden Eingaben, so wird eine weitere Anfrage gestellt nach einem Service, der als Ausgabe die Eingaben des vorher gefundenen Service enthält, usw. bis in dieser Kette schließlich ein Service auftaucht, dessen Eingaben sich mit den Eingaben der Anfrage decken. Es ist also durchaus möglich, dass obwohl keines der Services die Anfrage genau erfüllen kann, eine Kette von Services, die wenn nacheinander ausgeführt, diese erfüllen können. Dabei ist zu beachten, dass dabei nur die Eingaben und Ausgaben aber keine Vorbedingungen und Effekte berücksichtigt werden. Es wird also davon ausgegangen, dass alle Services in der Kette lediglich Datentransformationen durchführen.

2.2.4 METEOR-S

Im Projekt METEOR-S des LSDIS Lab an der University of Georgia wird ein Framework für Semantic Web Processes entwickelt. Semantic Web Processes sind im Kontext von METEOR-S WS-Flows, die SWS-Technologien verwenden. Ein Übersicht über METEOR-S gibt [CSV04]. Das Framework verfolgt einen Bottom-Up-Ansatz mit dem Ziel, wenn möglich auf bereits vorhandenen Standards durch Ausnutzen ihrer Erweiterungsmöglichkeiten aufzusetzen. Insbesondere werden WSDL zur Beschreibung der Services, BPEL als Grundlage für die Komposition von Prozessen und WS-Policy für Dienstgüte-Beschreibungen verwendet. WSDL-S ist aus METEOR-S entstanden.

METEOR-S geht von folgenden Phasen bei der Erstellung eines Prozesses aus: Entwicklung der im Prozess verwendeten Services, Annotation der Services durch Semantik, Publikation der Advertisements in einer Registry, Komposition und Ausführung des Prozesses.

Erstellung und Annotation der Services

Die Beschreibung von Web Services in METEOR-S basiert auf der Annotation von WSDL. Dazu werden drei Möglichkeiten unterstützt: Annotiertes WSDL 1.1, WSDL-S (basierend auf WSDL 2.0) und OWL-S-Service-Beschreibung mit einem WSDL-Grounding. METEOR-S unterstützt die Generierung dieser drei Beschreibungen aus dem Java-Quellcode heraus [RMVS04]. Dazu wird der Quellcode mit entsprechenden Metadaten annotiert. Ein *Semantic Description Generator* generiert daraus dann die gewünschten Service-Beschreibungen. Alternativ kann auch ein WSDL-S-Editor verwendet werden, der bereits existierende WSDL-Beschreibungen erweitert. Während bei der Annotation von Quellcode oder bereits existierendem WSDL der Entwickler die passenden Ontologien und deren Konzepte finden und zuordnen muss, wird in [POSV04] beschrieben wie dies semi-automatisch geschehen kann. Dabei versucht ein Algorithmus passende Konzepte für die entsprechenden WSDL-Elemente zu finden. Dies wird motiviert dadurch, dass es immmer mehr und immer größere Ontologien gibt und unter Umständen sogar mehrere gleichzeitig verwendet werden müssen, so dass manuelle Annotation zu umständlich und zeitaufwendig ist. Zusätzlich

zu den Eingaben, Ausgaben, Vorbedingungen und Effekten werden zu der Web-Service-Beschreibung auch Dienstgüte-Eigenschaften als WS-Policy-Assertions hinzugefügt.

Komposition und Ausführung

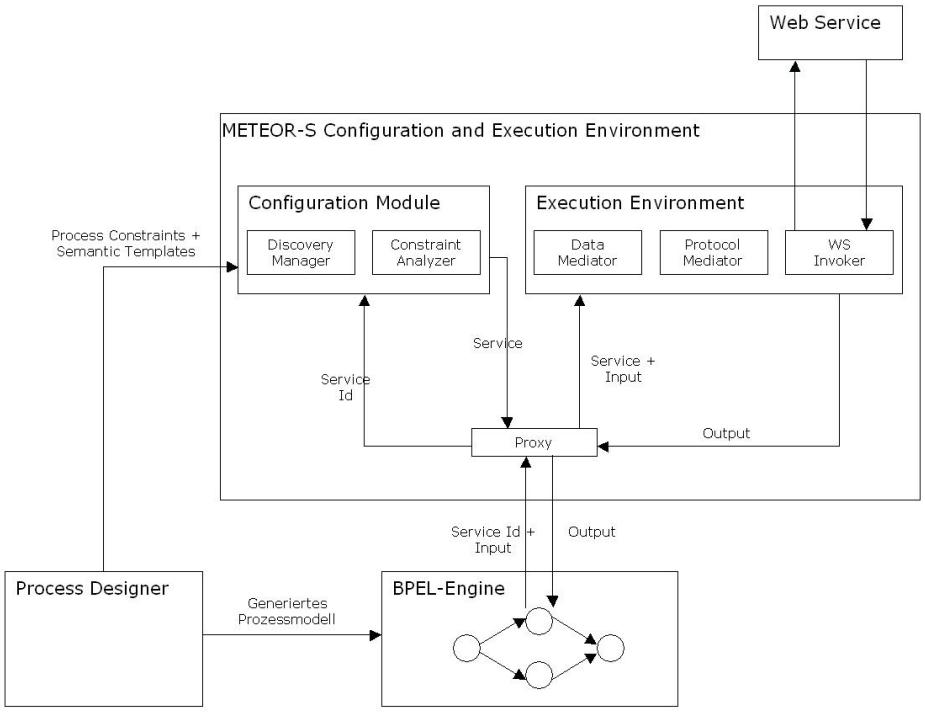

Abbildung 2.7: Dynamische Komposition in METEOR-S

Die Komposition von Prozessen in METEOR-S basiert auf BPEL [VAG+04, VGS+05]. Mit Hilfe des *Process Designers* wird der BPEL-Prozess modelliert. Zusätzlich zu den normalen BPEL-Sprachelementen, kann der Benutzer darin spezifizieren:

- *Virtuelle Partner*. Damit werden Partner Services bezeichnet, die erst zur Laufzeit in den Prozess eingebunden werden sollen. Sie werden durch *Service Templates* repräsentiert. Diese enthalten annotierte WSDL- und WS-Policy-Beschreibungen. Das *Service Template* berücksichtigt alle WSDL-Operationen eines virtuellen Partners, die im Prozessmodell verwendet werden.

- *Quantitative Prozess-Einschränkungen* (Process Constraints), wie z.B. maximale Kosten oder maximale Bearbeitungszeit des Prozesses. Jeder Partner Service bekommt entsprechende numerische Attribute zugewiesen, also z.B. Preis und Antwortzeit. Bei der dynamischen Auswahl von Partner Services wird dann überprüft, ob durch ihre Verwendung zusammen mit den restlichen Services die Prozesseinschränkungen noch erfüllt sind.

- *Qualitative Prozess-Einschränkungen*, die mit Hilfe von SWRL [HPS+04] spezifiziert werden. Ein Beispiel wäre: „Virtueller Partner 1 muss ein bevorzugter Partner sein“. Welche Partner bevorzugt sind, wird in diesem Fall in einer OWL-Ontologie definiert. Es

können auch Einschränkungen bezüglich mehrerer virtueller Partner gemacht werden, z.B. „Produkte der Partner 1 und 2 müssen kompatibel sein“. Diese Prozess-Einschränkungen können auf Scope- oder Prozess-Ebene definiert werden.

Der *Process Designer* generiert ein BPEL-Prozessmodell, wobei die virtuellen Partner im zugehörigen Partner Link einen Proxy-Service definieren. Die virtuellen Partner werden aus dem BPEL-Prozess heraus immer über einen Proxy-Service aufgerufen. Ob für jeden virtuellen Partner ein eigener Proxy generiert wird, oder ein generischer Proxy verwendet wird, wird aus [VGS+05] nicht klar. In Abbildung 2.7 ist die generische Variante abgebildet. Damit der generische Proxy weiß, um welchen Service es sich handelt, wird eine Service-Id übertragen.

Die *Semantic Templates* und die *Prozess-Einschränkungen* werden im *Configuration Module* publiziert. Wird der Proxy zur Laufzeit aufgerufen, so ruft er zunächst das *Configuration Module* auf, um den aufzurufenden Service zu erhalten. Handelt es sich um den ersten Aufruf eines Proxys in der Prozessinstanz werden zunächst alle virtuellen Partner unter Einbeziehung der Prozesseinschränkungen bestimmt. Bei folgenden Aufrufen von Proxys kann eine Rekonfiguration geschehen, wenn einer der virtuellen Partner nicht erreichbar ist.

Zur Bestimmung der passenden virtuellen Partner sucht der *Discovery Manager* zunächst anhand der *Semantic Templates* für jeden Partner einzeln passende Services. Die Suche geschieht anhand der Datensemanik, der funktionalen Semantik und nichtfunktionaler Eigenschaften. Der *Discovery Manager* liefert für jeden virtuellen Partner eine Liste von potentiell passenden Services zurück. Die Listen aller virtuellen Partner werden an den *Constraint Analyzer* weitergeleitet, der die bestmögliche Kombination anhand der Prozesseinschränkungen bestimmt.

Das *Configuration Module* liefert den aufzurufenden Service an den Proxy zurück. Der Proxy verwendet nun einen Data Mediator, um die Parameter in die vom Service gewünschte Datenstruktur zu transformieren und einen Protocol Mediator, der für den Aufruf des Services zuständig ist. Nach dem Aufruf des Services wandelt der Proxy wiederum die Rückgabewerte mit Hilfe des Data Mediators in das vom BPEL-Prozess gewünschte Datenformat um und liefert diesen Wert zurück.

2.2.5 WSMO

Web Service Modeling Ontology (WSMO) stellt einen weiteren Ansatz für Semantic Web Services dar. WSMO wird von der Web Service Modeling Ontology (WSMO) Working Group entwickelt und ist seit April 2005 eine W3C Submission [W3C05]. Ein konzeptionelles Modell für SWS wurde ursprünglich in [FB02] als Web Service Modeling Framework (WSMF) beschrieben. WSMO verfeinert und erweitert WSMF. Es beinhaltet die Unterprojekte WSML und WSMX. WSML definiert formale Sprachen für WSMO und ist als Konkurrenz zu OWL zu sehen. WSMX stellt eine Laufumgebung und gleichzeitig die Referenzimplementierung von WSMO dar.

WSMO

Das konzeptionelle Modell von WSMO beschreibt vier Hauptelemente, nämlich Ontologien, Web Services, Ziele (Goals) und Mediators.
Ontologien werden in WSMO definiert durch Konzepte, Attribute, Relationen, Funktionen, Instanzen und Axiome. Für die Definition wird eine der WSML-Sprachvarianten verwendet.
Web Services-Beschreibungen in WSMO bestehen aus Capabilities und Interfaces. Eine Capability beschreibt die Funktionalität des Web Service und besteht aus nichtfunktionalen Eigenschaften, Mediators, Preconditions, Postonditions, Assumptions und Effects. Ein Interface besteht aus der Choreography und der Orchestrierung und beschreibt damit wie der Web Service verwendet wird. Die Choreography beschreibt, wie der Service-Requester mit dem Service interagieren soll. Dazu gehört die Abfolge der Nachrichten, d.h. das externsichtbare Verhalten, und das WSDL-Grounding. Die Orchestrierung beschreibt wie der Service seine Funktionalität realisiert, indem er andere Services aggregiert.
Ziele (Goals) werden vom Service-Requester spezifiziert und bezeichnen die vom Service-Requester erwartete Funktionalität. Ziele werden technisch genauso definiert wie Web Service-Beschreibungen, also durch Capabilities und Interfaces, werden aber in WSMO im Gegensatz zu den anderen Frameworks konzeptionell von Web Services getrennt.
Mediators fungieren als Vermittler zwischen WSMO-Elementen, um verschiedene Arten von Heterogenität (Daten, Protokoll, Prozess) aufzulösen. So werden OO-Mediators verwendet, um Konflikte zwischen zwei Ontologien aufzulösen, während WG-Mediators einen Web Service auf einen Goal abstimmen. Die Motivation für Mediators wird in [FB02] dargelegt, wo argumentiert wird, dass man sie aufgrund ihrer Wichtigkeit bei der Integartaion von Anwendungen als „First-Class-Citizens“ definieren sollte. Bei anderen Ansätzen wie OWL-S werden Mediator-Aufgaben von der zugrundeliegenden Infrastruktur bzw. Implementierung übernommen und nicht direkt in der Service-Ontologie modelliert.

WSML

Die Web Service Modeling Language (WSML) umfasst mehrere Sprachvarianten zur Beschreibung von Ontologien, Web Services, Goals und Mediators. Bei gleicher Syntax werden in einem Schichtenmodell mehrere Sprachvarianten angeboten, die auf Formalismen wie Datalog, Beschreibungslogik und Frame-Logik basieren. WSML-Core unterstützt dabei z.B. nur die Schnittmenge von der Beschreibungslogik und der Hornlogik, während WSML-Full auf der Prädikatenlogik erster Stufe basiert und damit richtige Logikprogrammierung ermöglicht. Je nach Anwendungstyp kann die passende Sprache gewählt werden. WSML ist eine Frame-basierte Sprache und damit für den Menschen leichter lesbar als etwa OWL. Um Interoperabilität über das Web zu ermöglichen, werden aber Abbildungen zu XML, RDF und OWL definiert.

WSMX

Die Web Service Execution Environment (WSMX) stellt die Referenzimplementierung von WSMO dar. Sie unterstützt Publikation, Auffinden, Orchestrierung, Mediation und Aufru-

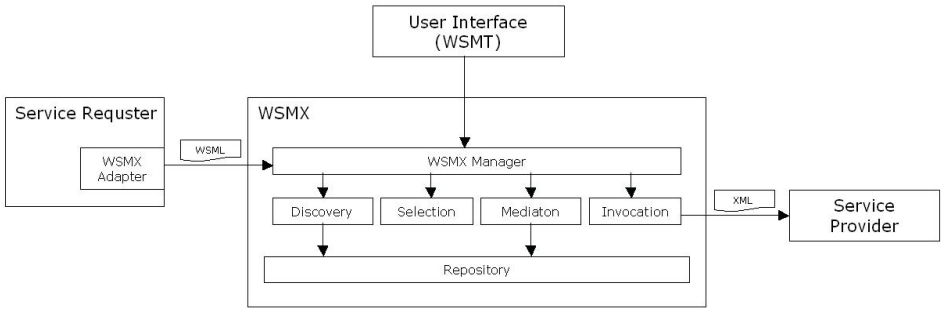

Abbildung 2.8: WSMX-Architektur

fen von Web Services. Eine stark vereinfachte Architektur von WSMX wird in der Abbildung 2.8 dargestellt [ZMH05].

WSMX fungiert als Broker in einer Hub-and-Spoke-Architektur, die für EAI-Systeme typisch ist, und ist selbst als Web Service erreichbar. Die Aufrufe zwischen Service-Requester und -Provider geschehen nie direkt, sondern gehen immer über WSMX.

Die *Publikation* von Service-Beschreibungen geschieht über das clientseitige GUI-basierte Web Service Modeling Toolkit (WSMT). Mit dessen Hilfe können Beschreibungen der WSMO-Elemente, nämlich Ontologien, Web Services, Goals und Mediators angelegt werden. Sie werden über den WSMX Manager an eine Compiler-Komponente übergeben werden, die diese validiert und in einer kompilierten Form im Repository speichert.

Die Schnittstelle von WSMX bietet drei Operationen an:

- *achieveGoal:* Diese Operation bekommt ein WSMO-Goal übergeben. WSMX führt die Suche, Auswahl und Aufruf eines Web Service durch und liefert das Ergebnis des Aufrufs an den Requester zurück.

- *getWebService:* Diese Operation bekommt auch ein WSMO-Goal übergeben. WSMX führt aber nur eine Suche und Auswahl durch und liefert die zum WSMO-Goal passenden Web Services an den Requester zurück.

- *invokeWebService:* Der Requester wählt einen der Services aus und lässt diesen von WSMX aufrufen.

Der Service-Requester spezifiziert die Suchanfrage an WSMX als WSMO-Goal, das in WSML beschrieben ist. Unterstützt der Requester dieses Format nicht direkt so wird ein WSMX Adapter verwendet, um die Transformation durchzuführen. Der WSMX Manager nimmt die Anfrage entgegen und leitet sie weiter an die Discovery-Komponente, die zu dem Goal die passenden Web Services im Repository sucht. Die Selector-Komponente wählt aus möglicherweise mehreren Suchergebnissen anhand von Präferenzen wie nichtfunktionalen Anforderungen den passendsten Web Service aus. Bei der Suche kann ein Mediator verwendet werden, um eventuelle Unterschiede zwischen Goals und Web Services zu überbrücken.

Beim Aufruf von Web Services hat WSMX vor allem die Aufgabe mit Hilfe eines Data Mediators und einer Choreography Engine Unterschiede bei verwendeten Ontologien bzw. Kommunikationsverhalten zwischen Requester und Provider aufzulösen. Der WSMX Manager hat die Funktion eines Micro-Kernels und basiert auf einer ereignisorientierten Architektur. Diese ermöglicht eine starke Entkopplung der Komponenten, ganz im Sinne von SOA. Die Komponenten rufen sich nicht gegenseitig auf, sondern kommunizieren über Nachrichten. Die Koordination geschieht über den WSMX Manager.

3 Lösungsansatz

Dieses Kapitel beschreibt einen Lösungsansatz für die Aufgabenstellung, die im Abschnitt 1.3 beschrieben wurde. Der Lösungsansatz wird in zwei Aspekte unterteilt, die Spezifizierung der Suchanfrage und die Architektur der notwendigen BPEL-Engine-Erweiterung. Dabei wird ausgehend von den Anforderungen eine mögliche Lösung vorgestellt und mit den in Kapitel 2 vorgestellten Ansätzen verglichen.

Der Inhalt der Suchanfrage wird im Lösungsansatz zunächst unabhängig von einer bestimmten SWS-Technologie spezifiziert, da es in diesem Bereich noch keinen Standard gibt. Es wird zunächst nur konzeptionell betrachtet, welche semantischen Informationen nötig sind, um die Anforderungen zu erfüllen. Die Umsetzung durch konkrete Technologien im Rahmen der erfolgten Implementierung wird dann im Kapitel 4 erläutert.

3.1 Spezifizierung der Suchanfrage

3.1.1 Anforderungen

Die Motivation für die Spezifizierung der Suchanfrage wurde im Abschnitt 1.2 bzw. [KLB05] erläutert. Sie wird innerhalb eines Evaluate-Konstrukts zu einer Invoke-Aktivität hinzugefügt, um für diese Invoke-Aktivität dynamische Suche, Auswahl und Aufruf eines Partner Services zu ermöglichen.

Die Anforderungen an die Suchanfrage sind die folgenden:

- Die Suche und Auswahl von Partner Services soll bzgl. der PortType- und Operation-Namen nicht an den vorkonfigurierten Service im BPEL-Prozessmodell gebunden sein.
- Die Suche soll nichtfunktionale Eigenschaften wie z.B. Sicherheit, Preis und Qualität berücksichtigen können.
- Die Suchanfrage soll BPEL-Übergangsbedingungen formulieren können.

3.1.2 Vorüberlegungen

Funktionale Beschreibung

Eine Invoke-Aktivität ruft eine WSDL-Operation eines Partner Services auf. Der Prozessmodellierer hat zur Modellierungszeit für diese Invoke-Aktivität einen bestimmten Partner Service ausgewählt und dessen WSDL-PortType-Beschreibung in dem zugehörigen Partner Link definiert. Die Auswahl des Partner Links bzw. der aufzurufenden Operation geschieht durch den Prozessmodellierer manuell. Er will durch die Invoke-Aktivität einen bestimmten Service im abstrakten Sinne erbracht haben. Er erkennt an der WSDL-Beschreibung des ausgesuchten Partners, dass dieser den gesuchten Service erbringen kann. Dazu liest er sich z.B. die textuelle Beschreibung zum WSDL-Dokument durch oder erkennt bei entsprechend aussagekräftiger Namensgebung der WSDL-Entitäten, welchen Service die WSDL-Operation implementiert. Wichtig ist aber zu beachten, dass die Semantik der WSDL-Operation nur implizit vorhanden und nur durch einen Menschen verstanden werden kann.

Da die Suche nach Partner Services unabhängig vom konfigurierten PortType *automatisch* durchgeführt werden soll, muss die vom Prozessmodellierer vorgesehene Semantik der durch die Invoke-Aktivität aufgerufenen Funktionalität explizit formuliert werden. Das Ziel der semantischen Beschreibung der Funktionalität in der Suchanfrage ist, die Semantik der aufgerufenen WSDL-Operation zu erfassen und damit die Abhängigkeit von einem spezifischen WSDL-PortType zu beseitigen.

Man kann bei einem Aufruf einer WSDL-Operation zwei Arten von Semantik unterscheiden, die funktionale und die Datensemantik. Die funktionale Semantik beschreibt, welche Funktion die Operation durchführt. Sie wird in WSDL-S beispielsweise durch Vorbedingungen, Effekte und durch eine Referenz auf ein Ontologiekonzept, das die Funktion beschreibt, spezifiziert. Die Datensemantik beschreibt die Input- und Output-Messages der Operation, in WSDL-S durch Referenzieren von Ontologiekonzepten.

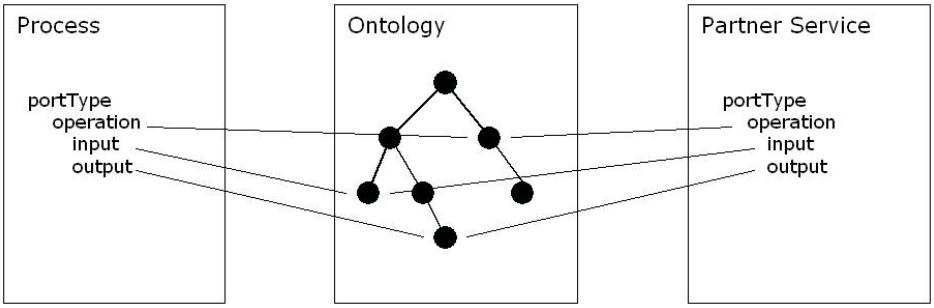

Abbildung 3.1: Annotation von WSDL-Operationen

Annotieren der vordefinierte PortType im Prozessmodell und der PortType des Partner Services dieselben Konzepte, wie in Abbildung 3.1 dargestellt, so wird der Partner Service durch einen semantischen Matchmaker als passend ausgewählt.

Für die Suche nach passenden Services ist die Annotation durch Ontologiekonzepte bereits ausreichend. In der Motivation wurde angenommen, dass die Input- und Output-Messages der gesuchten Partner Services dasselbe XML-Format aufweisen. Dies ist dann der Fall,

wenn sie domänenspezifische XML-basierte Standards wie RosettaNet implementieren. In diesem Fall verwenden alle Partner Services dieselben XML-Schema-Definitionen. Unter dieser Annahme kann nach einer erfolgreichen Suche der Aufruf ohne Datentransformationen durchgeführt werden.

Sind aber die Datenformate der Input- bzw. Output-Messages nicht standardisiert, so muss zusätzlich zu der Annotation definiert werden, wie der XML-Datentyp auf das Ontologie-Konzept abzubilden ist, damit der Partner Service aufgerufen werden kann. Es muss daher in der Suchanfrage spezifiziert werden wie die Input-Message der vorkonfigurierten WSDL-Operation, die typischerweise in XML-Schema definiert ist, auf das zugehörige Ontologie-Konzept abgebildet wird. Ontologiesprachen wie OWL haben eine XML-Serialisierung, entsprechend kann für eine solche Abbildung eine Sprache wie XSLT verwendet werden. Bei der Ausgabe ist es dagegen genau umgekehrt, d.h. es muss eine Abbildung von Ontologie-Konzepten auf die Output-Message definiert werden.

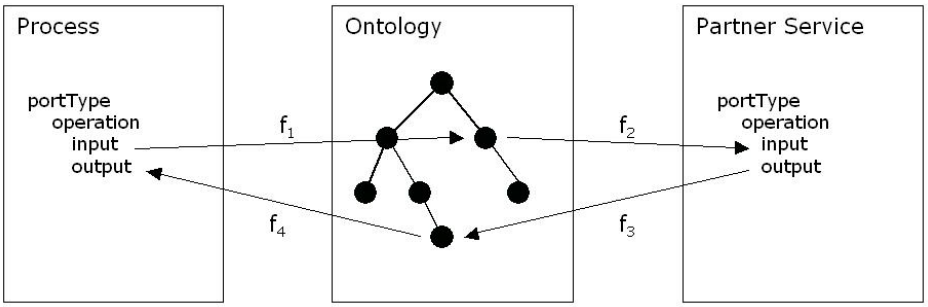

Abbildung 3.2: Mapping zwischen WSDL-Messages und Ontologie-Konzepten

Eine Servicebeschreibung dagegen spezifiziert die Abbildung genau umgekehrt, wie in Abbildung 3.2 gezeigt. Die Abbildungsfunktionen f1 und f4 werden in der Suchanfrage, f2 und f3 in der Servicebeschreibung definiert. Zur Laufzeit wird also zunächst f1 ausgeführt und damit die Daten aus der Eingabevariable der Invoke-Aktivität auf das zugehörige Ontologie-Konzept abgebildet, f2 konvertiert nun die Daten aus dem Ontologie-Konzept auf die Eingabe wie vom Service benötigt. Mit dieser Eingabe kann der Service aufgerufen werden. Bei der Ausgabe ist es genau umgekehrt.

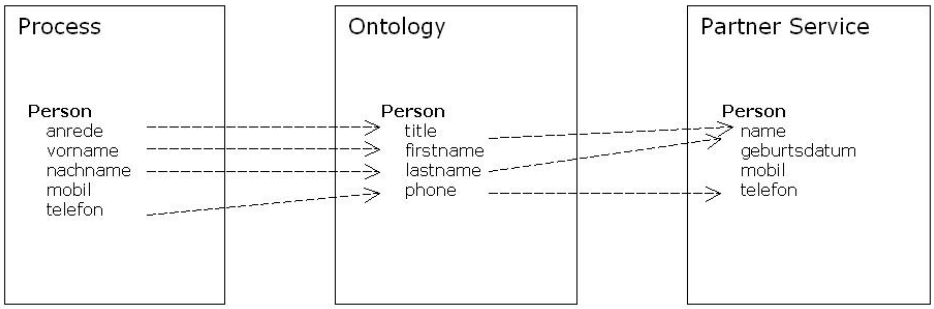

Abbildung 3.3: Mapping-Problematik

Bei der Transformation ist zu beachten, dass dabei eventuell Informationen verloren gehen,

wie in 3.2 gezeigt, bzw. unvollständig sind. Nicht immer ist auch eine sinnvolle Transformation möglich. Auch wird hier angenommen, dass eine gemeinsame Ontologie verwendet wird. Eine umfassendere Betrachtung dieser Probleme bzw. Literaturreferenzen diesbezüglich sind in der WSDL-S Spezifikation zu finden [AFM+05].

Im Abschnitt 2.2 wurden mehrere Ansätze vorgestellt, um Web Services semantisch zu beschreiben. Die obigen Betrachtungen wurden am Beispiel von WSDL-S ausgeführt. Für die Semantik-Beschreibung kann aber jede beliebige SWS-Technologie verwendet werden, solange sie durch ein Grounding eine Abbildung ihrer Konzepte auf WSDL-Operationen spezifiziert. Es würden also auch OWL-S, WSMO und METEOR-S in Frage kommen. Die Suchanfrage muss natürlich dieselbe Technologie wie die Services verwenden, nach denen gesucht werden soll.

Nichtfunktionale Beschreibung

Als nichtfunktionale Beschreibungen von Web Services können alle Metadaten betrachtet werden, die nicht in der WSDL-Beschreibung enthalten sind. WSDL beschreibt die Funktionalität von Services in Form von Operationen mit Eingabe- und Ausgabeparametern und wie man diese Operationen konkret über das Netzwerk aufruft. Nichtfunktionale Metadaten werden zur Zeit mit Hilfe von UDDI und WS-Policy spezifiziert.

Um die Services aufrufen zu können, müssen sie zunächst aufgefunden werden. Sie werden im UDDI-Verzeichnis publiziert und dabei durch Metadaten beschrieben. Diese Metadaten dienen der Suche und beeinflussen nicht die Ausführung des Services. Dazu gehören z.B. der Servicename, Service-Provider-Name und -Kontaktinformationen und die textuelle Beschreibung zur Verwendung des Services. Ein Service kann in bestimmte Kategorien eingeordnet werden, um eine Gelbe-Seiten-Suche zu ermöglichen. Standardisierte Kategorien, die innerhalb von UDDI verwendet werden, sind NAICS, UNSPSC, DUNS und ISO-3166. Es können aber beliebige Kategorien definiert werden.

WS-Policy dagegen definiert die technischen „Richtlinien“ für die Kommunikation bzw. den Zugriff auf einen Web Service. Dazu gehören die klassischen Dienstgüte-Kriterien: Transaktionen, Reliable Messaging und Sicherheit. Im Moment existieren domänenspezifische Vokabulare für Sicherheit und Reliable Messaging.

Die SWS-Technologien unterstützen für die Suche ähnliche Metadaten wie UDDI. So definiert das OWL-S-Profile auch einen Servicenamen und eine textuelle Beschreibung. Durch das Attribut „serviceCategory“ kann auf die o.g. standardisierten Kategorien verwiesen werden. Über „serviceParameter“ können beliebige weitere Beschreibungen über Name-Wert-Paare hinzugefügt werden, wobei auf die Werte über URIs zugegriffen wird. In WSDL-S kann ein PortType ebenfalls durch „category“-Elemente kategorisiert werden.

Zusammenfassend läßt sich also sagen, dass WS-Technologien im Wesentlichen die UDDI-Metadaten und WS-Policy verwenden. UDDI unterstützt aber keine semantische Suche. Wenn Services für die Suche im UDDI hinterlegt sind, so müssen diese entweder manuell zur Modellierungszeit in die Wissensbasis eines semantischen Matchmakers eingefügt werden. Oder man führt zur Laufzeit die Suche entsprechend in zwei Schritten durch: Zunächst wird anhand von Kategorisierungsinformationen eine Suche im UDDI durchgeführt,

um eine grobe Vorauswahl zu treffen. Anschließend werden die gefundenen Services an den semantischen Matchmaker übergeben, der dann die semantische Suche durchführt.

Bedingungen in der Suchanfrage

Die Suchanfrage soll BPEL-Übergangsbedingungen formulieren können. Die Motivation dafür wurde im Abschnitt 1.2.3 dargelegt. Das Ziel ist mehrere parallele Zweige, die mit Hilfe der Switch-Aktivität modelliert werden, zu einem zusammenfassen zu können, wobei die Übergangsbedingungen aus den Case-Elementen der Switch-Aktivität in die Suchanfrage wandern.

Dadurch ergeben sich in der Suchanfrage Auswahlregeln der Form: Wenn (Übergangsbedingung), Dann (Service hat Eigenschaft X).

Die Bedingungen werden typischerweise in XPath formuliert. Sie können auf die Laufzeitdaten der Prozessinstanz über die BPEL-Funktionen „bpws:getVariableData“, „bpws:getVariableProperty“ und „bpws:getLinkStatus“ zugreifen. Um sie auswerten zu können, ist also Zugriff auf den Prozessinstanzkontext notwendig.

3.1.3 Inhalt der Suchanfrage

Die Suchanfrage enthält folgende Elemente:

- *Semantische Beschreibung der Funktionalität:* Die semantische Beschreibung spezifiziert die funktionale Semantik und die Datensemantik der WSDL-Operation. Dabei werden wie oben dargestellt Abbildungen von den Input- und Output-Messages auf die dazugehörigen Ontologie-Konzepte spezifiziert.
- *Kategorisierungen:* Die Kategorisierungen spezifizieren Werte in den standardisierten Kategorien NAICS, UNSPSC, ISO-3166, DUNS. Diese erlauben eine Suche in der UDDI-Registry.
- *WS-Policy-Ausdruck:* Der WS-Policy-Ausdruck spezifiziert die technischen Anforderungen an den gefundenen Service.
- *BPEL-Übergangsbedingungen:* Diese enthalten andere Elementtypen aus der Suchanfrage und spezifizieren eine Bedingung, die typischerweise mit Hilfe von XPath ausgedrückt wird. Wenn die Bedingung „wahr“ ist, werden die enthaltenden Elemente bei der Suche berücksichtigt, ansonsten nicht.
- *Weitere Service-Eigenschaften:* Abhängig von der verwendeten Technologie ist ein Erweiterungsmechamismus sinnvoll, der beliebige Service-Eigenschaften definieren kann, wie bespielsweise Preis und Qualität (siehe Abschnitt 4.4.3).

Ein konkretes Format einer Suchanfrage wird bei der Umsetzung spezifiziert (siehe Abschnitt 4.4).

3.2 Erweiterung der BPEL-Engine

3.2.1 Anforderungen

Um die dynamische Suche und Aufruf von Partner Services anhand der Suchanfrage zu ermöglichen, muss die BPEL-Engine erweitert werden. Folgende Erweiterungen sind dabei nötig:

- *Berücksichtigung des Evaluate-Elements beim Parsen des Prozessmodells:* Die Suchanfrage wird innerhalb eines Evaluate-Elements spezifiziert, das als Erweiterungselement der Invoke-Aktivität fungiert. Das Evaluate-Element muss beim Einlesen/Parsen des Prozessmodells während des Deployment-Vorgangs erkannt und eingelesen werden. Die eingelesenen Daten müssen dann über den Prozesskontext zur Verfügung gestellt werden, so dass zur Laufzeit darauf programmatisch zugegriffen werden kann.
- *Parsen der Suchanfrage und Evaluierung der darin enthaltenen Bedingungen:* Spätestens beim Ausführen der zugehörigen Invoke-Aktivität muss die Suchanfrage eingelesen werden und die darin enthaltenen Bedingungen evaluiert werden.
- *Automatische Suche und Auswahl von passenden Services anhand der Suchanfrage:* Zur Laufzeit werden passende Services in einer Service-Registry gesucht und ausgewählt. Dabei müssen natürlich alle Kriterien in der Suchanfrage berücksichtigt werden, insbesondere semantische Beschreibungen und WS-Policy-Ausdrücke.
- *Aufruf des ausgewählten Service:* Nach Auswahl des passendsten Service muss dieser aufgerufen werden können, wobei u.U. Datentransformationen nötig sind. Ist der Aufruf nicht möglich, so wird der nächstbeste Service in der Liste genommen.

Da die Suchanfrage innerhalb eines Erweiterungselements spezifiziert und somit das BPEL-Metaprozessmodell erweitert wird, ist ein BPEL-Prozessmodell, das eine solche Erweiterung verwendet, konkret ein Evaluate-Element innerhalb einer Invoke-Aktivität, nicht mehr auf BPEL-Engine-Ebene portabel. Er kann zwar in jeder beliebigen BPEL-Engine installiert und ausgeführt werden, in diesem Sinne bleibt er portabel, das Evaluate-Element innerhalb der Invoke-Aktivität wird aber dabei nicht berücksichtigt. Um eine Portierung auf verschiedene BPEL-Engines zu erleichtern, sollte die Architektur den BPEL-Engine-spezifischen Teil möglichst sauber vom Rest der benötigten Erweiterung trennen.

3.2.2 Architektur

Die folgende Abbildung 3.4 zeigt eine mögliche Architektur für die benötigte Erweiterung einer BPEL-Engine.

Zur Deployment-Zeit des Prozessmodells muss zunächst das Evaluate-Element mit der Suchanfrage gelesen werden, um zur Laufzeit zur Verfügung zu stehen. Zum Zeitpunkt, wenn die Invoke-Aktivität mit Suchanfrage-Strategie ausgeführt wird, wird ein Adapter aufgerufen. Dieser liest die Suchanfrage aus dem Prozessinstanzkontext aus, evaluiert darin eventuell vorkommende Bedingungen und Variablenreferenzen, generiert eine Anfrage an

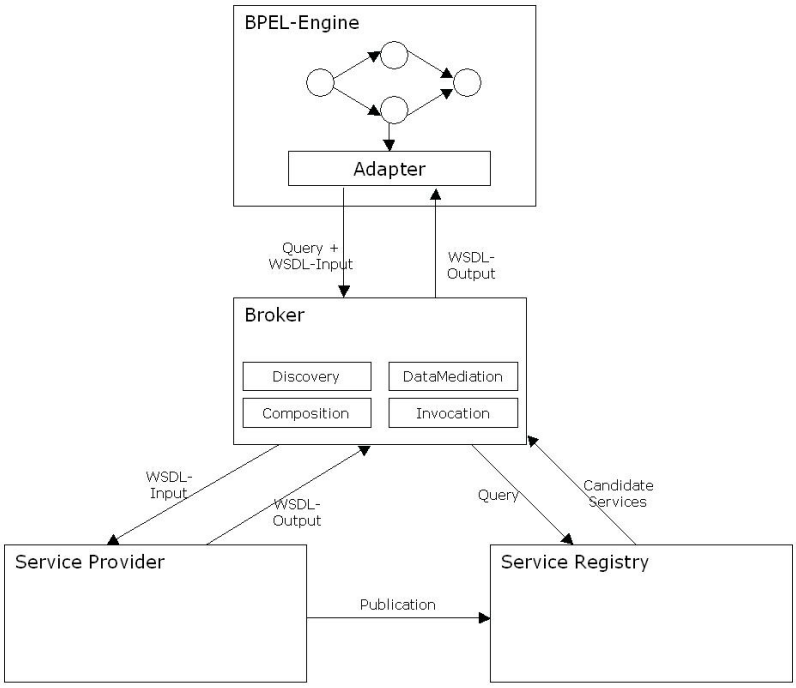

Abbildung 3.4: Architektur der BPEL-Engine-Erweiterung

den Broker und ruft diesen auf. Diese Anfrage enthält die Suchanfrage und WSDL-Input-Daten der Invoke-Aktivität. Der Broker ist zuständig für die Suche, Auswahl und Aufruf von Web Services, anhand der Suchanfrage, die u.a. semantische Beschreibungen enthält. Das Ergebnis des Aufrufs wird über den Adapter wieder zurück an die BPEL-Engine geliefert.

Im Folgenden werden die einzelnen Komponenten im Detail erklärt.

Parsen des Prozessmodells

Die BPEL-Engine muss dahingehend erweitert werden, dass sie das Evaluate-Element im Prozessmodell erkennt, ausliest und zur Laufzeit dem Adapter zur Verfügung stellt. Diese Erweiterung ist BPEL-Engine spezifisch, muss also für jede BPEL-Engine, die das Evaluate-Element unterstützen will, gesondert implementiert werden.

Adapter

Der Adapter ist ebenfalls eine BPEL-Engine spezifische Komponente und wird zur Laufzeit von der BPEL-Engine aufgerufen, wenn eine Invoke-Aktivität, die ein Evaluate-Konstrukt mit Suchanfrage enthält, ausgeführt werden soll. Der Adapter kapselt die Funktionalität, die BPEL-Engine spezifisch ist und macht dadurch den Broker davon unabhängig.

Der Adapter ist Teil der BPEL-Engine und hat im Gegensatz zum Broker Zugriff auf den gesamten Prozessinstanzkontext, d.h. alle Informationen der Prozessinstanz zur Laufzeit, insbesondere also auch Variablenwerte. Dadurch kann er Bedingungen über Variablenwerte auswerten bzw. Variablenwerte auslesen, die u.U. in der Suchanfrage enthalten sind.

Der Adapter führt folgende Schritte durch:

- Er wird bei Ausführung einer Invoke-Aktivität mit Suchanfrage von der BPEL-Engine aufgerufen. Er bekommt den Prozessinstanzkontext übergeben. Aus diesem liest er die Suchanfrage aus und wertet darin vorkommende Bedingungen bzw. Variablenzugriffe, die auf den Prozessinstanzkontext zugreifen, aus. Möglicherweise müssen zusätzliche WS-Policy-Ausdrücke, die beispielsweise am übergeordneten Scope hängen, mit berücksichtigt werden.
- Die Suchanfrage wird zusammen mit den Eingabedaten der Operation an den Broker geschickt.
- Der Broker sucht einen passenden Service zur Suchanfrage, ruft diesen auf und liefert die Ausgabedaten und Informationen zum aufgerufenen Service zurück. Die Ausgabedaten des Brokers werden in das BPEL-Engine-spezifische Format konvertiert und an die Engine zurückgegeben. Im Fall, dass der Broker eine Fehlermeldung zurückgibt, wird diese an die Engine weitergeleitet.

In der dargestellten Arbeitsweise des Adapters hat der Adapter und damit die BPEL-Engine keinen Einfluss auf den aufgerufenen Service, da der Broker die Suche und den Aufruf aus Sicht des Adapters in einem Schritt durchführt. Aus Sicht der BPEL-Engine ist der Service virtuell. Es kann nun aber sein, dass diese Virtualisierung in manchen Fällen nicht erwünscht ist. Man könnte sich vorstellen, dass für bestimmte Operationen der Prozessadministator nach einer Vorauswahl durch den Broker selbst manuell den Service auswählen möchte, um selbst die Kontrolle darüber zu behalten, welcher konkrete Service ausgewählt wird. Ein Grund dafür wäre mangelndes Vertrauen an die Auswahlkomponente des Service-Bus oder man möchte nicht die endgültigen Auswahlregeln preisgeben. Möglicherweise wurden auch keine Services gefunden, die allen in der Suchanfrage gestellten Kriterien genügen. In diesem Fall könnte es wieder sinnvoll sein eine Liste von potentiellen Services zurückzuliefern und den Prozessadministrator auswählen bzw. entscheiden zu lassen.

Für diesen Fall bietet der Broker eine zweite Schnittstelle an. Der Adapter ruft dazu den Broker zweimal auf. Beim ersten Aufruf übergibt er dem Broker die Suchanfrage und bekommt passende Services geliefert. Aus diesem wählt der Adapter einen aus bzw. läßt die Auswahl von einem Prozessadministrator durchführen und ruft dann den Broker ein zweites Mal auf, indem er ihm einen konkreten Service angibt und die Eingabedaten mitliefert. Der Partner Service wird also wieder über den Broker aufgerufen, da unter Umständen semantikbasierte Datentransformationen (siehe Abschnitt 3.1.2) nötig sind, die die BPEL-Engine nicht durchführen kann.

Broker

Der Broker ist angelehnt an das Broker-Pattern [BMR+96], was in verteilten Systemen verwendet wird, um die Kommunikation von verteilten Komponenten zu entkoppeln.

Der Broker koordiniert dabei die Kommunikation zwischen der Client- und der Server-Komponente, indem er Aufrufe des Clients an einen passenden Server weiterleitet und das Ergebnis des Aufrufs an den Client zurückliefert. Dies hat den Vorteil, dass der Client nicht wissen muss, wie er auf einer systemnahen Ebene mit dem Server kommunizieren muss. Zusätzlich kann der Broker einen passenden Server dynamisch bestimmen.

Die Architektur wie in Abbildung 3.4 dargestellt, hat Ähnlichkeit mit der Service-Bus-Architektur (siehe Abschnitt 1.1.2 bzw. [WCL+05], wobei hier die BPEL-Engine als Service-Requester fungiert und der Broker die Service-Bus-Funktionalität implementiert. Der Broker hat in unserem Kontext vor allem die Aufgabe auf Basis einer SWS-Technologie passende Services zu suchen und aufzurufen. Dazu gehören u.U. auch Datentransformationen und automatische Service-Komposition, also Funktionalität, die eine BPEL-Engine nicht unterstützt. Der Broker implementiert diese Funktionalität und bietet sie für Service-Requester transparent an. In einer Service-Bus-Architektur [WCL+05] ist der Broker als ein Teil des Service-Bus zu sehen.

Der Broker bietet für den Service-Requester zwei Schnittstellen an:

- *Suche und Aufruf in einem Schritt:* Dazu bekommt er vom Service-Requester die Suchanfrage und die Eingabedaten übergeben. Er versucht anhand der Suchanfrage einen passenden Service zu finden, u.U. auch durch Komposition mehrerer Services, und ruft diesen auf ohne zwischendurch mit dem Requester zu kommunizieren. Das Ergebnis des Aufrufs wird an den Requester zurückgeliefert.
- *Suche und Aufruf getrennt:* Der Broker liefert anhand der Suchanfrage zunächst eine Liste von passenden Services an den Requester zurück. Im nächsten Schritt bekommt er dann vom Requester den Service geliefert, der aufgerufen werden soll.

Der Broker ist unabhängig von einer BPEL-Engine-Implementierung. Er ist typischerweise eine verteilte Komponente, die beispielsweise über eine WSDL-Schnittstelle angesprochen werden kann. Zur Schnittstelle des Brokers gehört auch das Format der Suchanfrage.

Zu den Aufgaben des Brokers gehören die Suche, Auswahl, Komposition, Datentransformation und Aufruf von Semantic Web Services. Wie diese vom Broker umgesetzt bzw. gelöst werden, ist abhängig von der Mächtigkeit der zugrundeliegenden SWS-Technologie bzw. der Ausdrucksstärke der Suchanfrage, die der Service-Requester formulieren kann. Der Service-Requester formuliert eine Suchanfrage, die mindestens die Semantik des aufzurufenden Service spezifiziert und als weitere Einschränkungen die Input- und Output-Datentypen, die typischerweise in XML-Schema definiert sind, enthält. Aus seiner Sicht ist es nun egal, welche(n) Service(s) der Broker aufruft, solange die spezifizierte Funktionalität erbracht wird und die Ausgabe das richtige Format hat. Der Broker könnte also durchaus auch automatische Komposition betreiben, also dynamisch einen Prozess zusammenstellen und ausführen, der die spezifizierte Semantik des Service-Requesters erfüllt. Eine einfache Form der Service-Komposition wurde in der Implementierung umgesetzt (siehe Abschnitt 4.5.4).

Die *Suche* führt der Broker anhand der Suchanfrage durch Zugriff auf eine Service-Registry. Für semantische Suche ist eine UDDI-Registry nicht ausreichend, d.h. es muss (zusätzlich) ein semantischer Matchmaker zum Einsatz kommen. Ob die Service-Registry Teil des Brokers ist, ist implementierungsabhängig. Für die Suche ist innerhalb des Brokers eine

Discovery-Komponente zuständig. Sie interagiert mit der Service-Registry, die die Service-Beschreibungen enthält.
Die Discovery-Komponente erstellt anhand der Suchanfrage eine oder mehrere Anfragen an die Registry, vergleichbar mit einem SQL-Zugriff auf eine relationale Datenbank. Tatsächlich gibt es SQL-ähnliche Sprachen wie RDQL [Sea04], mit deren Hilfe auf eine solche Registry zugegriffen werden könnte. Die Discovery-Komponente implementiert typischerweise einen filterbasierten Algorithmus [VAG+04]. So wird sie zunächst eine grobe Anfrage an die Registry stellen und beispielsweise alle Services von der Service-Registry anfragen, die nur die Ausgaben der Suchanfrage unterstützen. Wird zuerst eine Suche im UDDI durchgeführt, dann könnten alle Services einer bestimmten Kategorie angefordert werden. Dann kann die erste Auswahl nach weiteren Kriterien wie Eingaben und funktionaler Semantik einschränkt werden. Die erste grobe Auswahl bietet die Möglichkeit, dass u.U. Services gefunden werden, die zusammen die geforderte Semantik bzw. Eingaben und Ausgaben unterstützen. Inwieweit Komposition sinnvoll möglich ist, ist auch abhängig von der Mächtigkeit der semantischen Beschreibungen. Mindestens sollte es aber möglich sein, Datentransformationen einzubeziehen, um Abweichungen in Datenformaten von Eingaben und Ausgaben auszugleichen (siehe Abschnitt 4.5.4).

Für den *Aufruf* eines Services bekommt der Broker die WSDL-Inputdaten vom Adapter geliefert. Zusätzlich benötigt er aus der Suchanfrage die Abbildungen von den Input- und Output-Messages auf die dazugehörigen Ontologie-Konzepte, um die notwendigen Datentransformationen durchführen zu können (siehe Abschnitt 3.1.2).

3.2.3 Vergleich mit anderen Ansätzen

Im Abschnitt 2.2 wurden zwei Ansätze zur BPEL-Erweiterung vorgestellt, die dem hier vorgestellten Broker-Ansatz ähnlich sind, nämlich das Projekt METEOR-S (Abschnitt 2.2.4) und der auf DAML-S basierende SDS (Abschnitt 2.2.3).

Beide verwenden einen Proxy-basierten Ansatz. Der Proxy entspricht dabei dem Broker und implementiert ähnliche Funktionalität. Der Unterschied ist aber und daher der Name „Proxy“, dass dieser in den Invoke-Aktivitäten, die eine dynamische Suche benötigen, als Partner Service definiert wird. Der Proxy besitzt dazu eine entsprechende WSDL-Schnittstelle, um als Web Service aus dem BPEL-Prozess heraus aufgerufen werden zu können. Es findet also keine Erweiterung des BPEL-Metamodells um ein Konstrukt wie Evaluate statt. Die Suchanfrage wird stattdessen als Eingabeparameter an den Proxy geschickt. Die Input-Message des Proxys enthält also einen Proxy-spezifischen Teil, der die Suchanfrage enthält, und die eigentliche Eingabe, die aus Prozesssicht an den Partner Service zu übergeben ist.

Der Vorteil des Proxy-Ansatzes liegt darin, dass der entsprechende BPEL-Prozess in jeder BPEL-Engine lauffähig ist, während in unserem Fall die Erweiterung des BPEL-Metamodells um das Evaluate-Konstrukt eine entsprechende Erweiterung der BPEL-Engine-Implementierung erfordert.

Der Proxy-Ansatz hat den Nachteil, dass der Proxy keinen Zugriff auf den Prozessinstanzkontext hat und somit Bedingungen und Variablenzugriffe in der Suchanfrage, die auf den

Prozesskontext zugreifen müssen, nicht ausgewertet werden können. Der Proxy kann weiterhin nur die Schnittstelle des Brokers implementieren, die Suche und Aufruf in einem Schritt durchführt. Die BPEL-Engine kann also keinen Einfluss darauf haben, welcher Service aufgerufen wird. Da die Suchanfrage als Teil der Input-Variable definiert werden muss, ist es schwierig einen bestehenden Prozess um dynamische Aufrufe zu erweitern. Mit Hilfe des Evaluate-Konstrukts ist dies denkbar einfach. Man hängt einfach ein Evaluate-Konstrukt mit einer Suchanfrage unter die entsprechende Invoke-Aktivität. Im Fall eines Proxys muss dagegen der entsprechende Partner Link umdefiniert und die verwendete Input-Variable erweitert werden, um die Suchanfrage aufnehmen zu können. Die Input-Variable enthält dann einen „technischen“ Teil, nämlich die Suchanfrage, und die eigentlichen „fachlichen“ Daten, es findet also eine unerwünschte Vermischung von technischen und fachlichen Aspekten statt.

Zusammenfassend lässt sich also sagen, dass die Portabilität des Proxy-Ansatzes auf Kosten der Funktionalität bzw. Flexibilität geht. Für die Aufgabenstellung in dieser Arbeit kommt der Proxy-Ansatz nicht in Frage, auch wenn man auf das Evaluate-Konstrukt verzichten würde, da der Zugriff auf den Prozessinstanzkontext benötigt wird.

Die Schnittstellen des Brokers sind angelehnt an WSMX (siehe Abschnitt 2.2.5). WSMX wäre eine mögliche Implementierung des Brokers. Um WSMX anzubinden, würde die Suchanfrage ein WSMO-Goal spezifizieren und ein WSMX-Adapter dieses als WSMLDocument verpackt an WSMX schicken.

4 Umsetzung

Dieses Kapitel beschreibt die Umsetzung des Lösungsvorschlags in eine Erweiterung der BPEL-Engine ActiveBPEL. Die Erweiterung umfasst die in Abschnitt 3.2 vorgestellten Architektur-Komponenten *Adapter*, *Broker* und *Service-Registry*.

Als SWS-Technologie wurde OWL-S gewählt, da diese zum Zeitpunkt des Implementierungsanfangs die ausgereifteste Software-Unterstützung hatte.

4.1 Verwendete Software

Im Rahmen der Umsetzung wurde folgende Software verwendet:

- *Java:* Die Implementierung wie auch alle eingesetzten Bibliotheken verwenden die Programmiersprache Java. Als Entwicklungsumgebung wurde Eclipse verwendet.

- *ActiveBPEL:* Als BPEL-Engine wird ActiveBPEL 1.2 [Act] verwendet. ActiveBPEL ist opensource. Als SOAP-Implementierung kommt Axis zum Einsatz.

- *Tomcat:* Apache Tomcat [Tom] in der Version 5.0 ist ein Servlet-Container, der eine Teilmenge der J2EE 1.4 Spezifikation implementiert. Er wird benötigt als Ablaufumgebung für die BPEL-Engine. Zusätzlich integriert er einen Webserver, der dazu verwendet wird, Service-Beschreibungen wie WSDL und OWL-S über HTTP zugreifbar zu machen.

- *OWL-S API:* Die Mindswap OWL-S API [OWL] ist eine Java-basierte Implementierung von OWL-S 1.1. Sie bietet Unterstützung für das Lesen, Schreiben und Ausführen von OWL-S-Beschreibungen. Als OWL-Reasoner wird u.a. Pellet unterstützt.

- *Pellet:* Mindswap Pellet [Pel] ist ein OWL-Reasoner und wird in Verbindung mit der OWL-S API in der Anwendung für das semantische Matchmaking verwendet.

- *WS-Policy-Implementierung:* Als WS-Policy-Implementierung wird [Pol] von Apache verwendet. Sie befindet sich noch im frühen Entwicklungsstadium und ist nur über das CVS erreichbar.

4.2 Kreditantragsprozess

Als Beispielanwendung wird im Folgenden ein Kreditantragsprozess (Loan Approval) betrachtet, der in einer ähnlichen Form auch in der BPEL-Spezifikation [ACD+03] vorkommt. An ihm sollen die implementierten Konzepte anschaulich aufgezeigt werden.

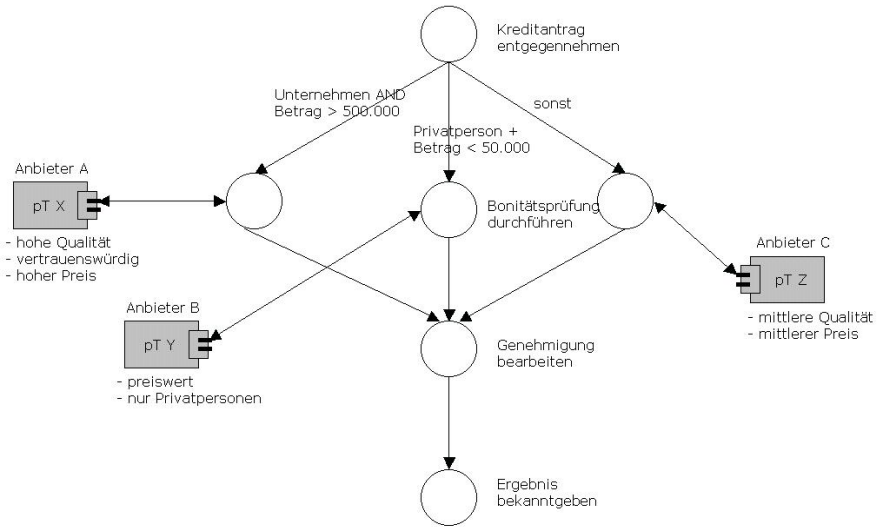

Abbildung 4.1: Kreditantragsprozess ohne Suchanfrage

In Abbildung 4.1 wird der Prozess dargestellt, wie er in BPEL ohne Suchanfrage aussehen könnte. Der Prozess wird von einer Bank implementiert und dient der Prüfung von Kreditanträgen, die von Privat- oder Geschäftskunden gestellt werden. Der Prozess wird gestartet durch das Entgegennehmen eines Kreditantrags. Dieser enthält den gewünschten Betrag und die Laufzeit. Anschließend wird eine Bonitätsprüfung durchgeführt, um das Risiko einzuschätzen, gefolgt von einer Genehmigung, die das Ergebnis der Risikoschätzung berücksichtigt. Schließlich wird das Ergebnis, ob der Antrag bewilligt wird, dem Antragsteller bekannt gegeben.

Die Bonitätsprüfung wird von externen Unternehmen durchgeführt. Wir nehmen an, die Bank habe drei Anbieter ausgewählt, die sich bezüglich nichtfunktionaler Eigenschaften wie Preis und Qualität unterscheiden. Die Bank möchte für hohe Anträge ab 500.000 Euro, die von Geschäftskunden gestellt werden, ausschließlich Anbieter A aufrufen, da dieser aus Sicht der Bank eine hohe Qualität liefert und hohe Vertrauenswürdigkeit besitzt. Anbieter B ist dagegen sehr preiswert und führt Prüfungen nur für Privatpersonen durch, entsprechend soll dieser Service für Beträge unter 50.000 Euro aufgerufen werden. Anbieter C liefert solide Qualität bei akzeptablem Preis sowohl für Unternehmen als auch für Privatpersonen. Die Aufrufe dieser drei Services wurden im Prozessmodell hartkodiert, die Auswahl geschieht über Übergangsbedingungen. Alle Services haben verschiedene PortTypes.

Die aufgezeigte Prozessmodellierung hat mehrere Schwächen, die durch eine dynamische Suche mit Hilfe der Suchanfrage, beseitigt werden können:

- Es findet keine dynamische Suche nach passenden Services statt. Die Services sind im Prozessmodell festkodiert. Tauchen mit der Zeit passendere Services auf, muss das Prozessmodell angepasst und neu deployt werden.
- Das Prozessmodell bildet einen Aufruf derselben Funktionalität bzw. Semantik auf drei Zweige ab. Für die drei Services müssen jeweils eigene Partner Links und Variablen definiert werden. Der Kontroll- und der Datenfluss bzw. allgemein das Prozessmodell sind dadurch komplexer als es (mit Suchanfrage) nötig wäre.

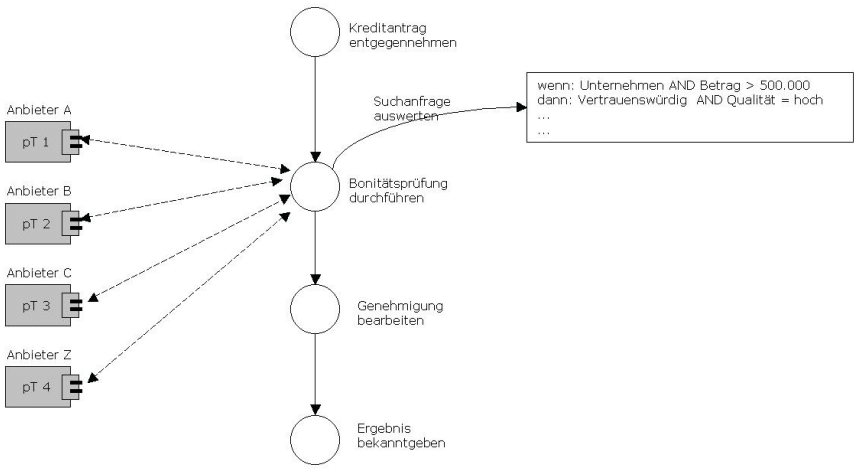

Abbildung 4.2: Kreditantragsprozess mit Suchanfrage

In Abbildung 4.2 wird dargestellt, wie das Prozessmodell mit Hilfe der Suchanfrage-Strategie vereinfacht werden kann. Die Übergangsbedingungen wandern in die Suchanfrage und werden als Wenn-Dann-Regeln modelliert. Die Kriterien, nach denen Services gesucht und gebunden werden sollen, werden deklarativ spezifiziert. Dadurch kann zur Laufzeit eine Suche nach den passendsten Services stattfinden.

4.2.1 Modellierung der domänenspezifischen Ontologie

Für die Bonitätsprüfung gibt es noch keine standardisierte Ontologie und auch keinen XML-basierten Standard wie etwa RosettaNet PIPs [Ros]. Solange es keinen Standard gibt, muss eine eigene OWL-Ontologie definiert werden, um SWS-Technologien nutzen zu können. Wir orientieren uns dabei an den vorhandenen WSDL-Beschreibungen der aufzurufenden Services und erstellen ein Domänenmodell in OWL.

Das Domänenmodell muss mindestens Konzepte für die Eingaben und Ausgaben und die Funktionalität der WSDL-Operation enthalten, um die Daten- bzw. funktionale Semantik spezifizieren zu können. Im Beispielaufruf sind das:

- *Eingabe:* OWL-Klasse `LoanRequester` bzw. ihre Unterklassen `PrivateRequester` und `BusinessRequester`.
- *Ausgabe:* OWL-Klasse `CreditReport`

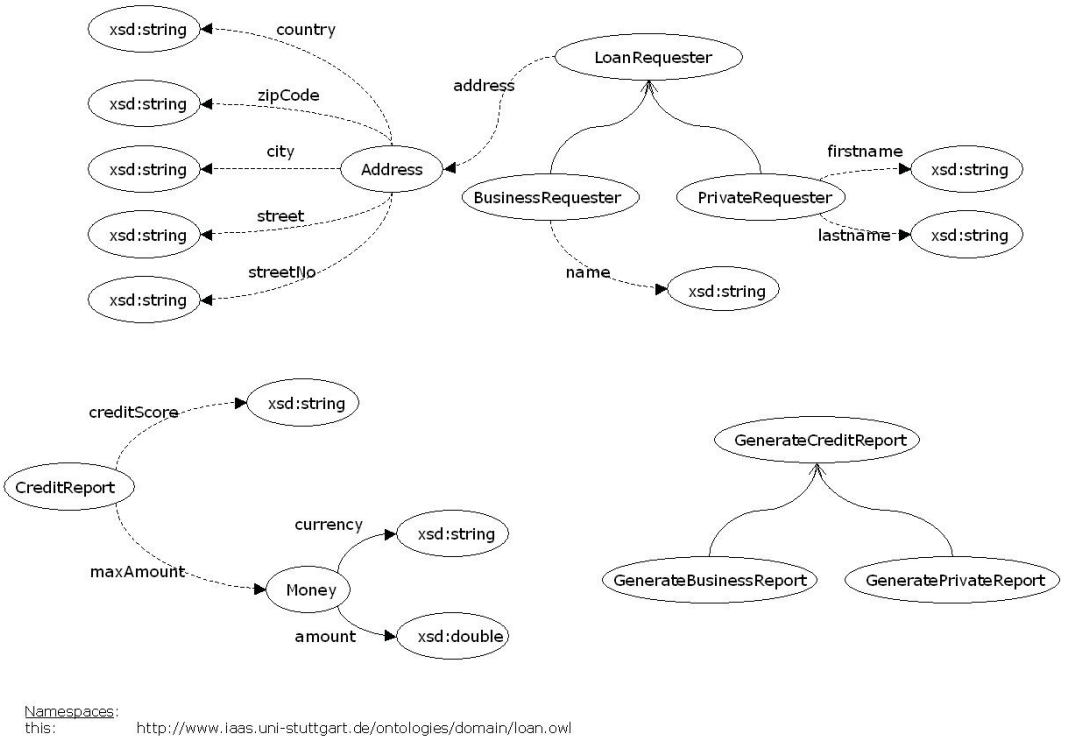

Abbildung 4.3: Ein Teil der OWL-Ontologie des Domänenmodells

- *Funktion:* OWL-Klasse `GenerateCreditReport` bzw. ihre Unterklassen `GeneratePrivateReport` und `GenerateBusinessReport`.

4.2.2 Erstellen der OWL-S-Servicebeschreibungen

Um nach den Services suchen bzw. sie aufrufen zu können, brauchen sie eine OWL-S-Beschreibung. Diese kann mit Hilfe des GUI-basierten Java-Programms *WSDL2OWL-S Converter*, das Bestandteil der OWL-S API ist, aus der WSDL-Beschreibung generiert werden. Die WSDL-Beschreibung wird häufig auch generiert, nämlich aus dem Programmcode selbst, der beispielsweise in Java geschrieben sein kann. Die umgekehrte Vorgehensweise, d.h. OWL-S -> WSDL -> Programmcode-Schnittstellen, würde dann Sinn machen, wenn OWL-S- bzw. WSDL-Beschreibungen standardisiert wären, was im Moment nicht der Fall ist.

Der *WSDL2OWL-S Converter* erlaubt die Erstellung von OWL-S-Services aus einzelnen WSDL-Operationen, d.h. zu einer WSDL-Operation wird ein kompletter OWL-S-Service erstellt, bestehend aus einem *Profile*, *AtomicProcess* und dem dazugehörigen *Grounding*.

Der Benutzer wählt dazu zunächst eine WSDL-Datei aus und daraus die WSDL-Operation. Zu jedem WSDL-Message-Part für die Input- und Output-Messages muss dann ein Ontologiekonzept als OWL-Typ zugewiesen werden und im Fall, dass es sich beim Message-Part um ein XML-Schema-ComplexType handelt, zusätzlich ein XSLT-Mapping.

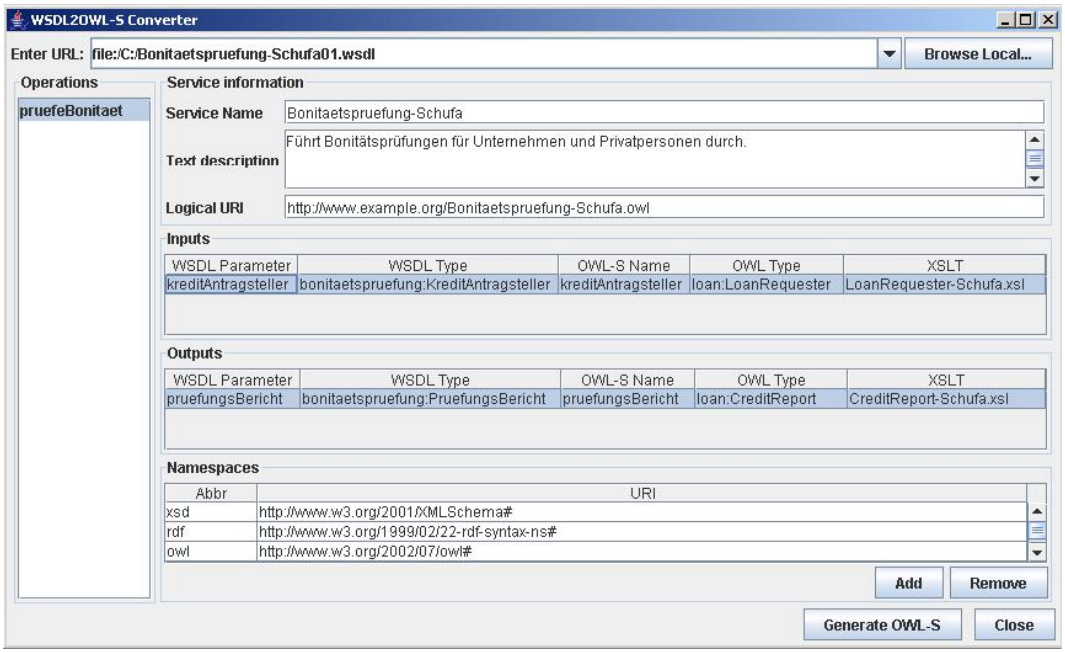

Abbildung 4.4: WSDL2OWL-S Converter

In Abbildung 4.4 wird eine Abbildung der WSDL-Operation „pruefeBonitaet“ gezeigt. Diese enthält als Input-Message einen Part namens „kreditAntragsteller“ mit dem ComplexType „KreditAntragsteller“. Dieses Part wird abgebildet auf den OWL-Typ „LoanRequester“. Die zugehörige XSLT-Definition in „LoanRequester-Schufa.xsl“ definiert wie die XML-Serialiserung des OWL-Types „LoanRequester“ auf den XML-Schema-ComplexType „Kreditantragsteller“ abzubilden ist. Bei der Output-Message ist es dagegen genau umgekehrt. Die dazugehörige XSLT-Definition definiert also wie ein „Pruefungsbericht“ auf den OWL-Typ CreditReport abgebildet wird. Diese Abbildungen werden zur Laufzeit beim Serviceaufruf für Datentransformationen benötigt, wie im Abschnitt 3.1.2 erklärt.

Mit Hilfe des *WSDL2OWL-S Converters* und den bisherigen Angaben läßt sich bereits eine ausführbare OWL-S-Beschreibung generieren. „Ausführbar“ bedeutet hier, dass eine OWL-S-Process-Engine, die Teil der OWL-S API ist, anhand der OWL-S-Beschreibung mit dem zugehörigen (WSDL-)Web Service kommunizieren kann. Dabei bekommt sie als Eingabe OWL-Instanzen, konvertiert diese in die zugehörige WSDL-Input-Message und ruft den Service auf (siehe Abschnitt 4.5.5).
Um nach der Beschreibung sinnvoll suchen zu können, müssen von Hand weitere Angaben hinzugefügt werden.

Ein OWL-S-Profile unterstützt über das Attribut „serviceCategory“ das Hinzufügen von wohlbekannten Kategorisierungen. Eine mögliche Kategorisierung ist in der folgenden Tabelle dargestellt:

Name	URI	Wert	Code
NAICS	www.naics.com	Credit bureaus	561450
UNSPSC	www.unspsc.org	Consumer credit reporting services	84141601
ISO-3166	www.iso.org	Baden-Württemberg, Deutschland	DE-BW

Tabelle 4.1: Kategorisierung

4.3 Erweiterung der OWL-S-Service-Ontologie

Die folgende Abbildung zeigt eine mögliche Erweiterung des OWL-S-Profiles um weitere nichtfunktionale Eigenschaften, die in einer Suche nützlich sein können.

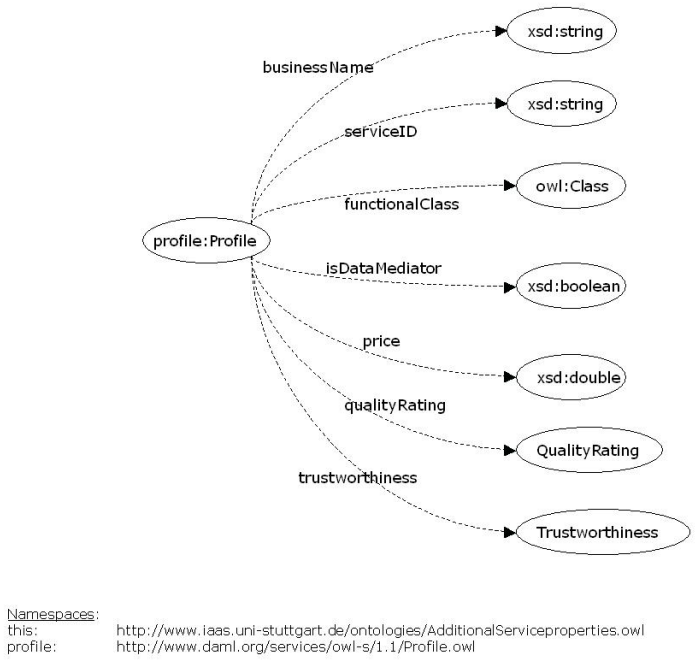

Abbildung 4.5: Zusätzliche Service-Eigenschaften

Das OWL-S-Profile wird um folgende Eigenschaften erweitert:

- *businessName:* Name des Unternehems, das den Service anbietet. Diese Information ist im UDDI vorhanden, nicht aber im OWL-S-Profile.
- *serviceID:* Ein eindeutiger Servicename. Dadurch wird es möglich, konkrete Services in der Suchanfrage zu referenzieren bzw. die Verwendung eines bestimmten Services zu erzwingen.
- *functionalClass:* Verweist auf ein Konzept, das die funktionale Semantik des Services modelliert.
- *isDataMediator:* Gibt an, ob es sich beim Service um einen Datamediator handelt. Ein Datamediator ist in diesem Kontext, ein Service mit einer WSDL-Schnittstelle, der eine

Datentransformation durchführt. Das OWL-Property wird benötigt bei der Service-Verkettung, siehe dazu Abschnitt 4.5.4.

- *price:* Gibt den Preis des Services an. Die Währung ist implizit bekannt, was das Vergleichen einfach macht.

- *qualityRating:* Gibt die Qualität des Services als Instanz der Klasse `QualityRating` an. Die möglichen Instanzen sind in der Abbildung nicht dargestellt, müssen aber definiert werden. Wenn Vergleiche möglich sein sollen, muss zusätzlich ein Property wie `isBetterQuality` für `QualityRating` definiert und entsprechend instanziert werden.

- *trustworthiness:* Gibt die Vertrauenswürdigkeit des Services an.

Es könnten beliebige weitere Eigenschaften hinzugefügt werden. Da diese nicht standardisiert sind, wird hier angenommen, dass diese unternehmensintern verwendet werden, um die Services, die in die unternehmensinterne Service-Registry aufgenommen werden, zu charakterisieren.

Man geht hier also davon aus, dass man vom Service-Provider lediglich die WSDL-Datei bekommt und daraus dann die OWL-S-Beschreibung generiert, siehe dazu Abschnitt 4.2.2. Dieser „minimalen„ OWL-S-Beschreibung fügt man dann Eigenschaften hinzu, wie die eben definierten, um sie so zu charakterisieren, dass man nach ihnen sinnvoll suchen bzw. sie auswählen kann. So würde man also ein „QualityRating“ oder „Trustworthiness“ nach eigener Einschätzung manuell den Services in der Registry hinzufügen.

Jeder OWL-S-Service wird typischerweise in einer eigenen Datei beschrieben. Diese enthält das Profile, Process und das Grounding. Das Profile enthält die Eingaben, Ausgaben und die Service-Kategorien. Die zusätzlichen Eigenschaften können in einer separaten Datei definiert werden und den Services zugeordnet werden, ohne die Service-Dateien anpassen zu müssen.

In der Tabelle 4.2 werden die möglichen Eigenschaften eines Service zur Bonitätsprüfung dargestellt:

Name	**Wert**
businessName	Schufa
serviceID	bonitaetspruefung-Schufa01
functionalClass	GenerateCreditReport
isDataMediator	false
price	0.15
qualityRating	QualityRatingExcellent
trustworthiness	HighTrustworthiness

Tabelle 4.2: Service-Properties

4.4 Suchanfrage

Im Folgenden wird der abstrakte Vorschlag aus dem Lösungsansatz (siehe Abschnitt 3.1) in ein konkretes Format für die Suchanfrage umgesetzt.

Die Suchanfrage wird in einem XML-Dokument definiert, das folgendes Format hat:

```
<query>
  <serviceSemantics>
    <semanticsURI uri='' />
  </serviceSemantics>

  <serviceCategories>
    <serviceCategory name='' value='' code='' /> *
  </serviceCategories>

  <serviceProperties>
    <serviceProperty name='' value='' comparator='' type='' /> *
  </serviceProperties>

  <servicePolicies>
    <wsp:Policy>
    ...
    </wsp:Policy>
  </servicePolicies>
</query>
```

Listing 4.1: Format der Suchanfrage

Sie wird entweder direkt ins Evaluate-Konstrukt eingebettet, dann wird sie im Attribut „substitute“ in einem CDATA-Abschnitt definiert oder alternativ über eine URL, die im „substitute“ steht, referenziert.

Die Suchanfrage besteht aus vier Elementtypen: der Semantikbeschreibung, die über eine URL referenziert wird, beliebig vieler Kategorisierungen, beliebig vieler Service-Properties und einer WS-Policy-Expression. Im Folgenden werden diese näher erläutert.

4.4.1 OWL-S-Anfragebeschreibung

Über das Element „semanticsURI“ bzw. die darin enthaltene URL wird eine OWL-S-Beschreibung referenziert. Diese wird wie im Abschnitt 4.2.2 beschrieben mit Hilfe des WSDL2OWL-S Converters aus der WSDL-Beschreibung des in der Invoke-Aktivität vordefinierten Partner Links generiert, beschreibt dadurch also die Semantik der Invoke-Aktivität, wie sie vom Prozessmodellierer durch die Auswahl einer bestimmten WSDL-Beschreibung implizit festgelegt wurde.

Die generierte OWL-S-Datei stellt nicht eine Servicebeschreibung dar, sondern eine Beschreibung der Anforderungen an einen Service, im Folgenden als Anfragebeschreibung bezeichnet. Was die beiden technisch unterscheidet, sind die im Grounding angegebenen XSLT-Transformationen (siehe weiter unten).

Die Anfragebeschreibung spezifiziert im *Profile* die Eingaben und Ausgaben des vordefinierten Service durch Referenzieren von Konzepten aus einer domänenspezifischen Ontologie. Das *Profile* wird verwendet, um nach dem Service zu suchen. Wie bereits erläutert ist das *Profile* sowohl für die Publikation seitens des Service-Providers als auch für die Spezifizierung der Anforderungen seitens des Service-Requesters gedacht (siehe Abschnitt 2.2.3). Die Aufgabe eines OWL-S-Matchmakers ist, die beiden *Profiles* zu vergleichen und zu entscheiden, ob der Service in Frage kommt. In der Anfragebeschreibung ist lediglich die Datensemantik spezifiziert, also der Teil der durch den WSDL2OWL-S Converter generiert wird. Die funktionale Semantik, Kategorisierungen und weitere Eigenschaften werden mit Hilfe der im Abschnitt 4.3 vorgestellten Erweiterungen spezifiziert.

Im *Grounding* wird spezifiziert wie die Abbildung von Ontologie-Konzepten auf WSDL-Datentypen, typischerweise in XML-Schema definiert, zu geschehen hat. Dies wurde bereits im Abschnitt 3.1.2 erläutert. Dabei wurde erklärt, dass die Abbildungen in der Anfragebeschreibung und der Servicebeschreibung genau andersrum zu spezifizieren sind. Die XSLT-Definitionen innerhalb der Anfragebeschreibung müssen also für die Eingabeparameter die Abbildung von XML-Schema-Datentypen auf Ontologie-Konzepte, für die Ausgabeparameter die Abbildung von Ontologie-Konzepten auf XML-Schema-Datentypen spezifizieren.

Zu beachten ist allgemein, dass das hier beschriebene Szenario, das *Profile* für die Suche und das *Grounding* für die Ausführung verwendet. Die Choreographie-Möglichkeiten des OWL-S-Process werden nicht verwendet, es wird immer nur ein AtomicProcess generiert.

4.4.2 Service-Kategorien

Im Element „serviceCategories“ kann mit Hilfe von beliebig vielen „serviceCategory“-Elementen die Kategorisierung spezifiziert werden. Dabei wird mit Hilfe des Namens die Kategorie ausgewählt und der Code spezifiziert einen konkreten Eintrag der Kategorie. Die Kategorien könnten zwar auch zur OWL-S-Anfragebeschreibung hinzugefügt werden, wurden aber zur besseren Lesbarkeit in die Suchanfrage direkt aufgenommen. Diese Kategorien würden typischerweise für eine Gelbe-Seiten-Suche im UDDI verwendet werden. Sind die Services in einer semantischen Wissensbasis (siehe Abschnitt 4.5.4) hinterlegt, so ist eine Suche anhand der funktionalen Klasse besser geeignet, da dabei auch Vererbungsbeziehungen berücksichtigt werden können.

4.4.3 Service-Properties

Service-Properties referenzieren Service-Eigenschaften, die wie im Abschnitt 4.3 aufgezeigt, als Erweiterung des *Profiles* definiert wurden.

Eine Service-Property besteht aus vier Elementen:

- *Name* referenziert eine OWL-Property eines *Profiles* und damit eine Eigenschaft des Services.

- *Value* ist ein Wert (in OWL-Sprache als „range“ bezeichnet) der referenzierten OWL-Property. Abhängig von der Definition der OWL-Property, kann der Wert einen numerischen Wert oder einen String als XML-Schema-Datentyp enthalten, oder auch einen OWL-Datentyp haben.

- *Comparator* dient zum Vergleichen zweier „Values“. Er wird bei der Suche verwendet, um zwischen Suchanfrage-Value und Service-Value zu vergleichen. Wenn der Vergleich „true“ ergibt, ist der Service bezüglich dieses Propertys passend. Im Fall von numerischen Werten werden =, <, >, <= und >= unterstützt. Ist der Wert des Propertys ein OWL-Typ muss in der Ontologie ein Vergleichsoperator definiert werden.

- *Type* gibt an, ob die Property zur Suche („discovery“) oder zur Auswahl („selection“) verwendet werden soll. Wenn der Typ „discovery“ verwendet wird bzw. das Attribut weggelassen wird, dann dient die Property zur Bestimmung passender Services. Im Fall von „selection“ werden passende Services nach dem Kriterium sortiert.

Beispiele

Die folgende Service-Property spezifiziert, dass der Preis des Services < 2.0 sein muss. Es wird hier angenommen, dass alle Services dieselbe Währung verwenden.

```
<serviceProperty
  name="http://localhost/ontology/AdditionalProps.owl#price"
  comparator="&lt;"
  value="2.0"
/>
```

Listing 4.2: Service-Property mit numerischem Wert

Die folgende Service-Property legt fest, dass die Qualität des Services besser als „gut“ sein muss.

```
<serviceProperty
  name="http://localhost/ontology/AdditionalProps.owl#qualityRating"
  comparator="http://localhost/ontology/AdditionalProps.owl#isBetterQuality"
  value="http://localhost/ontology/AdditionalProps.owl#QualityRatingGood"
/>
```

Listing 4.3: Service-Property mit einem OWL-Typ als Wert

Die folgende Property spezifiziert, dass die passenden Services nach Preis aufsteigend sortiert werden sollen.

```
<serviceProperty
  name="http://localhost/ontology/AdditionalProps.owl#price"
  comparator="&lt;"
  type="selection"
/>
```

Listing 4.4: Service-Property zur Sortierung

4.4.4 WS-Policy

Im „servicePolicies“-Element kann eine WS-Policy-Expression definiert werden. Diese wird dann ebenfalls in der Suche verwendet. Es findet dabei eine Policy-Intersection mit jedem potentiellen Service statt. Wenn die verglichenen Policies mindestens eine kompatible Alternative haben, wird der Service als passend angenommen.

4.4.5 Bedingungen in der Suchanfrage

Um den Kontrollfluss, wie im Abschnitt 4.2 gezeigt, vereinfachen zu können, muss es möglich sein, Übergangsbedingungen in der Suchanfrage zu spezifizieren. Dazu wurde einfach die Syntax von der BPEL-eigenen Switch-Aktivität übernommen. Ein Switch-Element innerhalb der Suchanfrage hat folgende Syntax:

```
<switch>
  <case condition="Boolesche-Bedingung">+
    switch|semanticsURI|serviceCategory|serviceProperty|wsp:Policy
  </case>
  <otherwise>?
    switch|semanticsURI|serviceCategory|serviceProperty|wsp:Policy
  </otherwise>
</switch>
```

Listing 4.5: Switch-Element in der Suchanfrage

Ein solches Switch-Element kann als Kindelement der vier Hauptelemente „serviceSemantics“, „serviceCategories“, „serviceProperties“ und „servicePolicies“ eingefügt werden. Switch-Elemente können geschachtelt werden.

Zur Formulierung von booleschen Bedingungen wird XSLT unterstützt. Die Bedingung kann dabei die BPEL-Funktionen „bpws:getVariableProperty“, „bpws:getLinkStatus“ und „bpws:getVariableData“ enthalten.

Das folgende Beispiel zeigt die Verwendung des Switch-Elements innerhalb des „serviceProperties“-Elements:

```
<serviceProperties>
  <switch>
    <case condition="bpws:getVariableData('request',
           'loanRequester','/loanRequester/type')='business'">
      <serviceProperty name="..."' value"'..."' />
    </case>
    <otherwise>
      <serviceProperty name="..."' value"'..."' />
    </otherwise>
  </switch>
</serviceProperties>
```

Listing 4.6: Verwendung des Switch-Elements

4.5 Erweiterung der BPEL-Engine

4.5.1 Überblick

Die Implementierung der Erweiterung der ActiveBPEL-Engine ist angelehnt an die im Abschnitt 3.2 beschriebene Broker-Architektur. Die folgende Abbildung zeigt die Architektur:

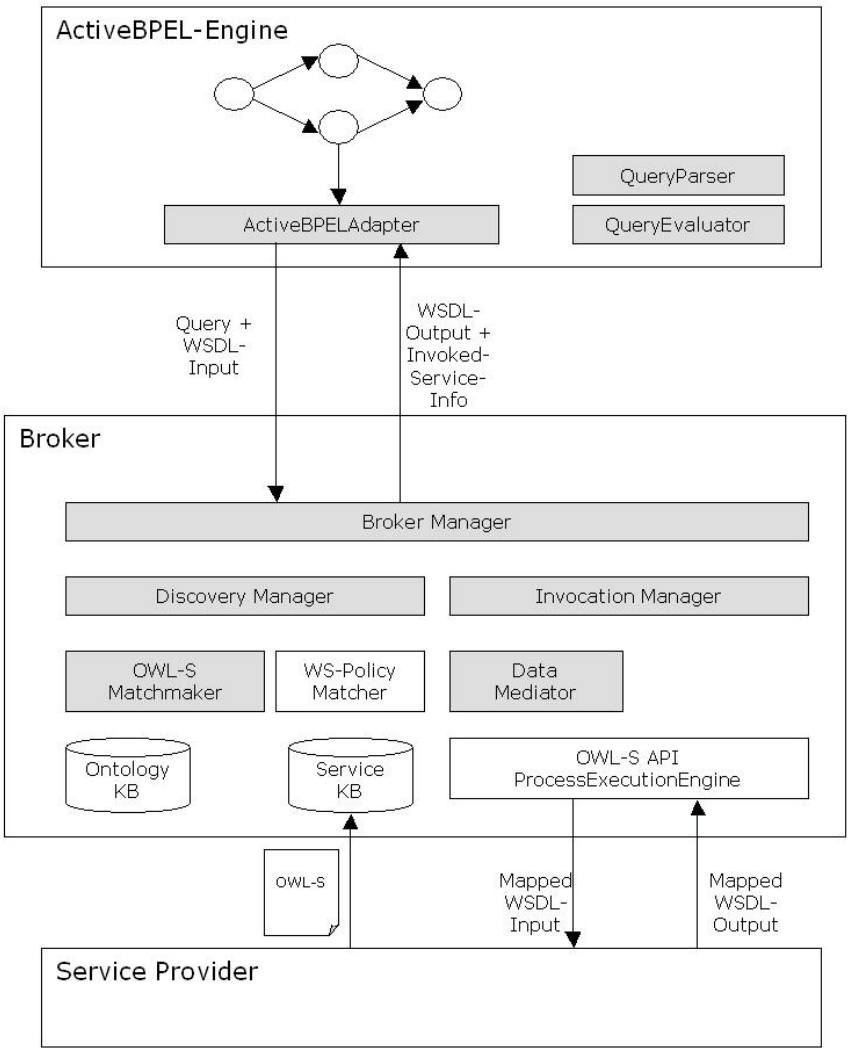

Abbildung 4.6: Architektur der ActiveBPEL-Erweiterung

Die grau untermalten Komponenten wurden komplett selbst implementiert. Die OWL-S API und die WS-Policy-Implementierung mussten angepasst werden, um den vorgestellten Kreditantragprozess ausführen zu können. So war der Policy-Intersection-Algorithmus in der WS-Policy-API noch nicht implementiert. Dieser wurde in einer Klasse `PolicyMatcher` umgesetzt. Größere Probleme gab es mit der OWL-S API. Diese unterstützte keine XML-Schema-ComplexTypes in Rückgabewerten und nicht alle XML-Schema-Datentypen, z.B.

keine Doubles. Dies ist erst aufgefallen, nachdem der Kreditantragsprozess etwas „realistischer“ modelliert worden war. Die API wurde soweit angepasst, damit das Beispiel sinnvoll ausgeführt werden konnte.
Im Folgenden werden die implementierten Komponenten im Detail beschrieben.

4.5.2 Adapter

Die notwendige Erweiterung der ActiveBPEL-Engine umfasste die Erweiterung des Prozessmetamodells um das Evaluate-Element und die Implementierung des Adapters.

Die Erweiterung des Prozessmetamodells um das Evaluate-Element und die anderen möglichen Strategien wurde im Rahmen einer anderen Diplomarbeit implementiert, die parallel zu dieser Arbeit durchgeführt wurde [Wut06]. Dabei wird das Evaluate-Element aus dem Prozessmodell ausgelesen, als Java-Klasse `Evaluate` zur Verfügung gestellt und kann zur Laufzeit vom Adapter ausgelesen werden.

Der Adapter wird durch die Java-Klasse `ActiveBPELAdapter` implementiert. Um von der ActiveBPEL-Engine aufgerufen werden zu können, implementiert sie das Interface `IAeInvokeHandler`. Wie der Name schon vermuten läßt, wird dieses von der Engine verwendet um alternative Implementierungen der Invoke-Aktivität zu unterstützen. Die Entscheidung, welche Implementierung tatsächlich verwendet wird, geschieht zur Laufzeit bei jedem Aufruf der Invoke-Aktivität durch die Klasse `AeInvokeHandlerFactory`. Diese wurde nun so erweitert, dass im Fall, dass die Invoke-Aktivität ein Evaluate-Element enthält, der Adapter aufgerufen wird, ansonsten die Standard-Implementierung. Das Interface `IAeInvokeHandler` definiert eine Methode `handleInvoke`, die den gesamten Prozessinstanzkontext in einem Parameter übergeben bekommt.

Nach Aufruf von `ActiveBPELAdapter` wird zunächst die Suchanfrage aus dem Evaluate-Element als XML-String oder über eine URL ausgelesen und an die Klasse `QueryParser` übergeben. Diese liest das XML-Dokument ein und erstellt einen Objekt-Graphen innerhalb von `Query`, der die Suchanfrage repräsentiert. Jeder Elementtyp der Suchanfrage wurde jeweils als eigene Klasse modelliert. Der `QueryParser` und die zugehörigen Klassen sind unabhängig von der BPEL-Engine-Implementierung. Die eingelesene `Query` wird nun an den `QueryEvaluator` übergeben, der die Bedingungen in der Suchanfrage auswertet. Die Klasse `QueryEvaluator` ist wiederum ActiveBPEL-spezifisch, da sie auf den Prozesseinstanzkontext zugreifen muss. Sie liefert als Ergebnis eine bereinigte `Query`, die keine Switch-Elemente und bpws:*-Funktionen mehr enthält.

Nun kann der Broker-Aufruf stattfinden. Dazu wird ein `VirtualInvocationRequest` erstellt, der die `Query` und in `WSDLMessageData` die Eingabedaten der aufzurufenden WSDL-Operation enthält. `WSDLMessageData` ist dabei eine Broker-spezifische Klasse und enthält eine `HashMap`, die Message-Part-Namen auf die entsprechenden Werte abbildet. Der Broker wird über die Methode `invokeVirtualService` aufgerufen und bekommt als Ergebnis eine `InvocationResponse`, aus der er die Ausgabedaten über `WSDLMessageData` ausliest und an die BPEL-Engine im erforderlichen Format zurückgibt.

4.5.3 Broker

Der Broker wird über ein Java-Interface `Broker` modelliert, das folgende Methoden enthält:

```
public interface Broker {

  InvocationResponse invokeVirtualService(
                        VirtualInvocationRequest request);

  DiscoveryResponse findServices(DiscoveryRequest request);

  InvocationResponse invokeService(InvocationRequest request);

}
```

Listing 4.7: Das Broker-Interface

Die Motivation für die Broker-Schnittstelle wurde im Abschnitt 3.2 erläutert. Es gibt also die Möglichkeit entweder die Methode `invokeVirtualService` aufzurufen oder die Methoden `findServices` und `invokeService` nacheinander. Der Broker ist in der Implementierung keine verteilte Komponente. Aus dem Broker-Java-Interface könnte aber leicht eine WSDL-Schnittstelle generiert werden.

Das Broker-Interface wird von der Klasse `BrokerManager` implementiert. Der `BrokerManager` implementiert die obengenannten Methoden, indem er entsprechend an den `DiscoveryManager` bzw. `InvocationManager` delegiert.

Der `DiscoveryManager` bekommt als Eingabe eine `Query` übergeben, ruft nacheinander den `OWLSMatchmaker` und `PolicyMatcher` auf und liefert eine Liste von `CandidateServices` zurück. Der `BrokerManager` liefert diese Liste in einem `DiscoveryResponse` zurück oder ruft den `InvocationManager` auf. Diesem übergibt er den ersten `CandidateService` aus der Liste, die OWL-S-Anfragebeschreibung aus der Suchanfrage (siehe Abschnitt 4.4) und den WSDL-Input. Der InvocationManager führt den Aufruf durch (siehe Abschnitt 4.5.5) und gibt den WSDL-Output an den `BrokerManager` zurück. Konnte der `InvocationManager` den Aufruf nicht durchführen, so wird er mit dem nächsten Service in der Liste aufgerufen usw.

Ein Service wird im Broker als Klasse `CandidateService` modelliert. Dieser kapselt im Wesentlichen einen `Service` der OWL-S API. Zusätzlich enthält ein `CandidateService` explizit die Attribute `portTypeName`, `operationName` und eine URL, die auf das zugehörige WSDL-Dokument zeigt. Diese sind zur Information für die BPEL-Engine bestimmt, da `CandidateServices` an die BPEL-Engine in den *Response-Messages zurückgeliefert werden.

4.5.4 OWL-S Matchmaker

Die Matchmaker-Komponente wird durch die Klasse `OWLSMatchmaker` implementiert. Sie bietet nach außen als Schnittstelle eine Methode `findServices`, die als Eingabe eine `Query`

bekommt und als Ausgabe eine Liste von `CandidateServices` liefert. Der Matchmaker berücksichtigt alle Suchkriterien aus der `Query` bis auf den WS-Policy-Ausdruck. WS-Policy-Matchmaking findet im darauf folgenden Schritt statt und bekommt als Eingabe die Ausgabe des OWL-S-Matchmakers. Sind in der `Query` ServiceProperty-Elemente vom Typ „Selection“ enthalten, so führt der `OWLSMatchmaker` auch die Sortierung anhand dieser Auswahlkriterien durch und liefert eine entsprechend sortierte Liste zurück.

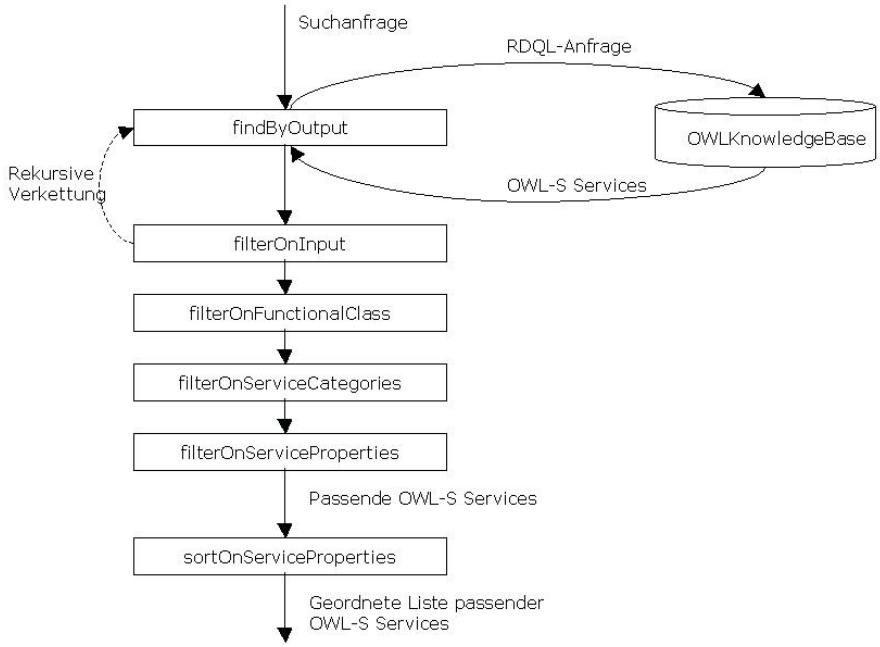

Abbildung 4.7: Matchmaking-Prozess

Wie das Matchmaking implementiert wurde, wird in Abbildung 4.7 dargestellt.

RDQL-Anfrage

Die publizierten OWL-S-Services und die domänenspezifischen Ontologien stehen in einer `OWLKnowledgeBase` zur Vefügung. `OWLKnowledgeBase` ist eine Klasse der OWL-S API und fungiert hier als Service-Registry. Sie bietet zum Zugriff auf die darin gespeicherten Ontologien sowohl Java-Methoden als auch eine RDQL-Schnittstelle an. RDQL ist eine SQL-ähnliche Anfragesprache für RDF-Modelle [Sea04].

Der Matchmaking-Prozess besteht wie in der Abbildung gezeigt aus mehreren Schritten. Zunächst findet eine grobe Vorauswahl statt, indem eine Suche nach den Ausgaben des gesuchten OWL-S-Service durchgeführt wird. Eine RDQL-Anfrage, die nach Services sucht, die als Ausgabe ein „CreditReport“ liefern, sieht folgendermaßen aus:

```
SELECT ?service
WHERE
 (?service rdf:type service:Service)
 (?service service:presents ?profile)
```

```
 (?profile profile:hasOutput ?output)
 (?output process:parameterType loan:CreditReport)
USING
 loan FOR <http://localhost:8080/ontology/loan.owl#>
 profile FOR <http://www.daml.org/services/owl-s/1.1/Profile.owl#>
 service FOR <http://www.daml.org/services/owl-s/1.1/Service.owl#>
```

Listing 4.8: RDQL-Anfrage

Der erste Schritt liefert als Ergebnis eine Liste von OWL-S-Services, deren Ausgaben mit denen in der Suchanfrage übereinstimmen. Als nächstes werden nun mehrere „Filter“ durchlaufen, um auch die restlichen Suchkriterien aus der Suchanfrage zu berücksichtigen. Die Filter verwenden dabei nicht mehr RDQL für Abfragen sondern arbeiten mit der objektorientierten Repräsentation eines OWL-S-Services. Diese wird von der OWL-S API als Klasse `Service` zur Verfügung gestellt. Diese enthält Methoden zum Zugriff auf die OWL-S-Service-Bestandteile Profile, Grounding und Process.

Service-Verkettung

Eine spezielle Beachtung verdient die Filterung an Eingaben. Dabei wird ein einfacher Algorithmus zur Service-Verkettung (service chaining) implementiert. Er orientiert sich an dem Vorschlag in [MM03] bzw. siehe dazu auch Abschnitt 2.2.3. Als Eingabe für den Input-Filter dienen Services deren Ausgaben mit denen in der Suchanfrage übereinstimmen. Nun wird für jeden Service überprüft, ob auch dessen Eingaben mit denen in der Suchanfrage übereinstimmen. Ist dies der Fall, geht der Service zur nächsten Filterstufe. Stimmen die Eingaben nicht überein, so wird der Service nicht sofort herausgefiltert, sondern es wird versucht rekursiv einen neuen Service zu finden, dessen Ausgaben mit den Eingaben (des alten Service) und dessen Eingaben mit den Eingaben in der Suchanfrage übereinstimmen. Werden dann diese beiden Services sequentiell ausgeführt, so haben sie von außen betrachtet die Eingaben und Ausgaben wie in der Suchanfrage spezifiziert. Da rekursiv gesucht wird, kann die Kette auch länger als zwei Services ausfallen. Es könnten bei der rekursiven Suche in jedem Schritt auch mehrere passende Services gefunden werden, dies wird der Einfachheit halber in der Implementierung unterbunden. Als Ergebnis ergibt sich also genau eine Kette von Services, die nacheinander ausgeführt die erwünschte Datensemantik erbringen.

Zu beachten ist, dass die Verkettung nur anhand der Datensemantik stattfindet. Es muss sichergestellt werden, dass die Kette auch die gesuchte funktionale Semantik erbringt. Die funktionale Semantik wird für einzelne Services in einem späteren Filterschritt anhand des ServicePropertys „functionalClass“ überprüft. Wenn ein Service in einer Kette aufgerufen wird, so wird zusätzlich in der Implementierung gefordert, dass alle Services in der Kette bis auf einen, nämlich denjenigen, der die eigentliche Funktionalität erbringt, Datamediators sind. Ein Datamediator ist in diesem Kontext ein Web Service, der nur eine Datentransformation durchführt. Wenn die Kette mehr als einen Service mit einer „functionalClass“ enthalten würde, der kein DataMediator ist, dann wäre die funktionale Semantik der Kette unklar. Das OWL-S-Profile wurde erweitert (siehe dazu Abschnitt 4.3), um Datamediators „kennzeichnen“ zu können, die Property „isDataMediator“ wird dabei entsprechend auf „true“ gesetzt.

Die Verkettung soll an einem einfachen Beispiel veranschaulicht werden. Wir nehmen an, wir wollen einen Service zur Bonitätsprüfung nutzen, dessen WSDL-Output ein spezielles deutsches Format des „CreditReport“ liefert, dessen Semantik mit dem vom BPEL-Prozess verwendeten nicht übereinstimmt. Die Transformation mit XSLT sei nicht möglich, es muss stattdessen ein Datamediator in Java implementiert werden. Wir annotieren die Ausgabe des Services mit einem Konzept „DECreditReport“. Der Datamediator bekommt als Eingabekonzept „DECreditReport“ und als Ausgabekonzept „CreditReport“. Wird der Matchmaking-Prozess durchgeführt, so wird zunächst der Datamediator gefunden, da er die gesuchte Ausgabe, nämlich „CreditReport“ besitzt. Als Eingabe besitzt er aber „DECreditReport“, gesucht ist stattdessen aber „LoanRequester“. Also wird eine rekursive Suche gestartet. Diese findet nun den Service zur Bonitätsprüfung mit der Ausgabe „DECreditReport“, die als Eingabe für den Datamediator dienen kann. Werden also der Service und anschließend der Datamediator aufgerufen, ergibt sich die gesuchte Datensemantik.

Die Verkettung, wie sie hier implementiert wurde, ist eine einfache Form der automatischen Service-Komposition. Sie kann dazu verwendet werden, um Datamediators automatisch aufzufinden und aufzurufen.

Filterung nach funktionaler Semantik

Die Filterung nach funktionaler Semantik geschieht auf Basis des „functionalClass“-Propertys, das als Erweiterung des OWL-S-Profiles definiert wurde (siehe Abschnitt 4.3). Der Wert dieses OWL-Propertys ist ein Konzept aus einer domänenspezifischen Ontologie, das eine bestimmte Funktionalität repräsentiert. In der Kreditantrag-Ontologie wurden beispielsweise eine OWL-Klasse „GenerateCreditReport“ und zwei Kindklassen „GenerateBusinessReport“ und „GeneratePrivateReport“ definiert.

Die Filterung nutzt die Vererbungsbeziehung der verwendeten Funktionsklassen. Dazu legen wir fest, dass wenn ein Service eine Vaterklasse im functionalClass-Property angibt, er dann auch die Funktionalität der unmittelbaren Kindklassen unterstützen muss. Diese Idee wurde im Zusammenhang mit dem OWL-S/UDDI Matchmaker (siehe Abschnitt 2.2.3) vorgestellt, dort allerdings für Eingaben und Ausgaben. D.h. wenn ein Unternehmen „GenerateCreditReport“ als funktionale Semantik angibt, dann muss es dieses sowohl für Unternehmen als auch für Privatpersonen, d.h. also die Funktionalität der unmittelbaren Kindklassen umfassen. Dies ist eine Vereinbarung, an die sich der Service-Provider beim Publizieren seines Service halten muss.

Bei der Filterung nach funktionaler Semantik, wobei in der Suchanfrage eine „Kindklasse“ spezifiziert ist, werden dann sowohl Services akzeptiert, die diese „Kindklasse“ als auch ihre unmittelbare Vaterklasse als „functionalClass“ spezifizieren. Wird also nach „GeneratePrivateReport“ gesucht, so werden auch Services der funktionalen Klasse „GenerateCreditReport“ als passend betrachtet.

Sortierung

Als letzter Schritt der Matchmaking-Prozedur findet eine Sortierung der gefundenen Services nach einem ServiceProperty-Element vom Typ „Selection“ statt. Ein solches

ServiceProperty-Element spezifiziert die Service-Eigenschaft und gibt den Vergleichsoperator an (siehe Abschnitt 4.4). Die Implementierung nutzt den Bubble-Sort-Algorithmus um die Services zu sortieren.

4.5.5 Serviceaufruf

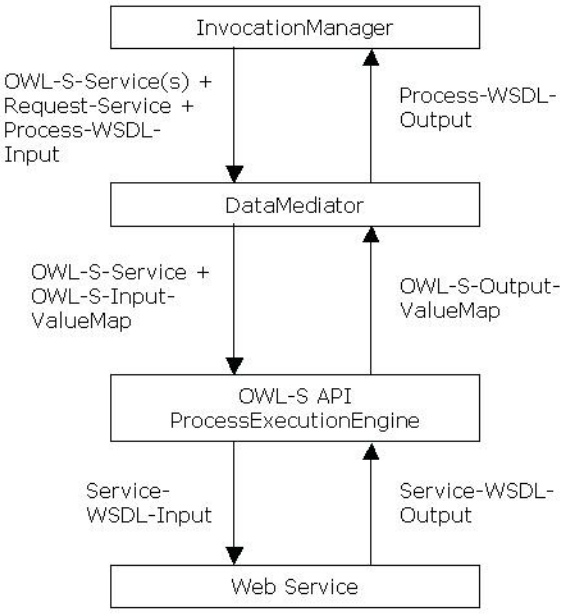

Abbildung 4.8: Serviceaufruf

In Abbildung 4.8 wird der Datenfluss beim Serviceaufruf dargestellt. Der `InvocationManager` wird vom `BrokerManager` aufgerufen. Dabei bekommt er als Eingabe einen OWL-S-Service oder eine Kette von OWL-S-Services, die aufgerufen werden sollen. Zusätzlich werden die WSDL-Input-Daten, die vom Prozess kommen, und die OWL-S-Anfragebeschreibung übergeben (siehe Abschnitt 4.4).

Der `DataMediator` (nicht zu verwechseln mit den Datamediators bei der Service-Verkettung) transformiert zunächst die WSDL-Input-Daten (des ersten Service in der Kette) durch eine XSLT-Tranformation, die in der OWL-S-Anfragebeschreibung spezifiziert ist, in eine `ValueMap`. Dies ist eine Klasse der OWL-S API und enthält eine Abbildung der OWL-S-Eingaben auf die Werte, die von den WSDL-Input-Daten stammen. Eine solche `ValueMap` dient als Eingabe für die `ProcessExecutionEngine`. Die `ProcessExecutionEngine` dient zum Ausführen von OWL-S-Processes. Die OWL-S API unterstützt auch *Composite-Processes*, in unserem Fall werden aber nur *AtomicProcesses* verwendet. Um einen solchen *AtomicProcess* auszuführen, wird das *Grounding* der OWL-S-Servicebeschreibung gelesen. Darin ist spezifiziert wie die OWL-S-Eingaben, die in der `ValueMap` modelliert sind, auf den WSDL-Input des aufzurufenden Web Services abzubilden sind. Diese Transformation wird von der `ProcessExecutionEngine` durchgeführt. Der Aufruf des Web Service wird dann mit Hilfe von Axis durchgeführt.

Die vom Web Service zurückgelieferten Outputs werden analog zu den Inputs in eine `ValueMap` der OWL-S-Ausgaben transformiert. Der `DataMediator` führt jetzt verschiedene Transformationen durch, abhängig davon ob es noch weitere Services in der Kette gibt, die aufgerufen werden sollen, oder nicht. War dies der letzte Service in der Kette so werden die Daten aus der `ValueMap` in WSDL-Outputs geschrieben, wie in der Anfragebeschreibung spezifiziert. Soll dagegen ein weiterer Service aufgerufen werden, so wird die `ValueMap` der Ausgaben des letzten Service-Aufrufs auf eine `ValueMap` der Eingaben für den nächsten Serviceaufruf abgebildet.

5 Zusammenfassung

Die Motivation der Arbeit bestand darin, eine Flexibilisierung von BPEL-Prozessen durch eine automatische Suche nach passenden Partner Services zu erreichen. Diese werden typischerweise bereits zur Deployment-Zeit des Prozesses festgelegt und können zur Laufzeit nicht gegen möglicherweise passendere Services durch eine Suche ausgetauscht werden. Eine mögliche Lösung dafür ist die Parametrisierung von Invoke-Aktivitäten, die es erlaubt zur Laufzeit, den zu verwendenden Partner Service unabhängig von den vorkonfigurierten PortType- und Operation-Namen auszuwählen. Um die „Parameter“ zu evaluieren, exisitieren mehrere Möglichkeiten, darunter eine manuelle Bestimmung durch einen Prozessadministrator. Die Aufgabe der Arbeit bestand aber darin, eine automatische Suche anhand einer Suchanfrage zu erlauben.

Dazu wurde zunächst festgestellt, dass eine automatische Suche unabhängig von WSDL-PortTypes nicht möglich ist, wenn nicht die Semantik der aufgerufenen WSDL-Operation in irgendeiner Form spezifiziert wird. Daraufhin wurden Ansätze untersucht, wie Web Services semantisch beschrieben werden können. Es wurden OWL-S, WSMO, WSDL-S und METEOR-S als SWS-Technologien vorgestellt. Keine dieser Technologien ist noch standardisiert. OWL-S und WSMO sind Top-down-Ansätze, die zunächst unabhängig von bestehenden WS-Technologien ein eigenes konzeptionelles Modell von einem „Semantic Web Service“ definieren, mit dem Ziel die Semantik möglichst sauber und vollständig zu erfassen. Sie geben dann aber in Form eines WSDL-Groundings an, wie ein Semantic Web Service über eine WSDL-Schnittstelle aufgerufen werden kann. Im Gegensatz dazu erweitern WSDL-S und METEOR-S die bestehenden Technologien in einem Bottom-Up-Ansatz. Die WSDL-Entitäten wie Input- und Output-Messages und Operationen werden dabei durch Annotation von Ontologiekonzepten erweitert.

Im darauf folgenden Lösungsansatz wurde auf Basis der vorgestellten SWS-Ansätze erörtert, wie Semantikbeschreibungen in die Suchanfrage aufgenommen werden können. Dabei wurde gezeigt, wie durch Annotation der WSDL-PortType-Entitäten durch Ontologiekonzepte eine Unabhängigkeit von PortType-Namen erreicht werden kann. Wenn die Input- und Output-Messages annotiert werden, muss zusätzlich ein Mapping angegeben werden, wie die XML-Datentypen der WSDL-Messages auf die XML-Serialisierung der Ontologiekonzepte abgebildet werden sollen. Dieses Mapping wird typischerweise in XSLT spezifiziert und wird benötigt, um die notwendigen Datentransformationen beim Aufruf eines Web Service durchzuführen.
Als Architektur für die BPEL-Engine-Erweiterung wurde ein Broker-basierter Ansatz gewählt. Die BPEL-Engine übergibt die Suchanfrage und die WSDL-Messagedaten mit Hilfe einer Adapter-Komponente an den Broker und dieser führt dann die semantische Suche

und den Aufruf durch. Der Broker führt dazu u.U. Datentransformationen durch bzw. kann sogar eine Service-Komposition erstellen, die die Anforderungen der Suchanfrage erfüllt. Es wurden in Anlehnung an WSMX zwei Schnittstellen des Brokers angedacht. Die erste führt die Suche und Aufruf des Partner Service in einem Schritt durch und führt somit zu einer Virtualisierung der Services aus Sicht des BPEL-Prozesses. Die zweite Schnittstelle erlaubt der BPEL-Engine nach einer Suche die Auswahl selbst vorzunehmen. Diese könnte beispielsweise manuell durch einen Prozessadministrator geschehen. Anschließend wird der ausgewählte Service wieder über den Broker aufgerufen, da u.U. Datentransformationen nötig sind.

Für die Umsetzung der vorgestellten Konzepte in einer Erweiterung der ActiveBPEL-Engine wurde OWL-S ausgewählt, da es zum Zeitpunkt der Implementierung die beste Software-Unterstützung hatte. Grundsätzlich würden aber auch die anderen vorgestellten SWS-Technologien in Frage kommen.
Die Suchanfrage enthält entsprechend eine OWL-S-Beschreibung, die deklarativ die Semantik der aufzurufenden WSDL-Operation beschreibt. Weiterhin können in der Suchanfrage standardisierte Kategorisierungssysteme wie NAICS, UNSPSC und ein WS-Policy-Ausdruck angegeben werden, um auch die technischen Anforderungen bei der Suche berücksichtigen zu können. Es wurde aufgezeigt, wie beliebige Service-Eigenschaften wie Preis und Qualität als Erweiterung der Service-Ontologie definiert werden können und wie nach ihnen gesucht werden kann. In die Suchanfrage können weiterhin Bedingungen aufgenommen werden, die den BPEL-Übergangsbedingungen entspechen.
Die Erweiterung der ActiveBPEL-Engine setzt den Broker-Ansatz um, der im Lösungsansatz beschrieben wurde. Besondere Beachtung verdient dabei der OWL-S-Matchmaker, der die semantische Suche durchführt. Dieser unterstützt datenbasierte Service-Verkettung und Suche anhand der Vererbungshierarchie in der Ontologie.

Durch die beschriebene Erweiterung der BPEL-Engine kann aus BPEL-Prozessen heraus eine dynamische Suche nach Partner Services unabhängig von ihrer PortType-Definition stattfinden. Es wird dabei angenommen, dass die Services eine entsprechende OWL-S-Beschreibung haben und ein OWL-Domänenmodell existiert, um die Services annotieren zu können. Da es wenige domänenspezifische Standards gibt und auch OWL-S kein Standard ist, können diese Annahmen für eine globale Suche nicht gemacht werden, d.h. auch wenn Services über ein öffentliches UDDI-Verzeichnis zur Verfügung gestellt würden, werden sie kaum dieselbe Ontologie verwenden und OWL-S-Beschreibungen haben. Realistischer ist ein Szenario, bei dem die vorgestellten Erweiterungen innerhalb eines großen Unternehmens mit möglicherweise wenigen ausgewählten Partnern genutzt werden. Diese könnten durchaus ein gemeinsames Domänenmodell erstellen, ihre WSDL-Schnittstellen entsprechend annotieren und damit die Flexibilität und Wiederverwendung ihrer Prozesse erhöhen.

Anhang A

Suchanfrage-Beispiel

Im Folgenden wird eine vollständige Suchanfrage dargestellt. Das Beispiel ist angelehnt an den Kreditantragsprozess, der im Abschnitt 4.2 vorgestellt wurde. Die Suchanfrage wird für eine Suche nach einem Service zur Bonitätsprüfung (LoanAssessor) spezifiziert.

A.1 Suchanfrage

Die Suchanfrage referenziert zunächst über eine URI eine OWL-S-Anfragebeschreibung (siehe Abschnitt A.2). Diese beschreibt die Datensemantik der vorkonfigurierten WSDL-Operation im BPEL-Prozessmodell. Anschließend wird eine Kategorisierung innerhalb der NAICS-Taxonomie angegeben.
Innerhalb des „serviceProperties“-Elements wird zunächst unterschieden, ob es sich beim Antragsteller um ein Unternehmen oder eine Privatperson handelt und entsprechend die funktionale Semantik spezifiziert. Es werden zwei weitere Bedingungen angegeben: Wenn ein Unternehmen einen Kreditbetrag > 500000 anfordert, so muss der Service sehr gute Qualität und Vetrauenswürdigkeit aufweisen. Wenn eine Privatperson einen Kreditbetrag < 50000 beantragt, so muss der Preis des Services < 0.5 sein. Die Services werden nach Preis sortiert. Abschließend wird eine WS-SecurityPolicy-Expression definiert.
In der Suchanfrage werden zwei OWL-Ontologien verwendet, das Domänenmodell (loan.owl) und die Erweiterung der Service-Eigenschaften (AdditionalServiceProperties.owl). Diese wurden in Abschnitten 4.2.1 bzw. 4.3 vorgestellt.

```
<query>

<serviceSemantics>
  <semanticsURI uri="loanassessorRequest.owls" />
</serviceSemantics>

<serviceCategories>
  <serviceCategory name="NAICS"
                   value="credit agency, credit bureau"
                   code="561450" />
</serviceCategories>

<serviceProperties>
```

```
 <switch>
  <case condition="bpws:getVariableData('creditReportRequest',
                   'loanRequestor' ,'/loanRequestor/type' )='business'">

   <serviceProperty
     name="AdditionalServiceProperties.owl#functionalClass"
     value="loan.owl#GenerateBusinessReport" />

   <switch>
    <case condition="bpws:getVariableData('loanApproverRequest',
                    'loanRequest' ,'/loanRequest/amount' )&gt;500000">
     <serviceProperty
       name="AdditionalServiceProperties.owl#qualityRating"
       comparator="AdditionalServiceProperties.owl#isHigherQualityRating"
       value="AdditionalServiceProperties.owl#GoodQualityRating" />
     <serviceProperty
       name="AdditionalServiceProperties.owl#trustworthiness"
       comparator="AdditionalServiceProperties.owl#isHigherTrustworthiness"
       value="AdditionalServiceProperties.owl#GoodTrustworthiness" />
    </case>
   </switch>
  </case>
  <otherwise>
   <serviceProperty
     name="AdditionalServiceProperties.owl#functionalClass"
     value="loan.owl#GeneratePrivateReport" />
   <switch>
    <case condition="bpws:getVariableData('loanApproverRequest',
                     'loanRequest' ,'/loanRequest/amount' )&lt;50000">
     <serviceProperty
       name="AdditionalServiceProperties.owl#price"
       comparator="&lt;"
       value="0.5" />
    </case>
   </switch>
  </otherwise>
 </switch>

 <serviceProperty
   name="AdditionalServiceProperties.owl#price"
   comparator="&lt;"
   type="selection" />

</serviceProperties>

<servicePolicies>
 <wsp:Policy
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
  <wsp:ExactlyOne>
   <wsp:All>
    <sp:SymmetricBinding>
     <wsp:Policy>
```

```
      <sp:ProtectionToken>
       <wsp:Policy>
        <sp:X509Token />
       </wsp:Policy>
      </sp:ProtectionToken>
      <sp:SignBeforeEncrypting />
      <sp:EncryptSignature />
     </wsp:Policy>
    </sp:SymmetricBinding>
    <sp:SignedParts>
     <sp:Body/>
     <sp:Header
       Namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"/>
    </sp:SignedParts>
    <sp:EncryptedParts><sp:Body/></sp:EncryptedParts>
   </wsp:All>
  </wsp:ExactlyOne>
 </wsp:Policy>
</servicePolicies>
</query>
```

A.2 OWL-S-Anfragebeschreibung

Die folgende OWL-S-Beschreibung spezifiziert die Datensemantik der vordefinierten WSDL-Operation im BPEL-Prozessmodell. Die Beschreibung kann mit Hilfe des WSDL2OWL-S Converters generiert werden (siehe Abschnitt 4.4.1).
Das OWL-S-Profile enthält jeweils eine Eingabe und Ausgabe. Die Eingabe verweist auf das Konzept `LoanRequester` des Domänenmodells (loan.owl). (Im Domänenmodell, das im Abschnitt 4.2.1 dargestellt wird, werden Unterklassen von `LoanRequester` modelliert. Diese werden hier nicht beachtet.) Die Ausgabe referenziert das Konzept `CreditReport`. Für die Suche sind diese Angaben bereits ausreichend.
Das WSDL-Grounding spezifiziert, wie die OWL-S-Eingaben und -Ausgaben auf die WSDL-Messages abzubilden sind. Im hier angezeigten Beispiel wird der WSDL-Part „loanRequestor“ der Input-Message „CreditReportRequest“ auf das Konzept `LoanRequester` abgebildet, der WSDL-Part „creditReport“ der OutputMessage ‘CreditReportResponse“ entsprechend auf das Konzept `CreditReport`. Es werden XSL-Transformationen referenziert, die die notwendigen Datentransformationen durchführen (siehe Abschnitt 4.4.1).

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
  xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
  xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
  xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
  xmlns:loanassessorWSDL="http://localhost:8080/queries/loanassessor.wsdl#"
  xmlns:loan="http://localhost:8080/ontology/loan.owl#"
```

```
  xml:base="http://localhost:8080/queries/loanassessorRequest.owls">

<service:Service rdf:ID="assessorService">
 <service:presents rdf:resource="#assessorProfile"/>
 <service:describedBy rdf:resource="#assessorProcess"/>
 <service:supports rdf:resource="#assessorGrounding"/>
</service:Service>

<profile:Profile rdf:ID="assessorProfile">
 <service:presentedBy rdf:resource="#assessorService"/>
 <profile:has_process rdf:resource="#assessorProcess"/>
 <profile:serviceName>Loan Assessor Query</profile:serviceName>
 <profile:textDescription>Creates Credit Report</profile:textDescription>
 <profile:hasInput rdf:resource="#LoanRequestor"/>
 <profile:hasOutput rdf:resource="#CreditReport"/>
</profile:Profile>

<process:AtomicProcess rdf:ID="assessorProcess">
 <service:describes rdf:resource="#assessorService"/>
 <process:hasInput rdf:resource="#LoanRequestor"/>
 <process:hasOutput rdf:resource="#CreditReport" />
</process:AtomicProcess>

<process:Input rdf:ID="LoanRequestor">
 <process:parameterType>loan:LoanRequester</process:parameterType>
</process:Input>

<process:Output rdf:ID="CreditReport">
 <process:parameterType>loan:CreditReport</process:parameterType>
</process:Output>

<grounding:WsdlGrounding rdf:ID="assessorGrounding">
 <service:supportedBy rdf:resource="#assessorService"/>
 <grounding:hasAtomicProcessGrounding
   rdf:resource="#assessorAtomicProcessGrounding"/>
</grounding:WsdlGrounding>

<grounding:WsdlAtomicProcessGrounding
  rdf:about="#assessorAtomicProcessGrounding">
 <grounding:owlsProcess rdf:resource="#assessorProcess"/>
 <grounding:wsdlDocument>
   http://localhost:8080/queries/loanassessor.wsdl
 </grounding:wsdlDocument>
 <grounding:wsdlOperation>
  <grounding:WsdlOperationRef>
   <grounding:portType>
     loanassessorWSDL:loanAssessorPT
   </grounding:portType>
   <grounding:operation>
     loanassessorWSDL:requestCreditReport
   </grounding:operation>
  </grounding:WsdlOperationRef>
 </grounding:wsdlOperation>
```

```
 <grounding:wsdlInputMessage>
   loanassessorWSDL:CreditReportRequest
 </grounding:wsdlInputMessage>

 <grounding:wsdlInput>
  <grounding:WsdlInputMessageMap>
   <grounding:wsdlMessagePart>
     loanassessorWSDL:loanRequestor
   </grounding:wsdlMessagePart>
   <grounding:owlsParameter rdf:resource="#LoanRequestor"/>
   <grounding:xsltTransformationURI>
     http://localhost:8080/queries/xslt/LoanRequestor.xsl
   </grounding:xsltTransformationURI>
  </grounding:WsdlInputMessageMap>
 </grounding:wsdlInput>

 <grounding:wsdlOutputMessage>
   loanassessorWSDL:CreditReportResponse
 </grounding:wsdlOutputMessage>

 <grounding:wsdlOutput>
  <grounding:WsdlOutputMessageMap>
   <grounding:wsdlMessagePart>
     loanassessorWSDL:creditReport
   </grounding:wsdlMessagePart>
   <grounding:owlsParameter rdf:resource="#CreditReport"/>
   <grounding:xsltTransformationURI>
      http://localhost:8080/queries/xslt/CreditReport.xsl
   </grounding:xsltTransformationURI>
   </grounding:xsltTransformationString>
  </grounding:WsdlOutputMessageMap>
 </grounding:wsdlOutput>

</grounding:WsdlAtomicProcessGrounding>

</rdf:RDF>
```

Literaturverzeichnis

[ABH+01] Anupriya Ankolekar, Mark H. Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne, Katia P. Sycara, and Honglei Zeng. DAML-S: Semantic Markup for Web Services. In *SWWS*, pages 411–430, 2001.

[ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services (BPEL4WS) 1.1, May 2003.

[ACKM03] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services*. Springer, November 2003.

[Act] ActiveBPEL. http://www.activebpel.org.

[AFM+05] Rama Akkiraju, Joel Farrell, John Miller, Meenakshi Nagarajan, Marc-Thomas Schmidt, Amit Sheth, and Kunal Verma. Web Service Semantics - WSDL-S. W3C Member Submission, November 2005.

[BBC+04a] Siddharth Bajaj, Don Box, Dave Chappell, Francisco Curbera, Glen Daniels, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Ashok Malhotra, Hiroshi Maruyama, Anthony Nadalin, Mark Nottingham, David Orchard, Hemma Prafullchandra, Claus von Riegen, Jeffrey Schlimmer, Chris Sharp, and John Shewchuk. Web Services Policy Framework (WS-Policy), 2004.

[BBC+04b] Siddharth Bajaj, Don Box, Dave Chappell, Francisco Curbera, Glen Daniels, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Ashok Malhotra, Hiroshi Maruyama, Anthony Nadalin, Mark Nottingham, David Orchard, Hemma Prafullchandra, Claus von Riegen, Jeffrey Schlimmer, Chris Sharp, and John Shewchuk. Web Services Policy Attachment (WS-PolicyAttachment), 2004.

[BCC+04] Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Marc Hadley, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, Steve Lucco, Steve Millet, Nirmal Mukhi, Mark Nottingham, David Orchard, John Shewchuk, Eugène Sindambiwe, Tony Storey, Sanjiva Weerawarana, and Steve Winkler. Web Services Addressing (WS-Addressing). W3C Member Submission, August 2004.

[BF03] S. Batres and C. Ferris. Web Services Reliable Messaging Policy Assertion (WS-RM Policy), February 2003.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[BvHH+04] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language. Reference. W3C Recommendation, February 2004.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Langauge (WSDL) 1.1, 2001.

[CMRW05] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Langauge (WSDL) Version 2.0 Part 1: Core Language, W3C Working Draft, August 2005.

[CSV04] Francisco Curbera, Amit P. Sheth, and Kunal Verma. Services Oriented Architecture and Semantic Web Processes. In *ICWS*, 2004.

[DLGHB+05] Giovanni Della-Libera, Martin Gudgin, Phillip Hallam-Baker, Maryann Hondo, Hans Granqvist, Chris Kaler, Hiroshi Maruyama, Michael McIntosh, Anthony Nadalin, Nataraj Nagaratnam, Rob Philpott, Hemma Prafullchandra, John Shewchuk, Doug Walter, and Riaz Zolfonoon. Web Services Security Policy Language (WS-SecurityPolicy), July 2005.

[FB02] Dieter Fensel and Christoph Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.

[GHM+03] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2. W3C Recommendation, June 2003.

[Haa04] Hugo Haas. Web Services Glossary. W3C Working Group Note, February 2004.

[HPS+04] 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. W3C Submission, May 2004.

[KHC+05] Dimka Karastoyanova, Alejandro Houspanossian, Mariano Cilia, Frank Leymann, and Alejandro Buchmann. Extending BPEL for Run Time Adaptability. In *9th International Enterprise Distributed Object Computing Conference (EDOC 2005)* , Enschede, The Netherlands, September 2005.

[KLB05] Dimka Karastoyanova, Frank Leymann, and Alejandro Buchmann. An Approach to Parameterizing Web Service Flows. In *3rd International Conference on Service-Oriented Computing* , December 2005.

[Ley01] Frank Leymann. Web Services Flow Language (WSFL). IBM Corporation, 2001.

[LR00] Frank Leymann and Dieter Roller. *Production workflow*. Prentice Hall PTR, 2000.

[MBH+04] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic Markup for Web Services. W3C Member Submission, November 2004.

[MBL+] David Martin, Mark Burstein, Ora Lassila, Massimo Paolucci, Terry Payne, and Sheila McIlraith. Describing Web Services using OWL-S and WSDL.

[MM03] Daniel J. Mandell and Sheila A. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In *International Semantic Web Conference*, pages 227–241, 2003.

[MPM+04] David L. Martin, Massimo Paolucci, Sheila A. McIlraith, Mark H. Burstein, Drew V. McDermott, Deborah L. McGuinness, Bijan Parsia, Terry R. Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia P. Sycara. Bringing Semantics to Web Services: The OWL-S Approach. In *SWSWPC*, pages 26–42, 2004.

[NL05] Eric Newcomer and Greg A. Lomow. *Understanding SOA with Web Services*. Addison-Wesley, 2005.

[Not04] OASIS Technical Note. Using WSDL in UDDI Registries v2.0.2, 2004.

[OWL] OWL-S API. MINDSWAP Group. http://www.mindswap.org/2004/owl-s/api/.

[Pel] Pellet. MINDSWAP Group. http://www.mindswap.org/2003/pellet/index.shtml.

[Pol] WS-Policy API. Apache Software Foundation. http://svn.apache.org/viewcvs.cgi/webservices/commons/trunk/policy/src/org/apache/ws/policy/.

[POSV04] Abhijit A. Patil, Swapna A. Oundhakar, Amit P. Sheth, and Kunal Verma. Meteor-s web service annotation framework. In *WWW*, pages 553–562, 2004.

[RMVS04] Preeda Rajasekaran, John A. Miller, Kunal Verma, and Amit P. Sheth. Enhancing Web Services Description and Discovery to Facilitate Composition. In *SWSWPC*, pages 55–68, 2004.

[Ros] RosettaNet Partner Interface Processes. www.rosettanet.org.

[SBF98] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. Knowledge engineering: Principles and methods. *Data Knowl. Eng.*, 25(1-2):161–197, 1998.

[Sea04] Andy Seaborne. RDQL - A Query Language for RDF, January 2004.

[SPS04] Naveen Srinivasan, Massimo Paolucci, and Katia P. Sycara. An Efficient Algorithm for OWL-S based Semantic Search in UDDI. In *SWSWPC*, pages 96–110, 2004.

[TC02] OASIS UDDI Specifications TC. UDDI Version 2 Specifications, 2002.

[Tha01] Satish Thatte. XLANG - Web Services for Business Process Design. Microsoft Corporation, 2001.

[Tom] Tomcat 5.0. Apache Software Foundation. http://tomcat.apache.org/.

[VAG+04] K. Verma, R. Akkiraju, R. Goodwin, P. Doshi, and J. Lee. On Accommodating Inter Service Dependencies in Web Process Flow Composition, 2004.

[VAG05] Kunal Verma, Rama Akkiraju, and Richard Goodwin. Semantic Matching of Web Service Policies, 2005.

[VGS+05] Kunal Verma, Karthik Gomadam, Amit P. Sheth, John A. Miller, and Zixin Wu. The METEOR-S Approach for Configuring and Executing Dynamic Web Processes, Technical Report. Technical report, LSDIS Lab, Computer Science Dept., UGA, June 2005.

[W3C04] W3C. Resource Description Framework (RDF), February 2004.

[W3C05] W3C. Web Service Modeling Ontology (WSMO) Submission, June 2005.

[WCL+05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, March 2005.

[Wut06] Daniel Wutke. Erweiterung einer Workflow-Engine zur Unterstützung von parametrisierten Web Service Flows. Diplomarbeit. Universität Stuttgart, 2006.

[ZMH05] Michal Zaremba, Matthew Moran, and Thomas Haselwanter. D13.4v0.2 WSMX Architecture, June 2005.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Stuttgart, den 17. Januar 2006 _____

(Branimir Wetzstein)