Technische Universität München
Fakultät für Informatik
Diplomarbeit in Informatik

Autonome Suche nach semantisch beschriebenen Web Services basierend auf ***OWLS***

Halgurt Mustafa Ali (halgurt@gmx.de)

Aufgabensteller: Prof. Michael Beetz PhD
Betreuer: Matthias Wimmer, Bernhard Kirchlechner
Abgabedatum: 15.03.2006

2. März 2006

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Garching den 24.02.2006
Halgurt Mustafa Ali

Abstract

Web Service ist eine Technologie, die eine plattformunabhängige und sprachneutrale Interaktion im Web ermöglicht. Derzeitige Standards wie WSDL und SOAP bieten nur syntaktische Interoperabilität und können nur beschränkt von Agenten bei der autonomen Suche, Komposition und Ausführung von Web Services benutzt werden. Dem Agenten oder dem Programmierer muss die Programmspezifikation bekannt sein, damit er einen Web Service nutzen kann.

Das Semantic Web bietet Standards zur Repräsentation von Wissen, so dass Informationen in einer maschineninterpretierbaren Form repräsentiert werden können. Semantic Web bietet Mechanismen zur Konzeptualisierung von Anwendungsdomänen. Informationen über eine Domäne werden als Konzepte, Attribute dieser Konzepte und Relationen zwischen diesen Konzepten repräsentiert. Ein semantisch beschriebener Web Service ist die Synthese dieser beiden Technologien, in der Informationen über Web Services wohldefiniert und maschinenverständlich in Form von Ontologien angegeben werden.

Die autonome Suche, Komposition und Ausführung von semantisch beschriebenen Web Services werden in dieser Arbeit behandelt. Die autonome Suche nach Web Services setzt eine Beschreibung dieser Services und der Anwendungsdomäne auf einer semantischen Ebene voraus. Agenten benötigen Semantiken nicht nur für die Suche, sondern auch für die Ausführung von Web Services. Agenten sollen in der Lage sein, autonom zu entscheiden, welche Eingabeparameter benötigt werden, um einen Service auszuführen und welche Ausgabeparameter zurückgeliefert werden sollen. Ein Agent soll ausgehend von deren Beschreibung in der Lage sein, Kompositionen autonom auszuführen. Die Ausführung von Kompositionen setzt eine autonome Dialogführung anhand der Kompositionsbeschreibung voraus. Diese Aspekte werden mittels einer Fallstudie vorgeführt.

Inhaltsverzeichnis

1 Motivation 6

2 Funktionalität von semantisch beschriebenen Web Services 8

2.1 Autonomous Service Discovery 10
2.2 Matching-Kriterien 14
2.3 Autonomous Service Composition 20
2.4 Autonomous Service Invocation 22

3 Verwendete Techniken 24

3.1 Agenten 24
3.2 Web Service-Technologie 25
3.2.1 *SOAP* (Simple Object Access Protocol) 28
3.2.2 WSDL (Web Services Definition Language) 30
3.2.3 UDDI (Universal Description, Discovery and Integration) . 35
3.3 *Semantic Web* 38
3.3.1 Einführung 38
3.3.2 Wissensrepräsentation und Schlussfolgerung 39
3.3.3 Web Ontology Language (OWL) 41
3.4 Semantisch beschriebene Web Services 44
3.4.1 ServiceProfile 47
3.4.2 Process Model 51
3.4.3 Service Grounding 57
3.5 Knowledge Interchange Format (KIF) 61
3.6 Inferenzmaschine (Reasoner) 63

4 Architektur & Algorithmen 67

4.1 Architektur 67
4.2 Algorithmus 70
4.3 Erfahrungen & Erkenntnisse 74

5 Seitenblick 76

6 Ausblick 79

7 Zusammenfassung 80

8 Screenshots 81

A Vorbedingung 86

B Nachbedingung 87

C Composite Process 88

D Grounding **91**

E WSDL-Grounding **94**

F Regeln zur Definition von Vor- und Nachbedingungen **96**

1 Motivation

Somewhere there is a service that does exactly what I want...

-Bob Sutor, IBM

Das WWW ist im Begriff, sich von einer statischen Informationsquelle zu einem höchst dynamischen Netzwerk zu entwickeln, in dem Ressourcen verteilt sind und Informationen auf Anfrage generiert werden. Um das Web besser ausnutzen zu können, werden gegenwärtig Business-Strategien wie B2B (Business-to-Business) entwickelt. Damit sind Organisationen in der Lage, Märkte schneller aufzusuchen, zu wechseln und Gewinne durch strategische Zusammenarbeit und Schließung von strategischen Partnerschaften zu optimieren [1]. Genauso wichtig ist das Web für wissenschaftliche Gemeinschaften, die ein Optimum bei der Ausnutzung von Ressourcen und dem Verteilen von Informationen benötigen. Aber auch für den normalen Benutzer spielt das Web eine immer größere Rolle. Bei diesen Unmengen an Informationen ist es oft schwierig, genau das zu finden, wonach man sucht. Ausserdem die sich schnell verändernde und unsichere Umgebung des WWW bedarf einer dynamischen und flexiblen Interoperabilität. Ein System, das mehrere Web Services verwendet, soll nicht wegen eines ausgefallenen Web Services seine Ausführung abbrechen müssen, sondern hat in der Lage zu sein, automatisch ähnliche Web Services zu finden und einzusetzen. Deswegen ist es wünschenswert, die Suche zu automatisieren, um Infrastrukturen autonom, flexibel und dynamisch zu gestalten.

Web Services können als Vermittler zwischen Client- und Server-Applikationen gesehen werden. Die Aufgaben eines Web Services sind somit klar definiert: Ein Web Service nimmt Anfragen seitens des Clients an, reicht diese Informationen an eine Serverkomponente durch, wartet ab, bis ein Ergebnis zurückgeliefert wird und leitet den Rückgabewert an den Client weiter. Web Services dienen der Kommunikation zwischen Rechnern in verteilten Systemen und im WWW. Eine allgemeine Anforderung an Web-Technologien ist die Interoperabilität zwischen unterschiedlichsten Systemen, die auf unterschiedlichsten Hardware-Typen laufen. Web Services bieten eine plattformunabhängige und sprachneutrale Interaktion im WWW.

Konventionelle Web-Service-Technologien bieten eine Syntax zur Beschreibung von Web Services und stellen logische Einheiten zur Angabe von verwendeten Protokollen und Schnittstellen zur Verfügung. Diese Beschreibungen sind von Maschinen nicht interpretierbar. Sie beschreiben Web Services auf einer syntaktischen Ebene, bieten aber keine Semantiken zur genaueren Spezifikation der Funktionalität von Web Services. Deswegen muss dem Benutzer - hier dem Programmierer des Agenten, der den Web Service benutzen will - die Spezifikation

des Web Services bekannt sein, damit er den gewünschten Web Service finden und ausführen kann. Es kann mehrere Services geben, die die gleiche Signatur besitzen und das selbe Protokoll unterstützen, aber unterschiedliche Berechnungen durchführen. Mit den Techniken von Web Services kann man zwischen diesen beiden Web Services nicht unterscheiden, denn beide Web Services erwarten die gleichen Eingabeparameter und liefern dieselben Ausgabeparameter zurück. Deswegen ist es schwer, automatisch den richtigen Web Service zu finden und auszuführen. Diese Problematik kann durch die Beschreibung der Web Services auf einer semantischen Ebene vermieden werden.

Für eine automatisierte Suche und Komposition muss ein Web Service Informationen über seine Signatur anbieten, die von Maschinen verstanden und interpretiert werden können. Semantisch beschriebener Web Service ist eine W3C-Initiative, die Standards bietet, um Informationen zu repräsentieren und zu verarbeiten, welche von Computern interpretiert werden können [5].

Die folgenden Funktionalitäten von semantisch beschriebenen Web Services werden in dieser Arbeit thematisiert:

1. Autonomous Service Discovery
   Die automatische Ermittlung von Web Services ist ein automatisierter Prozess zur Lokalisierung von Web Services, die bestimmte Funktionalitäten anbieten. Diese Funktionalitäten müssen Suchkriterien erfüllen, welche in einer Suchanfrage spezifiziert sind.

2. Matching-Kriterien
   Aus dem Verhältnis zwischen den Antworten, die ein Service tatsächlich liefert und den Antworten, die ein Benutzer auf seine Anfrage erwartet, ergeben sich unterschiedliche Matching-Kriterien. Dies ergibt sich aus der Klassifikation der Konzepte, die in der Anwendungsdomäne vorkommen.

3. Autonomous Service Composition
   Dieser Aspekt beinhaltet die automatisierte Selektion, Komposition und Interoperation von Web Services, um eine komplexe Aufgabe zu erfüllen.

4. Autonomous Service Invocation
   Service Invocation ist die automatische Ausführung von Web Services durch ein Computerprogramm oder durch einen Agenten, wenn eine deklarative Beschreibung der Web Services gegeben ist. Die Ausführung von Web Services kann als eine Menge von entfernten Prozeduraufrufen (RPC) aufgefasst werden.

2 Funktionalität von semantisch beschriebenen Web Services

Universal Description, Discovery, and Integration (UDDI) [4] ist die bekannteste Technologie zum Aufsuchen von Web Services. UDDI bietet Funktionalitäten, um Web Services zu suchen, die zu Schlagwörtern passen. Diese Schlagwörter werden mit Wörtern in der Beschreibung der Web Services verglichen. UDDI bietet weitere Funktionalitäten, um Web Services auf Basis von Beschreibungen im WSDL-Format zu suchen. Der Vergleich von gesuchten mit vorhandenen Web Services geschieht auf einer syntaktischen Ebene. Dies hat einige Nachteile: Erstens garantieren Syntaxvergleiche die gesuchte Funktionalität nicht. Zweitens können Agenten zwischen zwei Web Services mit derselben Signatur nicht unterscheiden(s. Beispiel 1 [1]), weil sie die angebotene, für Menschen gedachte, textuelle Beschreibung nicht verstehen können. Eine semantische Beschreibung der Anwendungsdomäne und der Web Services liegt nicht vor, deswegen können Agenten Web Services nicht automatisch suchen, ausführen, kombinieren oder ersetzen.

Agenten, die Web Services auf Basis von Web-Service-Techniken suchen, können zwischen Web Services mit derselben Signatur nicht unterscheiden. Ein Agent, der die Serviceparameter interpretiert, kann nicht deren Bedeutung ableiten, weil er die Parameter nicht lesen und verstehen kann, so wie es Menschen können. Alles, was der Agent versteht, ist, dass diese Parameter Eingabe- oder Ausgabeparameter vom Typ String, Integer oder andere primitive Datentypen sind. Ein menschlicher Entwickler, der einen Web Service in einem Clientprogramm nutzen will, muss die Syntax der Beschreibung lesen, sie interpretieren und dann das Clientprogramm entsprechend der Struktur des Web Services schreiben.[25].

Semantic Web ist eine Initiative, die maschineninterpretierbare Vokabularien zur Verfügung stellt, welche dann in Ontologien organisiert sind, um Domänen zu beschreiben. Diese Ontologien sind Konzeptualisierungen von Anwendungsdomänen. Semantic Web bietet die Möglichkeit, Web Services selber als Domäne zu konzeptualisieren und in Ontologien zu organisieren. Semantisch beschriebene Web Services sind Web Services, die auf Basis dieser Ontologien beschrieben sind. Agenten können diese Beschreibung benutzen, um die Merkmale der semantisch beschriebenen Web Services zu interpretieren. Autonome Agenten analysieren die Servicebeschreibungen und leiten daraus ab, ob der Web Service die gewünschten Funktionalitäten bietet, die in einer Suchanfrage spezifiziert sind [19].

[1] Die Variablen werden mit ? versehen,?teamNAme bedeutet hier: teamName ist eine Variable.

Beispiel 1:
Es wird ein Web Service gesucht, der als Eingabe einen Parameter vom Typ String bekommt und als Ausgabe einen Parameter vom Typ String zurückliefert. Offensichtlich werden die ersten beiden Web Services zurückgeliefert, denn beide erfüllen die Suchkriterien, obwohl sie unterschiedliche Berechnungen durchführen.

Service 1:
Dieser Web Service bekommt einen Parameter vom Typ String als Eingabe, der den Namen eines Vereins kennzeichnet und liefert als Ausgabe den Namen, das Alter aller Spieler dieses Vereins und die Anzahl der von den jeweiligen Spielern geschossenen Tore zurück.

Input:

$$String(?teamName)$$

Output:

$$String(?playerName) \wedge int(?leagueGoals) \wedge int(?age)$$

Service 2:
Dieser Web Service liefert alle Spieler eines Vereins zurück. Eingabe ist der Vereinsname als Stringwert, Ausgaben sind die Spielernamen und das Alter des jeweiligen Spielers.

Input:

$$String(?teamName)$$

Output:

$$String(?playerName) \wedge int(?age)$$

Service 3:
Dieser Web Service liefert die Anzahl der Tore, die ein Spieler in der Liga geschossen hat. Eingabe ist der Name eines Spielers als Stringwert, Ausgabe ist die Anzahl der Ligatore, die von diesem Spieler geschossen wurden (als Integerwert).

Input:

$$String(?playerName)$$

Output:

$$int(?leagueGoals)$$

Diese Ontologien sind als globale Ontologien zu verwenden, die beiden Seiten, sowohl dem Requester (Benutzer) als auch dem Provider (Anbieter), bekannt sind.

In diesen Ontologien werden die Konzepte der Anwendungsdomänen festgelegt. Es wird außerdem festgelegt, welche Attribute diese Konzepte besitzen. Man unterscheidet zwischen zwei Arten von Attributen: Manche Attribute sind primitive Datentypen, welche Eigenschaften eines Konzepts darstellen. Andere Attribute drücken Relationen und Beziehungen zwischen Konzepten aus (Beispiel 2). Der Provider benutzt Konzepte und Attribute dieser Domänen, um seine Web Services zu beschreiben. Der Requester sucht Web Services und benutzt die Beschreibung dieser Domänen, um genau zu spezifizieren, was er sucht. Bei konventionellen Web Services liegt weder eine Beschreibung der Anwendungsdomäne noch die Beschreibung der Web Service-Domäne vor. Die Spezifikation muss dem Requester bekannt sein, damit er die Web Services suchen und nutzen kann.

Beispiel 2:
Dieses Beispiel stellt einige Konzepte und Attribute der Fußballdomäne dar. Die Domänenontologie enthält Klassen und Attribute dieser Klassen. Die Klasse Spieler besitzt z.B. die folgenden Attribute:

- age (diese Property ist vom Typ xsd:int, welche das Alter eines Spielers darstellt).
- lastName (Nachname eines Spielers, Typ xsd:string)
- firstName (Vorname eines Spielers, Typ xsd:string)
- leagueGoals (Anzahl der Ligatore, die von diesem Spieler geschossen wurden, Typ xsd:int)

Eine weitere Klasse ist Team mit der Eigenschaft teamName vom Typ xsd:string. Abbildung 1 1 zeigt die erwähnten Klassen. Die gesamten Konzepte sind auf Abbildung 26 dargestellt

Nachfolgend werden die Funktionalitäten von semantisch beschriebenen Web Services vorgestellt, die in Kapitel 1 erwähnt wurden:

2.1 Autonomous Service Discovery

Transaktionen zwischen Web Services involvieren üblicherweise drei Parteien:

1. Provider, der Web Services zu bieten hat.
2. Requester, der Funktionalitäten benötigt, kann ein Agent sein
3. Mediator, der zwischen Client und Server vermittelt [25].

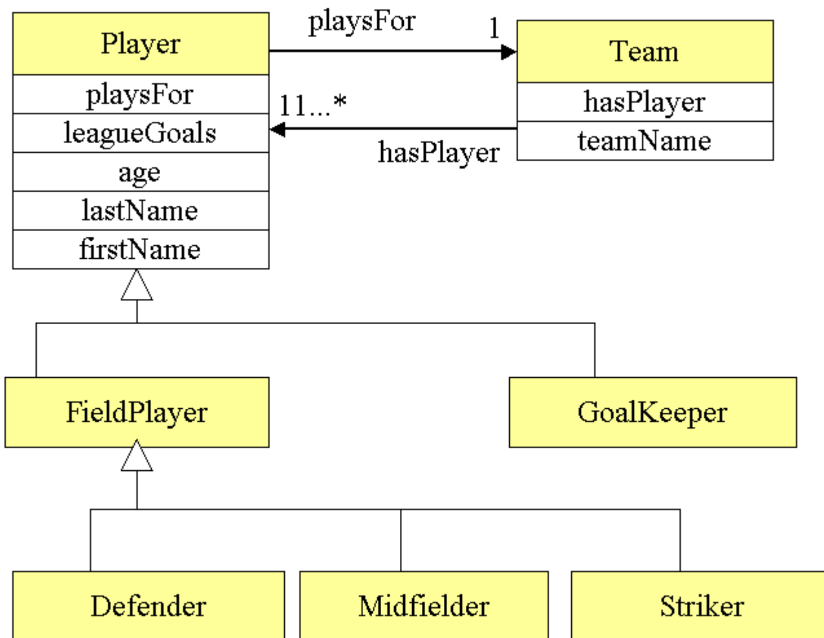


Abbildung 1: Einige Konzepte der Fußballdomäne

Abbildung 2 zeigt ein typisches Szenario mit diesen drei Parteien [20], das nachfolgend schrittweise erklärt wird. Service Discovery ist der Dienst eines Mediators, der über eine Liste ihm bekannter Web Services verfügt. Der Mediator hat eine Schnittstelle mit Funktionalitäten, um Web Services zu finden, die eine Menge von Suchkriterien erfüllen. Er wird von einem Requester benutzt, um Web Services zu suchen, die von diesem benötigte Funktionalitäten anbieten.

Der erste Schritt ist die Veröffentlichung von Web Services durch Provider bei einem Mediator. Informationen über einen Web Service werden in irgendeiner Form so lange gespeichert, bis der Provider sich entscheidet, diesen Web Service nicht mehr anzubieten. Auf der Requesterseite muss eine Beschreibung der benötigten Funktionalität kreiert werden, die dem Mediator als Anfrage geschickt wird. Abbildung 3 zeigt, wie Ontologien benutzt werden, um angebotene und gesuchte Web Services zu beschreiben.
Gesuchte und veröffentlichte Funktionalitäten müssen in einer maschineninterpretierbaren Form beschrieben werden, damit sie miteinander verglichen werden können. Der Mediator vergleicht gesuchte mit veröffentlichten Funktionalitäten und entscheidet, ob sie genügend ähnlich sind. Genügend ähnlich bedeutet, zu welchem Grad die angebotene Funktionalität der gesuchten entspricht (Beispiel 3).

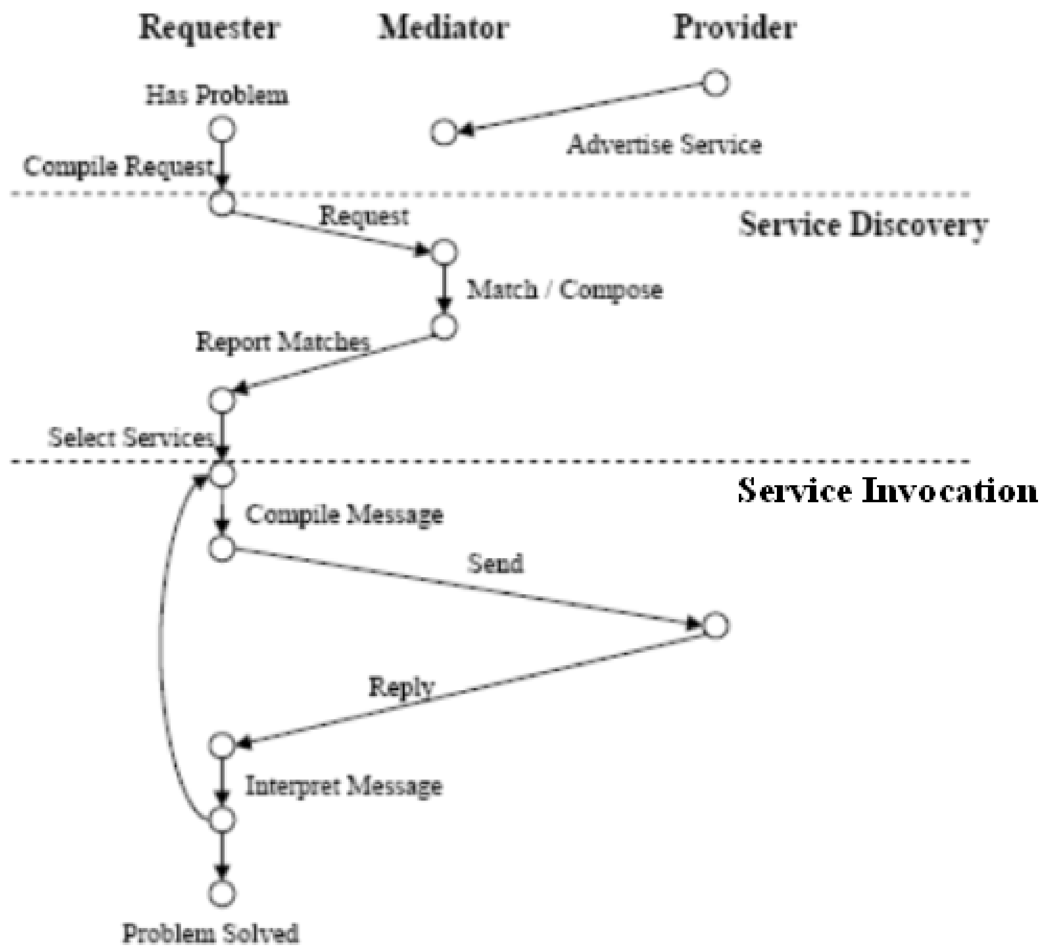


Abbildung 2: Ërvice Discoveryünd Interaktion mit Beteiligung von Provider, Requester und Mediator

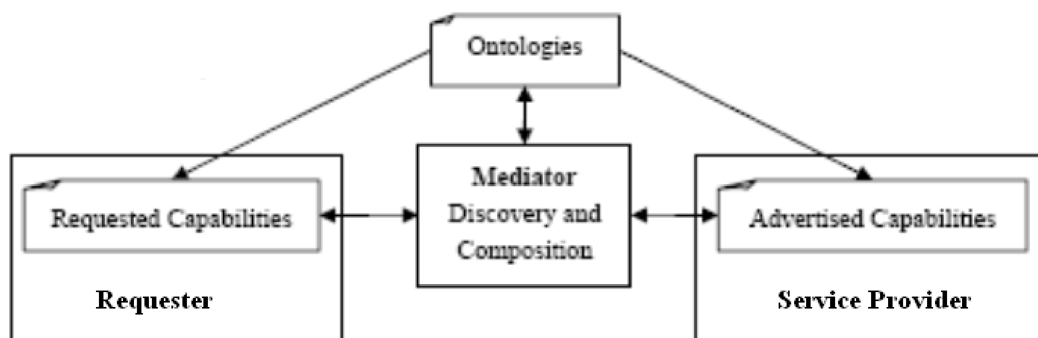


Abbildung 3: Nutzung von Ontologien zur Beschreibung von Web Services durch Provider, Requester und Mediator

Beispiel 3:
Es wird ein Web Service gesucht, der alle Stürmer eines Vereins, z.B. des FC Bayern München, zurückliefert. Eine mögliche Suchanfrage wäre:
Input:

$$String(?teamName)$$

Output:

$$String(?playerName)$$

Servicelokale Parameter, die weder Inputs noch Outputs sind, werden benötigt, um die Eingabe- und Ausgabeparameter entsprechend der Domänenontologie zu verbinden:
Local:

$$Team(?team) \wedge Player(?player)$$

Die Beziehungen zwischen den lokalen Parametern und den Ausgabeparametern werden als Vorbedingungen beschrieben:
Precondition:

$$teamName(?team, ?teamName)$$

$$playsFor(?player, ?team)$$

Die Verhältnisse zwischen lokalen Parametern und Ausgabeparametern werden als Nachbedingungen spezifiziert:
Effect:

$$lastName(?player, ?playerName)$$

Jeder Web Service hat entweder keinen oder mehrere Eingabeparameter, keinen oder mehrere Ausgabeparameter sowie keinen oder mehrere lokale Parameter. Mit Hilfe der lokalen Parameter werden Beziehungen und Relationen zwischen den Eingabe- und Ausgabeparametern entsprechend der Anwendungsdomäne spezifiziert. Diese Beziehungen und Relationen drücken die Vor- und Nachbedingungen aus, die vor bzw. nach der Serviceausführung gelten müssen. Die lokalen Parameter und die Verhältnisse zwischen den lokalen Parametern und den Eingabeparametern werden als Vorbedingungen spezifiziert. Die Verhältnisse zwischen Eingabe- und Ausgabeparametern werden als Nachbedingungen spezifiziert. Dies ermöglicht die Beschreibung der Web Services auf einer semantischen Ebene (Bei-

spiel 3 [2]).

Da konventionell die Suche nach Web Services auf Schlagwörtern basiert, werden, wenn richtige Web Services gefunden werden, nur exakte Web Services zurückgeliefert. Durch die Beschreibung der Anwendungsdomäne entstehen Hierarchien von Konzepten. Diese Hierarchien ermöglichen es, Web Services zu finden, die nur Teilantworten auf die Suchanfrage bieten. Dies führt zur nächsten Funktionalität von semantisch beschriebenen Web Services.

2.2 Matching-Kriterien

Semantisch beschriebene Web Services bieten die Möglichkeit, Web Services zu finden, die Teilantworten auf eine Anfrage geben. Aus der Konzeptualisierung der Anwendungsdomäne in Ontologien entstehen Hierarchien von Konzepten. Diese Hierarchien kann man wie Klassen-Konzepte in objektorientierten Programmiersprachen ansehen (Beispiel 4).

Beispiel 4:
Betrachten wir folgende Anfrage:
Alle Stürmer eines bestimmten Vereins.
In der Domänenontologie ist Stürmer als Unterklasse von Spieler definiert, d.h. Stürmer ist eine echte Teilmenge von Spieler. Weiter nehmen wir an, es gibt einen Service, der alle Spieler eines Vereins zurückliefert. Die Suchmenge *Stürmer*, ist eine Untermenge der Anzeigemenge *Spieler*.
Bei semantisch beschriebenen Web Services ist diese Art von Matching aufgrund der Stärke von Semantic Web möglich. Bei konventionellen Web Services funktioniert das nicht. Obwohl der Service eine Teilantwort liefert, würde hier kein Service gematcht.

Jeder Web Service liefert nach seiner Ausführung eine Menge an Einträgen zurück, diese Menge wird als ***Anzeigemenge*** bezeichnet. Ein Requester sucht nach Web Services, die eine von ihm erwartete Menge von Einträgen zurückliefern. Diese Menge wird als ***Suchmenge*** bezeichnet. Oft bekommt der Requester auf seine Anfrage nicht Web Services, die die exakte Menge an erwarteten Einträgen zurückliefern, sondern nur Teile davon. Die Schnittmenge der ***Anzeigemenge*** und der ***Suchmenge*** kann variieren. Unten sind verschiedene Möglichkeiten aufgelistet, wie sich zwei Mengen schneiden können:
Suchmenge Θ und Anzeigemenge Ψ:

$$\Theta \equiv \Psi \tag{1}$$

[2]Ein einstelliges Prädikat bedeutet, dass die Variable ?teamName vom Typ String ist

$$\Theta \subset \Psi \tag{2}$$

$$\Psi \subset \Theta \tag{3}$$

$$\Theta \bigcap \Psi = \phi \tag{4}$$

$$\Theta \bigcap \Psi \neq \phi \wedge \Theta \bigcap \Psi \neq \Theta \wedge \Theta \bigcap \Psi \neq \Psi \tag{5}$$

In der Mengenlehre bezeichnet man die erste Gleichung als eine Äquivalenz. Beide Mengen enthalten genau dieselben Einträge, d.h. man findet einen Web Service, der genau die Anfrage beantworten kann[3]. Wenn die Anzeigemenge eine echte Teilmenge der Suchmenge ist, bekommt der Requester einige gesuchte Einträge zurück, aber nicht alle.[4]. Gleichung Nummer zwei entspricht diesem Fall: Der Benutzer bekommt nur richtige Einträge zurück, aber nicht alle. Wenn die Suchmenge eine echte Teilmenge der Anzeigemenge ist, bekommt der Requester eine Menge von Einträgen, die alle seine gesuchten Einträge enthält, aber auch zusätzliche Einträge[5]. Gleichung Nummer drei entspricht diesem Fall: Der Requester muss zusätzlichen Aufwand betreiben, um die richtigen Einträge aus der Menge zu entnehmen (s. Beispiel 4). Die vierte Gleichung sagt aus, dass beide Mengen sich nicht schneiden. Dieser Fall tritt ein, wenn keiner der Web Services die Anfrage in irgendeiner Weise beantworten kann[6]. Die fünfte Gleichung entspricht einer echten Schnittmenge. Dieser Fall ist ein besonderer und schwieriger Fall und kann in einen der anderen Fälle enden[9]. Hier kann der Requester nicht wissen, ob Einträge der **Suchmenge** in der **Anzeigemenge** enthalten sind. Sind keine Einträge enthalten, würde dies Gleichung Nummer vier entsprechen.

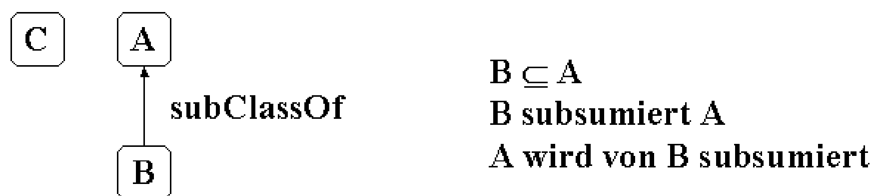
Die letzte Gleichung wurde von „Li, Lei und Horrocks, Ian" aufgestellt. Diese Art Matching ist stärker als Gleichung Nummer vier und schwächer als Gleichung Nummer drei. Sie wird vor allem bei Kompositionen betrachtet, aber nicht ausschließlich, denn dieser Fall kann bei Werteeinschränkungen eintreten, wie wir später sehen werden. Abbildung 4 stellt einige Matching-Kriterien dar:
Bis jetzt wurden Typeinschränkungen behandelt, semantisch beschriebene Web Services bieten aber auch die Möglichkeit, Werteeinschränkungen auszudrücken. Dies geschieht mittels Definition von Funktionen, die den Wertebereich von Datentypen einschränken. Diese Funktionen beeinflussen ebenfalls die Matching-Kriterien (Beispiel 5).

[3] In der englischsprachigen Literatur wird diese Menge oft als *exactMatch* bezeichnet

[4] In der englischsprachigen Literatur wird diese Menge oft als *plugin* bezeichnet

[5] In der englischsprachigen Literatur wird diese Menge oft als *subsume* bezeichnet

[6] Die ersten vier Klassifikationen sind von „Massimo Paolucci, Takahiro Kawamura, Terry R. Payne und Katia Sycara" identifiziert



Gesuchte Service	Verfügbare Service	Match-Typ
Input1: A	Input1: A	Exact
Input2: A	Input2: B	PlugIn
Input3: B	Input3: A	Subsume
Input4: A	Input: C	Fail
Output1: A	Output1: A	Exact
Output2: B	Output2: A	PlugIn
Output3: A	Output3: B	Subsume
Output4: A	Output4: C	Fail

Abbildung 4: Verschiedene Matching-Kriterien

Beispiel 5:
Betrachten wir folgende Anfrage:
Alle Stürmer eines bestimmten Vereins, die jünger als 25 Jahre sind
Bisher hatten wir Typeinschränkungen. Bei dieser Anfrage benötigen wir aber auch Werteeinschränkungen. Mit Hilfe der Domänenontologie können wir dies nicht erreichen. Wie man sich leicht denken kann, brauchen wir hier Funktionen, welche die Wertebereiche von Variablen einschränken. Hierfür gibt es eine Ontologie, die Funktionen und Einschränkungen auf Wertebereichen beschreibt; es handelt sich um die SWRLB [8] (Semantic Web Rule Language Builtins)
Eine der Funktionen ist lessThan(arg1, arg2). In unserem Beispiel würde es heißen:
lessThan(?age, 25)
Die fertige Anfrage würde dann so aussehen:
Input:

$$String(?teamName)$$

Output:

$$String(?playerName)$$

Local:

$$Team(?team) \wedge Player(?player)$$

Precondition:

$$teamName(?team, ?teamName) \wedge playsFor(?player, ?team)$$

Effect:

$$lastName(?player, ?playerName) \wedge Type(?player, Striker) \wedge lessThan(?age, 25)$$

Es bleibt zu erwähnen, dass Funktionen auch den Schnitt der Such- und Anzeigemenge beeinflussen. Stellen wir uns vor, wir wollen alle Spieler eines bestimmten Vereins haben, die jünger als 20 Jahre sind. Es gibt aber einen Service, der alle Spieler eines Vereins liefert, die jünger als 25 Jahre sind. Die Suchmenge ist in diesem Fall eine Untermenge der Anzeigemenge, also handelt es sich um ein Plugin. Sollten wir alle Stürmer eines Vereins suchen, die jünger als 20 Jahre sind, dann erhalten wir das Bild in Abbildung 5. Der karierte Kreis kennzeichnet diese Menge.

Ein schwierigerer Fall läge vor, wenn der Web Service wie bisher wäre und wir diese Anfrage hätten:
Alle Stürmer eines bestimmten Vereins, die älter als 20 Jahre sind
Abbildung 6 veranschaulicht diesen Fall. Die schwarze Schnittmenge entspricht Gleichung Nummer 5. Bei dieser Art Matching kann das Bild auf Abbildung 7 entstehen. In diesem Fall macht es überhaupt keinen Sinn, den Web Service zurückzugeben, denn dies entspricht keinem Match wie in Gleichung Nummer 4. Da man nicht von vornherein weiß, wie die Schnittmenge ausschaut, ist es schwer zu entscheiden, ob man diesen Service als Antwort betrachten soll oder nicht.

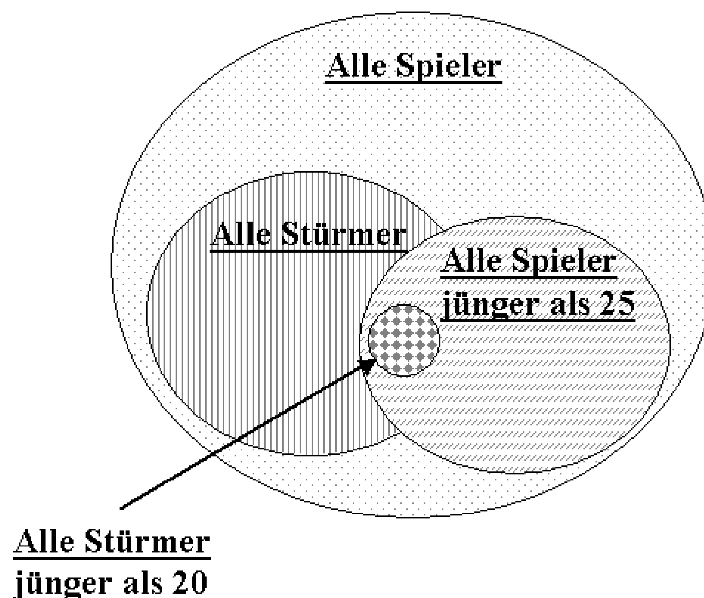


Abbildung 5: Alle Stürmer eines Vereins, die jünger als 20 Jahre alt sind

Bis jetzt haben wir nur Fälle betrachtet, in denen es einen Web Service gab, der die Anfrage entsprechend eines Matching-Kriteriums beantworten konnte. Aber in der Praxis gibt es Fälle, in denen kein Service alleine eine Anfrage beantworten kann, sondern nur eine Kombination von zwei oder mehr Web Services. Im nächsten Unterabschnitt betrachten wir solche Fälle.

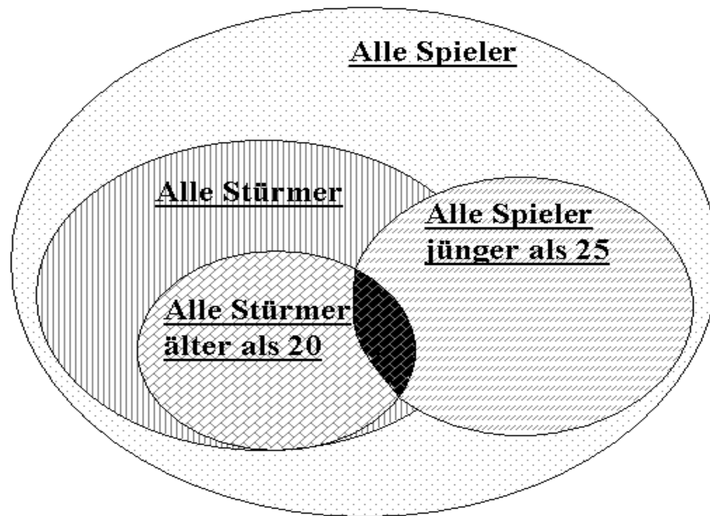


Abbildung 6: Alle Stürmer eines Vereins, die älter als 20 Jahre sind

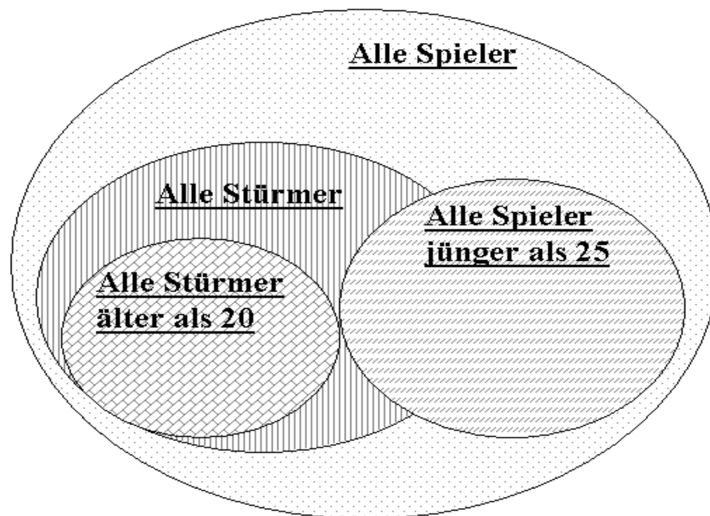


Abbildung 7: Schnittmenge zwischen alle Stürmer älter als 20 und alle Spieler jünger als 25 ist null.

2.3 Autonomous Service Composition

Eine Schlüsselfunktionalität ist die automatisierte Komposition, ausgehend vom Initialzustand bis hin zum Endzustand einer von einem Requester spezifizierten Anfrage. Die Zustände sind semantische Beschreibungen von Informationen. Unter Komposition versteht man eine Menge von Web Services, die in irgendeiner Weise kombiniert sind. Eine Komposition bildet eine Informationstransformation ausgehend von einem Initialzustand bis zu einem Endzustand. Die Konstruktion solcher Kompositionen wird durch die Abhängigkeiten zwischen Eingabeparametern, Ausgabeparametern und den Vor- und Nachbedingungen erreicht. Ein Web Service kann z.B. Eingabeparameter haben, die den Eingabeparametern einer Anfrage entsprechen, kann aber hingegen auch Ausgabeparameter haben, die mit den Eingabeparametern anderer Web Services übereinstimmen. Oder umgekehrt, ein Web Service kann Ausgabeparameter haben, die mit denen einer Anfrage übereinstimmen, aber Eingabeparameter, die den Ausgabeparametern anderer Web Services entsprechen.

Betrachten wir Abbildung 8. In dieser Abbildung suchen wir einen Web Service, der X als Eingabeparameter und Y als Ausgabeparameter hat. Wir haben eine Menge S von Web Services:

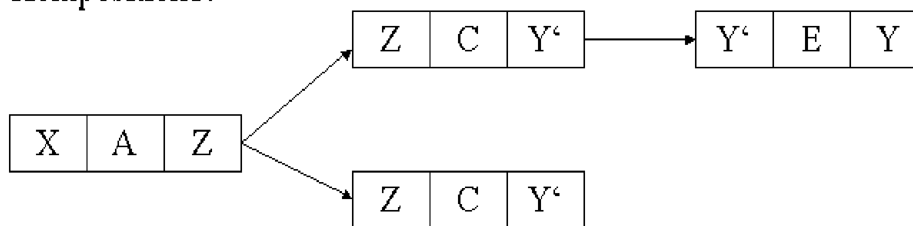
$$(A \in S \wedge B \in S \wedge C \in S \wedge D) \in S \tag{6}$$

Die Web Services A und B haben Eingabeparameter, die mit dem Eingabeparameter der Anfrage übereinstimmen. Wir wählen A und entfernen ihn aus der Menge S, um unendliche Schleifen zu vermeiden. Wir nehmen nun den Ausgabeparameter von Web Service A und berücksichtigen alle möglichen Expansionen, damit wir Web Services finden, deren Eingabeparameter mit dem Ausgabeparametern von einem anderen Web Service übereinstimmen und finden Web Service C. Web Service C hat Ausgabeparameter Y'. Schließlich finden wir Web Service E, der als Eingabeparameter den Ausgabeparameter von Web Service C und als Ausgabeparameter den des gesuchten Services hat, was eine Äquivalenz ist, denn Y ist äquivalent zu Y. Im Falle von Web Service C gibt es zwei Kompositionen, weil Ausgabeparameter Y' eine Untermenge von Y ist (Beispiel 6).

Notation: Inputs | Servicename | Outputs

Gesuchte Service: X | | Y

Komposition1:



Komposition2: X B K → K D Y

Abbildung 8: Komposition von Web Services als Graphen

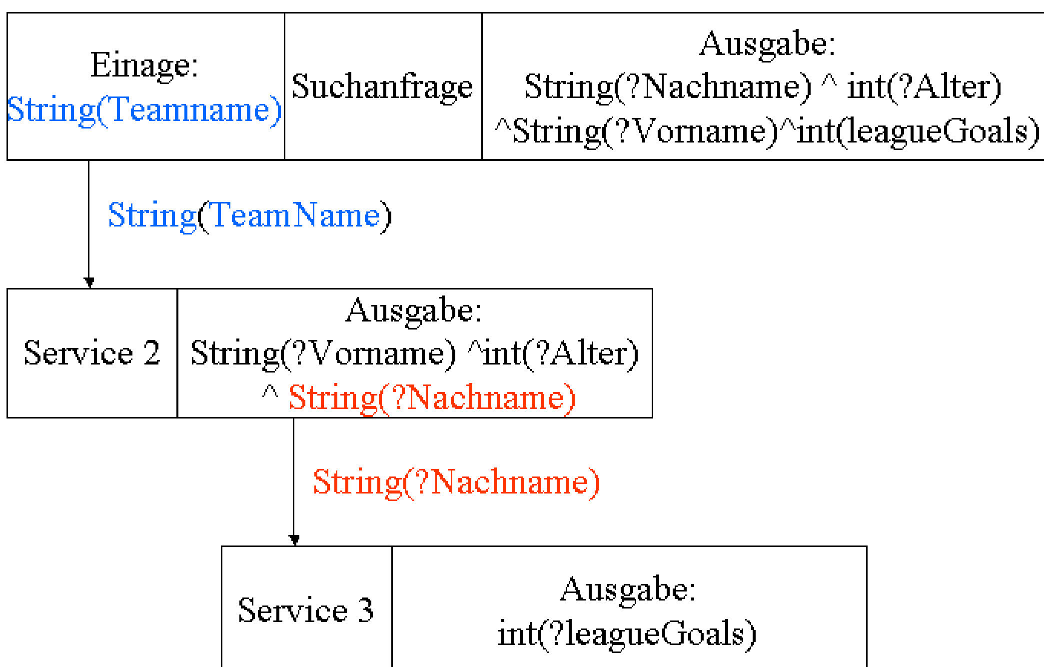


Abbildung 9: Flußkontrolle einer Service-Komposition

Beispiel 6:
Es wird ein Web Service gesucht, der als Eingabeparameter einen Stringwert erhält, der den Name eines Vereins spezifiziert. Als Ausgabeparameter werden Alter, Ligatore, Vorname und Nachname der Spieler dieses Vereins zurückgeliefert: **Input:**

$$String(?teamName)$$

Output:

$$String(?firstName) \land String(?lastName) \land int(?leagueGoals) \land int(?age)$$

Local:

$$Team(?team) \land Player(?player)$$

Precondition:

$$teamName(?team, ?teamName) \land playsFor(?player, ?team)$$

Effect:

$$lastName(?player, ?lastName) \land firstName(?player, ?firstName) \land Type(?player, Player) \land age(?player, ?age) \land leagueGoals(?player, ?leagueGoals)$$

Offenbar erfüllt keiner der drei Web Services aus Beispiel 1 unsere Anfrage. Aber eine Kombination aus Web Service 2 und Web Service 3 würde die Anfrage beantworten.

2.4 Autonomous Service Invocation

Sobald ein Requester einen Web Service von einem Mediator als Antwort auf seiner Anfrage bekommen hat, kann er diesen Web Service entweder in einem Schritt ausführen -Falls der Web Service ein atomarer Web Service ist- oder in mehreren Schritten -Falls der Web Service eine Komposition ist- entsprechend der Beschreibung der Flusskontrolle, des Datenverkehrs usw. Die Ausführung von semantisch beschriebenen Web Services kann als eine Kollektion von entfernten Prozeduraufrufen (RPC) angesehen werden. Semantisch beschriebene Web Services bieten eine deklarative, maschineninterpretierbare API, welche die Semantiken derjenigen Argumente enthält, die zur Ausführung dieser Aufrufe benötigt werden. Die Semantiken werden als Nachrichten zurückgeliefert, egal ob der Aufruf erfolgreich war oder gescheitert ist. Ein Agent soll in der Lage sein zu verstehen, welche Eingaben notwendig sind, um den Web Service ausführen zu können,

welche Ausgabeparameter zurückgeliefert werden und welche Änderungen in der Domäne durch diese Ausführung verursacht worden sind.

Semantisch beschriebene Web Services bieten semantische Beschreibungen des Datenverkehrs, der Reihenfolge der Ausführung der Teilkomponenten, der Iterationen bei Schleifen und der Dialogführung, falls die Ausführung einer Komposition vorliegt. Es gibt verschiedene Möglichkeiten, wie Daten zwischen den Komponenten fließen können. Semantisch beschriebene Web Services bieten Mechanismen, um verschiedene Flusskontrollen zu benutzen. Es werden folgende Punkte festgelegt:

- Die Reihenfolge der Ausführung der Teilkomponenten.
- Der Fluss der Daten zwischen den einzelnen Komponenten.
- Parallele Ausführung und Synchronisation.
- Schleifen und deren Typen und die Anzahl der Iterationen.
- Dialogführung bei interaktivem Austausch von Nachrichten.

Sobald Web Services gefunden worden sind, kann der Requester beginnen, mit ihnen zu interagieren. Nachrichten werden in dem von beiden Parteien vereinbarten Format generiert und ausgetauscht. Servicebeschreibungen müssen deswegen Informationen über die unterstützten Protokolle enthalten. Damit will man verhindern, dass eine Situation entsteht, in der ein Requester mit Web Services interagieren will, aber feststellt, dass es keine gemeinsamen Protokolle gibt. Deswegen ist es sinnvoll, Informationen über Protokolle während der Suche zu berücksichtigen. Der Mediator soll Web Services zurückliefern, die mit den Benutzerfähigkeiten kompatibel sind.

Die Semantiken während der Suche und der Komposition sind auf einer abstrakten Ebene und müssen nicht unbedingt bei der Interaktion betrachtet werden. Semantisch beschriebene Web Services bieten eine Beschreibung der Parameter, der Operationen und der Endpunkte auf einer konkreten Ebene. Die semantisch vermerkten Parameter können in Datenstrukturen konvertiert werden, die ausgetauscht werden. Das ist der Grund, wieso semantisch beschriebene Web Services konform mit Standards wie WSDL und UDDI sind.

3 Verwendete Techniken

Semantisch beschriebener Web Service ist die Synthese der beiden Technologien Semantic Web und Web Service. In diesem Kapitel werden beide Technologien Schritt für Schritt erklärt. Autonome Service Suche, Komposition und Ausführung sind in der Regel Funktionalitäten eines Agenten, deswegen erst eine kurze Einführung in die Agententechnologie.

3.1 Agenten

Die Agententechnologie stellt neue Wege zur Verfügung, um in verteilten Systemen Ressourcen zu repräsentieren und mit ihnen zu interagieren. In diesen Systemen sind Ressourcen als intelligente Entitäten dargestellt. Die Interaktion ist asynchron und dynamisch. Um diese neuen Konzepte zu verstehen, ist es sinnvoll, sie mit etwas gut Studiertem und Verstandenem zu vergleichen. Deswegen erfolgt die Erklärung der Agententechnologie an Hand des Vergleichs mit objektorientierten Technologien.

Agenten unterscheiden sich von Objekten in drei Punkten:

- Agenten sind autonom.
- Agenten interagieren durch Nachrichtenaustausch mit anderen Agenten.
- Agenten können sein.

Der erste Unterschied kommt aus dem Wesen der Agententechnologie. Autonomität bedeutet, dass Agenten verantwortlich für ihre Ausführungsreihenfolge sind. Agenten sind in der Lage, ihre Aktionen und Interaktionen auszuwählen [14]. Jeder Agent hat einen eigenen Ereigniszyklus, in dem die Reihenfolge der Ausführung von einem internen Modell festgelegt wird. Dieses Modell repräsentiert die Kernlogik des Agenten. Das Modell basiert in der Regel auf Wünschen (desires) und Zielen (goals). Ein Objekt hat keine Kontrolle über die Reihenfolge der Ausführung. Objekte bieten eine Reihe von Methoden, die in der Reihenfolge ihrer Aufrufe ausgeführt werden. Die Methoden können das Ausführen einer weiteren Reihe von internen Methoden hervorrufen. Der Initialakt des Methodenaufrufs wird nicht von dem Objekt selbst, sondern von einer externen Entität festgelegt.

Dies führt zum zweiten fundamentalen Unterschied zwischen Objekten und Agenten. Objekte interagieren durch Methodenaufrufe, Agenten durch Nachrichtenaustausch. Nachrichtenaustausch ist eine asynchrone Kommunikation zwischen Agenten und notwendig für autonomes Verhalten. Denn die Interpretation der

Nachricht ist eine Entscheidung, die vom Agenten selber getroffen wird. Dies verleiht dem Agenten die Fähigkeit zur Bevorzugung bestimmter Nachrichten, z.B. auf Metainformationen basierende Nachrichten wie Sender, Ontologie oder Sprachen. Wie und wann ein Agent eine ankommende Nachricht behandelt, hängt ganz vom internen Zustand des Agenten ab. Der Agent kann z.B., wenn er beschäftigt ist, eine Nachricht in einer Warteschlange für spätere Nutzung ablegen.

Um den Unterschied weiter zu veranschaulichen, ist es sinnvoll, reaktives und proaktives Verhalten zu unterscheiden. Reaktives Verhalten ist die Fähigkeit, auf Anfragen (requests) zu agieren. Agenten und Objekte sind in der Regel reaktiv, aber Objekte weisen eine stärkere Form der Reaktivität auf, indem sie eine Reihe von Funktionalitäten in Form von Methoden bereitstellen, die von externen Entitäten aufgerufen werden können. Agenten können reaktiv sein, aber wegen ihrer inhärenten Autonomie und ihrer Art der Interaktion durch Nachrichten haben sie eine flexiblere Form von Reaktivität. Auf ihrem Ereigniszyklus basierend können Agenten entscheiden, wann sie auf ein Ereignis reagieren. Diese Art von Parallelität ist üblich in vielen verteilten Systemen [15].

Objekte und Agenten sind sehr verschieden, wenn es um Proaktivität geht. Proaktiv bedeutet, in der Lage zu sein, selber Aktionen zu initialisieren, z.B. gemäß eines eigenen, internen Modells. Eine Anforderung hierfür ist die Autonomität. Objekte sind nicht autonom, deswegen können sie nicht proaktiv sein.

Abbildung 10 zeigt einen nutzenbasierten Agenten[7] [16] mit Weltmodell:
Eine wichtige Facette von Agentenfunktionalität ist der Austausch von Beweisen, die in einer vereinigenden Sprache geschrieben sind (Ontologien) [21]. Weitere zentrale Merkmale sind die digitalen Signaturen. Dabei handelt es sich um verschlüsselte Datenblöcke, welche von Computern und Agenten benutzt werden können, um zu beweisen, dass die angehängte Information von einer bekannten, glaubwürdigen Ressource kommt [21].

Der Mediator aus Abbildung 2 ist in der Regel ein Agent, der zur Ermittlung von semantisch beschrieben Web Services benutzt wird. Semantisch beschriebene Web Service sind Web Services, die auf einer semantischen Ebene beschrieben sind.

3.2 Web Service-Technologie

Web Service ist eine Technologie, die eine sprachneutrale und plattformunabhängige Interaktion im WWW ermöglicht. Diese Technologie bietet Funktionalität zur

[7] Für weitere Agentenmodelle wie reflexive/reaktive Agenten, Agenten mit internem Weltmodell, Agenten mit expliziten Zielen empfehlen sich: Stuart Russell: Rationality and Intelligence; Stuart Russell und Devika Subramanian: Provably bounded optimal agents

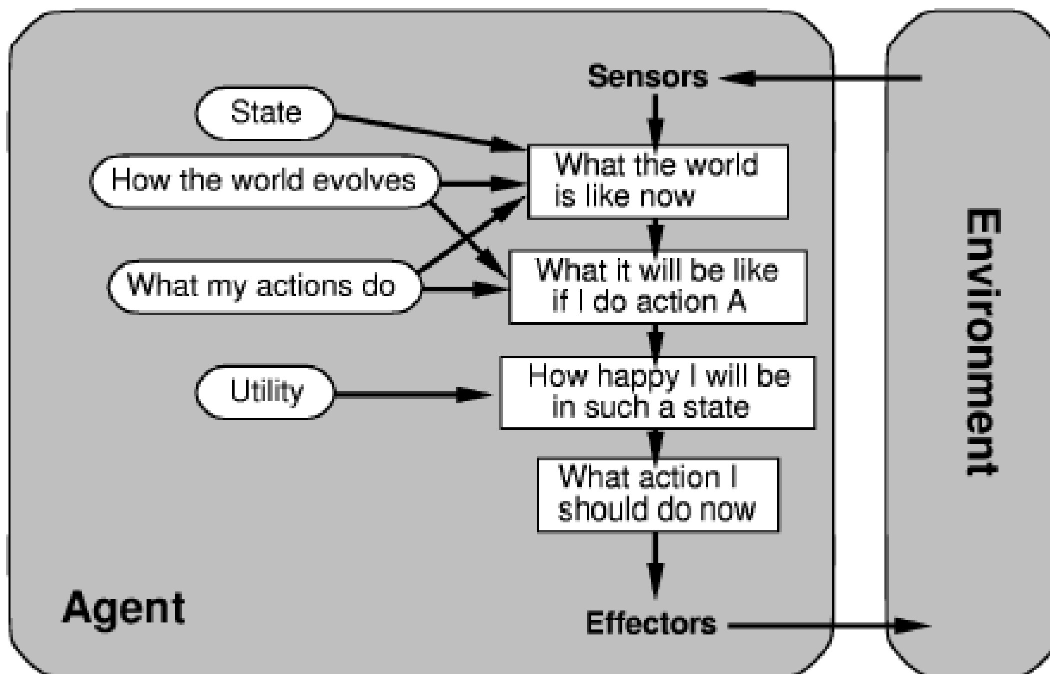


Abbildung 10: Nutzenbasierter Agent mit Weltmodell

Beschreibung, Registrierung, Suche und Ausführung von Web Services und Annotationen, um diese Funktionalitäten in einer plattform- und programmiersprachenunabhängigen Form zu repräsentieren. Diese Annotationen basieren auf XML und bilden ein Schichtenmodell. Abbildung 11 zeigt dieses Schichtenmodell.
Ein Web Service ist eine wohldefinierte, lose gekoppelte Applikation, die Anfragen von Clients entgegennimmt. Wohldefiniertheit bedeutet, dass der Service genügend Informationen zur Verfügung stellt, damit ein Client mit ihm interagieren kann. Lose gekoppelt soll ausdrücken, dass Services plattform- und programmiersprachenunabhängig sind. Dies ist das Gegenteil von komponentenbasierten Softwarearchitekturen, wo die Komponenten stark gekoppelt sind [10].
Das Konzept von Web Services führt zur Entwicklung des Paradigmas der „Service Oriented Architectures (*SOA*)" [1]. Die Idee von *SOA* ist die Verteilung von monolithischen Applikationen als Web Services auf vielen Rechnern, die über die Grenzen von Unternehmen verfügbar sind, um die Abwicklung von Geschäften einfacher zu realisieren. Abbildung 12 zeigt zwei Unternehmen, die einen Service teilen.

Im folgenden werden die Einzelnen Schichten des Web Service-Schichtenmodells, wie auf Abbildung 11 veranschaulicht, genauer erklärt.

Dienst-Entdeckung Schicht	UDDI
Dienst-Beschreibung Schicht	WSDL
XML-Nachrichten Sende Schicht	SOAP
Dienst-Transport Schicht	HTTP, SMTP

Abbildung 11: Web-Service-Schichtenmodell

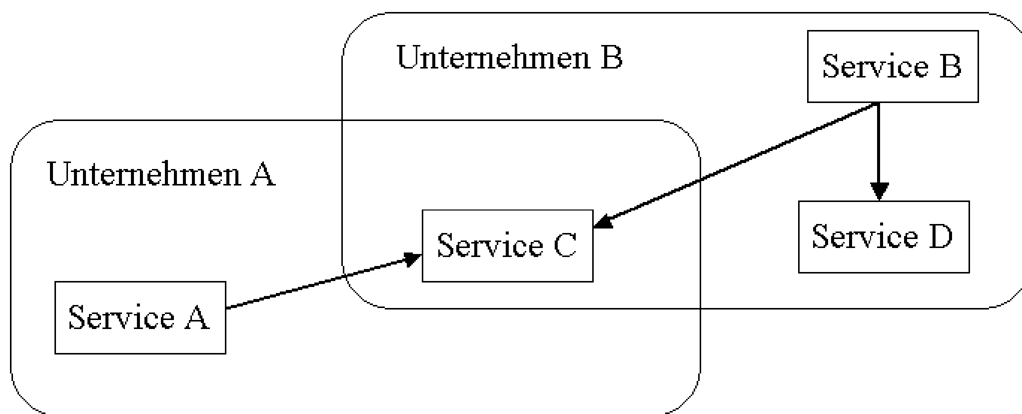


Abbildung 12: Servicenutzung quer durch Unternehmen, die Pfeile stellen Nutzung/Abhängigkeiten dar.

3.2.1 *SOAP* (Simple Object Access Protocol)

Das Simple Object Access Protocol legt eine Formatierung fest, mit der die Informationen, die von einem Rechner zum anderen übertragen werden, codiert werden müssen, um eine *globale* Verständigung zwischen den beteiligten Rechnern zu erreichen. SOAP basiert auf der XML-Syntax und kann daher von XML-Parsern eingelesen werden, ohne dass hierfür erst noch eigene Lösungen programmiert werden müssen [33].

SOAP ist ein Protokoll, um Informationen zwischen Applikationen auszutauschen. Das Protokoll spezifiziert ein Format, in dem Informationen verpackt werden können. Wenn Informationen in diesem Format verpackt sind, können sie von einer Applikation entpackt werden, die dieses Format unterstützt. SOAP ist deswegen ein Benachrichtigungsprotokoll, das Informationen, auf Abbildung 13 veranschaulicht, in eine Hülle verpackt [3].

SOAP-Nachrichten können unter Verwendung verschiedener Protokolle der darunter liegenden Schichten ausgetauscht werden. Eine Spezifikation, wie über SOAP-Nachrichten zwischen zwei Applikationen kommuniziert wird, heißt „SOAP Binding". Das meistbenutzte Protokoll ist logischerweise HTTP [33]

SOAP umfasst nur die Formatierung von Daten, nicht aber, wie die Daten von A nach B gelangen oder wie es technisch realisierbar ist, einen entfernten Methodenaufruf (RPC: Remote Procedure Call) über das Internet durchgzuführen. Das heißt, dass die Art, wie ein RPC auf dem Zielrechner ausgeführt wird, Sache des Dienstanbieters ist. Der Dienstekonsument muss wissen, wo er den Dienst wie aufruft.

Damit aus SOAP ein Allzweck-Protokoll wird, muss es in der Lage sein, nicht-XML-basierte Daten zu verpacken. SOAPs Lösung für dieses Problem ist die Nutzung eines definierten Datenmodells und eines Kodierungsschemas. Das Datenmodell spezifiziert, wie Daten als Graphen repräsentiert werden, so dass die Daten im Rumpf der SOAP-Nachricht (SOAP Body) eingefügt werden können. Damit die Daten auf einer Leitung transferiert werden können, müssen sie serialisiert werden und auf der anderen Seite der Leitung, wenn die Daten empfangen werden, müssen sie wieder deserialisiert werden. Die Abbildung zwischen Datenmodell und übertragenen Informationen ist durch das Kodierungsschema spezifiziert. Datenmodell und Kodierungsschema müssen deswegen zusammen verwendet werden, um Nicht-XML-Daten zu übertragen. Abbildung 13 zeigt den Aufbau von SOAP [33].

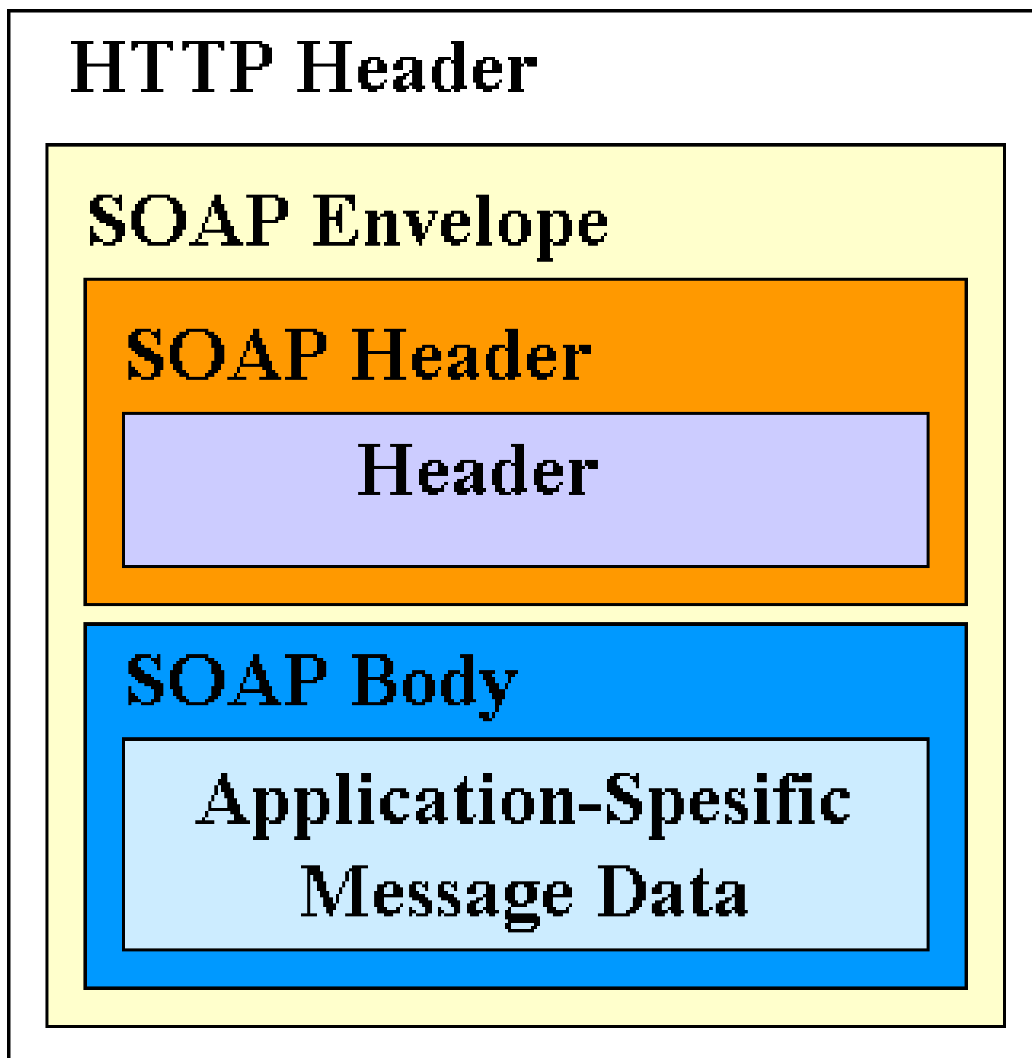


Abbildung 13: SOAP-Umschlag (envelope)-Layouts

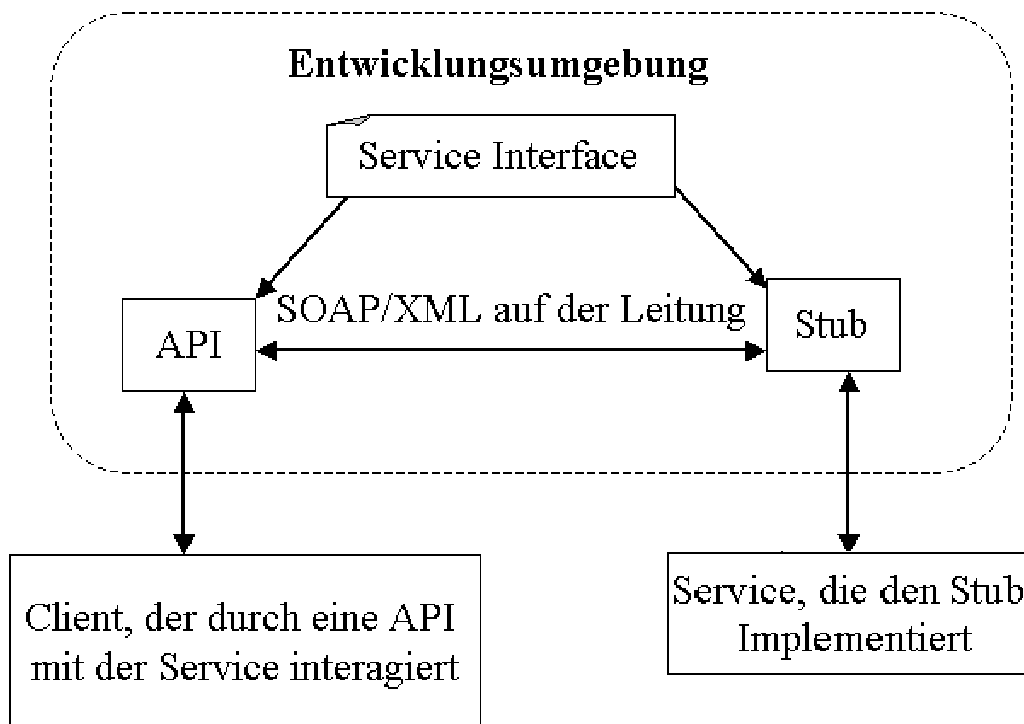


Abbildung 14: Konzeptuelle Arbeitsweise der Web-Service-Technologie

3.2.2 WSDL (Web Services Definition Language)

Die nächste Ebene des Web Service-Schichtenmodells ist WSDL, ein XML-Format zur Beschreibung von Netzwerk-Diensten als eine Ansammlung von Endpunkten, die über Nachrichten operieren. Die Nachrichten und Operationen sind abstrakt beschrieben. Die Nachrichten werden an ein konkretes Netzwerkprotokoll und ein Nachrichtenformat gebunden, um einen Endpunkt zu definieren. Relevante konkrete Endpunkte werden zu abstrakten Endpunkten kombiniert [2].

WSDL beschreibt Web Services, damit mit diesen interagiert werden kann. Es stellt für Clients Informationen bereit um den Standort der Web Services zu bestimmen, und um zu erfahren, welche Funktionalität diese Web Services bieten und welche Protokolle sie unterstützen. Ein Dokument, das all dies beschreibt, wird als Schnittstelle bezeichnet und sowohl vom Client als auch vom Service benutzt. Ein Service implementiert diese Schnittstelle und ein Client interagiert mit der API, die von der Schnittstellenbeschreibung generiert wird. Abbildung 14 zeigt die Arbeitsweise dieser Technologie.
WSDL ist eine auf XML basierende Sprache. Mit Hilfe von WSDL kann definiert werden, welche Methoden bei der Serverkomponente vom Client ausgeführt werden können, welche Parameter dabei übergeben werden müssen und was für einen Rückgabewert diese einzelnen Methoden liefern.

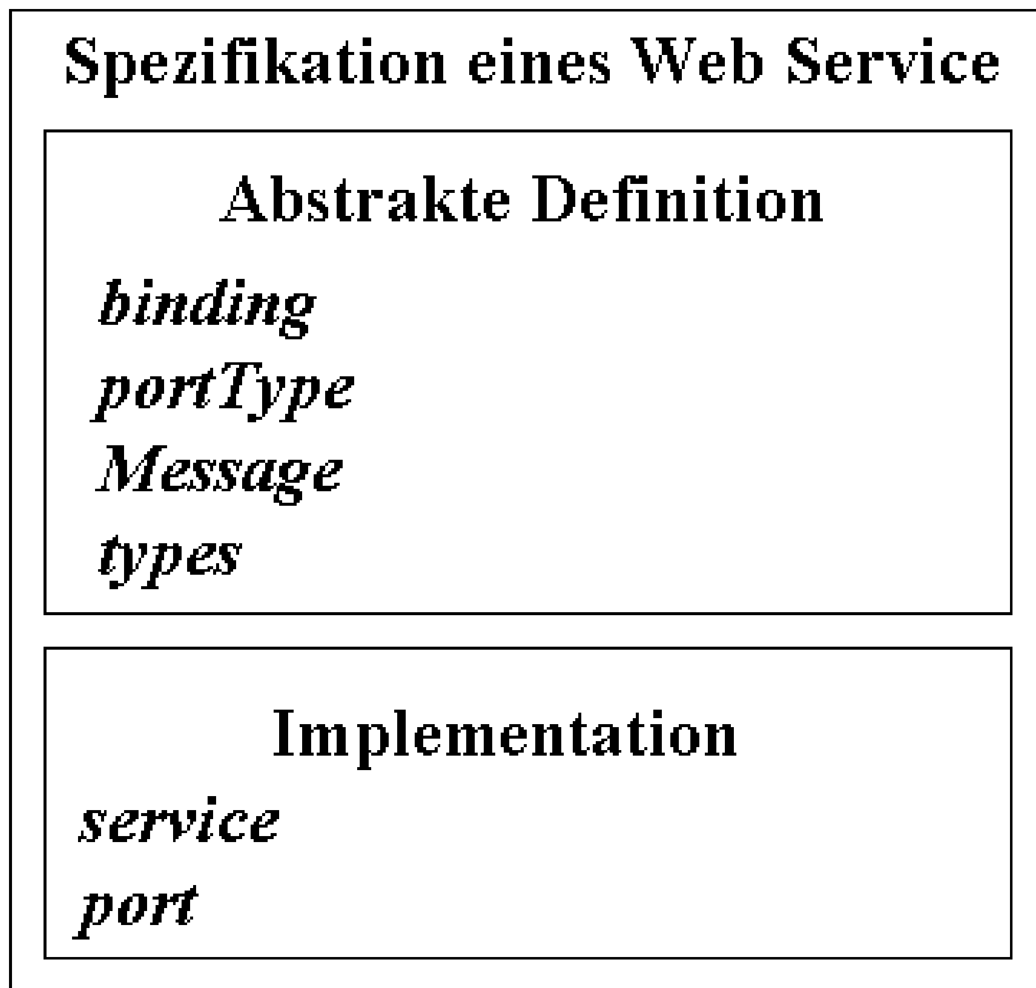


Abbildung 15: Aufbau eines WSDL-Dokuments für einen Web Service

Abbildung 15 zeigt den Aufbau einer Web-Service-Beschreibung. Ein WSDL-Dokument wird in zwei Teile aufgeteilt. Zum einen existiert eine abstrakte und wiederverwendbare Definition im oberen Teil und zum anderen eine implementierungsabhängige Definition im unteren Teil des Dokuments. Die erste ist unabhängig von dem jeweils benutzten Transportprotokoll. Das tatsächlich verwendete Transportprotokoll (SOAP, HTTP etc.) wird innerhalb eines Binding-Elements untergebracht. Dadurch wird der Web Service als solcher technologieunabhängig [33].
Nachfolgend werden die einzelnen WSDL-Elemente vorgestellt. Das *types*-Element ist eine Position im WSDL-Dokument, an der die Möglichkeit besteht, all die Datentypen zu definieren, die nicht vom XML-Schema-Standard erfasst werden. Dies sind sämtliche Hochtypen, die mittels complexType selbst definiert werden

müssen. Innerhalb des *types*-Elements werden nur die Datentypen definiert, die von den Methoden des Web Services entweder an den Client zurückgegeben werden oder die der Client einer dieser Methoden übergeben muss (s. Ausschnitt 1).

Ausschnitt 1: WSDL-Data-Type-Schema

```
1<wsdl:types>
2 <xsd:schema targetNamespace="http://atradig82.informatik.
3    tu-muenchen.de:8280/cocoon/halgurt/wsdl/
4         footballService.xsd1"
5    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/
6    encoding/"xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7    xmlns:xsd1="http://atradig82.informatik.tu-muenchen.de
        :
8    8280/cocoon/halgurt/wsdl/s.xsd1">
9   <xsd:complexType name="Player">
10        <xsd:sequence>
11            <xsd:element maxOccurs="1" minOccurs="0"
12                name="firstName" type="xsd:string"/>
13            <xsd:element maxOccurs="1" minOccurs="0"
14            name="leagueGoals" type="xsd:unsignedInt"/>
15            <xsd:element maxOccurs="1" minOccurs="1"
16                name="lastName" type="xsd:string"/>
17            <xsd:element maxOccurs="1" minOccurs="1"
18                name="age" type="xsd:unsignedInt"/>
19        </xsd:sequence>
20   </xsd:complexType>
21   <xsd:complexType name="PlayerList">
22        <xsd:sequence>
23           <xsd:element maxOccurs="unbounded"
24           minOccurs="0" name="Player" type="Player"/>
25        </xsd:sequence>
26   </xsd:complexType>
27 </xsd:schema>
28</wsdl:types>
```

In den *message*-Elementen werden die Nachrichten beschrieben, die zwischen Client und Server gesendet werden. Ein *message*-Element enthält die Parameter, die in den Nachrichten übergeben werden. Das Kindelement *part* stellt die Parameter dar, die einer Methode übergeben werden (s. Ausschnitt 2).

Ausschnitt 2: WSDL-Message-Parts

```
29<wsdl:message name="PlayerRequest">
30     <wsdl:part name="teamname" element="tns:TeamName"/>
31</wsdl:message> <wsdl:message name="PlayerResponse">
32     <wsdl:part name="PlayerSequence"
33          element="tns:PlayerList"/>
34</wsdl:message>
```

Nachdem im *message*-Element spezifiziert wurde, welche Nachrichten zwischen Client und Server fließen, können mit Hilfe des *portType*-Elements die am Web Service aufrufbaren Methoden definiert werden. Eine operation wird als Kindelement des portType-Elements spezifiziert. Alle Methoden, die der Web Service dem Client zur Verfügung stellen soll, werden mit einem *operation*-Element erfasst. Dessen Kindelemente *input* und *output* spezifizieren, welche Nachrichten (*message*-Elemente) beim Aufruf einer Methode hinein- bzw. hinausgehen (s. Ausschnitt 3).

Ausschnitt 3: WSDL-Port-Type

```
36     <wsdl:portType name="PlayerPortType">
37          <wsdl:operation name="getPlayerYoungerThan25">
38               <wsdl:input message="tns:PlayerRequest"/>
39               <wsdl:output message="tns:PlayerResponse"/>
40          </wsdl:operation>
41     </wsdl:portType>
```

Das *binding*-Element nimmt den größten Teil des WSDL-Dokuments ein. Es beschreibt, wie ein Web Service (Server) mit dem Client kommuniziert. Dies schließt auch das zu verwendende Trägerprotokoll (z.B. HTTP) mit ein. Das Element binding spezifiziert die Funktionsweise des zugrunde liegenden Web Services. Hierzu wird das style-Attribut eingefügt, welches entweder den Wert *rpc* oder den Wert *document* annehmen kann. Der Unterschied zwischen diesen beiden ist, dass die an den Client zurückgesendete SOAP-Nachricht anders beschrieben wird. Das zweite Attribut *transport* spezifiziert das zu verwendende Trägerprotokoll (s. Ausschnitt 4).

Ausschnitt 4: WSDL-Binding

```
42     <wsdl:binding name="PlayerBinding"
43                    type="tns:PlayerPortType">
44          <soap:binding style="rpc" transport="http://
45                    schemas.xmlsoap.org/soap/http"/>
46          <wsdl:operation name="getPlayerYoungerThan25">
47               <soap:operation
48                    soapAction="#getPlayerYoungerThan25"/>
```

```
49          <wsdl:input name="PlayerRequest">
50          <soap:body encodingStyle="http://schemas.xmlsoap.
51               org/soap/encoding/" parts="teamname"
52                    use="encoded"/>
53          </wsdl:input>
54          <wsdl:output name="PlayerResponse">
55          <soap:body encodingStyle="http://schemas.xmlsoap.
56               org/soap/encoding/" parts="PlayerSequence"
57               use="encoded"/>
58          </wsdl:output>
59     </wsdl:operation>
60     </wsdl:binding>
```

Das *service*-Element dient dazu, den Ort des Web Services zu bestimmen. Wenn der Client den Service kontaktieren möchte, muss er dessen URL als Zieladresse angeben. Mit dem *port*-Element wird der Registrierungsname des Web Services angegeben, wobei das *binding*-Attribut auf das zuvor spezifizierte *binding*-Element zeigen muss. Das *adress*-Element unterhalb von *port* enthält die URL des Web Services (s. Ausschnitt 5).

Ausschnitt 5: WSDL-Service-Location

```
61     <wsdl:service name="PlayerService">
62          <wsdl:port binding="tns:PlayerBinding"
63               name="PlayerPort">
64               <soap:address location="http://
65                    atradig82.informatik.tu-muenchen.de:8280/
66                    cocoon/halgurt/services/soap/playerAge"/>
67          </wsdl:port>
68     </wsdl:service>
```

Dies kann von einem Server, z.B UDDI 3.2.3, benutzt werden, um Suchanfragen von Interessenten zu beantworten, die nach solchen Services suchen.

WSDL-Beschreibungen sind zwar für Menschen verständlich, aber ein Rechner kann nicht interpretieren, was der Service genau tut. Die Operationen und Parameter sind in XML beschrieben und haben eine beschränkte semantische Ausdruckskraft. Betrachten wir die Nachrichtenbeschreibungen *message*s. Alles, was der Computer verstehen kann, ist, dass der Web Service einen String als Eingabe hat und als Ausgabe eine Sequenz von String- und Integerwerten. Damit mit Web Services automatisch interagiert werden kann, muss die Funktionalität des Web Service semantisch beschreiben werden, so dass Computer verstehen können, was

diese Services genau tun.

3.2.3 UDDI (Universal Description, Discovery and Integration)

Die oberste Schicht des Web Service-Schichtenmodells stellt UDDI dar. UDDI steht für „Universal Description, Discovery and Integration". Es ist die Registry für sämtliche Web Services. In der Einleitung wurde bereits der Zusammenhang zwischen Konsument, Dienstleister und dieser Registry gezeigt. Nachdem ein Web Service mittels WSDL definiert wurde, muss er veröffentlicht werden. Dieser Vorgang kann mit der Eintragung in eine Liste verglichen werden. Ein Beispiel aus der Nicht-Computerwelt sind die „Gelben Seiten". WSDL hat bereits beschrieben, wie die Web Services aufgebaut sind, mit denen ein Konsument den Service in Anspruch nehmen kann. UDDI ist daher ein Standard, mit dem man diese WSDL-Informationen zugänglich machen kann. Da Web Services für einen globalen Marktplatz gedacht sind, auf dem sich viele Unternehmen als Dienstleister anbieten, können in der Registry insgesamt drei verschiedene Informationstypen untergebracht werden: Business- , Service- und Technikinformationen. Diese werden auch als *White Pages*, *Yellow Pages* und *Green Pages* bezeichnet.[13]

In einer UDDI Registry werden drei verschiedene Informationstypen abgelegt [4]:

- Die *White Pages* erlauben die Abspeicherung von Informationen über Unternehmen und deren Kontaktpersonen. Hierzu zählen auch Informationen wie Steuernummern oder die in den USA gebräuchlichen D.U.N.S.-Nummern.

- Die *Yellow Pages* umfassen den eigentlichen Web Service, der von dem Unternehmen angeboten wird. In der Praxis kommt es oft vor, dass ein Unternehmen nicht nur einen Web Service anbietet, sondern gleich mehrere. Dementsprechend wird dem Konsumenten eine Möglichkeit geboten, alle Dienste dieses anbietenden Unternehmens zu erfragen.

- In den *Green Pages* werden die technischen Informationen des Web Services hinterlegt. Eine technische Information ist zum Beispiel ein WSDL-Dokument.

Zusammengefasst sind die *White Pages* dazu da, Web Services nach Anbietern (Unternehmen, einzelne Personen) zu suchen. *Yellow Pages* erlauben, Web Services nach ihren Diensttypen zu suchen, und über *Green Pages* findet man Web Services nach ihren technischen Informationen aufgelistet [12].

Insgesamt gibt es vier verschiedene Elementtypen in einem UDDI-Registry-Eintrag.

Eine *businessEntity* repräsentiert einen Eintrag in einer Registry. Sie umfasst mehrere Dienste, die als *businessService* deklariert werden. Jedes *businessService*-Element verweist auf ein *tModel*-Element, in dem die technischen Spezifikationen des Web Services enthalten sind [33].

Ein *tModel* repräsentiert eine technische Spezifikation innerhalb der UDDI Registry. Diese Spezifikation kann z.B. ein WSDL-Dokument sein, in welchem der Service technisch beschrieben ist. Bei der Registrierung eines Dienstes wird diesem von der Registry eine eindeutige ID zugewiesen (s. Ausschnitt 6), die ihn von anderen Diensten unterscheidet [33].

Ausschnitt 6: UDDI-tModel

```
1<tModel authorizedName="Beispiel-Applikation"
2     operator="nadaInc/services/uddi"
3          tModelKey="UUID:1234987653428299172524">
4     <name>Ein einfaches Beispiel</name>
5     <overviewDoc>
6          <overviewURL>
7            http://atradig82.informatik.tu-muenchen.de:8280/
8            cocoon/halgurt/wsdl/footballService.xsd1
9          </overviewURL>
10    </overvieDoc>
11</tModel>
```

Um die Menge der Suchergebnisse zu reduzieren, müssen diverse Informationen von Anbieterseite in dem entsprechenden Registry-Eintrag abgelegt werden. Hierzu dienen die Elemente *identifierBag* und *categoryBag*. Ein *identifierBag* repräsentiert eine oder mehrere identifizierende Attributmengen. Diese werden als *keyedReference* in das XML-Dokument eingetragen [33] (s. Ausschnitt 7).

Ausschnitt 7: UDDI-Key-Reference

```
12<identifierBag>
13    <keyedReference>
14         <keyName>
15         <!-- Name der Abteilung, die diesen Dienst
16              anbietet-->
17         Lehrstuhl Radig, Professor Beetz, TUM
18         </keyName>
19         <keyValue>
20              <!-- Kurzbezeichnung des Dienstes -->
21              FIPM
22         </keyValue>
23         <tModelKey>
```

```
24              <!-RegistryKey des Dienstes —>
25              UUID:123498765342829917252 4
26          </tModelKey>
27     </keyedReference>
28</identifierBag>
```

Es muss dabei beachtet werden, dass der Wert des Elements *tModelKey* auf ein existierendes *tModel*-Element verweist. Als weiteres kann auch ein *categoryBag* eingefügt werden. Ein *categoryBag* ist das Gleiche wie ein *identifierBag* mit dem Unterschied, dass ein *categoryBag* Informationen über die Taxonomie eines Web Services enthält [33].

Eine *businessEntity* umfasst mehrere Elemente vom Typ *businessService* und klassifiziert einen einzelnen Service. Jeder *businessService* beschreibt ein Serviceangebot eines Anbieters, welches nicht unbedingt technisch sein muss. Ein *businessService* besteht aus mehreren Kindelementen vom Typ *bindingTemplate*, in denen technische Informationen (URL des Web Services, Referenzen auf *tModel*-Elemente usw.) untergebracht werden [33].

Um die Suche, Ausführung und Komposition von Web Services automatisieren zu können, muss eine Beschreibung der Anwendungdomäne vorliegen. Anwendungsdomänen können mit Hilfe von Semantic Web beschrieben werden.

3.3 *Semantic Web*

The first step is putting data on the Web in a form that machines can naturally understand... This creates what I call a Semantic Web - a web of data that can be processed directly or indirectly by machines.

– Tim Berners-Lee [11]

3.3.1 Einführung

Semantic Web ist dabei, das WWW zur Ausschöpfung seines vollen Potenzials zu bringen [18]. Das WWW enthält über 3 Milliarden statische Dokumente, auf die von über 500 Millionen Benutzern zugegriffen wird. [8] Weil diese Zahlen rasch steigen, wird die Verwaltung von und die Suche nach Informationen immer schwieriger[19]. Semantic Web bietet die Möglichkeit, Wissen intelligent zu repräsentieren und zu verwalten mit dem Ziel, die Informationen im WWW in einer maschineninterpretierbaren Form zu gestalten und Wissen aus Wissen zu schlussfolgern, so dass Informationen im WWW nicht für Menschen, sondern auch für Maschinen verständlich sind. *Semantic Web* bietet ein Framework, welches erlaubt, dass Daten quer über Applikation, Unternehmen und Gemeinschaftsgrenzen verteilt und wieder verwendet werden können. Semantic Web basiert auf *RDF* (Resource Description Framework), welches verschiedene Applikationen integriert. *RDF* benutzt *XML*-Syntax und *URI*s zur Benennung von Ressourcen.

The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.

– Tim Berners-Lee, James[21]

Maschinenverständliche Informationen sind mit *Semantiken* annotiert. Annotationen können als *Metadaten* angesehen werden, welche die *Semantik* der Informationen beschreiben. Eine Ansammlung von Konzepten und deren Eigenschaften und Relationen, die in einer maschineninterpretierbaren Form annotiert sind, heißt eine Ontologie. Damit das Wissen einer Domäne in Ontologien repräsentiert werden kann, muss es in einer Sprache ausgedrückt werden, die in der Lage ist, die nötige Komplexität der Domäne zu vermitteln. *Beschreibungslogik* (*description logic -DL*) hat fortgeschrittene Fähigkeiten, um dies zu realisieren. Eine wirkungsvolle Abstraktion von *DL* ist die *Subsumtion*. *Subsumtion* drückt aus, ob ein Konzept ein anderes Konzept enthält oder nicht. *DL* ist in der Lage,

[8]Stand Dezember 2004

Wissen aus der Repräsentation einer Domäne abzuleiten.

Applikationen, die die *Semantiken* von Dokumenten interpretieren wollen, müssen Zugang zu den Ontologien haben, welche die Semantiken dieser Dokumente definieren. Da Ressourcen allgemein als *URI*s definiert sind, müssen diese Ressourcen erreichbar sein. Damit Ontologien in verteilten Systemen und quer durch Systeme benutzt werden können, muss es einen Standard geben, wie Wissen repräsentiert werden kann. Dieser Standard muss allgemein sein, aber zugleich auch stark genug, damit er die Anforderungen verschiedene Interessenten erfüllen kann. Aktuell sind *RDF* und *OWL* die Standards des Semantic Web. [22][17].

3.3.2 Wissensrepräsentation und Schlussfolgerung

Eine Ontologie repräsentiert das Wissen einer bestimmten Domäne. Sie ist eine formale Konzeptualisierung der Welt[19]. Konzeptualisierung bedeutet hier eine Sicht auf eine Domäne oder abstrakte Welt. Eine Ontologie spezifiziert eine Menge an Einschränkungen, die in einer möglichen Domäne eingehalten werden müssen. Jede mögliche Domäne muss die Einschränkungen einhalten, die in der Ontologie definiert sind. Eine gültige Domäne ist eine Domäne, die alle Einschränkungen einer Ontologie erfüllt.[24]

Beschreibungslogiken sind eine Familie von Sprachen zur Wissensrepräsentation. Die meisten Beschreibungslogiken sind eine Untermenge der Prädikatenlogik erster Stufe, im Gegensatz zu dieser aber entscheidbar. Dies ermöglicht es, über Beschreibungslogiken zu schließen, d.h. aus vorhandenem neues Wissen zu gewinnen [24]. Formal unterteilt man eine Beschreibungslogik in der Regel in eine Terminologiebox (terminological box -*TBox*) und eine Behauptungsbox (assertional box -*ABox*). Die *TBox* enthält hierbei das Wissen über die Konzepte und Relationen einer Domäne (das terminologische Wissen). Die *Abox* hingegen enthält das Wissen über Entitäten oder Instanzen dieser Konzepte sowie deren Beziehungen untereinander und repräsentiert den Zustand der modellierten Welt [24]. Konzepte und ihre Relationen können als physikalische Gesetze der Welt und Individuen als Population, die diese Gesetze befolgen, betrachtet werden. Ein Individuum, das die Gesetze eines Konzepts erfüllt, heißt ein gültiges Mitglied dieses Konzeptes.

Eine Wissensbasis besteht aus TBox und ABox:

$$\sum = (TBox, ABox) \tag{7}$$

Beschreibungslogik ist eine ausdrucksstarke Sprache mit enormen Möglichkeiten zur Repräsentation von komplexem Wissen. Eine detaillierte Diskussion würde

den Umfang dieser Arbeit sprengen, vgl. weiterführend: [17][19][23][24].

Konzepte werden als Mengen betrachtet. Subkonzepte sind Untermengen des Superkonzeptes. Dadurch entstehen Hierarchien von Konzepten. (s. Beispiel 7).

Beispiel 7:
Thing ist das Superkonzept aller Konzepte. *SomethingExisting* ist ein Subkonzept von *Thing*. Die Konzepte *Player* und *Team* sind als Subkonzepte von *SomethingExisting* definiert. Das Konzept Player hat zwei weitere Subkonzepte, nämlich *Goalkeeper* und *Fieldplayer*. *Fieldplayer* besitzt wiederum drei weitere Subkonzepte, das sind *Defender*, *Midfielder* und *Striker*.

$$\begin{aligned} SomethingExisting &\sqsubseteq Thing \\ Player &\sqsubseteq SomethingExisting \\ Team &\sqsubseteq Thing \\ Goalkeeper &\sqsubseteq Player \\ Fieldplayer &\sqsubseteq Player \\ Midfielder &\sqsubseteq Filedplayer \\ Defender &\sqsubseteq Fieldplayer \\ Striker &\sqsubseteq Fieldplayer \end{aligned} \tag{8}$$

Konzepte haben Attribute, die Eigenschaften dieser Konzepte sind. Subkonzepte erben die Eigenschaften des Superkonzepts (s. Beispiel 8)

Beispiel 8:
Das Konzept *Team* aus Beispiel 7 hat eine Eigenschaft *teamName*, die den Namen des Teams ausdrückt. Eine weitere Eigenschaft des Konzeptes *Team* ist *hasPlayer*, die die Beziehung zwischen dem Konzept *Team* und dem Konzept *Player* ausdrückt - Spieler spielen für Vereine. Das Konzept *Player* besitzt die Eigenschaften *firstName*, *lastName* und *playsFor*. All diese Eigenschaften sind binäre Relationen.

$$\begin{aligned} teamName &\sqsubseteq team/2 : Team \sqcap tname/2 : String \\ lastName &\sqsubseteq player/2 : Player \sqcap lname/2 : String \\ firstName &\sqsubseteq player/2 : Player \sqcap lname/2 : String \\ playsfor &\sqsubseteq player/2 : Player \sqcap team/2 : Team \\ hasPlayer &\sqsubseteq team/2 : Team \sqcap player/2 : Player \end{aligned} \tag{9}$$

Die Beschreibungslogik ist in der Lage, Einschränkungen auszudrücken. Einschränkungen sind Bedingungen, die ein Konzept einhalten muss (s. Beispiel

9). Es gibt unterschiedliche Arten von Einschränkungen. Manche betreffen die Häufigkeit des Vorkommens einer Eigenschaft. Andere drücken das Verhältnis zwischen Mengen aus, z.B. das Totalitätsprinzip. Eine weitere Einschränkung ist, wie sich Mengen schneiden und ob sie sich schneiden. Dies betrifft die Instanzen der Konzepte, ob eine Instanz eines Konzeptes die Instanz eines anderen Konzeptes sein darf oder nicht. Weitere Einschränkungen sind Beziehungen zwischen Attributen verschiedener Konzepte, z.B. dass ein Attribut das Gegenteil von einem anderen Attribut ist (s. Beispiel 9).

Beispiel 9:
Eine Einschränkung der Klasse $Team$ aus Beispiel 7 ist, dass exakt 11 Spieler für ein Team spielen dürfen.

$$Team \sqsubseteq \exists^{\geqslant 11} \mid team \mid hasPlayer \tag{10}$$

Das Totalitätsprinzip zwischen Konzepten besagt, dass z.B. die Menge aller $Goalkeeper$ und $Fielplayer$ genau die Menge aller $Player$ ist, mit anderen Worten: Ein $Player$ ist entweder ein $Goalkeeper$ oder ein $Fieldplayer$. Das gleiche gilt für $Fieldplayer$, $Defender$, $Midfielder$ und $Striker$.

$$\begin{aligned} Player &\sqsubseteq Fieldplayer \sqcup Goalkeeper \\ Fieldplayer &\sqsubseteq Striker \sqcup Defender \sqcup Midfielder \end{aligned} \tag{11}$$

Die Konzepte $Goalkeeper$ und $Fieldplayer$ sind disjunkte Konzepte. D.h. keine Instanz des Konzeptes $Goalkeeper$ darf eine Instanz des Konzeptes $Fieldplayer$ sein. Dies gilt für $Defender$, $Midfielder$ und $Striker$ sowie für $Team$ und $SomethingExisting$.

$$\begin{aligned} Goalkeeper &\sqsubseteq \neg Fieldplayer \\ Striker &\sqsubseteq \neg Midfielder \\ Striker &\sqsubseteq \neg Defender \\ Defender &\sqsubseteq \neg Midfielder \\ Team &\sqsubseteq \neg SomethingExisting \end{aligned} \tag{12}$$

Wenn ein Spieler für ein Team spielt, heißt das im Umkehrschluss, dass das Team diesen Spieler hat; dies ist eine Beziehung zwischen zwei Attribute zweier unterschiedlicher Konzepte.

$$hasPlayer \equiv playsFor^{-1} \tag{13}$$

3.3.3 Web Ontology Language (OWL)

Tabelle 1 zeigt einige DL-Konstrukte gegenüber ihrer OWL-Notation.

DL-Syntax	**Beschreibung**	**OWLSyntax**
$C_1 \sqcap ..... \sqcap C_n$	Schnitt	intersectionOf
$C_1 \sqcup ..... \sqcup C_n$	Vereinigung	unionOf
$\neg C$	Negation eines Konzepts	complementOf
$\{a_1, ..., a_n\}$	Eine Liste von Individuen	oneOf
$\forall R.C$	Werteeinschränkung mit Allquantor	allValuesFrom
$\exists R.C$	Existentielle Qualifikation von Konzepten	someValueFrom
$\exists R.\{a\}$	Existentielle Qualifikation von Individuen	hasValue
$\geq_n R.C$	Qualifizierte Mindestkardinalität	minCardinality
$\leq_n R.C$	Qualifizierte Höchstkardinalität	maxCardinality
$\equiv_n R.C$	Qualifizierte Exaktkardinalität	Cardinality

Tabelle 1: DL-Konstrukte mit der dazugehörigen OWL-Notation.

DL-Syntax	**Beschreibung**	**OWLSyntax**
$C_1 \sqsubseteq C_2$	Konzeptsubsumtion (C_1 Subsumiert C_2)	subClassOf
$C_1 \equiv C_n$	Äquivalente Konzepte	equivalentClass
$P_1 \sqsubseteq P_2$	Subsumtion von Eigenschaften	subPropertyOf
$P_1 \equiv P_2$	Äquivalente Eigenschaften	equivalentProperty
$C_1 \sqsubseteq \neg C_2$	Disjunkte Konzepte	disjointWith
$\{a_1\} \equiv \{a_2\}$	Gleiches Individuum	sameAs
$\{a_1\} \sqsubseteq \neg\{a_2\}$	Verschiedenes Individuum	differentFrom
$\geq_n R.C$	Qualifizierte Mindestkardinalität	minCardinality
$P_1 \equiv P_2^{-1}$	Inverse Eigenschaft	inverseOf
$(x, y) \in P \wedge (y, z) \in P \longrightarrow (x, z) \in p$	Transitive Eigenschaft	TransitiveProperty
$(x, y) \in P \longrightarrow (y, x) \in p$	Symmetrische Eigenschaft	SymmetricProperty
$(x, y_1) \in P \wedge (x, y_2) \in P \longrightarrow y_1 = y_2$	Eine Eigenschaft hat einen eindeutigen Wert für jede Instanz	FunctionalProperty
$(x_1, y) \in P \wedge (x_2, y) \in P \longrightarrow x_1 = x_2$	Eine Eigenschaft kann nur einer eindeutigen Instanz angehören	InverseFunctionalProperty

Tabelle 2: Einige DL-Axiome mit der dazugehörigen OWLSyntax

Rollen (Eigenschaften) in OWL können als binäre Eigenschaften angesehen werden, die zwei Klassen miteinander verknüpfen. OWL definiert einige Typen von Eigenschaften, z.B. *ObjectProperty* und *DataProperty*. *ObjectProperty* verknüpft die Instanz einer Klasse mit der Instanz einer anderen Klasse, während *DataProperty* eine Instanz einer Klasse mit einem XML-Schema-Datentyp verknüpft. *DataProperty* wird benutzt, um Daten in der Ontologie zu repräsentieren, während *ObjectProperty* Relationen zwischen Konzepten darstellt.

Eigenschaften werden in OWL mit Domäne(domain) und Bereich(range) Spezifiziert. Die Domäne ist die Klasse, die diese Eigenschaft besitzt und Range ist die Verknüpfung entweder mit einer anderen Klasse oder einem XML-Schema-Datentypen. Einschränkungen werden als Axiome dargestellt, sie verbinden Konzepte miteinander als Rollen. Auf Tabelle 2 sind einige Einschränkungen dargestellt.

Die Annotation einiger Ausdrücke aus Beispiel 7 werden in OWL-Notation in Ausschnitt 8 gezeigt.

Ausschnitt 8: Konzepte

```
1  <owl:Class rdf:ID="Team"/>
2  <owl:Class rdf:ID="SomethingExisting"/>
3  <owl:Class rdf:ID="Player">
4    <rdfs:subClassOf>
5      <owl:Class rdf:ID="SomethingExisting"/>
6    </rdfs:subClassOf>
7    <rdfs:subClassOf>
8      <owl:Class>
9        <owl:unionOf rdf:parseType="Collection">
10         <owl:Class rdf:ID="Fieldplayer"/>
11         <owl:Class rdf:ID="Goalkeeper"/>
12       </owl:unionOf>
13     </owl:Class>
14   </rdfs:subClassOf>
15 </owl:Class>
```

Die Eigenschaften (properties) von Konzepten lauten z.B.: *hasPlayer* ist eine Eigenschaft von *Team* vom Typ *Player* und ist die Inverse von *playsFor*. *playsFor* ist eine Eigenschaft des Konzepts *Player* (s. Ausschnitt 9)

Ausschnitt 9: Klassenattribute(Properties)

```
16 <owl:ObjectProperty rdf:ID="hasPlayer">
17   <rdfs:domain rdf:resource="#Team"/>
```

```
18     <rdfs:range rdf:resource="#Player"/>
19     <rdf:type rdf:resource="http://www.w3.org/2002/07/
20         owl#InverseFunctionalProperty"/>
21     <owl:inverseOf>
22       <owl:FunctionalProperty rdf:ID="playsFor"/>
23     </owl:inverseOf>
24   </owl:ObjectProperty>
```

Ausschnitt 10 zeigt das Verhältnis zwischen Konzepten und deren Schnittmengen.

Ausschnitt 10: Klassenhierarchie

```
25   <owl:Class rdf:about="#Fieldplayer">
26     <rdfs:subClassOf rdf:resource="#Player"/>
27     <rdfs:subClassOf>
28       <owl:Class>
29         <owl:unionOf rdf:parseType="Collection">
30           <owl:Class rdf:about="#Midfielder"/>
31           <owl:Class rdf:about="#Striker"/>
32           <owl:Class rdf:about="#Defender"/>
33         </owl:unionOf>
34       </owl:Class>
35     </rdfs:subClassOf>
36     <owl:disjointWith rdf:resource="#Goalkeeper"/>
37   </owl:Class>
```

Dies waren einige Auszüge aus der Fußballdomäne. Die gesamte Ontologie ist unter
http://atradig82.informatik.tu-muenchen.de:8280/cocoon/ontology/football.owl
verfügbar.

3.4 Semantisch beschriebene Web Services

Semantisch beschriebene Web Services stellen eine konzeptionelle Erweiterung des WebServiceParadigmas mit den oben beschriebenen Techniken des Semantic Webs dar. Dadurch sollen intelligente Softwareagenten in der Lage sein, nach ihren Bedürfnissen mit Hilfe spezieller ServiceOntologien semantisch beschriebene Web Services automatisch aufzufinden, aufzurufen, zusammenzustellen und zu überwachen [6].

Semantisch beschriebene Web Services selbst können wieder als intelligente Agenten betrachtet werden. Aus dieser homogenen Sicht heraus können also solche

Agenten ihr Verhalten mit Hilfe von ServiceOntologien modellieren und publizieren und sogar komplexe Verhaltensweisen selbständig durch Komposition verschiedener Agenten bzw. Dienste realisieren. Ziel ist analog zum Semantic Web auch hier, eine intelligente und flexible Automatisierung von Abläufen durch autonome Agenten zu erreichen [6].

OWLS (Früher DAML-S) ist eine Ontologie zur Definition von Web Services. Jede Instanz des Konzepts *Service* muss folgende Fragen (s. Abbildung 16) beantworten können:

- Was macht der Web Service?
- Wie funktioniert er?
- Wie kann er ausgeführt werden?

Jede Instanz des Servicekonzeptes besitzt drei Eigenschaften, die zur Beantwortung der oben genannten Fragen beitragen. Diese drei Eigenschaften sind Instanzen jeweils eines Konzeptes (s. Abbildung 16), sie sind wie folgt definiert:

- Service *Profile*: Dieses Konzept beantwortet die Frage: Was tut dieser Web Service für einen voraussichtlichen Client? Um diesen Blickwinkel einzufangen, muss jede Instanz des Konzeptes *Service* ein *ServiceProfile* mittels der Eigenschaft *presents* (s. Ausschnitt 11) präsentieren, der benutzt werden kann, um Services zu veröffentlichen und sie zu suchen.
- *process model*: Dieses Konzept beantwortet die Frage: Wie arbeitet dieser Web Service? Dieser Blickwinkel ist in der *ServiceModel*-Klasse eingefangen, welche eine detaillierte Beschreibung des Web Service und seine Operationen ermöglicht. Instanzen des Service-Konzeptes besitzen ein Attribut *describedBy* (s. Ausschnitt 11) um auf diesen Teil zu verweisen.
- Service *grounding*: Dieses Konzept beantwortet die Frage, wie auf einen Web Service durch Nachrichten zugegriffen und mit dem *Service* interagiert werden kann, welche Transport-Protokolle unterstützt werden usw. Jede Instanz des Konzepts *Service* besitzt ein Attribut *supports* (s. Ausschnitt 11) um auf diesen Teil zu verweisen.

Ausschnitt 11: Service-Instanz

```
1  <rdf:Description rdf:about="http://atradig82.informatik.
2      tu-muenchen.de:8280/cocoon/halgurt/profile/
3          PlayerAgeProfile.owl#PlayerAgeProfile">
```

```
4     <service:presentedBy>
5       <service:Service  rdf:ID="PlayerAgeService">
6         <service:presents  rdf:resource="http://atradig82.
7              informatik.tu-muenchen.de:8280/cocoon/halgurt/
8              profile/PlayerAgeProfile.owl#
9                  PlayerAgeProfile"/>
10        <service:supports  rdf:resource="http://atradig82.
11             informatik.tu-muenchen.de:8280/cocoon/halgurt/
12             grounding/PlayerAgeGrounding.owl#
13             AgeWSDLGrounding"/>
14        <service:describedBy     rdf:resource="http://
15             atradig82.informatik.tu-muenchen.de:8280/
16             cocoon/halgurt/processModel/
17             PlayerAgeProcessModel.owl#AgeAtomicProcess"/>
18      </service:Service>
19    </service:presentedBy>
20  </rdf:Description>
```

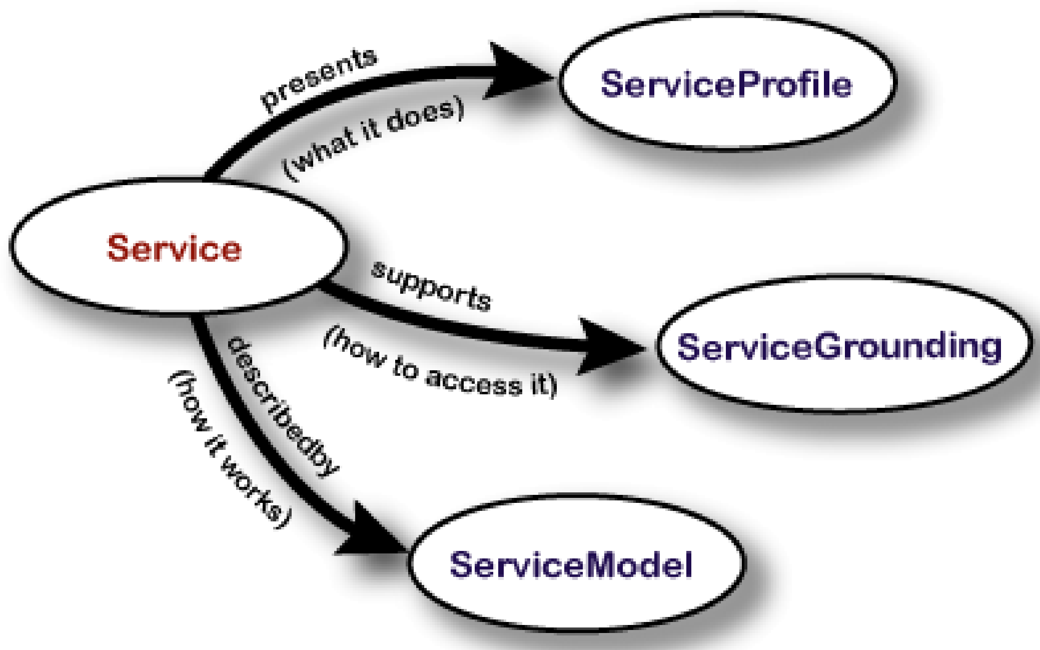


Abbildung 16: Die Service-Ontologie

Die obengenannten Konzepte werden, beginnend mit *Service Profile*, nachfolgend ausführlich erklärt.

Abbildung 17: Profile-Ontologie

3.4.1 ServiceProfile

Die wichtigsten Teile von *ServiceProfile* werden in vier Abschnitten dargestellt. Teil eins beschreibt die Eigenschaften, die *ServiceProfile* mit einer Instanz des Konzepts *Service* und einer Instanz von *ProcessModel* linken. Im zweiten Teil beschreiben wir die Kontaktinformationen und die Beschreibung des *Profile*s. Diese Informationen sind für menschliche Verbraucher und nicht für Agenten gedacht. Teil drei behandelt die funktionelle Repräsentation in Form von IOPEs (Input, Output, Precondition und Effect). Im letzten Teil beschreiben wir die nicht funktionellen Attribute von *ServiceProfile*. Abbildung 17 stellt alle Attribute von *Profile* graphisch dar.

1. **Service-Profile-Klasse**

   Das Konzept *ServiceProfile* bildet die Superklasse für alle Typen von Servicebeschreibungen. Es gibt eine Zwei-Wege-Relation zwischen *Service-Profile* und einer Instanz des Konzepts *Service*. Diese Relation ist durch die Eigenschaften (properties) *presents* und *presentedBy* dargestellt.

   - *presents*: Beschreibt eine Relation zwischen einer Instanz des Konzepts *Service* und einer Instanz eines *ServiceProfiles*. Es besagt eigentlich, dass der *Service* durch *ServiceProfile* beschrieben ist.

- *presentedBy*: Ist die Inverse von *presents* und gibt an, dass diese Instanz von *ServiceProfile* einen *Service* beschreibt(s. Ausschnitt 12 Zeile 2-5).

Ausschnitt 12: Profile-Instanz

```
1  <profile:Profile rdf:ID="PlayerAgeProfile">
2    <service:presentedBy rdf:resource="http://
3    atradig82.informatik.tu-muenchen.de:8280/
4    cocoon/halgurt/service/PlayerAgeService.owl
5    #PlayerAgeService"/>
6    <profile:hasOutput rdf:resource="#Team"/>
7    <profile:hasInput rdf:resource="#teamName"/>
8    <profile:hasOutput rdf:resource="#Player"/>
9    <profile:hasOutput rdf:resource="#lastName"/>
10   <profile:textDescription rdf:datatype="http://
11   www.w3.org/2001/XMLSchema#string"
12   >Getting all Players younger than 25 from a
13   given Team</profile:textDescription>
14   <profile:serviceCategory rdf:resource="#Football
       "/>
15   <profile:has_process rdf:resource="http://
       atradig82
16       .informatik.tu-muenchen.de:8280/cocoon/halgurt
           /
17
18       processModel/PlayerAgeProcessModel.owl#
19       AgeAtomicProcess"/>
20   <profile:hasOutput rdf:resource="#leagueGoals"/>
21   <profile:hasPrecondition
22       rdf:resource="#TeamCondition"/>
23   <profile:hasOutput rdf:resource="#age"/>
24   <profile:hasResult rdf:resource="#ServiceResult"/>
25   <profile:serviceName rdf:datatype="http://www.w3.
26   org/2001/XMLSchema#string"
27   >PlayerAge Service</profile:serviceName>
28   <profile:hasOutput rdf:resource="#firstName"/>
29 </profile:Profile>
```

2. **Service-Name, Kontakte und Beschreibungen**

   Einige Eigenschaften von Profile sind menschenlesbare Informationen, die nicht dazu gedacht sind, automatisch verarbeitet zu werden.

- *serviceName*: Ist der Name des angebotenen *Service*s und kann als Bezeichner von *Service* benutzt werden(s. Ausschnitt 12, Zeile 25-27).
- *textDescription*: Eine kurze Beschreibung der Services. Sie fasst zusammen, was der Service anbietet und beschreibt, was der Service benötigt, damit er funktioniert (s. Ausschnitt 12, Zeile 10-13).
- *contactInformation*: Bietet Mechanismen an mit Bezug auf Menschen und Individuen, die verantwortlich für den *Service*(oder Teile des Services) sind. Der Wertebereich dieser Eigenschaft ist innerhalb von OWLS nicht definiert, kann aber durch bestimmte Ontologien wie FOAF, VCard oder die jetzt veraltete Actor-Klasse der älteren OWLS-Version eingeschränkt werden.

3. **Funktionalitätsbeschreibung**

   Eine wesentliche Komponente des *Profile*s ist die Spezifikation der Funktionalität, die ein *Service* anbietet und die Spezifikation der Bedingungen, die für eine erfolgreiche Ausführung gelten müssen. Zusätzlich gibt *ServiceProfile* an, welche Effekte entstehen, wenn der *Service* ausgeführt wird. *ServiceProfile* repräsentiert zwei Aspekte der Funktionalität eines *Service*:

   (a) Informationstransformation, repräsentiert durch Inputs und Outputs

   (b) Zustandsänderungen, die durch die Ausführung eines Services veranlasst werden, repräsentiert durch Preconditions und Effects

   In unserem Beispiel soll vor der Ausführung der Services gelten, dass der Eingabestring ein gültiger Vereinsname ist, dies ist eine Precondition. Ein Effekt ist, dass der Ausgabeparameter *lastName* von Typ String der Nachname eines Spielers sein muss.

   ServiceProfile bietet keine Beschreibung der IOPE-Instanzen. Diese Beschreibung bietet aber *ProcessModel* an, wie im nächsten Unterabschnitt dargestellt wird. *ProcessModel* definiert die IOPEs und *ServiceProfile* zeigt einfach auf diese Instanzen.

   ServiceProfile bietet folgende Eigenschaften, um IOPEs zu beschreiben:

   - *hasParameter*: Verweist auf die Parameter eines *Services*
   - *hasInput*: Verweist auf die Eingabeparameter eines *Services* (s. Ausschnitt 12, Zeile 7).
   - *hasOutput*:Verweist auf die Ausgabeparameter eines *Services* (s. Ausschnitt 12, Zeile 6, 8, 9, 20, 23, 28).

- *hasPrecondition*: Verweist auf die Vorbedingung eines *Services* (s. Ausschnitt 12, Zeile 21-22).
- *hasResult*: Spezifiziert eines der Ergebnisse eines *Service*s. Es spezifiziert, unter welchen Bedingungen die Ausgabeparameter generiert werden. Darüberhinaus spezifiziert *Result*, welche Änderungen in der Welt während der Ausführung des *Service* verursacht werden (s. Ausschnitt 12, Zeile 24).

4. **Nicht funktionelle Attribute**

   Dies sind zusätzliche Attribute, die *Service*-Qualität garantieren, *Service*-Klassifikation ermöglichen und weitere zusätzliche Parameter, die ein *Service* spezifizieren möchte. Nachfolgend werden diese Attribute vorgestellt:

   (a) *serviceParameter*: Eine expandierbare Liste von Eigenschaften, die eine Servicebeschreibung zusätzlich begleiten. Der Wert dieser Eigenschaften ist eine Instanz der Klasse *ServiceParameter*. Sie besteht aus:

      i. *serviceParameterName*: Ist der Name der aktuellen Parameter, welcher nur ein Literal oder eine URI, die auf eine Instanz von *ServiceParameter* zeigt, sein kann.

      ii. *sParameter*: Zeigt auf den Wert der Parameter innerhalb von Ontologien.

   (b) *serviceCategory*: Sie beschreibt Kategorien von *Service*s basierend auf bestimmten Klassifikationen, die auch außerhalb von OWLS oder sogar außerhalb von OWL sein können. Wenn diese Klassifikationen außerhalb von OWL sind, müssen spezielle Reasoner benutzt werden, damit über diese inferiert werden kann. Ausschnitt 13 zeigt eine Instanz von *serviceCategory*.

Ausschnitt 13: serviceCategory-Instanz

```
1  <profile:ServiceCategory rdf:ID="Football">
2    <profile:value rdf:datatype="http://www.w3.org
3    /2001/XMLSchema#string"
4    >OK</profile:value>
5    <profile:taxonomy rdf:datatype="http://www.w3.
6    org/2001/XMLSchema#string"
7    >www.kicker.de</profile:taxonomy>
8    <profile:code rdf:datatype="http://www.w3.org
9    /2001/XMLSchema#string"
10   >12451221</profile:code>
11   <profile:categoryName rdf:datatype="http://www
12   .w3.org/2001/XMLSchema#string"
```

```
13      >Football</profile:categoryName>
14   </profile:ServiceCategory>
```

Die wichtigsten Elemente von *serviceCategory* sind:

i. *categoryName*: Ist der Name der aktuellen Kategorie, welche nur ein Literal oder evt. eine URI sein kann (s. Ausschnitt 13 Zeile 11-13).
ii. *taxonomy*: Speichert eine Referenz auf das Taxonomie-Schema. Es kann eine URI der Taxonomie oder eine URL, die auf die Taxonomie verweist, sein (s. Ausschnitt 13 Zeile 5-7).
iii. *value*: Zeigt auf einen Wert in einer spezifischen Taxonomie. Es kann mehrere Werte pro Taxonomie geben, hier sind keine Einschränkungen gemacht (s. Ausschnitt 13 Zeile 2-4).
iv. *code*: Für jeden Typ von Service wird der Code gespeichert, der zu einer Taxonomie gehört (s. Ausschnitt 13 Zeile 8-10).

(c) *serviceClassification*: Definiert eine Abbildung zwischen einem *ServiceProfile* und einer OWL-Ontologie von Services, wie die OWL-Spezifikation von NAICS.

(d) *serviceProduct*: Definiert eine Abbildung zwischen einem *ServiceProfile* und einer OWL-Ontologie von Produkten wie eine OWLSpezifikation von UNSPSC.

3.4.2 Process Model

Um eine detaillierte Perspektive darüber zu geben, wie mit einem *Service* zu interagieren ist, kann der *Service* als ein Prozess angesehen werden. Ein Prozess ist kein Programm, das ausgeführt werden kann, er ist vielmehr eine Spezifikation der Art und Weise, wie ein Client möglicherweise mit einem *Service* zu interagieren hat. Ein *atomic process* ist eine Beschreibung eines *Service*, der eine Nachricht (möglicherweise komplex) erwartet und eine Nachricht (möglicherweise komplex) zurückschickt. Ein *composite process* ist eine zustandshafte Beschreibung eines *Service*. Jede Nachricht, die ein Client schickt, treibt ihn (den Client) durch den Prozess voran [6].

Ein Prozess kann zwei Arten von Zweck erfüllen. Erstens kann er Informationen basierend auf erhaltenen Informationen und dem Weltzustand zurückliefern. Informationsproduktion ist durch *Input*s und *Output*s des Prozesses beschrieben. Zweitens kann er eine Änderung im Weltzustand produzieren. Diese Transition ist durch *Precondition*s und *Effect*s des Prozesses beschrieben. Abbildung 18 zeigt die Prozess-Ontologie.

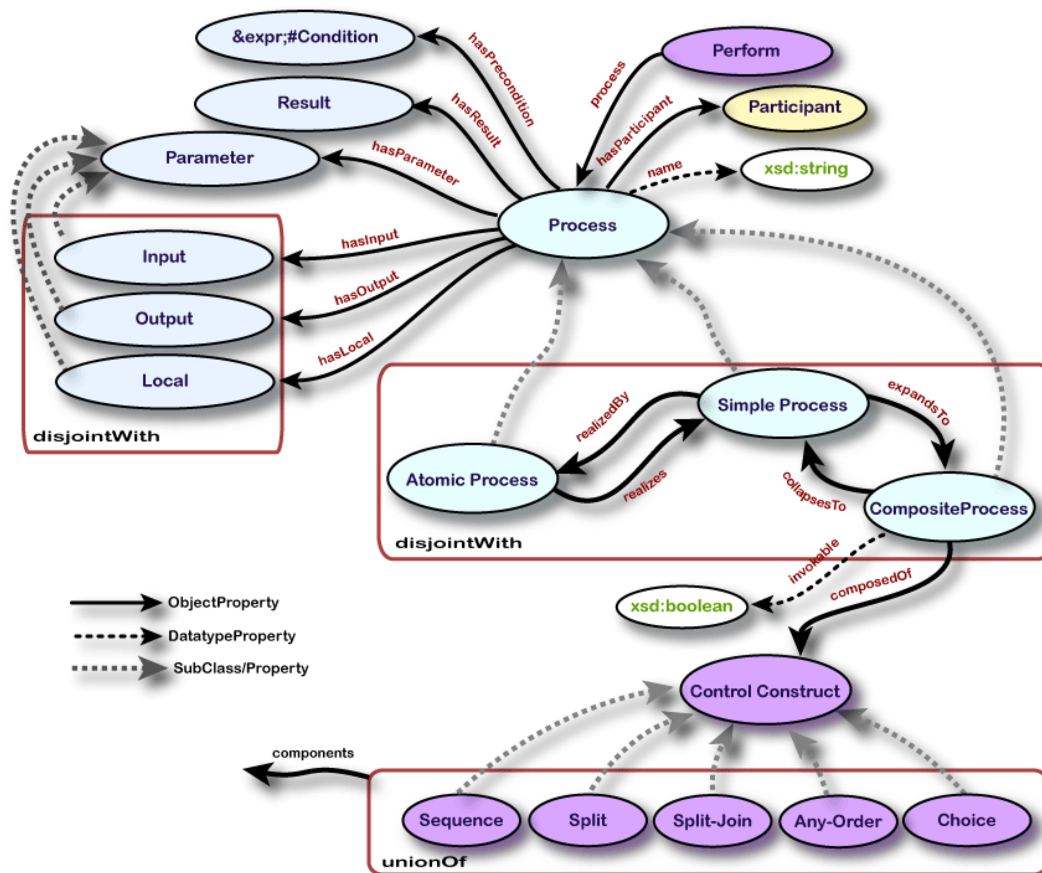


Abbildung 18: Prozess-Ontologie

Bevor auf die Detaills eingegangen wird, wie Prozesse arbeiten, ist es wichtig zu wissen, wie die *IOPE*s arbeiten. Die Eingabe- (*Inputs*= und Ausgabeparameter- (*Outputs*) sind Unterklassen einer allgemeinen Klasse namens *Parameter*. Es ist angebracht, *Parameter* mit den sogenannten *Variable*s in SWRL[7] — eine Sprache, um OWL-Regeln auszudrücken — zu identifizieren. Jeder *Parameter* hat einen Typ, der als URI spezifiziert wird. Dieser Typ ist nicht die OWL-Klasse, von dem der Parameter eine Instanz ist, sondern ist die Spezifikation von Klassen(oder Datentypen), von denen Werte der *Parameter* eine Instanz sind (s. Ausschnitt 14).

Ausschnitt 14: Inputs und Outputs eines Services

```
1  <process:Input rdf:ID="teamName">
2     <process:parameterType rdf:datatype="http://www
3      .w3.org/2001/XMLSchema#anyURI"
4     >http://www.w3.org/2001/XMLSchema#string<
5      /process:parameterType>
6  </process:Input>
```

```
 7
 8  <process:Output rdf:ID="lastName">
 9    <process:parameterType rdf:datatype="http://
10    www.w3.org/2001/XMLSchema#anyURI"
11    >http://www.w3.org/2001/XMLSchema#string<
12    /process:parameterType>
13  </process:Output>
```

Prozess-lokale Variablen werden als *Local* kodiert, sie dienen nur dazu, entsprechend der Domänenontologie die Eingabe- und Ausgabeparameter in den Vor- und Nachbedingungen zu spezifizieren. Die *Local*s müssen in den Vorbedingungen gebunden werden, damit sie später in den Nachbedingungen benutzt werden können (s. Ausschnitt 15).

Ausschnitt 15: Lokale Parameter eines Services

```
15  <process:Local rdf:ID="Player">
16    <process:parameterType rdf:datatype="http://
17    www.w3.org/2001/XMLSchema#anyURI">http://
18     atradig82.informatik.tu-muenchen.de:8280/
19     cocoon/ontology/football.owl#Player<
20    /process:parameterType>
21  </process:Local>
```

Ein *Process* kann nur dann ausgeführt werden, wenn seine Vorbedingungen erfüllt sind. Ein Prozess kann mehrere Nachbedingungen haben. Vor- und Nachbedingungen sind logische Ausdrücke. Es gibt mehrere Möglichkeiten, Vor- und Nachbedingungen auszudrücken, abhängig davon, wie nahe man an OWL/RDF bleiben möchte. Ausdrücke sind Literale, entweder String- oder XML-Literale. XML-Literale werden bei solchen Sprachen benutzt, deren Standardkodierung XML ist, wie z.B. SWRL, String-Literale werden z.B bei KIF oder PDDL (Planning Domain Definition Language) verwendet. Die Ontologie *http://www.daml.org/services/OWLS/1.1/generic/Expression.owl* definiert Ausdrücke (*expressions*) und ihre Eigenschaften.

Ein Ausdruck besteht aus einem Rumpf, *expressionBody* ist die Eigenschaft, die den Rumpf repräsentiert. Ein Rumpf besteht aus einer Liste von atomaren Ausdrücken, *SWRL* definiert diese Liste als *swrl:AtomList*. Jeder atomare Ausdruck wird als ein *swrl:Atom* bezeichnet.

SWRL definiert mehrere *Atom*-Typen, z.B. *swrl:ClassAtom*, um auszudrücken, dass eine Variable vom Typ einer Klasse ist. Ein *swrl:ClassAtom* besteht aus

zwei Eigenschaften, ein *swrl:classPredicate* verweist auf eine OWL-Klasse oder einen XML-Schema-Datentyp. *classPredicate* ist der Typ einer *swrl:Variable.* Und ein *swrl:Argument* kann auf die *swrl:Variable* — da alle *owls:Parameter swrl:Variablen* sind — verweisen. Die Zeilen 49-54 auf Appendix B zeigen ein *swrl:ClassAtom*.

Es gibt weitere sechs *Atom*-Typen, einer davon ist *swrl:DatavaluedPropertyAtom*, das *OWL-Property*s ausdrückt. *Ein swrl:DatavaluedPropertyAtom* hat drei Eigenschaften, ein *swrl:propertyPredicate*, das auf eine *OWL-Property* verweist und zwei Argumente, die auf ein Subjekt und ein Objekt verweisen, beide sind eine *swrl:Variable*e. Diese drei Eigenschaften drücken ein RDF-Triple [Prädikat, Subjekt, Objekt] aus. *swrl:DatavaluedPropertyAtom* dient dazu, mit Hilfe der lokalen *Parameter* Verhältnisse zwischen den Eingabe- und Ausgabeparametern herzustellen. Die Zeilen 6-15 auf Appendix A zeigen ein *swrl:DatavaluedPropertyAtom*.

Um Einschränkungen in Wertebereichen auszudrücken, definiert *SWRL swrl:BuiltinAtom*e. *swrl:BuiltinAtom* besteht aus einem *swrl:builtin* und eine Liste von Argumenten. Ein *swrl:builtin* drückt eine Funktion aus, die in der Ontologie: (http://www.w3.org/Submission/SWRLB/)
definiert ist. *SWRLB* definiert insgesamt 78 verschiedene *swrl:builtin*s. Die Zeilen 26-44 in Appendix B zeigen ein *swrl:builtin*, das ausdrückt, dass der Wert der Variable *age* kleiner als 25 ist. Appendix A zeigt einen Ausdruck, auf den von einem *owls:Process* durch die Eigenschaft *hasPrecondition* verwiesen werden kann. Appendix B zeigt noch einen Ausdruck, der ein *Effect* der Ausführung eines *owls:Process* ist. Die folgenden Eigenschaften der Prozess-Ontologie sind analog zu denen in der Profil-Ontologie:

- *hasInput*
- *hasOutput*
- *hasLocal*
- *hasPrecondition*
- *hasResult*

OWLS benutzt den Term *Result*, um Ausgabeparameter und Nachbedingungen zu koppeln. Ein *Result* ist ein mögliches Ergebnis der Ausführung eines Prozesses. Ein *Result* hat folgende Eigenschaften:

- *inCondition*: Zeigt, unter welcher Bedingung dieses Ergebnis produziert wird.

- *hasResultVar*: Deklariert Variable, die in der *inCondition* gebunden sind. *ResultVars* sind analog zu den lokalen Parametern, obwohl die lokalen Parameter erst in den Vorbedingungen gebunden werden müssen.
- *withOutput*: Zeigt, welche Ausgaben sich ergeben.
- *hasEffect*: Zeigt, welche Effekte sich ergeben.

OWLS definiert drei Typen von Prozessen, nachfolgend werden alle drei Typen vorgestellt.

- *AtomicProcess*:

  Entspricht einer Aktion eines Services, die in einer einzigen Interaktion mit dem Service durchgeführt wird. *AtomicProcess* hat keine Unterprozesse. Er bekommt eine Nachricht, tut etwas und schickt eine Nachricht zurück. *AtomicProcess* hat zwei Beteiligte, *TheClient* und *TheServer*, die einen Client und einen Server darstellen.

- *SimpleProcess*:

  SimpleProcess ist ähnlich zu *AtomicProcess*, mit dem Unterschied, dass *SimpleProcess* nicht ausführbar ist. *SimpleProcess* wird als Abstraktionselement benutzt. Es kann z.B. benutzt werden, um entweder eine abstrakte Sicht auf einen *AtomicProcess* zu bieten oder eine vereinfachte Repräsentation von *CompositeProcess*. *SimpleProcess* besitzt zwei Eigenschaften:

  1. *realizedBy*: *SimpleProcess* ist durch einen *AtomicProcess* realisiert.
  2. *expandsTo*: *SimpleProcess* expandiert zu einem *CompositeProcess*.

- *CompositeProcess*:

  Sind zerlegbar in andere, komplexe oder atomare Prozesse. Die Zerlegung wird durch Kontrollkonstrukte realisiert, wie z.B. *Sequence* oder $If - Then - Else$, die unten beschrieben werden. Kontrollkonstrukte besitzen Namen, die an Kontrollstrukturen in Programmiersprachen erinnern, deswegen kann ein fundamentaler Unterschied leicht übersehen werden: Ein *CompositeProcess* ist keine Handlung, die ein Service tun wird, sondern eine Handlung, die ein Client durch das Senden und Empfangen einer Serie von Nachrichten vollbringen kann. Wenn ein *CompositeProcess* als Ganzes einen Effekt hat, muss der Client alle Teile des *CompositeProcess* ausführen, um diesen Effekt zu bekommen.

Ein $CompositeProcess$ hat eine Eigenschaft namens $composedOf$, welche die Komposition durch ein Kontrollkonstrukt zum Ausdruck bringt. Jedes Kontrollkonstrukt besitzt eine Eigenschaft namens *components*, um auf die eingebetteten Kontrollkonstrukte und deren Ordnung zu verweisen, aus denen es besteht. Jede Sequenz von Kontrollkonstrukten hat einen Wertebereich vom Typ *ControlConstructList* (s. Appendix C, Zeile 47-60), der den Wert von *components* darstellt.

Jeder $CompositeProcess$ kann als ein Baum angesehen werden. Die Knoten sind mit Kontrollkonstrukten versehen. Die Triebe eines Knotens sind die Komponenten des Kontrollkonstrukts. Die Blätter sind Zugriffe auf andere Prozesse, die durch Instanzen der Klasse $Perform$(s. Appendix C, Zeile 53-56) dargestellt werden. Die Klasse $Perform$ besitzt eine Eigenschaft *process*, die auf einen Prozess, der ausgeführt werden soll, verweist. Abbildung 18 zeigt verschiedene Prozesse und ihre Eigenschaften. Es gibt folgende Kontrollkonstrukte in OWLS:

1. *Sequence*: Eine Liste von Konstrukten, die geordnet aufgeführt werden sollen (s. Appendix C, Zeile 45-62).
2. *Split*: Eine Liste von Konstrukten, die gleichzeitig ausgeführt werden dürfen. Ein *Split* ist zu Ende, sobald alle seine Komponenten ausgeführt sind. *Split* definiert partielle Synchronisation.
3. $Split + Join$: Ein Bündel von konkurrierenden Prozessen mit partieller Synchronisation. $Split + Join$ ist zu Ende, wenn alle seine Komponenten ausgeführt sind.
4. $Any - Order$: Eine Liste von Konstrukten, die in jeder Reihenfolge ausgeführt werden können, aber nicht gleichzeitig.
5. *Choice*: Auswahl eines einzigen Prozesses aus einer Liste.
6. $If - Then - Else$: Eine Klasse von Kontrollkonstrukten, die die Eigenschaften $ifCondition$ *then* und *else* hat, mit derselben Semantik wie in Programmiersprachen.
7. *Iterate*: Dieses Konstrukt macht keine Annahmen über die Anzahl der Iterationen, den Zeitpunkt der Instanziierung und den der Terminierung. Die Initialisierungs-, Terminierungs- oder Fortsetzungsbedingungen könnten durch $whileCondition$ oder $untilCondition$ wie unten spezifiziert werden.
8. $Repeat - While$ und $Repeat - Until$: Beide Konstrukte iterieren, bis eine Bedingung wahr oder falsch wird, wie in üblichen Programmiersprachen. $Repeat - While$ kann möglicherweise nie ausgeführt werden, während $Repeat - Until$ mindestens einmal ausgeführt wird.

Bei der Komposition von OWLS-Prozessen kommt es oft vor, dass die Ausgabeparameter eines Prozesses als Eingabeparameter an einen anderen Prozess weitergegeben werden, während an anderen Stellen die Eingabeparameter direkt vom Client übergeben werden. Es gibt mehrere solcher Konstellationen, deswegen wird eine Datenfluss-Spezifikation benötigt (s. Beispiel 10).

Beispiel 10:
Stellen wir uns Folgendes vor: Wir haben eine Komposition aus zwei Services. Ein Service S_1 erhält einen Parameter vom Typ String, der der Name eines Fussballvereins ist und liefert die Namen aller Spieler, die für diesen Verein spielen. Der zweite Service S_2 bekommt den Namen eines Spielers als Eingabeparameter und liefert die Anzahl der von diesem Spieler geschossenen Tore und das Alter des Spielers. Die Komposition dieser beiden Services wird zu einem neuen Service, der einen Vereinsnamen erhält und für jeden Spieler dieses Vereins den Namen, das Alter und die Ligatore zurückliefert. Es ist offensichtlich, dass zuerst S_1 ausgeführt werden muss, damit S2 den Ausgabeparameter von S1 als Eingabeparameter bekommen kann.
Output S_1 = Input S_2

OWLS bietet das Element *Binding* an um anzuzeigen, dass ein Parameter seinen Wert von einem anderen Parameter bekommt. *Binding* drückt ein abstraktes Objekt mit zwei Eigenschaften aus:

1. *toParam*: der Name eines Parameters(s. Appendix C, Zeile 20)
2. *valueSpecifier*: eine Beschreibung des Werts eines Parameters; es gibt 4 verschiedene *valueSpecifier*-Sorten:
    (a) *valueSource*: die einfachste Sorte der *valueSpecifier*. Der Wertebereich von *valueSource* ist ein simples Objekt vom Typ *ValueOf*, das durch seine Eigenschaften *theVar* und *fromProcess* spezifiziert wird. Ein Binding mit *toParam* = P und *valueSource* = s mit *theVar* = v und *fromProcess* = R bedeutet, dass Parameter P von diesem Prozess = Parameter v von Prozess R (s. Appendix C, Zeile 21-36).
    (b) *valueFunction*: ein XML-Literal, das als Funktion gelesen wird
    (c) *valueData*: ein XML-Literal, das eine Konstante ist
    (d) *valueType*: eine URI, die auf eine OWL-Klassendefinition zeigt

3.4.3 Service Grounding

Die bisher vorgestellten Teilontologien von OWLS dienen der abstrakten Spezifikation von Eigenschaften und Funktionen eines Web Services. Das *Grounding*

beschreibt nun, wie die abstrakten Ein und Ausgaben eines atomaren Prozesses transformiert und in konkrete Nachrichten verpackt werden müssen, um die Kommunikation mit dem Web Service zu ermöglichen. Das *Grounding* spezifiziert hierbei die Nachrichtenformate nicht selbst, sondern bedient sich bereits existierender Standards wie WSDL und bildet abstrakte Beschreibungselemente von OWLS auf konkrete Elemente der Nachrichtenformate des betreffenden Kommunikationsstandards ab. Das Grounding stellt somit das Bindeglied zwischen den semantisch höheren, beschreibungslogischen Parametertypen des Prozessmodells und den Typen des Kommunikationsformalismus dar. Es wird demnach immer durch die Beschreibung der Kommunikationsprotokolle in einem anderen Formalismus wie z.B. WSDL und SOAP ergänzt. Hieraus folgt, dass eine GroundingOntologie, welche die Elemente zur Abbildung auf konkrete Komunikationsmechanismen beschreibt, immer auf den Kommunikationsformalismus zugeschnitten sein muss und nicht allgemein definiert werden kann. Es ist offensichtlich, dass durch diese Trennung der Verantwortlichkeiten verschiedene Groundings für das selbe Prozessmodell definiert werden können [6].

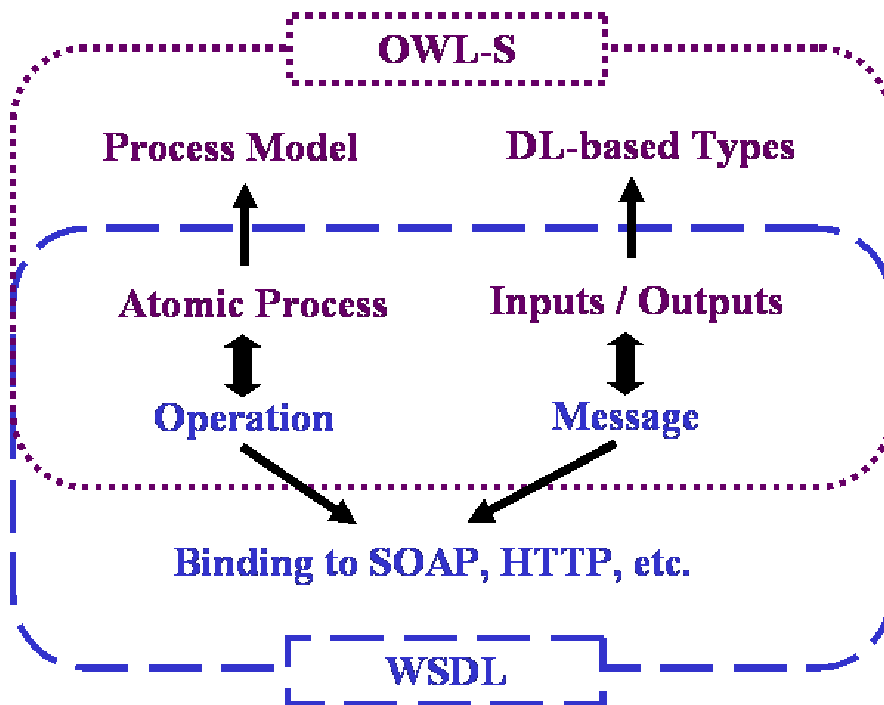


Abbildung 19: WSDL-Grounding

WSDL (s. Kapitel 3.2.2) ist erweiterbar, um Endpunktbeschreibungen zu erlauben, unabhängig davon, welches Netzwerkprotokoll oder Nachrichtenformat be-

nutzt wird. Es ist offensichtlich, dass OWLS-Grounding-Konzepte mit den Konzepten von WSDL-Binding generell konsistent sind.

Ein OWLS-/WSDL-Grounding basiert auf folgenden drei Korrespondenzen zwischen OWLS und WSDL (Abbildung 19):

1. *AtomicProcess* korrespondiert mit WSDL-Operation.
2. Die Menge von Eingabe- und Ausgabeparametern von OWLS korrespondiert mit WSDL-Nachrichten-Konzepten. Genauer: OWLS-Eingabeparameter korrespondieren mit den *part*s der *message*s der WSDL-*operation*s (s. Kapitel 3.2.2).
3. Die Typen (OWL-Klassen) der Eingabe- und Ausgabeparameter des *AtomicProcess* korrespondieren mit WSDLs erweiterbaren Notationen abstrakter Typen (s. Kapitel 3.2.2).

Weil OWLS eine XML-basierte Sprache ist und ihre *AtomicProcess*-Deklaration und Eingabe- und Ausgabeparametertypen anpassbar an WSDL sind, ist es einfach bereits existierende WSDL-*Binding*s zur Nutzung mit OWLS zu erweitern, wie z.B. *SOAP-Binding*. In den folgenden zwei Punkten lassen sich die Erweiterungen darstellen:

1. **WSDL-Spezifikation:**

   WSDL-Erweiterungen sind wie folgt definiert:

   (a) Das *part*-Element von der Definition der WSDL-*message* wird um das Attribut $OWLS-parameter$ erweitert. Diese Erweiterung ermöglicht es, den vollqualifizierten Namen des OWLS-Parameters zu referenzieren, mit dem dieses *part*-Element korrespondiert. Dieses Attribut ist vor allem in solchen Fällen nützlich, wo das *part*-Element ein OWL-Typ und nicht im WSDL-Dokument definiert ist. Der Typ kann aus der Eigenschaft *parameterType* des OWLS-Parameters entnommen werden (s. Appendix E Zeile 33).

   (b) Wenn das *part*-Element einen OWL-Typ benutzt, kann dem Attribut *encodingStyle* innerhalb des WSDL-*binding*-Elements ein Wert wie *http://www.w3.org/2002/07/owl* gegeben werden. Diese Angabe zeigt, dass die *messageparts* als Klasseninstanzen von OWL serialisiert werden müssen (s. Appendix E Zeile 50 und 53).

   (c) Jedes WSDL-*operation*-Element wird um das Attribut $OWLS-process$ erweitert. Dieses Attribut verweist auf einen OWLS-*AtomicProcess*,

mit dem diese WSDL-*operation* korrespondiert (s. Appendix E Zeile 39). [9]

2. **OWLS-Grounding-Klasse:**

   Bis jetzt wurde gezeigt, wie WSDL-Definitionen auf korrespondierende OWLS-Deklarationen verweisen. Es bleibt, einen Mechanismus zu entwickeln, in dem auf mögliche WSDL-Konstrukte innerhalb von OWLS Bezug genommen wird. Die OWLS-Klasse $WSDL - Grounding$ (s. Appendix D, Zeile 15-57) ist eine Unterklasse von *Grounding* und kann zu diesem Zweck benutzt werden. Jede *WSDL-Grounding*-Instanz enthält eine Liste von *WSDL-AtomicProcess-Grounding*-Instanzen (s. Appendix D, Zeile 17-54).

   Eine $WSDL - AtomicProcess - Grounding$-Instanz verweist durch die Nutzung einiger Eigenschaften auf bestimmte WSDL-Elemente:

   (a) $wsdlVersion$: Eine URI, die auf die benutzte WSDL-Version verweist.
   (b) *wsdlDocument*: Eine URI, die auf ein WSDL-Dokument verweist, zu dem dieses *grounding* gehört (s. Appendix D, Zeile 49-51).
   (c) *wsdlOperation*: Die URI der WSDL-Operation im Bezug auf den angegebenen *AtomicProcess* (s. Appendix D, Zeile 18).
   (d) *wsdlService* und *wsdlPort* (optional): Die URI des WSDL-Services (oder Ports), die diese angegebene Operation anbietet (s. Appendix D, Zeile 47-49 und Zeile 9-11).
   (e) $wsdlInputMessage$: Ein Objekt, das die URI der WSDL-*message*-Definition enthält, die auf die Eingabeparameter des gegebenen *AtomicProcess* verweist (s. Appendix D, Zeile 51-53).
   (f) *wsdlInput*: Ein Objekt, das Abbildungspaare für ein *message part* der WSDL-Eingabenachricht enthält. Jedes dieser Paare ist durch eine Instanz von $WsdlInputMessageMap$ repräsentiert. Ein Element des Paars, das durch die Eigenschaft $wsdlMessagePart$ ausgedrückt wird, verweist durch eine URI auf das *message part*. Das andere Element verweist auf einen OWLS-Eingabeparameter. In einfachen Fällen, wo der *message part* direkt mit einem einzigen OWLS-Eingabeparameter korrespondiert, wird die Eigenschaft $owlsParameter$ benutzt. In allen anderen Fällen gibt die Eigenschaft $xsltTransformation$ ein XSLT-Skript an, das diesen *message part* aus der Instanz des $AtomicProcess$ generiert. Das Skript kann als eingebetteter String in der Grounding-Instanz oder als URI angegeben werden (s. Appendix D, Zeile 37-42).

[9]WSDL erlaubt bereits die Nutzung von neuen Attributen in den *messagepart*-Elementen und benutzerdefinierten Werten für das Attribut *encodingStyle*. Punkt drei ist also die einzige Modifikation der aktuellen WSDL-Spezifikation.

(g) *wsdlOutputMessage*: Ähnlich wie *wsdlInputMessage*, aber für Ausgabeparameter (s. Appendix D, Zeile 20-22).

(h) *wsdlOutput*: Ähnlich wie *wsdlInput*, aber für Ausgabeparameter (s. Appendix D, Zeile 22-29).

Um aus OWL-Axiomen komplexe Regeln aufzustellen werden Inferenzmaschinen benötigt. KIF ist eine Sprache zur Definition von Regeln. Diese Regeln werden einer Inferenzmaschine als Wissensbank bereit gestellt, sodass über diese Regeln inferiert werden kann.

3.5 Knowledge Interchange Format (KIF)

Knowledge Interchange Format (KIF) ist eine Sprache, die für den Austausch von Wissen zwischen verschiedenen Computersystemen entworfen wurde. KIF ist kein Format, um Informationen zwischen Rechner und Mensch auszutauschen. Verschiedene PC-Systeme interagieren mit ihren Benutzern auf verschiedene Arten entsprechend ihrer Applikationen. KIF ist keine interne Repräsentation von Wissen innerhalb von Computersystemen. Wenn ein Computersystem Wissen in KIF liest, konvertiert es die Daten normalerweise in seiner internen Form. Alle Berechnungen werden in dieser internen Form durchgeführt. Wenn ein Computersystem mit anderen Computersystemen kommunizieren will, bildet die interne Form auf KIF ab. Der Sinn von KIF ist der Austausch von Wissen zwischen verschiedenen Computersystemen. Folgende kategorischen Features sind wesentlich für den Entwurf von KIF:

1. Die Sprache hat deklarative Semantiken, es ist ohne Zuhilfenahme von evt. manipulierenden Interpretern möglich, die Ausdrücke der Sprache zu verstehen.In dieser Hinsicht ist KIF anders als andere Sprachen, die einen Interpreter brauchen, wie Emycin oder Prolog.

2. Die Sprache ist logisch umfassend — in ihrer Allgemeinheit stellt sie für beliebige Ausdrücke logische Sätze zur Verfügung. Auf dieser Weise unterscheidet sie sich von SQL oder Prolog.

3. Die Sprache ist dazu geeignet, Wissen über Wissen zu repräsentieren. Dadurch kann der Benutzer Entscheidungen über Wissensrepräsentation explizit treffen und neue Wissensrepräsentationskonstrukte definieren, ohne die Sprache zu wechseln.

Zusätzlich besitzt KIF diese Features:

1. Auch wenn KIF nicht dazu gedacht ist, kann es innerhalb von Programmen als Repräsentations- oder Kommunikationssprache benutzt werden.

2. KIF ist zwar nicht als eine Interaktionssprache zwischen Mensch und PC gedacht, aber dafür kann KIF auch benutzt werden.

Die Grundlage der Semantiken von KIF ist eine Konzeptualisierung der Welt in Objekten und Relationen zwischen Objekten.

Ein Interessen-Universum ist eine Ansammlung aller Objekte, die vermutlich oder hypothetisch in der Welt existieren. Objekte können konkret (z.B. Konfuzius, die Sonne) oder abstrakt (Integer 2, die Reihe von natürlichen Zahlen, das Konzept der Gerechtigkeit) sein. Objekte können primitiv oder komplex sein (z.B ein Kanal, der aus vielen Unterkanälen besteht). Objekte können fiktional sein (z.B. ein Einhorn, Sherlock Holmes).

Verschiedene Benutzer einer deklarativen Repräsentationssprache (wie KIF) können verschiedene Interessen-Universen haben. KIF ist konzeptuell vielfältig in dem Sinne, dass nicht alle Benutzer das selbe Universum teilen müssen. Aber andererseits muss jedes Universum bestimmte Basisobjekte enthalten.

Diese Basisobjekte müssen in jedem Universum vorkommen:

1. Alle Zahlen, real und komplex.
2. Alle ASCII-Zeichen.
3. Eine endliche Zeichenreihe aus ASCII-Zeichen.
4. Wörter.
5. Eine endliche Anzahl von Objekten.
6. bottom – Ein Objekt, das vorkommt, wenn eine Funktion auf Objekte angewandt wird, für die sie keinen Sinn macht.

Hier ist zu beachten, dass der Benutzer beliebig viele Nicht-Basisobjekte, die er für sinnvoll hält, benutzen kann.

In KIF haben Beziehungen zwischen verschiedenen Objekten die Form von Relationen. Eine Relation ist definiert als eine beliebige Ansammlung endlicher Listen von Objekten (möglicherweise mit unterschiedlichen Längen). Jede Liste ist die Auswahl von Objekten, die diese Relation erfüllen. Z.B. die $< -Relation$ auf Zahlen, die diese Liste $< 2, 3 >$ enthält, besagt dass 2 kleiner ist als 3.

Funktionen sind spezielle Relationen. Für jede endliche Sequenz von Objekten (Argumente) assoziiert die Funktion ein Objekt (den Rückgabewert). [10]

[10]Für mehr Informationen zu KIF vgl.: http://logic.stanford.edu/kif/dpans.html

Ausschnitt 16 zeigt ein Beispiel in KIF-Syntax. Das Beispiel besteht aus 3 Regeln. Regel 1 (Zeile 1-6) heißt *profile* und drückt aus, dass entsprechend dem RDF-Triple von [Prädikat, Subjekt, Objekt] das Prädikat OWL:type die Variable ?x mit dem Konzept *Profile* aus der Ontologie Profile.owl verbindet, mit anderen Worten: Die Variable ?x ist vom Typ *Profile*. Regel 2 (Zeile 9-14) drückt das gleiche für Variable ?y, Prädikat *type* und Konzept *Service* aus der Ontologie Service.owl aus. Regel 3 (Zeile 16-21) besagt, dass die Variablen ?x und ?y durch die Relation (Property) *presentedBy* aus der Ontologie Service.owl verbunden sind.

Ausschnitt 16: Effekte eines Services

```
1
2   (<=(profile ?x)
3     (|http://www.w3.org/1999/02/22-rdf-syntax-ns#|::|type|
4       ?x
5       |http://www.daml.org/services/OWLS/1.1/Profile.owl#
6       |::|Profile|)
7   )
8
9
10  (<=(service ?y)
11    (|http://www.w3.org/1999/02/22-rdf-syntax-ns#|::|type|
12    ?y
13    |http://www.daml.org/services/OWLS/1.1/Service.owl#
14    |::|Service|)
15  )
16
17  (<=(presentedBy ?x ?y)
18    (|http://www.daml.org/services/OWLS/1.1/Service.owl#
19    |::|presentedBy|
20    ?x
21    ?y)
22  )
```

Die Regeln, die in KIF definiert werden, werden einer Inferenzmaschine übergeben, die darüber inferieren kann.

3.6 Inferenzmaschine (Reasoner)

Eine Inferenzmaschine (teilweise auch Reasoner oder Theorem-Beweiser genannt) ist ein Software-Tool, um neue Fakten oder Assoziationen aus vorhandenen Informationen zu erzeugen. Man unterteilt diese in drei Schritte: Akquisition (Wissenserhebung und -erfassung), Repräsentation (Darstellung und Speicherung des

Wissens in der Wissensbasis) und Inferenz (Verarbeitung des Wissens zur Lösung bestimmter Probleme) [28].

Oft wird behauptet, dass eine Inferenz-Maschine die menschliche Fähigkeit des Schlussfolgerns nachahmt. Indem man aus den Informationen und Beziehungen ein Modell erstellt, ermöglicht man dem Reasoner, logische Schlussfolgerungen auf Basis dieses Modells zu ziehen. Ein typisches Beispiel für ein Reasoning ist, Modelle von Menschen und ihren Beziehungen zu anderen Personen zu erstellen, um neues Wissen zu erhalten. Durch eine genaue Betrachtung eines solchen Netzwerks können Inferenzmaschinen Beziehungen erkennen, die vorher nicht explizit definiert wurden [28]. So könnte beispielsweise aus dem Wissen, dass A Vater von B und C Bruder von A ist, geschlussfolgert werden, dass C Onkel von B ist. Vorausgesetzt natürlich, dass die Regeln, wie eine Onkel-Beziehung dargestellt wird, dem System bekannt sind [34].

Bei Inferenzmaschinen werden naturgemäß zwei Arten des Schlussfolgerns unterschieden. Auf der Basis einer Regel, die in der Form:

wenn A, dann B

repräsentiert wird, kann eine einfache Schlussfolgerung gezogen werden. Häufig will man jedoch komplexere Schlussfolgerungen aus mehreren gegebenen Regeln ziehen. Eine Möglichkeit hierzu besteht in der Verkettung von Regeln. Die Vorwärtsverkettung (forward chaining) geht dabei transitiv vor, d.h. aus einem Faktum wird anhand einer Regel und einer Inferenzmethode (z.B. modus ponens) eine Schlussfolgerung gezogen, die wiederum als Prämisse und mittels einer weiteren Regel für eine weitere Schlussfolgerung verwendet wird usw. Da von einem meist fallspezifischen Faktum ausgegangen wird, bezeichnet man diese Inferenzstrategie auch als datengetriebene Inferenz. Ebenso wie die Vorwärtsverkettung basiert die Rückwärtsverkettung (backward chaining) auf einer transitiven Verknüpfung von Regeln. Man geht dabei jedoch vom Zielobjekt aus und prüft nur die Regeln, die das Ziel in der Schlussfolgerung haben. Falls der Wert eines Objektes in der Prämisse einer solchen Regel unbekannt ist, wird versucht, diesen aus anderen Regeln herzuleiten. Gelingt dies nicht, so wird der Wert schließlich vom Benutzer erfragt. Man nennt dieses Verfahren auch zielorientierte Inferenz [34].

Des Weiteren unterscheidet man zwei Arten von Wissen: *ABox* und *TBox*. Unter *TBox* wird das intensionale, also unveränderliche Wissen in Form einer Terminologie bezeichnet (terminologisches Wissen), welches sich auch beim Inferenzschritt zur Erhebung neuen Wissens nicht verändert. *ABox* hingegen beinhaltet das extensionale Wissen spezifisch für die Instanzen einer Domäne (auch Individuals genannt - assertions on individuals, daher: assertorisches Wissen). Dieses Wissen

kann sich im Laufe der Zeit verändern (meist vergrößern)[34].

In OWL findet auch diese Unterscheidung in *ABox*- und *TBox*-Wissen statt: OWL-Facts wie Typ-Zusicherungen, Eigenschaftszusicherungen, Gleichheit oder Ungleichheit von Individuen sind *ABox*-Zuweisungen, *OWL*-Axiome wie Unterklassen, Untereigenschaften, die Domäne oder der Bereich einer Property hingegen sind *TBox*-Wissen [29].

Es gibt verschiedene Inferenzmaschinen, unten sind einige aufgelistet mit einer kurzen Beschreibung dazu.

- **JESS:**

  Das Java Expert System Shell JESS (aktuell: Version 7.0a6) wird von den Sandia National Laboratories in Livermore, California, USA entwickelt (siehe: http://herzberg.ca.sandia.gov/JESS). Es wurde in der Programmiersprache Java implementiert und liefert eine sehr detaillierte API mit. Es implementiert den Rete-Algorithmus und erweitert diesen um Optimierungen, backward chaining und ein Konstrukt namens *defquery*, um direkte Anfragen an die Wissensbasis zu stellen. JESS baut auf Jena auf und es existieren bereits Übersetzer von OWLS nach JESS. JESS ist ein eigenständiges Programm, welches von anderen Systemen mittels Interface angesprochen werden kann. Dort erwartet JESS die Regeln, Axiome und Instanzen einer Ontologie als Fakten. Es ist also keine direkte Verarbeitung von OWL-Konstrukten aus Jena heraus möglich [34].

- **Pellet:**

  Eine weitere Inferenzmaschine auf Basis von Java ist Pellet, welches in der MINDSWAP Research Group der University of Maryland, USA entwickelt wird. Pellet basiert ebenfalls auf Jena, kann direkt die dort eingegebenen Fakten übernehmen und Anfragen auf deren Basis bearbeiten. Es ermöglicht eine *Reparatur* von OWL-Full auf OWL-DL, falls eine Anfrage in OWL-Full nicht zu beantworten wäre. Pellet basiert auf Tableaux-Algorithmen für ausdrucksstarke Beschreibungslogik und steht unter der Lizenz des MIT, der Quelltext kann also einfach eingesehen werden. Pellet wird im Moment für SWRL weiterentwickelt und stellt bereits für ein Reasoning auf OWL- oder RDF-Basis eine ausführliche Java API zur Verfügung [34].

- **Java Theorem Prover:**

  Der Java Theorem Prover JTP wurde, wie der Name bereits sagt, in Java implementiert und besitzt eine hybride Reasoning-Architektur. Er un-

terstützt backward-chaining-Reasoner, um Anfragen auszuführen und den Beweis für eine Antwort zu ermitteln, sowie forward-chaining-Reasoner, um der Wissensbasis neue Informationen hinzuzufügen und Schlussfolgerungen zu ziehen. Der JTP ermöglicht es, Unterklassen von Reasonern einzubinden, um mittels eines Dispatchers Anfragen an verschiedene Reasoner zu verteilen. Der JTP baut auf dem Jena-Framework auf und kann sowohl DAML+OIL als auch OWL-Konstrukte verarbeiten. Der Java Theorem Prover setzt als einzige Inferenz-Maschine unter Java auf der First-Order Logic auf [26][27][34].

4 Architektur & Algorithmen

4.1 Architektur

In diesem Abschnitt wird die Architektur des Programms vorgestellt. Abbildung 20 und 22 zeigen die Schnittstellen, die Benutzern und Anbietern von semantisch beschriebene Web Services zur Verfügung gestellt werden, um mit dem System zu interagieren. Folgende Komponenten werden in der Architektur benutzt:

1. Apache Tomcat Server:

   Apache Tomcat stellt eine Umgebung zur Ausführung von Java-Code auf Webservern bereit, die im Rahmen des Jakarta-Projekts der Apache Software Foundation entwickelt wird. Es handelt sich um einen in Java geschriebenen Servletcontainer, der mit Hilfe des JSP-Compilers Jasper auch JavaServer Pages ausführen kann.

2. Apache Cocoon Framework:

   Cocoon ist ein Publishing Framework Servlet, das mittels seines modularen Reaktorkonzepts ein XML-Quelldokument je nach anfragendem Client in ein beliebiges Zielformat transformiert.

3. OWQL Server (OWL Query Language) basierend auf JTP:

   Die Web Ontology Query Language (OWL-QL) ist der Entwurf eines standardisierten Zugriffs auf Wissensbasen und Reasoner-Systeme, in diesem Fall JTP (s. Kapitel3.6). Die zugrunde liegende Web Ontology Language wird in (s. Kapitel 3.3.3) beschrieben. Die OWL-QL-Spezifikation definiert zwei Bereiche der Client-Server-Kommunikation: das Nachrichtenformat und ein Kommunikationsprotokoll. Zum einen wird die Struktur von Frage- und Antwort-Dokumenten in XML-Schemata definiert, zum anderen wird ein Protokoll für den Dialog-Prozess zwischen Client und Server vorgeschlagen.

4. Eine Wissensbasis zur Speicherung von registrierten, semantisch beschriebenen Web Services in OWL-Format (s. Kapitel 3.4).

5. Eine Domänen-Ontologie zur Beschreibung der Anwendungsdomäne (s. Kapitel 3.3.3).

Abbildung 20 zeigt ein Szenario, in dem ein Anbieter eine Domänenontologie benutzt, um die von ihm angebotenen Web Services semantisch zu beschreiben, und diese bei einem *SemanticDirectory* registriert oder bereits von diesem Anbieter

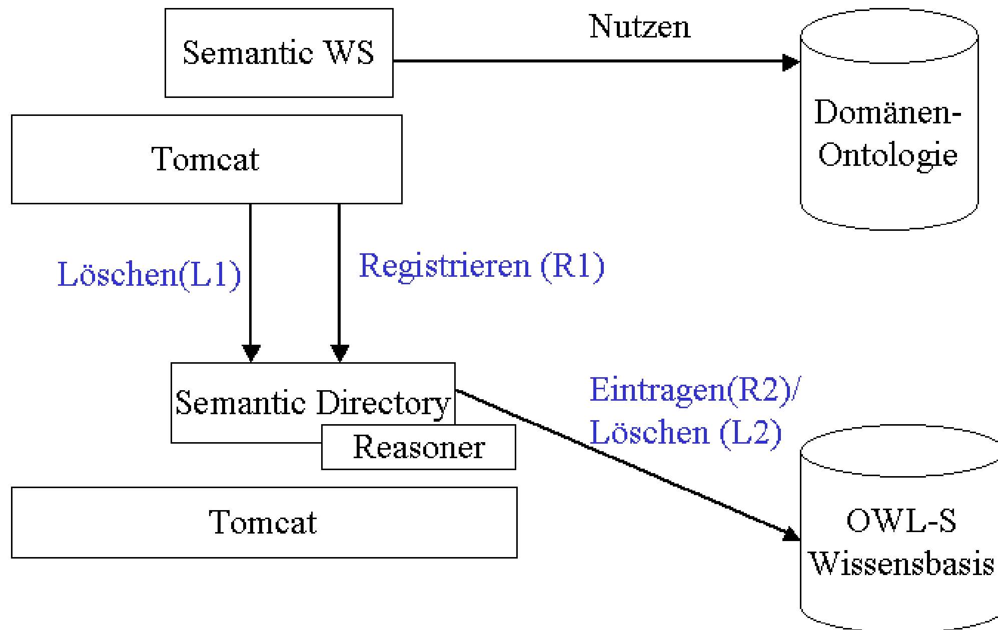


Abbildung 20: Löschen oder Eintragen von semantisch beschriebene Web Services

registrierte semantisch beschriebene Web Services löscht. Folgende Schnittstellen sind auf Abbildung 20 dargestellt.

1. Registrieren(R1):
   Diese Schnittstelle bietet einem Anbieter die Möglichkeit, die von ihm angebotenen Web Services beim *SemanticDirectory* zu registrieren.

2. Eintragen(R2)/Löschen(L2):
   Das *SemanticDirectory* trägt die von einem Anbieter erhaltenen semantisch beschriebene Web Services in die Wissensbank ein oder löscht sie aus der Wissensbank, falls der Anbieter dies veranlasst.

3. Löschen(L1):
   Diese Schnittstelle bietet einem Anbieter an, bereits von ihm registrierte Semantic Web-Services zu löschen, wenn er entscheidet, einen Service nicht mehr anzubieten.

Abbildung 22 zeigt ein Szenario, in dem ein Benutzer eine Domänenontologie benutzt, um seine Suchanfrage zu spezifizieren und diese Suchanfrage einem *Semantic Directory* schickt. Das *Semantic Directory* sucht entsprechend der

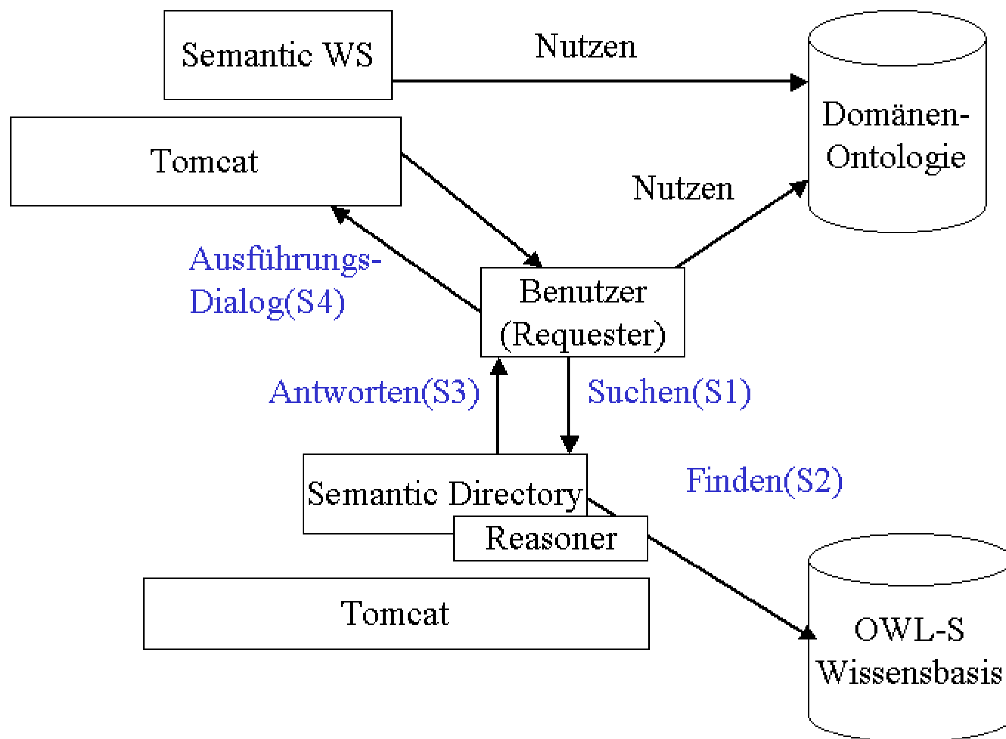


Abbildung 21: Suche nach semantisch beschriebene Web Services

Suchanfrage in einer OWLS-Wissensbasis nach semantisch beschriebene Web Services und gibt dem Benutzer die gefundenen Serviceinstanzen aus.

1. Suchanfrage an
   Semantic Directory (S1):
   Diese Schnittstelle bietet dem Benutzer die Möglichkeit, dem *Semantic Directory* eine Suchanfrage zu schicken, in der seine Suchkriterien spezifiziert sind. Abbildung 23 zeigt ein Suchformular, das zu diesem Zweck benutzt werden kann.

2. Suchanfrage an OWLS-Wissensbank(S2):
   Semantic Directory benutzt ein Reasoner - in unserem Programm „Java Theorem Prover"-, der eine OWLS-Wissensbank benutzt, in der semantisch beschriebene Web Services eingetragen sind, um die Benutzeranfrage zu beantworten.

3. Antworten(S3):
   Sobald der Reasoner Semantic-Web-Service-Instanzen ermittelt hat, werden sie dem Benutzer als Antwort auf seine Anfrage zurückgeschickt.

4. Ausführungsdialog(S4):
   Nachdem ein Service gefunden wurde, kann der Benutzer mit dem Anbieter interagieren und den Service ausführen. Es findet ein Dialog statt, Parameter werden übergeben und die Ergebnisse zurückgeliefert.

4.2 Algorithmus

Einen effizienten Algorithmus zur Ermittlung von semantisch beschriebene Web Services entsprechend einer Menge von Suchkriterien zu schreiben, ist die schwierigste Aufgabe. Jeder Service hat eine Menge von IOPEs. Semantisch beschriebene Web Services werden anhand dieser IOPEs ermittelt. Jeder Benutzer definiert seine Suchkriterien mit Hilfe einer Domänenontologie. Der erste Schritt ist die Erweiterung der Wissensbasis um diese Domänenontologie. Der zweite Schritt ist die Definition von Regeln entsprechend OWLS (s. Kapitel 3.4). Im weiteren Verlauf dieses Abschnitts definieren wir schrittweise die Regeln in KIF-Syntax (s. Kapitel 3.5).

OWLS-Konzepte, deren Attribute und Relationen müssen als Regeln definiert werden. Ausschnitt 17 zeigt die Definition von drei Regeln. Die Zeilen 1-5 zeigen die Definition der ersten Regel, die eine OWLS-Profile-Instanz darstellt. Die Zeilen 7-11 zeigen eine weitere Regel, die eine OWLS-Service-Instanz darstellt. Die dritte Regel ist durch die Zeilen 13-16 definiert, die eine Relation zwischen einer Service-Instanz und einer Profile-Instanz darstellt.

Ausschnitt 17: Service- und Profile-Instanz

```
1  (<=(isProfile ?profile)
2     (|http://www.w3.org/1999/02/22-rdf-syntax-ns#|::|type|
3     ?profile |http://www.daml.org/services/OWLS/1.1/
4     Profile.owl#|::|Profile|)
5  )
6
7  (<=(isService ?service)
8     (|http://www.w3.org/1999/02/22-rdf-syntax-ns#|::|type|
9     ?service |http://www.daml.org/services/OWLS/1.1/
10    Service.owl#|::|Service|)
11 )
12
13 (<=(presentedBy ?profile ?service)
14    (|http://www.daml.org/services/OWLS/1.1/Service.owl
15    #|::|presentedBy| ?profile ?service)
16 )
```

Der Algorithmus basiert auf einer Regel, die nach und nach erweitert wird. Diese Regel ist eine Implikation der Form „wenn A dann B”. Die Nutzung dieser Regel bindet Service- und Profile-Instanzen, die eine Menge von Kriterien erfüllen, an zwei Variablen: $?Service$ und $?Profile$. Die linke Seite der Regel besteht aus: Name der Regel, in diesem Fall *getService*, und obengenannte Variablen (s. Ausschnitt 18).

Ausschnitt 18: Eine Regel zur Ermittlung von Serviceinstanzen

```
1  (<=(getService ?Profile ?Service)
2      (presentedBy ?Profile ?Service)
3  )
```

Die rechte Seite benutzt eine unbegrenzte Menge an Regeln und Axiomen. Die Axiome sind OWL-Konzepte und Relationen zwischen diesen Konzepten, die in OWL-Ontologien organisiert sind. Ein Axiom ist *presentedBy*, das in der Ontologie $ProcessModel$ (s. Kapitel 3.4.2) definiert ist. *presentedBy* ist eine Relation, die eine Instanz der Klasse Profile (Domäne) an eine Instanz der Klasse Service bindet (s. Kapitel 3.4.2). Ausschnitt 17 zeigt diese Regel. Die linke Seite der Regel *getService* besteht aus dieser Regel. Dies impliziert, dass die Variable $?Profile$ vom Typ $owls : Profile$ und die Variable $?Service$ vom Typ $owls : Service$ sind (s. Kapitel 3.4.2).

Die Regel *getService* würde in dieser Form alle Profile- und Service-Instanzen aus der Wissensbasis zurückliefern, weil keine Suchkriterien angegeben sind. Die Suchkriterien sind durch die IOPEs definiert, die nachfolgenden Regeln in Ausschnitt 19 zeigen die Definition von Regeln, welche die IOPEs ausdrücken. Die Zeilen 1-4 definieren eine Regel, die ein Axiom aus der Ontologie $owls : Profile$ benutzt. Dieses Axiom ist eine Relation zwischen einer Instanz der Klasse $Profile$ und einer Instanz der Klasse *Output* (s. Kapitel 3.4.1). Die Zeilen 6-9 definieren *Input*- und die Zeilen 11-15 *Local*- Parameter (s. Kapitel 3.4.1). Die Zeilen 17-21 benutzen das Attribut *parameterType* der Klasse $Parameter$, um den Typ einer Variable anzugeben (s. Kapitel 3.4.2). Der Typ einer Variable ist entweder ein OWL-Objekttyp (s. Kapitel 3.3.3) oder ein XML-Schema-Datentyp.

Ausschnitt 19: Parameter Definitionen

```
1  (<=(Output ?Profile ?Output )
2      (|http://www.daml.org/services/OWLS/1.1/
3      Profile.owl#|::|hasOutput| ?Profile ?Output)
4  )
5
6  (<=(Input ?Profile ?Input )
```

```
7     (|http://www.daml.org/services/OWLS/1.1/
8     Profile.owl#|::|hasInput| ?Profile ?Input)
9   )
10
11  (<=(Local ?Profile ?Local )
12    (|http://www.daml.org/services/OWLS/1.1/
13    Process.owl#|::|hasLocal|
14    ?Profile ?Local)
15  )
16
17  (<=(parameterType ?Parameter ?parameterType)
18    (|http://www.daml.org/services/OWLS/1.1/
19    Process.owl#|::|parameterType| ?Parameter
20    ?parameterType)
21  )
```

Die Regel *getService* kann jetzt um die Regeln in Ausschnitt 19 erweitert werden:

Ausschnitt 20: Erweiterte Regel zur Ermittlung von Serviceinstanzen

```
1   (<=(getService ?Profile ?Service)
2     (Input ?Profile ?teamName)
3     (parameterType ?teamName
4      "http://www.w3.org/2001/XMLSchema\#string")
5     (Local ?Profile ?Player)
6     (parameterType ?Player
7      "http://atradig82.informatik.tu-muenchen.de:
8      8280/cocoon/ontology/football.owl\#Player")
9     (Output ?Profile ?age)
10    (parameterType ?age
11     "http://www.w3.org/2001/XMLSchema\#int")
12    (Output ?Profile ?legueGoals)
13    (parameterType ?legueGoals
14     "http://www.w3.org/2001/XMLSchema\#int")
15    (Local ?Profile ?Team)
16    (parameterType ?Team
17     "http://atradig82.informatik.tu-muenchen.de:
18     8280/cocoon/ontology/football.owl\#Team")
19    (presentedBy ?Profile ?Service)
20  )
```

Diese Regel würde in dieser Form Service-Instanzen zurückliefern, die eine Variable vom Typ *xsd : string* als Eingabe, zwei lokale Variablen, eine vom Typ

Player und eine vom Typ *Team* sowie zwei Variablen von Typ *xsd* : *int*, haben. Das Verhältnis zwischen der Eingabe-, der Ausgabe- und den lokalen Variablen wird in Vor- und Nachbedingungen (s. Kapitel 3.4.2) definiert. Appendix F zeigt die entsprechenden Regeln in KIF-Syntax (s. Kapitel 3.5). Die Zeilen 28-32 zeigen eine Regel über Atome entsprechend SWRL-Definitionen (s. Seite 53). Die Zeilen 34-47 zeigen eine rekursive Regel, die prüft, ob ein Atom in einer Atomliste enthalten ist.

Es gibt 6 verschiedene Atome (s. Seite 53), jedes Atom besitzt ein Attribut (*predicate*) und eine unterschiedliche Anzahl an Argumenten. *ClassAtom* hat z.B. nur ein Argument, *DataValuedPropertyAtom* hingegen zwei (s. Seite 53). Die entsprechenden Regeln werden in Appendix E gezeigt.

Damit der Benutzer erfahren kann, in welchem Verhältnis die gefundenen Services zu den von ihm angegebenen Suchkriterien stehen, erweitern wir die Regel *getService* um die Variable *MatchType*. Diese Variable gibt an, welches Matching-Kriterium einer Service-Instanz entspricht. Dazu müssen Regeln definiert werden, die unterschiedliche Matching-Kriterien berechnen. Der Wertebereich der Variable *MatchType* liegt zwischen 0 und 2. 0 entspricht einer Äquivalenz, 1 besagt, dass die Suchmenge die Obermenge der Anzeigemenge ist. 2 bedeutet, dass die Suchmenge die Anzeigemenge subsumiert. Ausschnitt 21 zeigt vier Hilfsregeln. Die Zeilen 1-6 zeigen eine Regel, die für die Funktion *swrlb* : *lessThan* das Verhältnis zwischen dem Argument des Builtins (s. Seite 53) und dem Wert, der vom Benutzer angegeben ist, ausdrückt. Beispielsweise liefert ein Service Spieler zurück, die jünger als 25 sind und ein Suchkriterium, dass der Spieler jünger als 20 sein soll.

Die Zeilen 8-13 auf Ausschnitt 21 zeigen eine Regel für die Funktion *swrlb* : *greaterThan*. Die Zeilen 15-30 zeigen eine Regel, die durch die Klassenhierarchie der Anwendungsdomäne das Matching-Kriterium berechnet. Dazu werden die OWL-RDF-Schema-Konzepte benutzt. Für jedes Kriterium wird ein eigener *MatchType* berechnet. Zum Schluss werden alle *MatchType*s miteinander verglichen und der größte Wert drückt das Verhältnis der Suchmenge zu der Anzeigemenge aus. Diese Aufgabe übernimmt die Funktion *max*.

Ausschnitt 21: SWRL-Funktionen

```
1     (<=(lessThanMatchType ?arg1 ?arg2 ?MatchType1)
2           (or (and(&gt; ?arg1 ?arg2)(= ?MatchType1 1))
3                (and (&gt; ?arg2 ?arg1)(= ?MatchType1 2))
4                (and (= ?arg2 ?arg1)(= ?MatchType1 0))
5           )
6     )
7
```

```
 8     (<=(greaterThanMatchType ?arg1 ?arg2 ?MatchType1)
 9          (or (and(&gt; ?arg2 ?arg1)(= ?MatchType1 1))
10               (and (&gt; ?arg1 ?arg2)(= ?MatchType1 2))
11               (and (= ?arg2 ?arg1)(= ?MatchType1 0))
12          )
13     )
14
15     (<=(ClassMatchType ?class1 ?class2 ?param ?paramType
16               ?MatchType2)
17          (or
18               (and(parameterType ?param ?paramType)
19                    (= ?MatchType2 0)
20               )
21               (and (|http://www.w3.org/2000/01/rdf-schema#
22                    |::|subClassOf| ?class2 ?class1)
23                    (= ?MatchType2 2)
24               )
25               (and (|http://www.w3.org/1999/02/
26               22-rdf-syntax-ns#|::|type| ?class1 ?class2)
27                    (= ?MatchType2 1)
28               )
29          )
30     )
```

Appendix F Ausschnitt 28 zeigt die Regel *getService* für die Suche nach einem Service, der als Eingabe einen Parameter vom Typ String erhält, welcher dem Namen eines Vereins entspricht. Für jeden Spieler des Vereins, der jünger als 25 Jahre ist, soll der Service das Alter und die Anzahl der Ligatore zurückliefern.

4.3 Erfahrungen & Erkenntnisse

Die Reihenfolge der Regeln spielt im Bezug auf die Antwort keine Rolle, aber sie beeinflusst die Antwortzeiten. Der Grund hierfür liegt daran, dass Reasoner in der Regel Vorwärts- oder Rückwärtsverkettungen betreiben (s. Kapitel 3.6). Regeln, die die Menge der relevanten Suchergebnisse auf eine Anfrage stark beschränken, sollten bei Rückwärtsverkettung am Anfang der Kette stehen, andere hingegen am Ende der Verkettung. Die Regel *presentedBy* schränkt die Menge z.B. gar nicht ein, sondern legt die Instanz der Klasse Service fest. Diese Regel am Anfang der Verkettung zu schreiben, reduziert den Suchumfang nicht. Dies führt dazu, dass sich die Antwortzeit erhöht. Die Angabe der Vor- und Nachbedingungen am Anfang der Verkettung schränkt hingegen die Menge der relevanten Suchergebnisse sehr stark ein und führt zu einer kürzeren Antwortzeit. Die nachfolgende

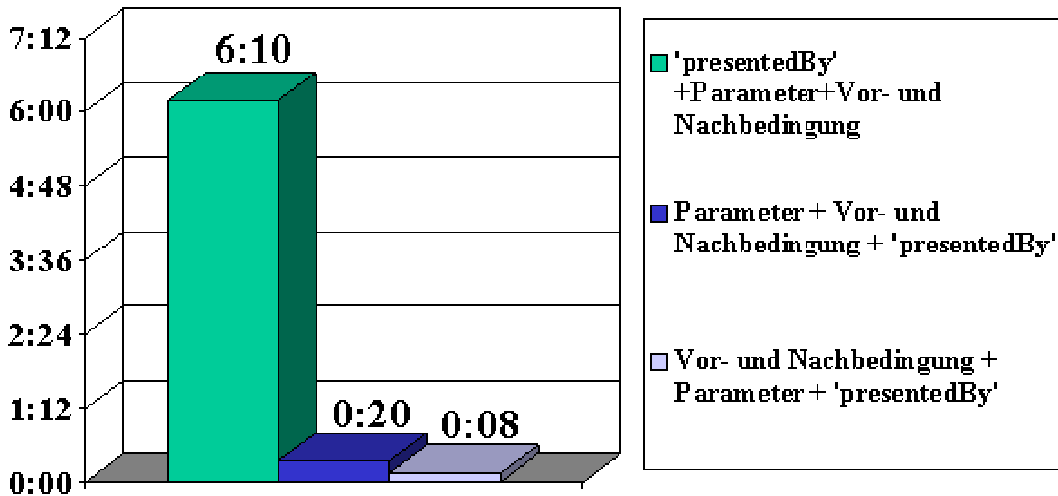


Abbildung 22: Antwortzeit in Stunden : Minuten bei verschiedenen Reihenfolgen der Regeln

Abbildung zeigt die Unterschiede zwischen den Antwortzeiten unterschiedlicher Verkettungen derselben Anfrage.
OWLS definiert eine Klasse *Parameter* zur Angabe von Serviceparametern. Diese Klasse besitzt ein Attribut *parameterType*, das dazu dient, den Typ der Service-Parameter anzugeben. Dieses Attribut ist vom Typ $xsd : anyURI$. JTP interpretiert dieses Attribut als eine Zeichenkette und nicht als Verweis auf eine OWL-Klasse. Diese Einschränkung führt dazu, dass keine Schlussfolgerung über die Position der Klasse in der Klassenhierarchie möglich ist. Dies macht es unmöglich, das Verhältnis zwischen der Anzeigemenge und der Antwortmenge genau zu bestimmen. Deswegen ist es sinnvoll, für Serviceparameter, die kein XML-Schema-Datentyp sind, ein $swrl : classAtom$ als Nachbedingung zur Angabe des Typs zu definieren.

5 Seitenblick

Autonome Servicekomposition ist eine der interessantesten Aufgaben im Bereich von semantisch beschriebenen Web Services, vor allem solche Kompositionen, die bedingte Ausdrücke enthalten. Fest beschriebene Kompositionen sind nicht so interessant wie solche, die auf Anfrage generiert werden. In diesem Abschnitt werden auf Anfrage generierte Kompositionen behandelt. Eine Komposition ist nichts anderes, als einen Plan zu finden, in dem mehrere kleinere Aufgaben (Web Service) zusammen eine komplexere Aufgabe (Komposition) erfüllen, eine komplexe Aufgabe ist eine Aufgabe, die nicht in einem Schritt erledigt werden kann. Das Kompositionsproblem kann als ein Planungsproblem in KI (Künstliche Intelligenz) angesehen werden.

Automatisierte Komposition von semantisch beschriebenen Web Services kann durch die Nutzung von KI - Techniken erreicht werden. „Hierarchical Task Network" (HTN) ist zu diesem Zweck geeignet. SHOP2 ist ein HTN-Planungssystem, das mit OWLS- Beschreibungen benutzt werden kann.

SHOP2 ist ein domänenunabhängiges HTN-Planungssystem. HTN ist eine Planungsmethodologie, die Pläne durch Aufgabendekomposition kreiert. Das Ziel der HTN-Planer ist die Produktion einer Sequenz von Aktionen, die eine Aufgabe erfüllt. Die Beschreibung einer Planungsdomäne enthält eine Liste von Operatoren, ähnlich wie in der klassischen Planung, und eine Liste von Methoden. Jede Methode ist eine Vorschrift, wie eine Aufgabe in kleinere Aufgaben zerlegt werden kann. Die Erstellung eines Plans wird durch die rekursive Zerlegung von Aufgaben in kleinere Aufgaben erreicht, bis der Planer nur noch primitive Aufgaben vorfindet. Primitive Aufgaben werden durch einen Planungsoperator in einem Schritt ausgeführt.

Ein Unterschied zwischen SHOP2 und den meisten HTN-Planungssystemen ist, dass SHOP2 Aufgaben in derselben Reihenfolge plant, wie sie später ausgeführt werden. Dies macht es möglich, in jedem Schritt des Planungsprozesses den aktuellen Zustand der Welt zu kennen. Dadurch kann der Mechanismus von SHOP2 zur Evaluierung von Vorbedingungen eine beträchtliche Schlussfolgerungsfähigkeit erreichen und externe Programme aufrufen.

SHOP2 benötigt Wissen über eine Domäne, damit es darin planen kann. Die Wissensbasis von SHOP2 enthält Operatoren und Methoden (plus für keine Aktion relevante Fakten und Axiome). Jeder Operator ist eine Beschreibung, was getan werden muss, um eine primitive Aufgabe auszuführen. Jede Methode beschreibt, wie eine komplexe Aufgabe in partiell geordnete Teilaufgaben zerlegt werden kann.

Definition 1 (Operator) *Ein SHOP2-Operator ist ein Ausdruck der Form* $(h(\overrightarrow{v})$ *Pre Del Add), wobei:*

- $h(\overrightarrow{v})$ *ist eine primitive Aufgabe mit einer Liste von Eingabeparametern* $\overrightarrow{v}$.
- ***Pre*** *repräsentiert die Vorbedingungen des Operators.*
- ***Del*** *ist die Lösch-Liste des Operators, die Dinge enthält, welche nach der Ausführung des Operators falsch werden.*
- ***Add*** *repräsentiert die Add-Liste des Operators, welche Dinge enthält, die nach der Ausführung des Operators wahr werden.*

Vorbedingungen enthalten logische Atome mit Variablen, die entweder in h enthalten oder existentiell quantifiziert sind. Logische Atome können durch logische Konnektive wie Konjunktion, Disjunktion, Negation, Implikation und Allquantor kombiniert werden. *Add*- und *Del*-Listen sind generell als Konjunktion von logischen Atomen definiert, aber bedingte Ausdrücke und Allquantoren können auch benutzt werden.

Definition 2 (Methode) *Eine SHOP2-Methode ist ein Ausdruck der Form* $(h(\overrightarrow{v})\ Pre_1\ T_1\ Pre_2\ T_2\ ....)$, *wobei:*

- $h(\overrightarrow{v})$ *ist eine kombinierte Aufgabe mit einer Liste von Ausgabeparametern.* $\overrightarrow{v}$
- *Jeder* Pre_i *ist ein Vorbedingungsausdruck.*
- *Jede* T_i *ist eine partiell geordnete Liste von Teilaufgaben.*

Die Bedeutung dieser Definition ist analog zu bedingten Ausdrücken. Sie sagt SHOP2: Wenn Bedingung Pre_1 erfüllt ist, dann benutze T_1, andernfalls wenn Pre_2 erfüllt ist, dann benutze T_2 usw. Eine Aufgabenliste enthält atomare Aufgaben und andere Aufgabenlisten. Eine Aufgabenliste kann *geordnet* oder *ungeordnet* definiert werden. Die Aufgaben in einer *geordnet*en Liste sollen sequentiell erledigt werden, während Aufgaben in einer *ungeordnet*en Liste in jeder Reihenfolge erledigt werden können. Verknüpfungen von *geordnet*en und *ungeordnet*en Aufgabenlisten können benutzt werden, um komplexere Ordnungseinschränkungen zu erreichen. Eine Aufgabe repräsentiert selber eine auszuführende Aktivität, die entweder primitiv oder komplex ist. Eine primitive Aufgabe kann durch einen Planungsoperator erreicht werden. Eine komplexe Aufgabe muss durch Methoden in kleinere Teilaufgaben zerlegt werden. Es kann mehrere Methoden geben, die diese Aufgabe zerlegen. SHOP2 versucht jede Zerlegung und wählt eine andere Komposition, falls eine scheitert.

Zusätzlich zu den logischen Atomen, Vorbedingungen von SHOP2, Methoden und Operatoren können Aufrufe zu externen Programmen und Variablenzuweisungen existieren. Dies ist nützlich, um Planungen mit Anfragen an Informationsquellen über das WWW zu integrieren. Der Ausdruck

$$(\text{assign } v \ (\text{call } f \, t_1 \ t_2....t_n))$$

verbindet z.B. die Variable v mit dem Ergebnis des Aufrufs der externen Prozedur f mit den Argumenten $t_1 \ t_2....t_n$.

Definition 3 (Planungsproblem)*Ein SHOP2-Planungsproblem ist ein Tripel (S, T, D), wobei S der Initialzustand, T eine Aufgabenliste und D eine Domänenbeschreibung ist. Bei Eingabe (S, T, D), liefert SHOP2 einen Plan P =* $(p_1 p_2....p_n)$*, dies ist eine Sequenz initialisierter Operatoren zum Lösen von T, ausgehend von S in D*[31].

Definition 3 stellt ein *vollständig informiertes Planungsproblem* dar. Wenn Informationen über den Initialzustand der Welt oder weitere Zustände im Suchbaum nicht vollständig vorhanden sind, ist das Planungsproblem wie folgt definiert: Ein unvollständig informiertes Planungsproblem ist definiert als ein Tupel P^I = (J, X, T, D), J ist eine Menge von Grundatomen, die zu Beginn bekannt sind. X ist eine Liste von Informationsquellen, T ist eine komplexe Aufgabe und D ist eine HTN-Domänenbeschreibung[32].

Algorithmen zur Übersetzung von OWLS-Services in die SHOP2-Notation sind in [31] und [32] beschrieben. Z.B Ein OWLS-Atomic-Process wird als ein SHOP2-Operator interpretiert. Ein OWLS-Composite-Process als eine SHOP2-Methode usw.

Beispiel 11: **Vorbereitungen für eine Reise treffen:**

1. Den Transport reservieren.
   (a) Finde Transport-Services
   (b) filtere die Services heraus, die keinen Transport im gewünschten Datum bieten
   (c) Wähle einen Service aus, der deine Kreditkarte akzeptiert und einen guten Preis bietet
2. Hotel Reservierung (Füttere diesen Service mit Daten (z.B Ankunftsdatum) aus dem vorigen Service)
   .....
3. Trage Kosten in deinen finanziellen Organisierer ein (berechne die gesamte Kosten der letzten Schritte)

6 Ausblick

Die Suche nach Web Services, die semantisch beschrieben sind, involviert mehrere Parteien und ist wesentlich komplexer als bei konventionellen Web Services. Deswegen sind die Antwortzeiten in der Regel höher. Eine mögliche Lösung dieses Problems ist die Verteilung der Last auf mehrere Knoten in einem Netzwerk. Jeder Knoten soll einen Teil der Aufgabe übernehmen und Services nach kleineren Kriterien suchen. Ein Verteilerkonto nimmt eine Anfrage entgegen, zerlegt sie und verteilt die Teilaufgaben auf andere Knoten des Netzwerks. Jeder Knoten schickt eine Menge an relevanten Services zurück und der Verteilerknoten wählt Web Services aus, die von jedem Knoten zurückgeschickt wurden.

Jeder Knoten soll Wissen über andere Knoten besitzen. Wenn ein Knoten ausfällt, soll die Teilaufgabe an einen anderen Knoten geschickt werden. Um zu vermeiden, dass das System total ausfällt, soll jeder Knoten in der Lage sein, als Verteilerknoten zu fungieren.

Ähnlich wie bei UDDI soll es universelle Server geben, bei denen Verteilerknoten registriert sind, damit Benutzer diese finden können, oder wenn ein Verteilerknoten eine Anfrage nicht beantworten kann, sucht er bei einem universellen Server nach anderen Netzwerken, um die Anfrage zu beantworten.

7 Zusammenfassung

Die Kommunikation zwischen Rechnern im verteilten Systemen wurde durch den Einsatz von Middleware-Technologien ermöglicht. Eine bekannte Middleware im B2B-Bereich ist *CORBA*. Web Service Technologien wurden entwickelt um die Kommunikation über Unternehmens- und Organisationsgrenzen hinaus zu ermöglichen, sodass B2B-Strategien besser durchgesetzt werden können.

Web Service Technolgien erreichen die Interoperabilität zwichen unterschiedlichen Systemen durch eine sprachneutrale und plattformunanhängige Interaktion. Sie bieten Vokabularien zur Beschreibung von Web Services auf einer syntaktischen Ebene, sodass diese Web Services veröffentlicht, gesucht und ausgeführt werden können. Beschreibungen auf einer syntaktischen Ebene bieten keine maschineninterpretierbare Informationen, sodass Maschinen Web Service automatisch suchen, kombinieren und ausführen können.

Semantic Web ist eine Erweiterung der gegenwärtigen Form des Webs, die Information mit einer wohldefinierten Bedeutung versieht um die verbesserte Zusammenarbeit zwischen Mensch und Computer zu ermöglichen. Ziel des semantischen Webs ist es WWW-übertragene Daten durch Menschen mit Semantiken anzureichern für die Verarbeitung durch Maschinen und Nutzung durch Menschen. Semantic Web bietet Mechanismen zur Konzeptualisierung von Anwendungsdomänen. Die Konzepte, Attribute dieser Konzepte und die Relationen zwischen diesen Konzepten werden auf einer semantischen Ebene beschrieben.

Senamtisch beschriebene Web Services verbinden Web Service Technologien und das Semantic Web. Ein semantisch beschriebener Web Service kann automatisch durch Agenten gesucht und ausgeführt werden. Falls kein Service gefunden wird, bieten semantisch beschriebene Web Services die möglichkeit, dass ein Agent prüfen kann, ob eine Kombination von mehreren Services die gesuchte Funktionalität bieten kann. Diese Funktionalitäten von semantisch beschriebenen Web Services, nämlich autonome Web Service Suche, Ausführung, Komposition wurden in dieser Arbeit an Hand einer Fallstudie vorgestellt.

8 Screenshots

Nachfolgend sind einige Screenshots, die das Programm bildlich darstellen.

Fussball-Service-Portal

Ermittelt Spieler eines Vereins, die junger als 25 Jahre sind	Logiksyntax
Ermittelt alle Spieler eines Vereins	Logiksyntax
Ermittelt Name, Vereinsname und Anzahl der Ligatore eines Spielers	Logiksyntax
Ermittelt Informationen über alle Spieler eines Vereins	Logiksyntax

Suche nach geeigneten Services

Suchformular
Beispielesuche (nur Exact match)
Beispielesuche mit match type
Semantic web services in Logik Syntax
XSL-Datei kommentiert
Query-Datei kommentiert für das obige Beispiel
Services registrienen

Abbildung 23: Hauptseite der Fussballdomäne

Inputs

Parameter	Parameter Type	Type	Parameter Value
?teamName	Input	xsd.string	
?Player	Parameter	Player	
?age	Output	xsd.int	
?legueGoals	Output	xsd.int	
?Team	Parameter	Team	

Preconditions

Atom Type	Predicate/Builtin	Argument1	Argument2
DatavaluedPropertyAtom	teamName	?Team	?teamName

Effects

Atom Type	Predicate/Builtin	Argument1	Argument2
BuiltinAtom	swrlb.lessThan	?age	20
DatavaluedPropertyAtom	age	?Player	?age
ClassAtom	Striker	?Player	

Global Ontology

Ontology
http://atradig82.informatik.tu-muenchen.de:8280/cocoon/ontology/football.owl

Cancel these values | Submit

Abbildung 24: Suche nach einem Service, der alle Stürmer eines Vereins zurückliefert, die jünger als 20 Jahre sind

Semantic Web Services

Service	Profile	Match Type	Execute
PlayerAgeService	PlayerAgeProfile	2	Execute This Service

Abbildung 25: Ermittlung der Services, die die Anfrage auf dem letzten Bild beantworten

Abbildung 26: alle Konzepte der Fußballdomäne

Semantic Web Services Execution

teamName FC Bayern Muenchen

Cancel these values Submit

Abbildung 27: Parametereingabe zur Ausführung der ermittelten Services

Ausschnitt 22: Ergebnis der Serviceausführung

```
1 <?xml version="1.0" encoding="ISO-8859-1"?><result
2 xmlns:owql="http://maria_sharapova.html"
3 xmlns:soccer="http://atradig82.informatik.tu-muenchen.de
     :8280/cocoon/ontology/football.owl"
4 xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
5 xmlns:owl="http://www.w3.org/2002/07/owl#"
6 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
7 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
8 xmlns:tkb="http://www.daml.org/2001/03/daml+oil-ex#"
9 xmlns:iw="http://www.ksl.stanford.edu/software/IW/spec/iw
     .daml#"
10 xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
11 xmlns:owl-ql="http://www.w3.org/2003/10/owl-ql-syntax#"
12 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope
     /"
13 xmlns="http://www.w3.org/1999/xhtml"><SOAP-ENV:Envelope
14 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/
     encoding/">
15 <SOAP-ENV:Body> <owl-ql:answerBundle>
16  <owl-ql:queryPattern><owl-ql:KIF>
17 (playsForTeam ?Player ?playsFor "FCBayernMuenchen" ?
     lastName
18 ?leagueGoals ?firstName) </owl-ql:KIF></owl-ql:
     queryPattern>
19  <owl-ql:answer>
20    <owl-ql:binding-set>
21      <var:Player rdf:resource="http://atradig82.
          informatik.tu-muenchen.de:8280/cocoon/ontology/
          footballInstances.owl#bastian-schweinsteiger"/>
22      <var:lastName rdf:resource="Schweinsteiger"/>
```

```
23         <var:firstName rdf:resource="Bastian"/>
24         <var:playsFor rdf:resource="http://atradig82.
               informatik.tu-muenchen.de:8280/cocoon/ontology/
               footballInstances.owl#fc-bayern-muenchen"/>
25         <var:leagueGoals rdf:resource="2"/>
26      </owl-ql:binding-set>
27      <owl-ql:answerPatternInstance>
28
29 (playsForTeam
30 http://atradig82.informatik.tu-muenchen.de:8280/cocoon/
      ontology/footballInstances.owl#bastian-schweinsteiger
31 http://atradig82.informatik.tu-muenchen.de:8280/cocoon/
      ontology/footballInstances.owl#fc-bayern-muenchen
32 "FCBayernMuenchen" Schweinsteiger 2 Bastian)
33
34      </owl-ql:answerPatternInstance>
35   </owl-ql:answer>
36   <owl-ql:continuation>
37      <owl-ql:termination-token>
38         <owl-ql:end/>
39      </owl-ql:termination-token>
40   </owl-ql:continuation>
41 </owl-ql:answerBundle> </SOAP-ENV:Body>
42 </SOAP-ENV:Envelope></result>
```

A Vorbedingung

Ausschnitt 23: Preconditions eines Services

```
1
2  <expr:SWRL-Condition rdf:ID="TeamCondition">
3    <expr:expressionBody>
4      <swrl:AtomList rdf:ID="AtomList_4">
5        <rdf:first>
6          <swrl:DatavaluedPropertyAtom
7          rdf:ID="DataAtom_3">
8            <swrl:argument2 rdf:resource="#
9            teamName"/>
10           <swrl:argument1 rdf:resource="#Team"/>
11           <swrl:propertyPredicate rdf:resource=
12           "http://atradig82.informatik.tu-
13           muenchen.de:8280/cocoon/ontology/
14           football.owl#teamName"/>
15         </swrl:DatavaluedPropertyAtom>
16       </rdf:first>
17       <rdf:rest>
18     <swrl:AtomList rdf:ID="AtomList_24">
19       <rdf:first>
20         <swrl:DatavaluedPropertyAtom
21         rdf:ID="DataAtom_4">
22           <swrl:argument2 rdf:resource="#Team"/>
23           <swrl:argument1 rdf:resource="#Player"/>
24           <swrl:propertyPredicate rdf:resource=
25           "http://atradig82.informatik.tu-
26           muenchen.de:8280/cocoon/ontology/
27           football.owl#playsFor"/>
28         </swrl:DatavaluedPropertyAtom>
29       </rdf:first>
30       <rdf:rest rdf:resource="http://www.w3.org/
31       1999/02/22-rdf-syntax-ns#nil"/>
32     </swrl:AtomList>
33       </rdf:rest>
34     </swrl:AtomList>
35   </expr:expressionBody>
36   <expr:expressionLanguage rdf:resource="http://
37   www.daml.org/services/OWLS/1.1/generic/
38   Expression.owl#SWRL"/>
39 </expr:SWRL-Condition>
```

B Nachbedingung

Ausschnitt 24: Effects eines Services

```
1
2  <expr:SWRL-Condition rdf:ID="ResultEffect">
3    <expr:expressionLanguage rdf:resource="http://
4    www.daml.org/services/OWLS/1.1/generic/
5    Expression.owl#SWRL"/>
6    <expr:expressionBody>
7      <swrl:AtomList rdf:ID="AtomList_1">
8        <rdf:first>
9         <swrl:DatavaluedPropertyAtom
10        rdf:ID="DataAtom_1">
11          <swrl:propertyPredicate rdf:resource=
12          "http://atradig82.informatik.tu-muenchen
13          .de:8280/cocoon/ontology/football.owl#
14          age"/>
15          <swrl:argument2 rdf:resource="#age"/>
16          <swrl:argument1 rdf:resource="#Player"/>
17        </swrl:DatavaluedPropertyAtom>
18       </rdf:first>
19       <rdf:rest>
20        <swrl:AtomList rdf:ID="AtomList_2">
21         <rdf:rest>
22          <swrl:AtomList rdf:ID="AtomList_3">
23           <rdf:rest rdf:resource="http://www.w3.org
24           /1999/02/22-rdf-syntax-ns#nil"/>
25            <rdf:first>
26             <swrl:BuiltinAtom rdf:ID="Builtin_1">
27              <swrl:builtin rdf:resource="http://www.
28              w3.org/2003/11/swrlb#lessThan"/>
29               <swrl:arguments>
30                <rdf:List rdf:ID="List_1">
31                 <rdf:first rdf:resource="#age"/>
32                 <rdf:rest>
33                  <rdf:List rdf:ID="List_2">
34                   <rdf:rest rdf:resource="http://www.
35                   w3.org/1999/02/22-rdf-syntax-ns#
36                   nil"/>
37                   <rdf:first rdf:datatype="http://www.
38                   w3.org/2001/XMLSchema#int"
39                   >25</rdf:first>
```

```
40                          </rdf:List>
41                         </rdf:rest>
42                        </rdf:List>
43                       </swrl:arguments>
44                      </swrl:BuiltinAtom>
45                     </rdf:first>
46                    </swrl:AtomList>
47                   </rdf:rest>
48                  <rdf:first>
49                   <swrl:ClassAtom rdf:ID="ClassAtom_1">
50                    <swrl:classPredicate rdf:resource="http://
51                    atradig82.informatik.tu-muenchen.de:8280/
52                    cocoon/ontology/football.owl#Player"/>
53                    <swrl:argument1 rdf:resource="#Player"/>
54                   </swrl:ClassAtom>
55                  </rdf:first>
56                 </swrl:AtomList>
57               </rdf:rest>
58            </swrl:AtomList>
59         </expr:expressionBody>
60      </expr:SWRL-Condition>
```

C Composite Process

Ausschnitt 25: Servicekomposition

```
1
2  <process:InputBinding rdf:ID="InputBinding_4">
3    <process:valueSource>
4      <process:ValueOf rdf:ID="ValueOf_5">
5        <process:fromProcess rdf:resource="http://www.daml
             .org/services/OWLS/1.1/Process.owl#
             TheParentPerform"/>
6      </process:ValueOf>
7    </process:valueSource>
8    <process:toParam rdf:resource="http://atradig82.
         informatik.tu-muenchen.de:8280/cocoon/halgurt/
         profile/PlayerTeamProfile.owl#teamName"/>
9  </process:InputBinding>
10
11 <process:ValueOf rdf:ID="ValueOf_6">
```

```
12     <process:fromProcess rdf:resource="http://www.daml.org
          /services/OWLS/1.1/Process.owl#TheParentPerform"/>
13     <process:theVar>
14       <process:Input rdf:ID="teamName"/>
15     </process:theVar>
16   </process:ValueOf>
17
18
19   <process:InputBinding rdf:ID="InputBinding_9">
20     <process:toParam rdf:resource="http://atradig82.
          informatik.tu-muenchen.de:8280/cocoon/halgurt/
          processModel/PlayerGoalProcessModel.owl#PlayerName
          "/>
21     <process:valueSource>
22       <process:ValueOf rdf:ID="ValueOf_10">
23         <process:fromProcess>
24           <process:Perform rdf:ID="Perform_2">
25             <process:process rdf:resource="http://
                  atradig82.informatik.tu-muenchen.de:8280/
                  cocoon/halgurt/processModel/
                  PlayerTeamProcessModel.owl#
                  PlayerTeamAtomicProcess"/>
26             <process:hasDataFrom>
27               <process:InputBinding rdf:ID="InputBinding_5
                    ">
28                 <process:toParam rdf:resource="http://
                      atradig82.informatik.tu-muenchen.de
                      :8280/cocoon/halgurt/processModel/
                      PlayerTeamProcessModel.owl#teamName"/>
29                 <process:valueSource rdf:resource="#
                      ValueOf_6"/>
30               </process:InputBinding>
31             </process:hasDataFrom>
32           </process:Perform>
33         </process:fromProcess>
34         <process:theVar rdf:resource="http://atradig82.
              informatik.tu-muenchen.de:8280/cocoon/halgurt/
              processModel/PlayerTeamProcessModel.owl#
              PlayerName"/>
35       </process:ValueOf>
36     </process:valueSource>
37   </process:InputBinding>
38
```

```
39 <process:CompositeProcess rdf:ID="
     PlayerTotalInformationCompositeProcess">
40   <process:hasResult rdf:resource="#TotalInfResult"/>
41   <process:hasInput rdf:resource="#teamName"/>
42   <process:hasOutput rdf:resource="#NameOfTeam"/>
43   <process:hasOutput rdf:resource="#leagueGoal"/>
44   <process:composedOf>
45     <process:Sequence rdf:ID="Sequence_1">
46       <process:components>
47         <process:ControlConstructList rdf:ID="
             ControlConstructList_3">
48           <list:first rdf:resource="#Perform_2"/>
49           <list:rest>
50             <process:ControlConstructList rdf:ID="
                 ControlConstructList_8">
51               <list:rest rdf:resource="http://www.daml.
                   org/services/OWLS/1.1/generic/
                   ObjectList.owl#nil"/>
52               <list:first>
53                 <process:Perform rdf:ID="Perform_7">
54                   <process:process rdf:resource="http://
                       atradig82.informatik.tu-muenchen.de
                       :8280/cocoon/halgurt/processModel/
                       PlayerGoalProcessModel.owl#
                       GoalAtomicProcess"/>
55                   <process:hasDataFrom rdf:resource="#
                       InputBinding_9"/>
56                 </process:Perform>
57               </list:first>
58             </process:ControlConstructList>
59           </list:rest>
60         </process:ControlConstructList>
61       </process:components>
62     </process:Sequence>
63   </process:composedOf>
64   <process:hasOutput rdf:resource="#Team"/>
65   <process:hasPrecondition rdf:resource="#
       TotalInfPrecondition"/>
66   <process:hasOutput rdf:resource="#Player"/>
67   <process:hasOutput rdf:resource="#PlayerName"/>
68   <service:describes rdf:resource="http://atradig82.
       informatik.tu-muenchen.de:8280/cocoon/halgurt/
       service/PlayerTotalInformationService.owl#
```

```
        PlayerTotalInformationService"/>
69  </process:CompositeProcess>
70</rdf:RDF>
```

D Grounding

Ausschnitt 26: Service Grounding

```
1<rdf:RDF>
2  <grounding:WsdlOutputMessageMap rdf:ID="
      WsdlOutputMessageMap_5">
3    <grounding:owlsParameter rdf:resource="http://
        atradig82.informatik.tu-muenchen.de:8280/cocoon/
        halgurt/profile/PlayerAgeProfile.owl#Team"/>
4    <grounding:wsdlMessagePart rdf:datatype="http://www.w3
        .org/2001/XMLSchema#anyURI"
5    >http://atradig82.informatik.tu-muenchen.de:8280/
        cocoon/halgurt/wsdl/PlayerAgeService.wsdl#Team</
        grounding:wsdlMessagePart>
6  </grounding:WsdlOutputMessageMap>
7
8  <grounding:WsdlOperationRef rdf:ID="AgeWsdlOperationRef
      ">
9    <grounding:portType rdf:datatype="http://www.w3.org
        /2001/XMLSchema#anyURI"
10   >http://atradig82.informatik.tu-muenchen.de:8280/
        cocoon/halgurt/wsdl/PlayerAgeService.wsdl#
        PlayerPortType</grounding:portType>
11   <grounding:operation rdf:datatype="http://www.w3.org
        /2001/XMLSchema#anyURI"
12   >http://atradig82.informatik.tu-muenchen.de:8280/
        cocoon/halgurt/wsdl/PlayerAgeService.wsdl#
        getPlayerYoungerThan25</grounding:operation>
13 </grounding:WsdlOperationRef>
14
15 <grounding:WsdlGrounding rdf:ID="AgeWSDLGrounding">
16   <grounding:hasAtomicProcessGrounding>
17     <grounding:WsdlAtomicProcessGrounding rdf:ID="
          AgeWsdlAtomicProcessGrounding">
18       <grounding:wsdlOperation rdf:resource="#
            AgeWsdlOperationRef"/>
```

```
19          <grounding:wsdlOutput rdf:resource="#
                WsdlOutputMessageMap_5"/>
20          <grounding:wsdlOutputMessage rdf:datatype="http://
                www.w3.org/2001/XMLSchema#anyURI"
21          >http://atradig82.informatik.tu-muenchen.de:8280/
                cocoon/halgurt/wsdl/PlayerAgeService.wsdl#
                PlayerResponse</grounding:wsdlOutputMessage>
22          <grounding:wsdlOutput>
23            <grounding:WsdlOutputMessageMap rdf:ID="
                  WsdlOutputMessageMap_6">
24              <grounding:wsdlMessagePart rdf:datatype="http
                    ://www.w3.org/2001/XMLSchema#anyURI"
25              >http://atradig82.informatik.tu-muenchen.de
                    :8280/cocoon/halgurt/wsdl/PlayerAgeService.
                    wsdl#leagueGoal</grounding:wsdlMessagePart>
26              <grounding:owlsParameter rdf:resource="http://
                    atradig82.informatik.tu-muenchen.de:8280/
                    cocoon/halgurt/profile/PlayerAgeProfile.owl
                    #leagueGoal"/>
27            </grounding:WsdlOutputMessageMap>
28          </grounding:wsdlOutput>
29          <grounding:wsdlOutput>
30            <grounding:WsdlOutputMessageMap rdf:ID="
                  WsdlOutputMessageMap_1"/>
31          </grounding:wsdlOutput>
32          <grounding:wsdlOutput>
33            <grounding:WsdlOutputMessageMap rdf:ID="
                  WsdlOutputMessageMap_8"/>
34          </grounding:wsdlOutput>
35          <grounding:wsdlPort rdf:datatype="http://www.w3.
                org/2001/XMLSchema#anyURI"
36          >informatik.tu-muenchen.de#PlayerPortType</
                grounding:wsdlPort>
37          <grounding:wsdlInput>
38            <grounding:WsdlInputMessageMap rdf:ID="
                  WsdlInputMessageMap_4">
39              <grounding:wsdlMessagePart rdf:datatype="http
                    ://www.w3.org/2001/XMLSchema#anyURI"
40              >http://atradig82.informatik.tu-muenchen.de
                    :8280/cocoon/halgurt/wsdl/PlayerAgeService.
                    wsdl#teamName</grounding:wsdlMessagePart>
41            </grounding:WsdlInputMessageMap>
42          </grounding:wsdlInput>
```

```
43          <grounding:wsdlOutput rdf:resource="#
               WsdlOutputMessageMap_4"/>
44          <grounding:wsdlOutput>
45            <grounding:WsdlOutputMessageMap rdf:ID="
                 WsdlOutputMessageMap_7"/>
46          </grounding:wsdlOutput>
47          <grounding:wsdlService rdf:datatype="http://www.w3
               .org/2001/XMLSchema#anyURI"
48          >http://atradig82.informatik.tu-muenchen.de:8280/
               cocoon/halgurt/services/playerAge/playerAge</
               grounding:wsdlService>
49          <grounding:wsdlDocument rdf:datatype="http://www.
               w3.org/2001/XMLSchema#anyURI"
50          >http://atradig82.informatik.tu-muenchen.de:8280/
               cocoon/halgurt/wsdl/PlayerAgeService.wsdl</
               grounding:wsdlDocument>
51          <grounding:wsdlInputMessage rdf:datatype="http://
               www.w3.org/2001/XMLSchema#anyURI"
52          >http://atradig82.informatik.tu-muenchen.de:8280/
               cocoon/halgurt/wsdl/PlayerAgeService.wsdl#
               PlayerRequest</grounding:wsdlInputMessage>
53          <grounding:owlsProcess rdf:resource="http://
               atradig82.informatik.tu-muenchen.de:8280/cocoon
               /halgurt/processModel/PlayerAgeProcessModel.owl
               #AgeAtomicProcess"/>
54        </grounding:WsdlAtomicProcessGrounding>
55      </grounding:hasAtomicProcessGrounding>
56      <service:supportedBy rdf:resource="http://atradig82.
           informatik.tu-muenchen.de:8280/cocoon/halgurt/
           service/PlayerAgeService.owl#PlayerService"/>
57    </grounding:WsdlGrounding>
58
59    <grounding:WsdlInputMessageMapList rdf:ID="
         AgeWsdlInputMessageMapList"/>
60    <grounding:WsdlOutputMessageMapList rdf:ID="
         AgeWsdlOutputMessageMapList">
61      <list:first rdf:resource="#WsdlOutputMessageMap_6"/>
62      <list:first rdf:resource="#WsdlOutputMessageMap_8"/>
63      <list:rest rdf:resource="#AgeWsdlOutputMessageMapList
           "/>
64      <list:first rdf:resource="#WsdlOutputMessageMap_4"/>
65      <list:first rdf:resource="#WsdlOutputMessageMap_7"/>
66      <list:first rdf:resource="#WsdlOutputMessageMap_5"/>
```

```
67   </grounding:WsdlOutputMessageMapList>
68</rdf:RDF>
```

E WSDL-Grounding

Ausschnitt 27: WSDL-Grounding

```
1<?xml version="1.0" encoding="UTF-8"?>
2 <wsdl:definitions name="PlayerService"
3     targetNamespace="http://atradig82.informatik.tu-
        muenchen.de:8280/cocoon/halgurt/wsdl/
        PlayerAgeService.wsdl"
4     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
5     xmlns:playerProcess="http://atradig82.informatik.tu-
        muenchen.de:8280/cocoon/halgurt/processModel/
        PlayerAgeProcessModel.owl"
6     xmlns:tns="http://atradig82.informatik.tu-muenchen.de
        :8280/cocoon/halgurt/wsdl/PlayerAgeService.wsdl"
7     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
8     xmlns:OWLS-wsdl="http://www.daml.org/services/OWLS/
        wsdl/"
9     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
10    <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap
        .org/wsdl/">Created using Cape Clear Studio SOA
        Editor - http://www.capeclear.com</wsdl:
        documentation>
11    <wsdl:types>
12        <xsd:schema
13            targetNamespace="http://atradig82.informatik.
                tu-muenchen.de:8280/cocoon/halgurt/wsdl/
                footballService.xsd1"
14            xmlns:SOAP-ENC="http://schemas.xmlsoap.org/
                soap/encoding/"
15            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
16            xmlns:xsd1="http://atradig82.informatik.tu-
                muenchen.de:8280/cocoon/halgurt/wsdl/s.xsd1
                ">
17            <xsd:complexType name="Player">
18                <xsd:sequence>
19                    <xsd:element maxOccurs="1" minOccurs
                        ="0" name="firstName" type="xsd:
                        string"/>
```

```
20                     <xsd:element maxOccurs="1" minOccurs
                          ="0" name="leagueGoals" type="xsd:
                          unsignedInt"/>
21                     <xsd:element maxOccurs="1" minOccurs
                          ="1" name="lastName" type="xsd:
                          string"/>
22                     <xsd:element maxOccurs="1" minOccurs
                          ="1" name="age" type="xsd:
                          unsignedInt"/>
23                 </xsd:sequence>
24             </xsd:complexType>
25             <xsd:complexType name="PlayerList">
26                 <xsd:sequence>
27                     <xsd:element maxOccurs="unbounded"
                          minOccurs="0" name="Player" type="
                          Player"/>
28                 </xsd:sequence>
29             </xsd:complexType>
30         </xsd:schema>
31     </wsdl:types>
32     <wsdl:message name="PlayerRequest">
33         <wsdl:part name="teamname" element="tns:TeamName"
              OWLS-wsdl:OWLS-parameter="playerProcess:#
              teamName"/>
34     </wsdl:message>
35     <wsdl:message name="PlayerResponse">
36         <wsdl:part name="PlayerSequence" element="tns:
              PlayerList"/>
37     </wsdl:message>
38     <wsdl:portType name="PlayerPortType">
39         <wsdl:operation name="getPlayerYoungerThanAnAge"
              OWLS-wsdl:OWLS-process="playerProcess:#
              AgeAtomicProcess">
40             <wsdl:input message="tns:PlayerRequest"/>
41             <wsdl:output message="tns:PlayerResponse"/>
42         </wsdl:operation>
43     </wsdl:portType>
44     <wsdl:binding name="PlayerBinding" type="tns:
          PlayerPortType">
45         <soap:binding style="rpc" transport="http://
              schemas.xmlsoap.org/soap/http"/>
46         <wsdl:operation name="getPlayerYoungerThan25">
47             <soap:operation
```

```
48                    soapAction="capeconnect:PlayerService:
                         PlayerPortType#getPlayerYoungerThan25
                         "/>
49                   <wsdl:input name="PlayerRequest">
50          <soap:body encodingStyle="http://www.w3.org
               /2002/07/owl" parts="teamname" use="encoded"
               namespace="http://atradig82.informatik.tu-
               muenchen.de:8280/cocoon/ontology/football.owl
               "/>
51          </wsdl:input>
52          <wsdl:output name="PlayerResponse">
53          <soap:body encodingStyle="http://www.w3.org
               /2002/07/owl" parts="PlayerSequence" use="
               encoded" namespace="http://atradig82.informatik
               .tu-muenchen.de:8280/cocoon/ontology/football.
               owl"/>
54          </wsdl:output>
55      </wsdl:operation>
56      </wsdl:binding>
57      <wsdl:service name="PlayerService">
58          <wsdl:port binding="tns:PlayerBinding" name="
               PlayerPort">
59              <soap:address location="http://atradig82.
                   informatik.tu-muenchen.de:8280/cocoon/
                   halgurt/services/soap/playerAge"/>
60          </wsdl:port>
61      </wsdl:service>
62</wsdl:definitions>
```

F Regeln zur Definition von Vor- und Nachbedingungen

```
1 (<=(hasPrecondition ?Profile ?Precondition )
2    (|http://www.daml.org/services/OWLS/1.1/
3    Profile.owl#|::|hasPrecondition| ?Profile
4    ?Precondition)
5 )
6
7 (<=(hasResult ?Profile ?Result)
8    (|http://www.daml.org/services/OWLS/1.1/
9    Profile.owl#|::|hasResult| ?Profile ?Result)
```

```
10 )
11
12 (<=(hasEffect ?result ?effect)
13    (|http://www.daml.org/services/OWLS/1.1/
14    Process.owl#|::|hasEffect| ?result ?effect)
15 )
16
17 (<=(exBody ?Precondition ?body)
18    (|http://www.daml.org/services/OWLS/1.1/
19    generic/Expression.owl#|::|expressionBody|
20    ?Precondition ?body)
21 )
22
23 (<=(atomList ?body)
24    (|http://www.w3.org/1999/02/22-rdf-syntax-ns#
25    |::|type| ?body |http://www.w3.org/2003/11/
26    swrl#|::|AtomList|)
27 )
28 (<=(Atom ?atom)
29    (|http://www.w3.org/1999/02/22-rdf-syntax-ns#
30    |::|type| ?atom |http://www.w3.org/2003/11/
31    swrl#|::|Atom|)
32 )
33
34 (<=(InAtomList ?atomList ?atom)
35    (and (atomList ?atomList)
36       (or
37          (and
38             (hasFirst ?atomList ?atom)
39          )
40          (and
41             (hasRest ?atomList ?atomList1)
42             (atomList ?atomList1)
43             (InAtomList ?atomList1 ?atom)
44          )
45       )
46    )
47 )
48
49 (<=(hasFirst ?atomList ?first)
50    (|http://www.w3.org/1999/02/22-rdf-syntax-ns#|::|first|
51     ?atomList ?first)
52 )
```

```
53
54 (<=(hasRest ?atomList ?first)
55   (|http://www.w3.org/1999/02/22-rdf-syntax-ns#|::|rest|
56    ?atomList ?first)
57 )
58
59 (<=(classPredicate ?atom ?classPredicate)
60   (|http://www.w3.org/2003/11/swrl#|::|classPredicate|
61   ?atom ?classPredicate)
62 )
63
64 (<=(propertyPredicate ?atom ?propertyPredicate)
65   (|http://www.w3.org/2003/11/swrl#|::|propertyPredicate|
66   ?atom ?propertyPredicate)
67 )
68
69 (<=(predicate ?atom ?predicate)
70   (or (propertyPredicate ?atom ?predicate)
71       (classPredicate ?atom ?predicate)
72   )
73 )
```

Ausschnitt 28: Erweiterte Regel zur Ermittlung von Serviceinstanzen

```
21 (<=(getService ?Profile ?Service ?MatchType)
22    (and (hasResult ?Profile ?Result)
23       (hasEffect ?Result ?Effect)
24       (exBody ?Effect?body)
25       (BuiltinAtom ?atom1 |http://www.w3.org/2003/
26         11/swrlb#|::|lessThan| ?age ?zahl)
27       (lessThanMatchType ?zahl 20 ?MatchType1)
28       (predicate ?atom3?predicat3)
29       (argument1 ?atom3 ?Player)
30       (predicate ?atom2 |http://atradig82.informatik.
31          tu-muenchen.de:8280/cocoon/ontology/
32          football.owl#|::|age|)
33       (argument1 ?atom2 ?Player)
34       (argument2 ?atom2 ?age)
35       (get3List ?body?atom3 ?atom2 ?atom1)
36       (hasPrecondition ?Profile ?Precondition)
37       (exBody ?Precondition ?body1)
38       (predicate ?atom5 |http://atradig82.informatik.
39           tu-muenchen.de:8280/cocoon/ontology/
```

```
40             football.owl#|::|teamName|)
41         (argument1 ?atom5 ?Team)
42         (argument2 ?atom5 ?teamName)
43         (predicate ?atom6 |http://atradig82.informatik.
44             tu-muenchen.de:8280/cocoon/ontology/
45             football.owl#|::|playsFor|)
46         (argument1 ?atom6 ?Player)
47         (argument2 ?atom6 ?Team)
48         (get2List ?body1 ?atom5 ?atom6)
49         (Input ?Profile ?teamName)
50         (parameterType ?teamName
51             "http://www.w3.org/2001/XMLSchema#string")
52         (Parameter ?Profile ?Player)
53         (Output ?Profile ?age)
54         (parameterType ?age
55             "http://www.w3.org/2001/XMLSchema#int")
56         (Output ?Profile ?legueGoals)
57         (parameterType ?legueGoals
58             "http://www.w3.org/2001/XMLSchema#int")
59         (Parameter ?Profile ?Team)
60         (parameterType ?Team "http://atradig82.informatik.
61            tu-muenchen.de:8280/cocoon/ontology/
62            football.owl#Team")
63         (ClassMatchType ?predicate3
64           |http://atradig82.informatik.tu-muenchen.de:8280/
65           cocoon/ontology/football.owl#|::|Striker| ?Player
66           "http://atradig82.informatik.tu-muenchen.de:8280/
67           cocoon/ontology/football.owl#Striker" ?MatchType2)
68         (max ?MatchType ?MatchType1 ?MatchType2)
69         (presentedBy ?Profile ?Service)
70     )
71 )
```

Literatur

[1] M.W.A, Strating, P., Lankhorst, M.M., Doest, H. ter & Iacob, M.-E. Service-Oriented Enterprise Architecture. To appear in: Zoran Stojanovic & Ajantha Dahanayake

(Eds.), Service-Oriented Software System Engineering: Challenges and Practices. IDEA Group, 2004.

[2] Web Services Description Language (WSDL). W3C Note 15 March 2001. http://www.w3.org/TR/wsdl

[3] SOAP Version 1.2 Part 0: Primer. W3C Recommendation 24 June 2003. http://www.w3.org/TR/soap12-part0/

[4] UDDI Technical White Paper. September 6, 2000.

[5] Semantic Web http://www.w3.org/2001/sw/

[6] OWLS: Semantic Markup for Web Services. http://www.daml.org/services/OWLS/1.1/overview/

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

[8] http://www.w3.org/Submission/SWRLB/

[9] Li, Lei and Horrocks, Ian. A Software Framework for Matchmaking Based on Semantic Web Technology. In Proceedings International WWW Conference, Budapest, Hungary. (2003)

[10] Gregor Hohpe. Web Services: Pathway to a Service-Oriented Architecture? ThoughtWorks, Inc., 2002.

[11] Tim Berneers-Lee. Weaving the Web. Harper San Francisco, 1999.

[12] Web Services Essentials, Ethan Cerami, O'reilly, February 2002.

[13] Understanding Web Services: XML, WSDL, SOAP, and UDDI, Eric Newcomer, May 2002.

[14] Agent Technology: Enabling Next Generation Computing, A Roadmap for Agent Based Computing. AgentLink, 2003.

[15] Tannenbaum, A., Steen, M. Distributed Systems - Principles and Paradigms. Prentice Hall, 2002.

[16] Vorlesung Wissensbasierte Systeme/Kuenstliche Intelligenz WS 2004/2005. Dozent: Prof. Michael Beetz. Rationale Agenten: Definitionen, rationale Agenten, Implementierung von Agenten, PAGE Beschreibung, Kategorien von Agenten, BDI Agenten.

[17] OWL Web Ontology Language Overview. W3C Recommendation 10 February 2004. http://www.w3.org/TR/owl-features/

[18] Daconta, Obrst, Smith. The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management. Wiley Publishing, Inc. 2003.

[19] Grigoris Antoniou and Frank van Harmelen. A Semantic Web Primer. The MIT Press, 2004.

[20] Massimo Paolucci, Katia Sycara, and Takahiro Kawamura. Delivering Semantic Web Services. In Proceedings of the Twelve's World Wide Web Conference (WWW2003), Budapest, Hungary, May 2003, pp 111- 118

[21] Scientific American: The Semantic Web A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities By Tim Berners-Lee, James Hendler and Ora Lassila.

[22] RDF Primer. W3C Recommendation 10 February 2004

[23] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. The Description Logic Handbook. Cambridge University Press, 2002.

[24] Wissensbasierte Systeme 2: Chapter 2: Description Logics http://wwwradig.in.tum.de/vorlesungen/wibas2.SS04/material/vorlesung02.pdf

[25] Massimo Paolucci, Katia Sycara, Takuya Nishimura, and Naveen Srinivasan. Toward a Semantic Web e-commerce. To appear in Proceedings of BIS2003.

[26] R. Fikes, G. Frank und J. Jenkins. JTP: A System Architecture and Component Library for Hybrid Reasoning, 2003. http://ksl.stanford.edu/pub/KSL Reports/KSL- 03-01.pdf

[27] R. Fikes, G. Frank und J. Jenkins. JTP: An Object Oriented Modular Reasoning System, 2004. http://www.ksl.stanford.edu/software/JTP/

[28] Semantic Interoperability Community of Practice SICoP. Introducing Semantic Technologies and the Vision of the Semantic Web, February 2005

[29] E. Sirin und B. Parsia. Planning for Semantic Web Services, 2004. http://www.ai.sri.com/SWS2004/final-versions/SWS2004-Sirin-Final.pdf

[30] Pellet: An OWL DL Reasoner Bijan Parsia and Evren Sirin MINDSWAP Research Group University of Maryland, College Park, MD bparsia@isr.umd.edu, evren@cs.umd.edu

[31] HTN Planning for Web Service Composition Using SHOP2. Evren Sirin, Bijan Parsia, Dan Wu, James Hendler, Dana Nau.

[32] Information Gathering During Planning for Web Service Composition. Ugur Kuter, Evren Sirin, Bijan Parsia, Dana Nau, James Hendler.

[33] Web Services W3C Standards I: SOAP, WSDL und UDDI http://ibis.in.tum.de/teaching/ws04_05/pr_iis_wae/Zusammenfassung.pdf

[34] Ontologie-basierte Modellierung und Synthese von Geschäftsprozessen. Florian Lautenbacher, Report 2005-16 September 2005. http://www.informatik.uni-augsburg.de/lehrstuehle/swt/vs/publikationen/reports/2005-16/TechnicalReport.pdf