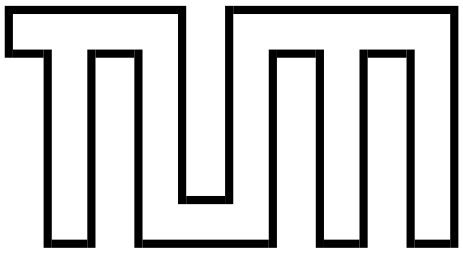


Technische Universität München

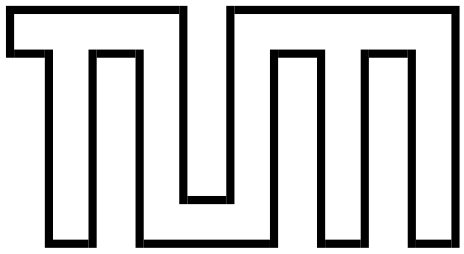
Fakultät für Informatik

Forschungs- und Lehreinheit Informatik IX

Semantische Repräsentation der universitären Organisationsstruktur und deren automatische Präsentation im WWW

Systementwicklungsprojekt

Marius Morawski und Matthias Seidl



Technische Universität München

Fakultät für Informatik

Forschungs- und Lehreinheit Informatik IX

Repräsentation der universitären Organisationsstruktur
und deren automatische Präsentation im WWW

Systementwicklungsprojekt

Marius Morawski und Matthias Seidl

Themensteller: Prof. Michael Beetz, PhD

Betreuer: Dipl.-Inform. Matthias Wimmer

Abgabetermin: 01.05.2005

Inhaltsverzeichnis

1 Einleitung

Seit Menschengedenken hat der Mensch Informationen physisch festgehalten, in zunehmend ausgeklügelter Art und Weise: Felszeichnungen, Hieroglyphen, Schrift, Bits&Bytes.
Wichtige Motivationen dafür waren und sind das Weitergeben von Information an Zeitgenossen und nachfolgende Generationen, aber auch das Festhalten der Information zum eigenen späteren Gebrauch.

Mit der Speicherung von Daten in (heutzutage vernetzten) Computersystemen haben sich vollkommen neue Möglichkeiten aufgetan: Informationen können schnell und unkompliziert weltweit ausgetauscht werden, maschinell weiterverarbeitet werden und durch die Verknüpfung mit anderen Informationen noch nützlicher gemacht werden.

1.1 Verstehen von Informationen

Die neuen Möglichkeiten beinhalten aber auch neue sowie auch alte Herausforderungen. Dazu gehört eine Herausforderung, die zwar schon seit Erfindung der Sprache bekannt ist: Die Information muss verstanden werden, damit man etwas damit anfangen kann.
Diese Herausforderung hat mit dem Einzug der elektronischen Datenverarbeitung eine ganz neue Dimension angenommen: Während man früher primär mit verschiedenen Sprachen kämpfte, hat man es heute zusätzlich mit dem Verstehen von Informationen durch den Computer zu tun - auf niedriger Ebene mit dem Verstehen fremder Formate, auf höherer Ebene damit, Informationen sinnvoll zu verwerten.
Computer können Informationen zwar in großer Geschwindigkeit verarbeiten, ihnen aber keine Bedeutung zuweisen, was ihre Nützlichkeit in vielen Gebieten einschränkt.
Eine Suchmaschine z.B. kann zwar zu einem Suchbegriff ähnliche Zeichenketten in großen Datenmengen auffinden, hat aber Schwierigkeiten, in Zeichenketten enthaltene Informationen zu korrelieren. Letztendlich beruhen heutige Suchmaschinene auf der Annahme, dass ähnlichen Zeichenketten ähnliche Bedeutungen entsprechen - das stimmt zwar oft, aber eben bei weitem nicht immer.

Was viele der heute gebräuchlichen Speicherungsarten gemeinsam haben, ist, dass sie keinerlei Informationen über die Bedeutung der gespeicherten Daten enthalten. Um mit solchen Daten sinnvoll umgehen zu können, müssen Informationen über Bedeutung und Zusammenhang der Daten also bei demjenigen vorhanden sein, der damit arbeitet – z.B. in Form von Allgemein- oder Fach-Wissen. Computer verfügen nicht über derartiges Wissen, sie können nur rein syntaktisch mit den Daten arbeiten.

Zwei sich ergänzende Ansätze, diese Herausforderung anzugehen, sind

- die im entsprechnden Kontext gerade nötigen Informationen in die Programmlogik mit einfließen oder lernen zu lassen
- das Problem von der anderen Seite anzugehen und den Daten Meta-Daten mitzugeben, die Hinweise über über Kontext, Bedeutung und Zusammenhang enthalten.

1.2 Nachhelfen mit Meta-Informationen

Im folgenden wird es um Technologien gehen, die den zweiten Ansatz implementieren. Dieser hat zwar den Nachteil, nicht direkt mit eventuell vorhandenen Datanbestnden umgehen zu können, allerdings kann der erste Ansatz ergänzend eingesetzt werden, indem die Daten mit entsprechenden Meta-Daten angereichert werden. Liegen die Daten erstmal in einer geeigneten Form vor, kann der zweite Ansatz seine Vorteile voll ausspielen:

- Die Daten über Kontext, Bedeutung und Zusammenhänge können von verschiedenen Programmen benutzt werden, da sie explizit in den Daten enthalten sind, statt wie beim ersten Ansatz implizit im unzugänglichen Programmcode.
- Es kann direkt mit den Metdaten gearbeitet werden, statt sie erst mühsam extrahieren zu müssen, was zu Geschwindigkeitsvorteilen führen kann.
- Je nach Qualität der Meta-Daten können im Vergleich zu gelernten Zusammenhängen sehr viel besser Schlussfolgerungen aus den Daten gezogen werden, weil gelernte Zusammenhänge immer fehlerbehaftet sind.

Eine Technologie, die sich anbietet, um solche Meta-Daten einfließen zu lassen, sind Datenbanken. Über Spaltennamen können Daten näher beschrieben werden, z.B. ob es sich um einen Familiennamen oder eine Adresse handelt. Ein Vorteil von Datenbanken ist, dass sie schnell auf großen Datenbeständen arbeiten können. Eine andere Möglichkeit ist XML (EXtensible Markup Language): Mit Hilfe von XML kann man sich eine Syntax definieren, die Abschnitten in textuellen Informationen Bedeutungen zuweist. XML hat gegenüber Datenbanken Vorteile:

Flexibilität/Erweiterbarkeit Datenbanken sind nicht sonderlich flexibel. Eine Tabelle hat eine fixe Menge von Spalten und die möglichen Datentypen sind durch die jeweilige Datenbank beschränkt. Eigene Erweiterungen sind schwierig oder garnicht realisierbar.
XML dagegen ermöglicht es, eigene Datentypen zu definieren und eine variable Anzahl von Unterelementen zu haben, die wiederum selbst von beliebiger Struktur sein können.

Portabilität und Lesbarkeit die Daten sind in die Datenbank "eingegossen". Andere Programme können auf diese Daten nur über die jeweilige Datenbankanwendung nur durch passende Queries zugreifen, was die potentiellen Einsatzmöglichkeiten einschränkt.
XML speichert die Daten inklusive Metadaten als Textdateien, was gute Portabilität gewährleistet. Die Daten sind z.B. über einen beliebigen Texteditor auch für den Menschen leicht zugänglich.

Ein einfaches XML-Dokument kann z.B. so aussehen:

Einfaches XML-Dokument:

```
<?xml version="1.0"?>
<Termin>
<Zeit><Tag>Morgen</Tag>um
<Uhrzeit>11 Uhr</Uhrzeit></Zeit>
findet in <Ort>H"orsaal 2</Ort> ein
<Anlass><TerminArt>Vortrag</TerminArt>
zum Thema <Thema>Semantic Web</Thema>
</Anlass> statt.
</Termin>
```

Das ist schonmal ein Schritt vorwärts, einzelnen Teilen eines Satzes werden hier Bedeutungen zugewiesen, die auch einem Computer Informationen zugänglich machen können, die ein Mensch mit Hilfe seines Allgemeinwissens aus dem Kontext erkennen könnte. Ein Programm, dem derartige Daten vorliegen, könnte damit Anfragen nach Ort, Zeit oder Thema dieses Termins beantworten.

1.3 Das Problem

Aber Moment, könnte es das wirklich? Muss es dazu nicht wiederum erst die Bedeutung von Tags wie ¡Ort¿ kennen? Ja, müsste es.
Muss man sich also letztendlich doch auf einige wenige standardisierte XML-Dialekte beschränken und die Bedeutung der Auszeichnungen in einem Programm hardcoden?
Nicht ganz. Es muss nur eine standardisierte Möglichkeit existieren, eigene XML-Dialekte zu definieren und grundlegende Beziehungen zwischen diesen Elementen (Vererbung, Assoziation, Aggregation, ...) mitsamt Beschränkungen zu definieren.

1.4 Die Lösung?

XML Schema reicht für diese Zwecke nicht aus. RDF bzw. seine Erweiterung OWL dagegen bietet Möglichkeiten, eigene XML-Sprachen mit Elementen zu definieren, die in logischen Beziehungen zueinander stehen. Außerdem existiert eine Implementierung, die auf XML-Dokumenten in mit OWL Sprachen logische Schlüsse ziehen kann und Anfragen ausführen und wieder als XML ausgeben kann, die in einer passenden Query-Sprache (OWL-QL) formuliert sind. Mehr zu OWL in Kapitel 2.3
Man kann also z.B. mit Hilfe von OWL in einem eigenen XML-Dialekt eigene Domänen von Objekte mit eigenen Regeln beschreiben, über das dann vollautomatisch Schlüsse gezogen werden können. Damit ist die Grundlage für das sogenannte Semantic Web gelegt, eine Erweiterung des Internet um maschinenverstehbare Informationen.

1.5 Paralleluniversen?

Aber wird da nicht eine Parallelwelt zum althergebrachten Web aufgebaut? Ein Web-Browser kann mit solchen selbstdefinerten XML-Dialekten schließlich nicht viel anfangen.
Die Antwort ist nein. Zumindest ist das nicht zwingend das Ergebnis:

1. (X-)HTML und andere XML-Dialekte schließen sich nicht gegenseitig aus. XML ist so ausgelegt, dass Elemente aus verschiedenen Dialekten (z.B. OWL und XHTML) im selben Dokument vorkommen können. Ein Programm kann dabei die Elemente ignorieren, die aus seiner Sicht keinen Sinn machen.
   So könnte z.B. ein XHTML-Dokument zwischen den normalen HTML-Tags auch OWL-Tags aufweisen, die die selbe Information, die ein Mensch im Web-Browser sieht, auch für Programme zugänglich machen, die mit den enthaltenen OWL-Tags etwas anzufangen wissen.

2. Mit Hilfe von CSS kann man im Prinzip beliebige XML-Dokumente formatieren. Allerdings macht das hauptsächlich dann Sinn, wenn das Dokument dazu ausgelegt ist, von Menschen gelesen zu werden - sonst kommt im besten Fall eine tabellarische Ansicht der Daten raus.

3. XML-Dialekte können mit Hilfe von Transformationssprachen wie XSLT (siehe Kapitel 2.1 auf Seite 6) ineinander übergeführt werden. Diesen Weg sind wir bei unserem Projekt gegangen.

Bevor wir aber zu unserem Projekt kommen, werden wir zuerst etwas näher auf die verwendeten (und teilweise schon angesprochenen) Technologien eingehen.

2 Verwendete Techniken

2.1 XSLT&Family

XSLT ist ein Teil von XSL (Extensible Stylesheet Language) einer "Familie" von drei Sprachen zur Umwandlung und Formatierung von XML-Dokumenten:

XSLT ist das zentrale Element, mit dem Transformationen von XML-Dokumenten in andere XML-Dokumente definiert werden können.

XPath eine Ausdrucks-Sprache, um Teile eines XML-Dokuments zu adressieren; wird in dieser Funktion bei XSLT eingesetzt.

XSL-FO ist ein XML-Dialekt, der exakte Formatierungen definiert und dadurch gut dazu geeignet ist, in Formate wie PDF oder übersetzt zu werden. Cocoon enthält z.B. einen Serializer, der XSL-FO in PDF umwandeln kann, so dass man z.B. ein XML-Dokument mit XSLT und anderen Transformatoren in XSL-FO transformieren kann, um es dann als PDF auszugeben. Da diese Sprache aber bei unserem Projekt nicht relevant war, wird dieser Dialekt im folgenden nicht erläutert.

Weil XSLT und XPath sehr eng zusammenhängen, kann man sie bei der Erläuterung der beiden Sprachen auch nicht vollständig getrennt behandeln. Aus diesem Grund soll hier zunächst mal anhand eines kleinem Beispiels das Prinzip hinter XSLT nähergebracht werden.
Um das Beispiel anschaulich, leicht verständlich und lesbar zu halten, wird hier kein Beispiel aus unserem Projekt vorgestellt. Statt dessen geht es im folgenden Beispiel um eine einfache Speisekarte in einem einfachen XML-Format, die zur Darstellung auf der Webseite eines Restaurants umgewandelt werden soll in XHTML:

Speisekarte als XML-Datei

```
<?xml version="1.0"?>
<Speisekarte>
  <Tag>Montag</Tag>
  <Gericht>
    <Name>Wiener Schnitzel</Name>
    <Beilage>Salat</Beilage>
    <Preis waehrung="Euro">9.99</Preis>
  <Gericht>

  <Gericht>
    <Name>Schweinebraten</Name>
```

```
    <Preis waehrung="Euro">12.99</Preis>
  <Gericht>
</Speisekarte>
```

Ein Webbrowser kann sowas natürlich nicht sinnvoll anzeigen, er kann mit den Tags nichts anfangen (außerdem möchte man manchmal vielleicht auch eine auswahl der Informationen angeben, die angezeigt werden). Besser wäre da eine Form wie die folgende:

Gewünschte Ausgabedatei:

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="de">
  <head><title>Speisekarte Montag</title></head>
  <body>
    <h2>Montag:</h2>
    <table>
      <tr><th>Gericht</th><th>Preis</th></tr>
      <tr><td>Wiener Schnitzel</td><td>9.99 Euro</td></tr>
      <tr><td>Schweinebraten</td><td>12.99 Euro</td></tr>
    </table>
  </body>
</html>
```

Wie man sich vorstellen kann, muss, um eine sinnvolle Transformation zu bewerkstelligen, der Transformator auf die spezielle Struktur des Eingabedokuments ausgelegt sein - die prinzipielle Struktur des Dokuments müsste also beim Programmieren eines Transformators schon a priori bekannt sein. Diesem Dilemma begegnet man durch XSLT-Transformatoren, die die eigentlichen "Regeln" zur Transformation eines XML-Dokuments aus XSLTs auslesen. Man kann XSLTs also auch als Regeln für einen Transformator verstehen. Das aber nur nebenbei.

Abbildung 1 zeigt schematisch den Ablauf der Transformation eines XML-Dokuments in ein anderes im einfachsten Fall (nur eine Transformation).

Ein XSL-Transformator liest als Eingaben das zu transformierende Dokument sowie das XSLT, das die Transformation in das Ausgabeformat definiert, und erzeugt daraus das Ausgabedokument. In komplizierteren Fällen können auch mehrere Transformatoren hintereinandergeschaltet werden, so dass die Ausgabe des einen Transformators als Eingabe für den dahintergeschalteten dient - mehr

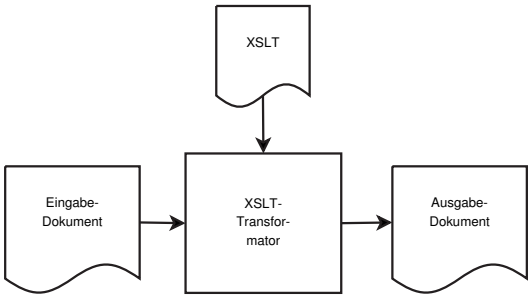


Abbildung 1: Transformation per XSLT

dazu in Kapitel 2.2, wo es unter anderem um sogenannte Sitemaps geht.
Aber wie sieht nun so ein XSLT-Programm oder -Regelwerk zur Transformation der obigen Speisekarte nach XHTML aus?

2.1.1 XSLT

XSLT (XSL-Transformations), der zentrale Teil der XSL-Familie, ist (wie bereits erwähnt) eine Sprache, um XML-Dokumente in andere XML-Dokumente zu transformieren. XSLT ist sehr mächtig (sogar berechenbarkeits-universell), weist aber zwei Sonderheiten auf:

1. XSLT benutzt XML-Syntax. XSLT-"Programme" werden also selbst in XML geschrieben. Falls man nicht schon mit anderen Sprache nauf XML-Basis wie z.B. Jelly- oder Ant- Skripten gearbeitet hat, kann das etwas gewöhnungsbedürftig sein.

2. der "Programm"-Ablauf ist datengesteuert, d.h. das Eingabe-Dokument "steuert" im wesentlichen den Programmfluss bzw. das Programm reagiert auf Daten (also im Prinzip eine Unterart der ereignisbasierten Programmierung).

Zurück zum Beispiel Wie so ein datengesteuertes XML-basiertes Programm in der Praxis aussieht, lässt sich anhand unseres Beispiel-Problems gut darstellen. Deswegen hier vor der eigentlichen Erklärung zunächst ein XSLT, mit dem ein XSLT-Transformator die gewünschte Transformation der Speisekarte vornehmen kann.

XSLT zur Transformation der Speisekarte:

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0"
```

```
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml">

  <xsl:output method="html" version="1.0"
      encoding="UTF-8"
      indent="yes"
      doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
      doctype-system="http://www.w3.org/1999/xhtml"
      />

  <xsl:template match="/Speisekarte">
    <html>
      <head>
        <title>Speisekarte <xsl:value-of select="Tag"/></title>
      </head>
      <body>
        <h2><xsl:value-of select="Tag"/></h2>
        <table>
          <tr><th>Gericht</th><th>Preis</th></tr>
          <xsl:apply-templates select="Gericht"/>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="Gericht">
    <tr>
      <td><xsl:value-of select="Name"/></td>
      <td>
        <xsl:value-of select="Preis"/>
        <xsl:value-of select="Preis/@waehrung"/>
      </td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

Namespaces Was an diesem Beispiel wohl zuerst auffällt, ist die Mischung von HTML-Tags mit Tags, die den Präfix `xsl:` verwenden. Hier kommt die Fähigkeit von XML, verschiedene XML-Dialekte in einem Dokument zu erlauben, zur Anwendung. Dazu müssen die verschiedenen Dialekte nur durch zuvor definierte Namespaces auseinandergehalten werden - im Beispiel ist das Präfix

`xsl:` dem Namenraum `http://www.w3.org/1999/XSL/Transform` zugeordnet, XHTML braucht in dem Fall kein Präfix, da es im Beispiel dem Default-Namensraum zugeordnet ist.

Dieses noch recht einfache Beispiel verwendet schon einige interessante Mechanismen.

Templates Das grundlegende Konstrukt von XSLT zur Verarbeitung von XML-Dokumenten ist das Template. Ein Template definiert, wie ein Teil eines XML-Baums[1] transformiert wird. In Templates können Unterbäume durch verschiedene Mechanismen manipuliert werden, z.B. können weitere Unterbäume einfach durch Angabe der entsprechenden Tags erzeugt werden.

Delegieren Es können aber auch Unterbäume an andere Templates delegiert werden. Beim sehr gebräuchlichen "Push"-Ansatz[2] überlässt man es dem XSLT-Transformator, für einen Teilbaum ein passendes Template zu finden:
`<xsl:apply-templates select="Gericht" />` z.B. bewirkt, dass sämtliche `Gericht`-Unterelemente durch das erste anwendbare Template behandelt werden, das im XSLT gefunden wird.

Matching Mit dem `match`-Attribut kann man dabei pro Template definieren, auf welche Teilbäume es anwendbar sein soll.
`<xsl:template match="/Speisekarte">` z.B. definiert ein Template, das auf ein Element namens `Speisekarte` anwendbar sein soll; das führende Zeichen '`/`' schränkt dabei zusätzlich ein, dass es das Wurzelelement sein muss (in dem Fall also ganz analog zu Pfadangaben in Unix). Die Syntax für diese "Matchings" ist durch die Sprache XPath definiert, die auch noch viel komplexere Ausdrücke erlaubt und in Kapitel 2.1.2 auf Seite 10 ausführlicher beschrieben wird.

Iterieren mit for-each Ein weiterer wichtiger Baustein von XSLT ist das `for-each`-Konstrukt. Es ermöglicht das Iterieren über eine Menge von XML-Elementen, die mit dem `select`-Attribut selektiert werden können. Das folgende Beispiel zeigt, wie man bei unserer Speisekarte

2.1.2 XPath

XPath definiert eine Syntax für Ausdrücke, um Teile von XML-Dokumenten zu adressieren und legt damit die Grundlage, mit der Transfomationen per XSLT

[1] XML-Dokumente sind in einer Baumstruktur organisiert - der Wurzelknoten heißt bei XML auch Wurzelelement (definiert durch das äusserste Paar von Anfangs- und End-Tags

[2] Der "Pull"-Ansatz ist, Unterbäume nach bestimmten Kriterien auszusuchen und an namentlich adressierte Templates zur weiteren Bearbeitung zu übergeben - ganz ähnlich zum Funktionsaufruf in funktionalen Sprachen.

überhaupt erst möglich werden. In diesem Kapitel soll nun die allgemeine Syntax und der Funktionsumfang von XPath nun näher vorgestellt werden.

Adressieren von Elementen (und Attributen) XPath-Ausdrücke repräsentieren stets einen Teilbaum oder eine Menge von Teilbäumen von XML-Bäumen, die über verschiedene Eigenschaften adressiert werden können, wie z.B. über die ensprechenden Element-Namen und ihre Verschachtelung im XML-Dokument. Im Kapitel 2.1 kamen im Beispiel für ein einfaches XSLT auf Seite 8 ja schon einfache XPath-Ausdrücke vor, allerdings erlaubt XPath sehr viel mehr als das. Hier die grundlegenden "Adressierungsarten":

absolute Pfadangabe `/Speisekarte/Gericht/Name` z.B. adressiert die Menge aller Elemente "Name", die Unterelemente von "Gericht" sind, die selbst wiederum Unterelemente des Wurzelknotens "Speisekarte" sind.

relative Pfadangabe `Preis` adressiert alle Elemente namens "Preis", die direkt unter dem aktuellen Element liegen

beliebige Position im Dokument `//Preis` adressiert alle Elemente mit Namen "Preis", egal wo im Dokument sie vorkommen.

Wildcards `/Speisekarte/*/Preis` adressiert alle Elemente "Preis", die in einem beliebigen Element vorkommen, dass selbst Unterelement von "Speisekarte" ist.

Attribute `//Preis/@Waehrung` adressiert alle "Waehrung"-Attribute aller "Preis"-Elemente im Dokument.

Das sieht teilweise recht ähnlich zu einer Pfadangabe in einem Unix-Dateisystem aus – allerdings sind XML-Elementnamen anders als Dateinamen keine Bezeichner für einzelne Individuen (ein Element/Knoten kann mehrere gleichnamige Unterknoten haben), sondern eher mit Klassen/Typ-Bezeichnern vergleichbar, so dass auch auf den ersten Blick scheinbar eindeutige Pfade wie `/Speisekarte/Gericht/Name` nicht ein Objekt, sondern eine Menge von Elementen adressieren. Die Unterscheidung zwischen Knoten (nodes) und Knotenmengen (node sets) sollte man dabei niemals aus dem Auge verlieren, was besonders bei einelementigen Mengen schnell mal unterlaufen kann.

Untermengen und Individuen durch Prädikate auswählen Um aus einer Menge von Elementen eine Untermenge oder auch einzelne Elemente auszuwählen, kann man durch Prädikate in eckigen Klammern weitere Einschränkungen definieren:

Attribute und Unterelemente Man kann unter anderem einschränken, dass Elemente bestimmte Bedingungen in Bezug auf ihre Attribute oder Unterelemente erfüllen müssen. `Gericht[Beilage]` selektiert alle Gerichte mit Beilage. Man kann auch noch genauer einschränken, indem man Beschränkungen über die Inhalte definiert wie bei den folgenden drei Beispielen

- `Gericht[Beilage = 'Salat']`
- `Gericht[Beilage != 'Salat']`
- `Gericht[Preis < 10.00]`

Genauso funktioniert das auch für Attribute, nur dass man den Namen der Attribute ein "@" voranstellt:
`Gericht[Preis/@Waehrung = 'Euro']` selektiert alle Gerichte, deren Preis in Euro angegeben ist. Hier sieht man auch eine weitere Möglichkeit: Die eingeklammerten Prädikate selbst können wiederum Pfade enthalten.

Auswahl per ID Eine Möglichkeit, die sich durch Auswahl per Attributwert anbietet, ist die Auswahl von Elementen anhand ihrer ID: Viele XML-Dialekte definieren eindeutige IDs (wie z.B. die id-Attribute in RDF und HTML). Diese kann man dann zu Hilfe nehmen, um bestimmte einzelne Elemente zu adressieren (auch unabhängig von Position und Tiefe im XML-Baum).
`Gericht[@id = 'schnitzel']` adressiert z.B. die Menge aller Gerichte, die ein Attribut mit dem Namen "id" und dem Inhalt "schnitzel" haben (im XML-Dokument z.B. definiert als `<Gericht id="schnitzel" />`). Falls "id" eindeutig ist, ist das eine einelementige Menge – aber eben immer noch eine Menge! Auf Mengen sind jedoch andere Operationen definiert als auf Einzelelementen. Um an das einzelne Element heranzukommen, braucht man also einen zusätzlichen Mechanismus...

Position im Menge Wirklich eindeutig kann man Elemente nur über ihre Position in einer Menge definieren. Beim aktuellen Beispiel kann man z.B. `Gericht[@id = 'schnitzel'][1]` verwenden, um das erste (und in dem Fall hoffentlich einzige) Element der Menge zu adressieren. Man kann dazu beliebige Zahlenwerte oder die Funktion `last()` (die das letzte Element einer Menge adressiert) angeben. Es sei noch darauf hingewiesen, dass die Schreibweise `[1]` eine Abkürzung ist für `[position() = 1]` – es sind also auch andere Prädikate möglich wie z.B. `[position() != last()]`

Weitere Funktionen Neben `last()` und `position()` definiert die W3C-spezifikation auch noch eine Reihe weiterer Funktionen (`http://www.w3.org/TR/xpath#corelib`), die hier aufzulisten den Rahmen sprengen würde.

Kombinatioen

Kombination von verschiedenen Adressierungsmechanismen Die Kombination von relativen bzw.absoluten Pfadangaben mit anderen Mechanismen ermöglicht praktisch beliebig komplexe Adressierungsarten; das obige Beispiel `/Speisekarte/*/Preis` kombiniert so z.B. eine absolute Pfadangabe mit einer Wildcard, aber es lassen sich auch beispielsweise Prädikate mit Wildcards oder miteinander kombinieren.

Oder-Operator Auch der Oder-Operator steht beim Arbeiten mit XPath zur Verügung (Beispiel für die gesundheitsbewussten: verb+Gericht[Beilage='Salat' or Nachspeise='Obst']+ selektiert alle gerichte mit entsprechenden Unterelementen, `Beilage or Nachspeise` selektiert alle Beilagen und Salate).

Achsen Wer mit XPath arbeitet, wird früher oder später mit Achsen zu tun tun haben – genaugenommen hat man sogar ständig damit zu tun, nur dass es in den meisten Fällen nicht offensichtlich wird: Achsen stecken in jedem XPath-Ausdruck, nur in der Regel (und in den bisherigen Beispielen) in der Kurzschreibweise.
Um Funktion und Sinn von Achsen zu verstehen, muss man wissen, wie ein XPath-Pfad syntaktisch aufgebaut ist: Ein Pfad besteht aus einem oder mehreren "Schritten", durch verb+/+ getrennt, und jeder "Schritt" setzt sich zusammen aus "Achse", "Knotentest" und null oder mehreren "Prädikaten". Die Knotentests sind die bereits bekannten Elementnamen bzw. Wildcards und auch die Prädikate sind weiter oben schon beschrieben worden.
Von den Achsen hingegen war bisher nichts zu sehen – das liegt daran, dass wir bisher implizit meist eine bestimmte Achse verwendet haben: Die Kind-Achse. Jeder weitere XPath-Schritt adressiert Kindknoten, wie von Dateisystemen gewohnt.
Die explizite Schreibweise zur Kurzform `/Speisekarte/Gericht/Name` ist `/child::Speisekarte/child::Gericht/child::Name`. Die Achse beschreibt also quasi die Art der Schrittes.
Ein paar weitere gebräuchliche Achsen bzw. Kombinationen von Achse und Knotentests, die teilweise auch schon in bisherigen Beispielen benutzt wurden, sind:

`attribute::<name>` selektiert Attribute mit Name `<name>` – in den bisherigen Beispielen wurde immer die Kurzform "@" benutzt.

`self::node()` adressiert aktuelle Knoten – Kurzform ".".

`parent::node()` Elternknoten – Kurzform "..".

`descendant-or-self::node()` adressiert Knoten selbst oder beliebige Unterknoten (also nicht nur direkte Kindknoten) – Kurzform "//".

Die XPath-Spezifikation (`http://www.w3.org/TR/xpath#axes`) beschreibt noch ein paar weitere Achsenarten, aber mit den hier genannten sollten die meisten Anwendungsfälle eigentlich abgedeckt sein.

2.2 Cocoon

2.2.1 Eigenschaften und Funktionsweise von Cocoon

Cocoon ist ein XML-basiertes Web-Publishing Framework von Apache und wurde in der Programmiersprache Java als Servlet implementiert [COC]. Hauptziel von Cocoon ist die Trennung von Inhalt, Präsentation, Dynamik und Management, also die "Separation of Concerns". Dies ist in der Literatur auch als MVC (Model View Controller) - Architektur bekannt. Im Interesse eines Erstellers einer Webanwendung liegt, dass die Applikation möglichst vielen Benutzern zugänglich ist. Letzere können aber verschiedenste Ausgabegeräte (Handy, Browser(Opera, IE), Handheld) verwenden, von denen jedes ein anderes Ausgabeformat benötigt. Aufgrund der MVC-Architektur kann dies nun leicht realisiert werden. Hierbei kapselt man den Inhalt in einem XML-File und transformatiert dieses bei Bedarf mit Hilfe eines XSLT-Stylesheet in das jeweils benötigte Format, wobei für jedes Format ein eigenes Stylesheet benötigt wird. Cocoon unterstützt dabei alle gängigen Ausgabeformate wie z.B. HTML, XML, PDF, WML, RTF und kann um neue Formate erweitert werden. Ferner wird durch die MVC-Architektur die Zerlegung des Gesamtprojekts in Teilaufgaben erleichtert; so wird ermöglicht, dass mehrere Personen unabhängig und gleichzeitig daran arbeiten können. Auch die Erweiterbarkeit der Anwendung wird hierdurch unterstützt.

Cocoon ist außerdem stark durch seine komponenten-basierte Software-Architektur geprägt. Die vorgefertigten Komponenten müssen nur noch in einer dafür vorgesehenen Konfigurationsdatei, der Sitemap, zu sogenannten Pipelines zusammengebaut werden. Hauptbestandteile einer solchen Pipeline sind Generatoren (G), Transformatoren (T) und Serialisierer(S). Dabei ist zu beachten, dass jede Pipeline aus einem Generator, beliebig vielen oder gar keinem Transformator und einem Seralisierer bestehen muss, oder als reglärer Ausdruck: GT*S. Eine Sitemap kann beliebig viele solcher Pipelines beinhalten. Für einen Http-Request wird nun eine Pipeline über einen sogenannten Matcher ausgewählt. Andere mögliche Kontrollflußsteuerungselemente sind Selektoren sowie Aktionen. Ist die Pipe gewählt, erzeugt der dazugehörige Generator einen SAX-Event Strom, der dem nachgeschalteten Transformer als Input dient. Dieser transformiert die Sax-Events nun unter Verwendung eines XSLT-Stylesheets und gibt das Ergebnis erneut als SAX-Event Strom an den Serialisierer weiter. Er gibt das Ergebnis der Pipelineabarbeitung schliesslich als Response zurück. Wie in Abbildung 2 ersichtlich, verfügt Cocoon außerdem noch über ein weiteres Element, den Aggregator. Dessen Aufgabe ist es mehrere XML-Event-Streams zu einem Einzigen

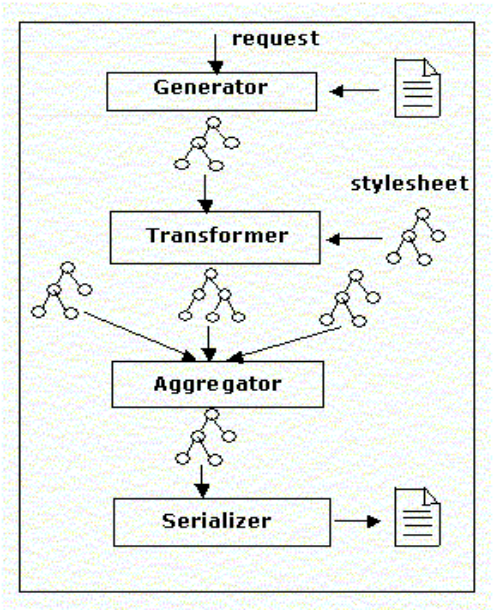


Abbildung 2: Funktionsweise des Pipelinekonzeptes [BK04]

zusammenfügen. Dazu werden die zusammenzufügenden XML-Bäume einfach an ein neues Wurzel-Element gehängt, welches dann als Event-Strom ausgegeben wird. Will man mehrere XML-Dateien gleichzeitig verwenden, kann man alternativ zum Aggregator auch die XPATH-Funktion (siehe Kapitel 2.1.2) document() verwenden. Diese gibt den entsprechenden XML-Baum zurück, den man am besten in einer Variablen speichert und so bequem weiterverarbeiten kann.

Wichtige Vertreter der Generatoren sind FileGenerator, RequestGenerator. FileGenerator und RequestGenerator erzeugen den Event-Strom, wie der Name schon sagt, aus einer Datei bzw. einem Request. TraxTransformer ist der wichtigste Vertreter der Klasse der Transformer. Er transformiert den ankommenden Event-Strom unter Verwendung eines XSLT-Stylesheets. Für unsere Webanwendung kam außerdem ein OWLQL-Transformer zum Einsatz. Dieser bekommt als Eingabe eine OWLQL-Query, die auch in KIF-Syntax geschrieben sein kann, und gibt das Ergebnis der Anfrage zurück. Der Serialisierer gibt nur noch die gewünschte Ressource als Ergebnis zurück. Dies können beim HTMLSerialiser bzw. XMLSerialiser HTML bzw. XML Dateien sein.

- **XSP (eXtesible Server Pages):** Wird Logik benötigt, so stellt Cocoon hierfür XSP bereit. Hiermit lässt sich in XML-eingebetteter Programmcode einer beliebigen Programmiersprache ausführen; meist wird dafür jedoch JAVA verwendet. Um XSP zu benutzen wird in der Sitemap der ServerpagesGenerator benötigt. Dieser erzeugt aus dem JAVA-Code der XSP-Datei

ein .java File, welches einen eigenständigen Generator implementiert, und kompiliert es. Dadarch dass dieser einen SAX-Strom zurückgibt, wird die Integrierbarkeit von XSP in das Pipelinekonzept sichergestellt. In unserer Beispielwebanwendung (siehe Kapitel 2.2.2) benötigten wir zweimal XSP. Das erste Mal, um für ein gegebenes Datum den Wochentag zu berechnen, das andere Mal zur Generierung von wöchentlich wiederkehrenden Terminen. Bei letzterem wurde bei Eingabe eines Datums, die sich wöchentlich wiederholenden Folgetermine berechnet. Dies hätte man sicherlich auch mit XSLT berechnen können, jedoch mit erheblich mehr Aufwand, da JAVA die dazu benötigten Methoden schon in seiner Standardbibliothek bereitstellt. Jede XSP-Datei beginnt mir dem Rootelement <xsp:page>. Die java-Logik wird in <xsp:logik> eingebettet. Wird statischer Text benötigt, so kann er mittels verschacheltem <page><content> angegeben werden. Der im <xsp:logic>-Tag befinierte Code wird mittels <xsp:expr> aufgerufen. Sollte die aufgerufene Methode einen Rückgabewert besitzen , so wird das <xsp:expr>-Tag einfach durch diesen substituiert [LM03].

2.2.2 Sitemap unserer prototypischen Webanwendung

Im Rahmen unseres Software-Entwicklungsprojektes sollten wir auch eine Webanwendung erstellen, welche unter Verwendnung unserer Ontologie die Lehrstuhlseite dynamisch generiert; für deren Realisierung haben wir Cocoon benutzt. Die Lehrstuhlwebsite ist unterteilt in die Kategorien Mitarbeiter-, Lehr-, Forschungs- und Publikationsseite und verfügt außerdem eine Hauptseite sowie eine Homepagesitemap. Für jede Kategorie gibt es eine Indexseite, die eine Liste von Hyperlinks zu den Kategorieinstanzen enthält, d.h. z.B. für die Kategorie Mitarbeiter enthält die Indexseite Hyperlinks zu den einzelnen Mitarbeitern des Lehrstuhls. Jede Seite wird automatisch mit Hilfe von Queries an die Wissensbank generiert. Die Information, die benötigt wird, um festzustellen, welche Anfrage gerade aufzurufen ist, wird statisch in den Links codiert. Da der Ablauf der HTML-Seitengenerierung für jede Kategorie immer gleich abläuft, wurde das Erstellen der Seite in folgende Ressource ausgelagert, die dann je nach Kategorie und der Tatsache, ob es sich um die Index- oder Instanzseite handelt, mit unterschiedlichen Parametern aufgerufen wird.

```
<map:resources>
  <!-- Pipeline zum Erstellen der verschiedenen Seiten -->
  <map:resource name="create-page">


  <!-- Query-Pipeline aufrufen -->
  <map:generate type="file"
src="http://localhost:8080/cocoon/sep/transform/{category}/{instance}/{page-type}"/>

  <map:select
```

```
type="request">
    <map:parameter name="parameter-name" value="display"/>
    <map:when test="query">

<map:transform type="xslt" src="out-html/display-query.xsl">
        <map:parameter name="category"
value="{category}"/>
        <map:parameter name="instance" value="{instance}"/>
        <map:parameter
name="page-type" value="{page-type}"/>
      </map:transform>
    </map:when>
    <map:otherwise>
    <!--
Erzeuge aus Query-Ergebnis Zwischenform der Seite -->
      <map:transform type="xslt"
src="{category}/{page-type}.xsl">
        <map:parameter name="instance" value="{instance}"/>

</map:transform>

      <!-- Sonderbehandlung teaching page-Seite -->

      <map:select type="parameter">

<map:parameter name="parameter-selector-test" value="{category}"/>
        <map:when test="teaching">

<map:transform type="xslt" src="teaching/page2.xsl"/>
        </map:when>
      </map:select>

    <!-- endgueltige HTML-Seite mit entsprechendem Design erzeugen -->
      <map:transform type="xslt"
src="out-html/{design}.xsl"/>
    </map:otherwise>
  </map:select>

  <!-- HTML-Seite ausgeben -->
  <map:serialize type="html"/>
</map:resource>
```

Bei Resourcen gelten die gleichen Bestimmungen bezüglich Generatoren, Transformern und Serialisieren. Der Generator der Resource ruft eine Pipe auf, in der ein OWLQL-Transformer eine Anfrage an die Wissensbank stellt. Bevor der OWLQL-Transformer die Anfrage stellen kann, wird diese erst mittels des Sty-

lesheets build-query.xsl erzeugt. Dies ist notwendig, damit man zum Beispiel einzelne Instanzen (wie z.B. einen speziellen Mitarbeiter des Lehrstuhls), die erst zur Laufzeit bekannt sind, abfragen kann. Dabei wird einfach an die Stelle in der Querie das XML-Tag <instance> eingefügt, welches dann bei der Transformation einfach durch die entsprechende Instanz ersetzt wird. Dieses Vorgehen verkürzt die Dauer der Anfrage enorm, da nicht jede Instanz abgefragt werden muss. Die soeben erstellte Query wird nun an den OWLQL-Transformer weitergeleitet, der die Anfrage an die Datenbank stellt. Im Fall "otherwise" (Der if Teil wird nur für Testzwecke benutzt) wird das Ergebnis der Anfrage mittels eines der Kategorie entsprechenden XSLT-Stylesheet in eine Zwischenform transformiert. Zuletzt erfolgt dann die abschließende Transformation in HTML. Je nach gewähltem Design, das als Requestparameter übergeben wird, wird dies über das entsprechende XSLT-Stylesheet erreicht. Wie bei jeder Pipeline kommt zuletzt ein Serialisierer zum Einsatz, der in diesem Fall HTML ausgibt.

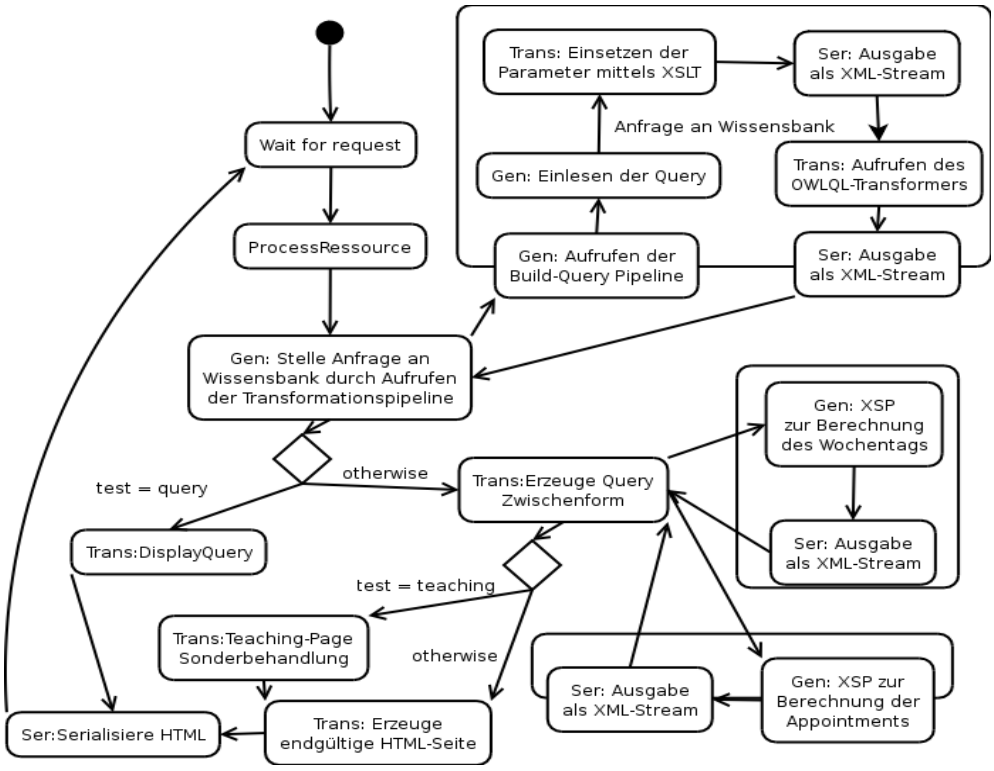


Abbildung 3: Globaler Kontrollfluss unserer Webanwendung dargestellt als UML-Aktivitätsdiagramm

2.3 OWL

2.3.1 Grundlagen

Die Web Ontology Language OWL ist ein Webstandard zur Beschreibung einer Ontologie, d.h. von Konzepten einer Anwendungsdomäne und Relationen zwischen diesen. Sie gehört zum Bereich des Semantic Web, dessen Vision es ist, Webpages durch Anreichern mit zusätzlicher semantischer Information, sogenannter Metainformation, für Agentenprogramme les- und verarbeitbar zu machen. Beispielsweise sollen hiermit bessere Suchmaschinen erstellt werden können, oder auch Systeme, die sich selbständig die notwendigen Informationen aus dem Netz beschaffen.

OWL ist ein Teil des Stacks, der vom W3C [W3C] gegebenen Empfehlungen bezüglich des Semantic Web (siehe Abbildung 4). Grundlage dieses Stacks ist XML sowie XML-Schema. Damit ist es möglich, Dokumente mit sogenannten Metainformationen anzureichern. Leider kann eine Maschine hieraus nicht auf die Bedeutung der XML-Elemente schliessen. Dazu nötig ist RDF, die nächst höhere Stufe des Stacks. Nun ist es möglich Konzepte zu definieren und mit Hilfe sogenannter Tripel in Beziehung zu setzen. Ein Tripel besteht dabei immer aus Subjekt, Prädikat und Objekt und setzt Instanzen zweier Klassen mit Hilfe des Prädikats in Beziehung. Maschinen können nun mit Hilfe der Abfragesprache RDQL Wissen aus diesen Tripeln ableiten und weiterverarbeiten. Leider reicht dies nicht aus, um daraus neues Wissen zu generieren. Diese Lücke soll nun OWL schliessen, indem es RDF um Einschränkungen, die in einer der Prädikatenlogik ähnlichen Syntax verfasst sind, erweitert. Technisch gesehen ist OWL aus der Vereinigung von DAML+OIL entstanden. Logik, Proof and Trust sind noch nicht implementiert. Sie zielen darauf ab, sicherzustellen, ob die Informationen, die im Semantik Web ausgetauscht werden, auch vertrauenswürdig sind.

Um dem Wissensingenieur die Möglichkeit zu geben, zwischen Ausdrucksstärke und Berechenbarkeit zu wählen, wurden drei Teilsprachen von OWL eingeführt. Hier wird nur auf die wichtigesten Unterschiede zwischen diesen Sprachklassen eingegangen. Benötigt man detaillierte Informationen, so sei auf die Seiten des W3C zu OWL verwiesen.

- **OWL** Lite hat ein stark eingeschränktes Vokabular, dafür wird es von allen Inferenztools unterstützt und garantiert volle Berechenbarkeit sowie Entscheidbarkeit. Erlaubt sind Klassen, Properties und Einschränkungen, letztere jedoch nur in sehr begrenztem Umfang. So ist es zum Beispiel erlaubt, die Kardinalität einer Beziehung auf 0 oder 1 einzuschränken.

  Folgende Elemente sind in OWL Lite nicht erlaubt:

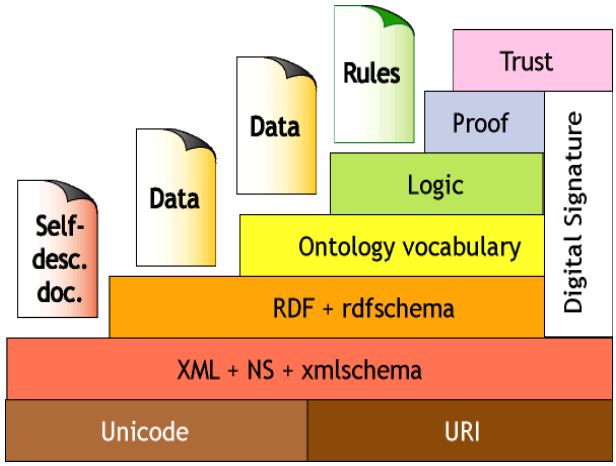


Abbildung 4: Semantik Web Stack des W3C [SEM]

- *owl:oneOf*
- *owl:unionOf*
- *owl:complementOf*
- *owl:hasValue*
- *owl:DataRange*

Bei anderen Tripeln sind Subjekt bzw. Objekt eingeschränkt. Subjekte sind eingeschränkt bei *owl:equivalentClass* und *rdfs:subClassOf*, wohingegen den Objekten bei *owl:allValuesFrom*, *owl:someValuesFrom*, *rdf:type*, *rdfs:domain*, *rdfs:range* gewisse Einschränkungen unterliegen [OWL04b].

- Die nächst mächtigere Subsprache von OWL ist **OWL DL**, wobei DL für description logics steht. Hier wird maximale Ausdrucksstärke erreicht bei gleichzeitiger Erhaltung der Berechen- bzw. Entscheidbarkeit. Hier sind alle Sprachkonstrukte von OWL erlaubt, jedoch mit gewissen Einschränkungen. So kann z.B. eine Klasse nicht die Instanz einer weiteren Klasse sein.

  Außerdem sind die Menge der Objekt Properties und Datatype Properties disjunkt. Daher können für die Elemente

  - *inverse of*
  - *inverse functional*
  - *symmetric*
  - *transitive*

keine Datatype Properties definiert werden.

Wichtig ist auch die paarweise Unterscheidung von Klassen, Datentypen, Datatype- Object- , Annotation - und Ontology-Properties, Instanzen, Datenwerten und den selbst definierten Datenwerten. So ist nicht möglich, dass z.B. eine ObjectProperty gleichzeitig auch eine DatatypeProperty ist [OWL04b].

- Die letzte Sprachklasse ist **OWL Full**. Hier sind alle Sprachkonstrukte von OWL erlaubt, jedoch kann die Berechen- und Entscheidbarkeit nicht sicher gestellt werden. Es gibt keine Inferenzmaschinen die OWL-Full vollständig unterstützen. Dies kann dazu führen, dass Wissen, das mit OWL -FULL ausgezeichnet ist, nicht schlussgefolgert und damit verloren gehen kann. [OWL04b].

2.3.2 Wichtige Sprachkonstrukte

Im folgenden Abschnitt werden die wichtigsten OWL Auszeichnungselemente nämlich Klassen, Properties und Einschränkungen genauer erklärt.

- **Klassen** werden in OWL mit owl:Class definiert. Sie beschreiben ähnlich den objekt-orientierten Sprachen wie z.B. Java eine Menge von Objekten bzw. Individuen. Es können mit der sub-class Anweisung Vererbungsbeziehungen definiert werden. Gibt es in Java nur die einfache Vererbung, so lässt OWL auch multiple Vererbung zu. Dies ist in vielen Fällen nützlich, da es z.B. keine Interfaces gibt. Ähnlich wie in Java gibt es in OWL eine Oberklasse, von der alle anderen Klassen abgeleitet sind; die Klasse Thing. Außerdem können mittels *disjointWith* zwei Klassen als disjunkt definiert werden. Ebenfalls gibt es alle wichtigen Mengenoperationen wie z.B. Schnittmenge, Vereinigung und Komplement.

```
<owl:Class rdf:ID="BachelorThesis">
    ...
    <rdfs:subClassOf>
      <owl:Class rdf:about="#StudentProject"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#DiplomaThesis"/>
    </owl:disjointWith>
     ...

</owl:Class>
```

In diesem Beispiel wird die Klasse *BachelorThesis* definiert. Sie ist eine Unterklasse von *StudentProject*. *BachelorThesis* ist disjunkt zu *DiplomaThesis*,

d.h. ihre Schnittmenge ist leer [OWL04a].

- Es gibt vier verschiedene Arten von **Properties**: Datatype-, Object-, Annotation und OntologyProperties. DatatypeProperties werden verwendet, wenn man die Beziehung zwischen einem Datentyp und einem Konzept beschreiben will. Als Datentypen können alle primitiven Datentypen von XML-Schema wie z.B. string, boolean, float, double etc. benutzt werden [XML]. Objectproperties beschreiben die Relation zwischen zwei Instanzen. Besonders ist, dass die Eigenschaften auch als transitiv, symmetrisch und invers gekennzeichnet werden können. Dies erhöht die Ausdrucksstärke der Ontologie enorm, da hiermit neues Wissen schlussgefolgert werden kann. Annotation- und OntologyProperties werden verwendet, um Metainformationen über die Ontologie abzuspeichern.

  ```
  <owl:TransitiveProperty rdf:ID="isSubOrganisationOf">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
      <owl:inverseOf rdf:resource="#hasSubOrganisation"/>
      <rdfs:domain rdf:resource="#Organisation"/>
      <rdfs:range rdf:resource="#Organisation"/>
  </owl:TransitiveProperty>
  ```

  In diesem Beispiel wird die transitive Property *isSubOrganisationOf* definiert. Es handelt sich dabei um eine Objektproperty, weshalb domain und range definiert werden. Zusätzlich gibt es die inverse Property *hasSubOrganisation*. Ist A eine Unterorganisation von B und B eine von C, so kann ein Reasoner daraus schliessen, dass auch A eine Unterorganisation von C ist. Da zu *isSubOrganisationOf* eine Inverse existiert, kann daraus geschlossen werden, dass B die Unterorganisation A hat. Sowohl die Transitivät als auch die Inversenbeziehung ermöglichen einfache Schlussfolgerungen. Dies erlaubt es, den Speicherplatzbedarf der A-Box, d.h. der Datei, in der die Instanzen gespeichert werden, geringer zu halten, als dies bei relationalen Datenbanken möglich wäre, da z.B. nicht die gesamte transitive Hülle abgespeichert werden muss [OWL04a].

- Zusätzlich können auch noch Einschränkungen auf den Properties definiert werden, sogenannate **Restrictions**. Dies können im einfachen Fällen Kardinalitätsbeschränkungen sein. So ist es sinnvoll, dass die Relation *hatVater* die Kardinalität 1 hat, da jeder genau nur einen Vater hat. Definiert werden die Einschränkungen über das Tag *owl:restriction*, wobei die Property, die eingeschränkt wird mittels *owl:onProperty* angegeben werden muss. Kompliziertere Einschränkungen werden mit *allValuesFrom* bzw. *someValuesOf* definiert. *allValuesFrom* ist dem Allquantor der Prädikatenlogik verwandt, *someValuesOf* dem Existenzquantor. Mit beiden kann die Range der Property eingeschränkt werden.

```
<owl:ObjectProperty rdf:ID="hasOffice">
    <rdfs:range rdf:resource="http://wwwradig.in.tum.de/
           ontology/space-basic#GeographicalThing"/>
    <rdfs:domain rdf:resource="#PersonThing"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Person">
    ...
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:allValuesFrom rdf:resource="http://wwwradig.in.tum.de/
          ontology/space-basic#Office"/>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasOffice"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    ...
</owl:Class>
```

Zuerst wird die Property *hasOffice* definiert mit der Domain *PersonThing* und der Range *GeographicalThing*. Während der Definition der Klasse Person wird der Range von *hasOffice* eingeschränkt. Als Werte sind nur Instanzen von Office erlaubt, welches eine Subklasse von GeographicalThing ist. Wird *hasOffice* im Kontext einer Person benutzt, so kann aus der Range-Restriction geschlossen werden, dass es sich dabei um ein Office handeln muss. Dies ist von besonderem Interesse, wenn es weitere Restrictions von *hasOffice* gibt [OWL04a].

3 Architektur der OWL-Teile

Um die Generierung der Lehrstuhlwebpage mittels geeigneter Anfragen zu bewerkstelligen, benötigen wir eine aussagekräftige Ontologie, welche sowohl zeitliche, räumliche als auch organisatorische Aspekte umfasst. Im Rahmen unseres SEP lag der Schwerpunkt hauptsächlich auf der Erstellung der "Organisations"-ontologie, wohingegen wir zur Realisierung der räumlichen und zeitlichen Aspekte zwei bereits bestehende Ontologien verwendeten und diese unseren Bedürfnissen anpassten. Dies hat den Vorteil, dass die verwendeten Ontologien bereits von mehreren Personen getestet wurden und damit weniger fehleranfällig sind. Ausserdem wird durch Wiederverwendung erheblich Zeit gespart. Für die zeitlichen Aspekte verwenden wir die "time-basic"-Ontologie [TIM] des Cobra-Projektes [CoB], welche eine vereinfachte Version der DAML "time"-ontologie darstellt, für die räum-

lichen Aspekte benutzen wir die “location“-Ontologie. Unsere “Organisations“-Ontologie wurde, wie bei Wissensbanken üblich, in eine A-Box bzw. T-Box aufgeteilt. Die Datei organsiation.owl entspricht der A-Box und beinhaltet die eigentlichen Definitionen über Organisationen, wohingegen organisation-radig.owl die T-Box repräsentiert und die Instanzen, d.h. das aktuelle Wissen über die Domäne, enthält.

Mit Hilfe dieser Ontologie können nun geeignete Anfragen an die Wissensbank gestellt werden, um daraus die entsprechenden HTML-Seiten zu generieren. So werden z.B. für eine Mitarbeiterseite schon alle drei Teilontologien benötigt. In einer einzigen Anfrage werden die persönlichen Daten eines Mitarbeiters (Name, Nachname, Telfon-, Fax- und Raumnummer etc.), die geleiteten Lehrveranstaltungen, die Forschungsschwerpunkte und -interessen), die betreuten Studentenprojekte(Bachelor-, Master- und Diplomarbeit) sowie persönliche Links extrahiert und dann wie in Abschnitt 2.2.2 beschrieben weiterverarbeitet.

3.1 Organisatorische Aspekte

Die Abbildung 5 gibt einen Überblick über die zentralen Klassen unserer Organisationsontologie. *PersonThing*, *GroupOfPersons* und *ThingHasWebsite* sind die in der Vererbungshierarchie allgemeinsten Klassen und besitzen eine ähnliche Funktionalität wie JAVA interfaces, indem sie sehr allgemeine Eigenschaften an speziellere Klassen weitervererben. *GroupOfPersons* beschreibt eine Gruppe von Leuten mittels der Property *hasPersonMember* sowie einen Leiter der Gruppe mittels *isHeadedBy*. Ausserdem besitzt es die Eigenschaft *hasOffice*, welche sie von *PersonThing* erbt. *ThingHasWebsite* beschreibt die Eigenschaften einer Website wie z.B. Bild, Link, Beschreibung sowie NewsItem.

Die wichtigen Klassen, um die organisatorische Struktur des Lehrstuhls beschreiben zu können sind *Organisation*, *Person*, *Course*, *Publication*, *ResearchProject* sowie *Appointment*.

- **Organisation:** Kapselt die wichtigsten Eigenschaften einer Organisation wie den Namen, mögliche Unterorganisationen sowie die Zahl der Mitarbeiter. Außerdem besitzt sie die zwei Unterklassen *Company* und *SientificOrganisation*, wobei letztere ihre Eigenschaften an die *University*, *ResearchGroup*, *Department* und *Chair* weiterleitet. Diese Klassen ermöglichen es, die verschieden Organisationseinheiten einer Universität und damit des Lehrstuhls zu beschreiben.

- **Person:** Diese Klasse beschreibt die Merkmale einer Person. Zu den Datatypeproperties gehören Name, Vorname, Telefonnummer, Geburtstag, Email, zu den Objectproperties *hasScientificTitle*, *hasOffice* etc. Von Person abgeleitet sind *Student* und *Employee*, wobei *ScientificEmployee* die letzt genannte Klasse nochmals verfeinert, und *Professor* und *Assistent* in einer Generalisierungsbeziehung stehen. Die Klassen *Person* und *Organisation* bilden bereits ein Grundgerüst, um die Anfrage zur Erstellung der Mitarbeiterindexseite in OWLQL zu formulieren.

- **Appointment:** Diese Einheit dient zur Beschreibung von Terminen. *hasParticipant* erlaubt es, die Teilnehmer an einer Besprechung zu speichern. Wichtigste Unterklasse ist *SingularAppointment*. Da sie von *loc:ThingHasLocationContext* und *tme:IntervalEvent* abgeleitet ist, kann man hiermit alle für Termine benötigten Informationen, d.h. vor allem Beginn und Ende sowie Ort, ablegen und sollte deshalb hauptsächlich Verwendung finden.

- **Course:** Verhält sich wie eine abstrakte JAVA-Klasse, die allgemeine Eigenschaften für Lehrveranstaltungen kapselt. Hat eine Referenz auf *FirstLesson*, welche wiederum von *SingularAppointment* abgeleitet wurde und besitzt damit einen Anfangs- und Endzeitpunkt, sowie einen Ort. Außerdem wird abgespeichert, ob es sich um eine Vorlesung im Grund- bzw.

Hauptstudium handelt. Werden Vorlesungen vorausgesetzt, so können diese mittels *hasPrerequisite* referenziert werden. Mit Unterklassen können Kurse in Vorlesungen, Seminare und Praktika eingeteilt werden.

- **Publication:** Wird verwendet, um die Publikationsseite der Webpage zu generieren, und enthält dafür notwendige Information wie Titel, Publizist, Veröffentlichungsdatum, Verweise auf andere Publikationen etc.
- **ResearchProject:** Wird verwendet, um die Forschungsseite zu erstellen und beinhaltet neben Titel des Projects, auch Mitglieder bzw. Leiter der Gruppe und weitere Eigenschaften, die in *GroupOfPersons* definiert wurden.

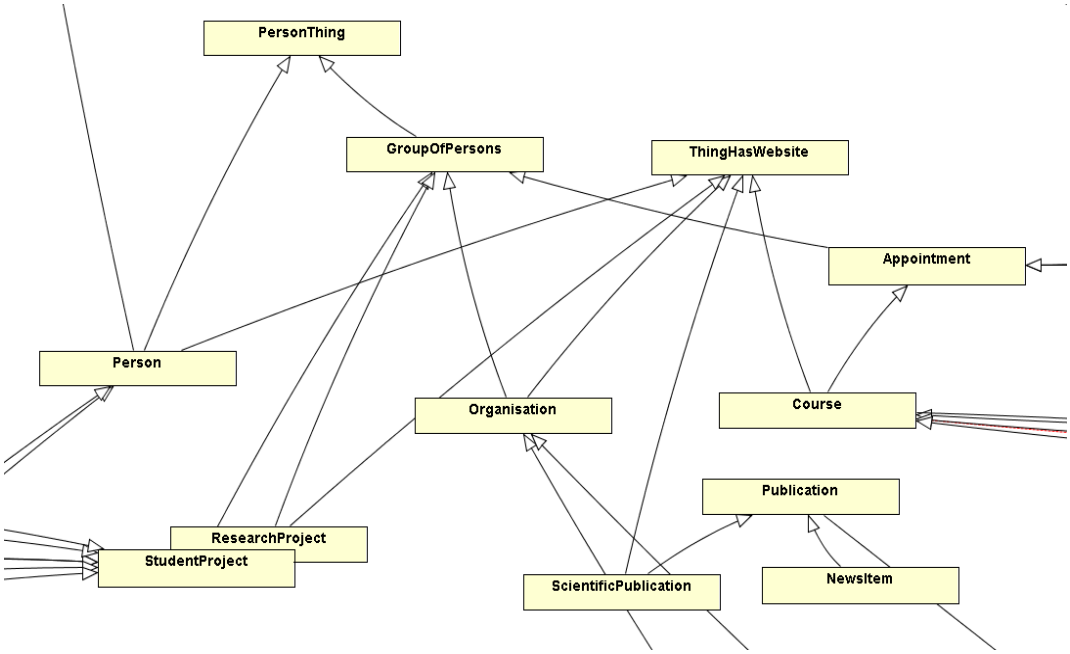


Abbildung 5: Überblick über die wichtigsten Klassen und Vererbungsbeziehungen der “Organisations“-Ontologie

3.2 Zusammenhang der Ontologien

tme:InstantEvent, *tme:IntervalEvent*, *loc:LocationContextDescription* und *loc:ThingHasLocationCont*
sind Klassen, die zu den importierten Ontologien gehören und mittels Vererbung
an unsere Ontologie angepasst werden, wohingegen *spc:Office* und *spc:hasFixedStructure*
durch Restrictions referenziert werden. Daneben gibt es noch eine Reihe von Propertyreferenzen, welche jedoch aus Gründen der Übersichtlichkeit nicht im Diagramm erscheinen. *Person* befindet sich jedem Zeitpunkt an einen bestimmten

LocationContext und erbt deshalb von *loc:ThingHasLocationContext* die Property *loc:locationContext* und kann somit auf beliebige LocationContext's verweisen. *SingularAppointment* erbt von den beiden Klassen *tme:IntervalEvent* und *loc:ThingHasLocationContext*. Damit ist es möglich Besprechungen Ort und Zeit (Beginn, Ende) zuzuweisen. *Publication* und *Birthday* sind Ereignisse, die festen Zeitpunkten zugewiesen werden können und erben deshalb von *tme:InstantEvent*.

Wird *hasOffice* im Context von Organisationen verwendet, so verweist sie mindestens einmal auf die Klasse *spc:FixedStructure*; wird sie jedoch von der Klasse *Person* verwendet, so handelt es sich dabei immer um ein *Office*. Die Eigenschaft *loc:locationContext* verweist bei Personen immer auf *loc:PersonLocationContext*

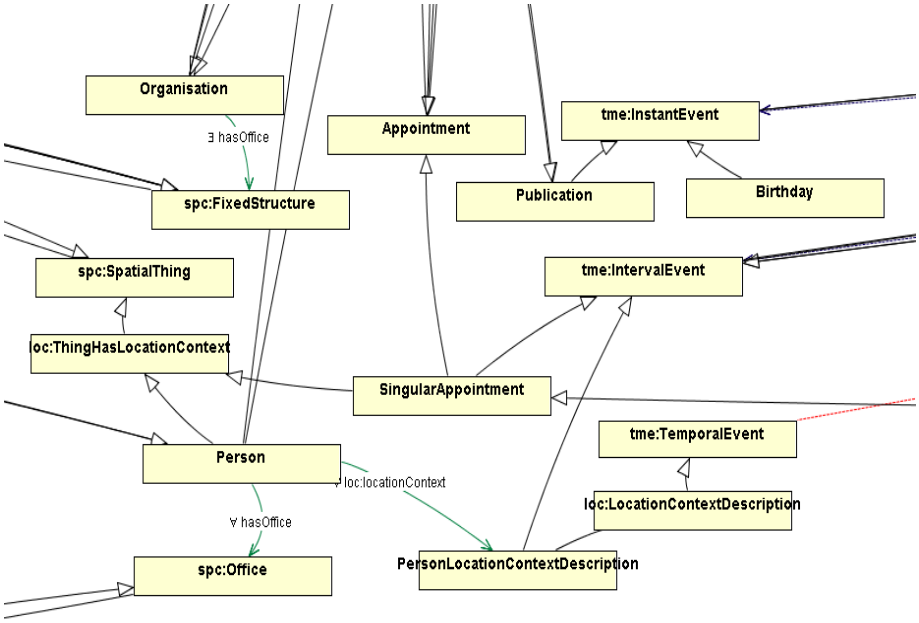


Abbildung 6: Berührungspunkte der verwendeten Ontologieen

Literatur

[BK04] Prof. Brüggemann-Klein. *XML-Praktikum*, 2004. http://www11.in.tum.de/ brueggem/Praktika/xmlTechnologieWS2004/Folien/G.Cocoon.pdf.

[CoB] *Context Broker Architecture (CoBrA)*. http://cobra.umbc.edu/about.html.

[COC] *The Apache Cocoon Project.* http://cocoon.apache.org/.

[LM03] Jeremy Aston Lajos Moczar. *Cocoon, Developer's Handbook.* Developer's Library, 2003.

[OWL04a] *OWL Web Ontology Language Overview*, 2004. http://www.w3.org/TR/owl-features/.

[OWL04b] *OWL Web Ontology Language Guide*, 2004. http://www.w3.org/TR/2004/REC-owl-guide-20040210/.

[SEM] *W3C FI und W3C Semantic Web.* http://www.w3c.tut.fi/talks/2002/0923sw-vtt-on/.

[TIM] *Entry Sub-Ontology of Time.* http://www.isi.edu/ pan/damltime/time-entry-documentation.txt.

[W3C] *World Wide Web Consortium.* http://www.w3.org/.

[XML] *XML Schema Part 2: Datatypes Second Edition.* http://www.w3.org/TR/xmlschema-2/datatypes.html.