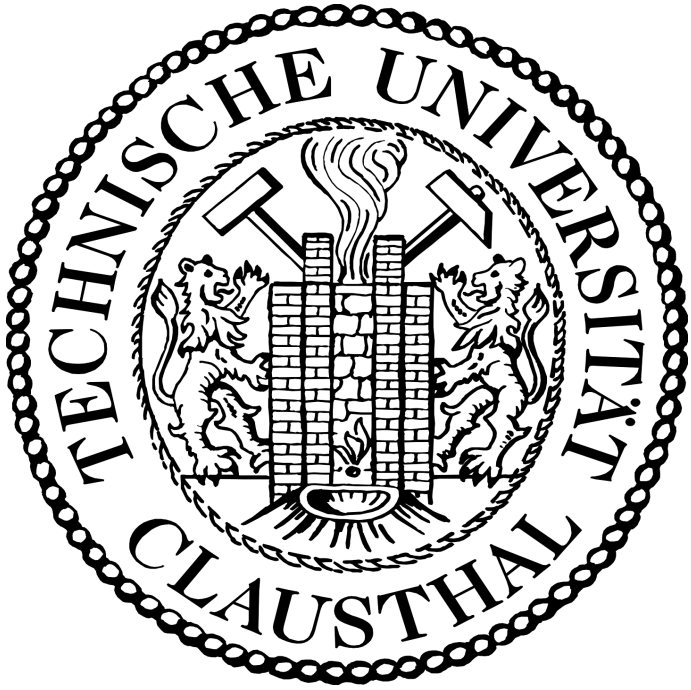
Entwicklung eines RDF-Web-Services mit Trigger-Funktionalität

Diplomarbeit

Elke von Lienen



Betreut von

Professor Dr. Wolfgang May
Universität Göttingen

Technische Universität Clausthal
Institut für Informatik

Versicherung

Hiermit versichere ich, die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Clausthal, den 7. März 2006

Elke von Lienen

Vorwort

Das Erstellen dieser Diplomarbeit stellte mich immer wieder vor Probleme und dauerte länger als geplant. Ohne Unterstützung wäre ich wohl nicht so weit gekommen. Ich möchte diese Gelegenheit nutzen, um einigen Leuten einmal „Danke“ zu sagen.
Zunächst danke ich Herrn Professor May für die Möglichkeit, eine Diplomarbeit, die in Zusammenhang mit Datenbanken steht, schreiben zu können, obwohl wir in Clausthal zur Zeit wieder keinen Datenbankprofessor haben. Ihm und seinen Mitarbeitern danke ich außerdem für die gute Betreuung während der Arbeit und die schnelle Hilfe bei Problemen. Insbesondere danke ich Franz Schenk, der viel Zeit darin investierte, die Laufzeitumgebung in Göttingen an mein Programm anzupassen.
Des Weiteren danke ich Herrn Professor Dix für die Übernahme der Zweitkorrektur dieser Arbeit.
Ich danke meinen Freunden und meiner Familie für die Unterstützung und die erfreuliche Ablenkung die sie mir in dieser Zeit zukommen ließen. Besonderer Dank geht dabei an Florian, der immer für mich da war, und Markus, der immer ein offenes Ohr für meine Probleme hatte. Ich danke außerdem Sebastian für den kreativen Austausch und gemeinsame Göttingen-Fahrten, Kai und Ansgar, die ich immer mit meinen Linux- und LaTeX-Problemen nerven konnte und Stefan, der mich erkennen ließ, dass ich mit den Jena-Problemen nicht alleine bin.
Herzlichen Dank außerdem an alle fleißigen Korrekturleser, die sich durch diese Arbeit gekämpft haben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Diese Arbeit ist im Zusammenhang mit dem Semantic Web zu sehen. Dabei werden Daten so dargestellt und vernetzt, dass ihr Zusammenhang automatisch von Maschinen erfasst werden kann. Zudem können Maschinen Daten aus anderen Anwendungen für ihre Zwecke nutzen, ohne dass diese Daten speziell für diesen Einsatzbereich erstellt wurden.
Als ein Einsatzgebiet eignet sich dabei zum Beispiel die Planung und Buchung von Reisen. Im konventionellen Web ist es zwar durchaus möglich, eine Reise online zu buchen und auch zu bezahlen, jedoch muss der Benutzer hier noch einiges an eigenem Wissen einbringen. Zum Beispiel muss der Reisende bei einer Flugreise wissen, wo die Flughäfen mit den kürzesten Entfernungen von seinem Ziel- und dem Ausgangsort liegen. Für die Planung ist es außerdem hilfreich, wenn der Benutzer weiß, wie er sich die Fahrpläne der örtlichen Busgesellschaft beschaffen kann, die ihn vom Flughafen oder einem Bahnhof zum Zielort bringt.
Mit Hilfe des Semantic Webs soll es möglich sein, solche Informationen automatisch zu ermitteln. Auch auf Änderungen wie gestrichene Flüge sollte das System reagieren können, indem der Reisende informiert wird und ihm mögliche Alternativen vorgeschlagen werden.
Das Semantic Web gibt es so zur Zeit noch nicht, aber es gibt einige Grundlagen dazu, die in verschiedenen Projekten weiterentwickelt werden. Um die Inhalte des World Wide Webs (WWW) zu verarbeiten, werden dabei sogenannte Metadaten eingesetzt. Dies sind Daten über Daten, wie der Autor oder das Datum der letzten Änderung des Dokuments, aber auch eine Beschreibung des Inhalts zählt zu den Metadaten. Zur Darstellung von Metadaten wird im Semantic Web normalerweise RDF (Resource Description Framework) verwendet. Hierdurch können nicht nur einfache Informationen weitergegeben werden, sondern auch die Beziehungen zwischen den Daten ausgedrückt werden. Für diesen Zweck werden Ontologien bereitgestellt, welche die grundlegenden Begriffe und Zusammenhänge eines Wissensgebiets beschreiben.

Das ganze Potential des Semantic Webs ist jedoch nicht mit der Verfügbarkeit der Metadaten ausgeschöpft. Das Semantic Web kann, wie auch das WWW, nicht als statische Datenquelle verstanden werden. Einer der großen Vorteile ist, dass es sich ständig ändert und so aktuelle Daten zur Verfügung stellen kann. Es ist daher wichtig, dass diese Änderungen erfasst werden und anderen Parteien bekannt gemacht werden.
Für diesen Zweck können ECA-Regeln eingesetzt werden. E steht dabei für Event (Ereignis), C für Condition (Bedingung) und A für Action (Aktion). Wenn das hier vorgegebene Event eintritt, werden die angegebenen Bedingungen ausgewertet und schließlich eine Aktion ausgeführt.
Im Beispiel der Reiseplanung ist ein mögliches Event, dass ein Flug Verspätung hat. Die Bedingung ist, dass die Verspätung rechtzeitig bekannt ist, zum Beispiel einige Stunden vor dem Abflug. Als Aktion werden die Fluggäste über die Verspätung informiert.
Eine spezielle Form von ECA-Regeln sind Trigger. Sie sind normalerweise auf dem gleichen Server implementiert, auf dem die Änderungen stattfinden und können nur auf diese reagieren. Auch die Bedingung kann nur mit den Daten auf diesem Server zusammenhängen. Aktionen sind häufig ebenfalls lokale Änderungen an den Daten, es können jedoch auch Fehlermeldungen oder Nachrichten ausgegeben werden.

In dieser Arbeit werden Trigger für RDF-Daten entwickelt. Die Trigger reagieren auf Änderungen an den Daten und können selbst lokale Änderungen vornehmen, aber auch Nachrichten an andere Web-Services versenden. Dort können diese Nachrichten als Events in anderen ECA-Regeln wirken.
Hierfür wurde ein Server entwickelt, der RDF-Daten bereitstellt. An diese Daten können Anfragen gestellt werden und es können Veränderungen vorgenommen werden. Das Hinzufügen, Löschen und Ändern kann dabei die Trigger auslösen. Wie auch die RDF-Daten werden die Trigger in einer Datenbank gespeichert, so dass diese nach einem Neustart des Servers nicht erneut eingegeben werden müssen. Die Kommunikation mit dem Server erfolgt dabei vollständig über SOAP, einem XML-Format zum Versenden von Nachrichten über das Internet.
Außerdem entstand in dieser Arbeit ein Client mit einer graphischen Oberfläche, der es ermöglicht, die Daten und Trigger übersichtlich darzustellen. Anfragen, Änderungsoperationen und Trigger können über den Client bequem erstellt und versandt werden. Mit Beispieleingaben und anderen Hilfen, wie der Erkennung von Schlüsselwörtern unabhängig von der Groß- und Kleinschreibung, wurden diese Programme möglichst benutzerfreundlich gestaltet.

In Kapitel 2 werden einige grundlegende Begriffe eingeführt, die für diese Arbeit von Bedeutung sind. Kapitel 3 beschäftigt sich mit dem Resource

Description Framework (RDF). Hier wird auch näher auf die Syntax und RDQL, eine Anfragesprache für RDF, eingegangen.
Für das Programm wurde das Jena Framework für das Semantic Web genutzt, welches in Kapitel 4 betrachtet wird. In Kapitel 5 folgen dann einige allgemeine Aspekte, die ich zu den hier entwickelten Programmen anmerken möchte. Der Client wird in Kapitel 6 beschrieben, in Kapitel 7 wird auf den Server eingegangen. Die Trigger selbst werden jedoch erst im folgenden Kapitel ausführlich behandelt. In Kapitel 9 werden noch zwei kleine zusätzlich entwickelte Programme vorgestellt, die für die Funktion zur Zeit notwendig sind. In Kapitel 10 werden weitere Möglichkeiten für die Auslösung von Triggern diskutiert. Schließlich folgen in Kapitel 11 mögliche Weiterentwicklungen und Verbesserungen der Programme.

Kapitel 2

Einordnung und grundlegende Begriffe

2.1 Semantic Web und REWERSE

Unter dem Semantic Web versteht man eine Erweiterung des World Wide Webs (WWW). Das Ziel ist es hierbei das Web „intelligenter“ zu machen, indem die Inhalte für Maschinen interpretierbar werden. Die Semantik der Information wird dabei über eine Ontologie festgelegt. Dies ist eine Art Vokabular, durch das Informationen wie URLs mit einer Bedeutung verbunden werden. Über die Ontologie können weitere Daten abgeleitet werden. Zur Darstellung der Daten und Ontologien werden dabei RDF und OWL verwendet. Diese werden in Kapitel 3 näher erläutert.

REWERSE ist die Abkürzung eines EU-weiten Projekts, welches im Zusammenhang mit dem Semantic Web steht. REWERSE steht dabei für „Reasoning on the Web with Rules and Semantics“. Das Ziel dieses Projektes ist es, sogenannte „Reasoning“-Sprachen für das Web zu entwickeln und diese zu testen. Es sollen außerdem die nötigen technischen Grundlagen für den Einsatz dieser Sprachen entwickelt werden. Reasoning steht im Englischen für Schlussfolgern, was hier automatisch mit der festgelegten Semantik erfolgen soll.
Innerhalb von REWERSE gibt es verschiedene Arbeitsgruppen. Die Gruppe I5 trägt den Titel „Evolution and Reactivity“ und beschäftigt sich mit Änderungen, die mit der Zeit im Semantic Web auftreten. Diese Änderungen müssen durch das Semantic Web weitergegeben werden. Ein großes Problem ist dabei die Frage, wie alle Knoten von den für sie interessanten Änderungen erfahren. Beispielsweise möchte ein Reisebüro von Änderungen im Zugfahrplan informiert werden. Ein neues Rezept für Pflaumenkuchen ist für diesen Service jedoch nicht von Interesse.
Für diesen Zweck schlägt die Arbeitsgruppe ECA-Regeln vor. Der Benutzer

definiert hierin ein Event, auf das reagiert werden soll. Tritt dieses Event ein, so werden die definierten Bedingungen überprüft. Schließlich wird die festgelegte Aktion ausgeführt. Eine einmal festgelegte Regel wird bei jedem Auftreten des Events automatisch ausgeführt.
Für die Realisierung der ECA-Regeln müssen verschiedene Komponenten entwickelt werden. Die Funktion der einzelnen Komponenten kann an Abbildung 2.1 nachvollzogen werden. Dies dient auch der Einordnung des in dieser Arbeit entwickelten Programms.
In der ECA-Engine können globale ECA-Regeln festgelegt werden. Es wird hier genau definiert, wie das Event aussehen muss, was für eine Bedingung gelten soll und welche Aktion erfolgen soll. Die **Domain-Knoten** stellen hier die eigentlichen Daten zur Verfügung. Hier können von den Benutzern Änderungen an den Daten vorgenommen werden. Auf manche dieser Änderungen reagieren die dort vorhandenen Trigger. Zu den **Domain-Knoten** gehört auch der in dieser Arbeit entwickelte Server. Nach Updates können die Trigger in diesem Server nicht nur lokale Änderungen vornehmen, sondern auch Events weitersenden. Diese Events werden an den sogenannten **Eventbroker** geschickt. Der Eventbroker schickt die Events an die bei ihm registrierten **Atomic Event Matcher**. Ein Atomic Event Matcher ermittelt die nötigen Variablenbindungen für die bei ihm registrierten Events. Diese werden entweder direkt von der **ECA-Engine** registriert oder von einem **Composite Event Detector**. Letzterer setzt nach festgelegten Regeln aus den einfachen ‚Atomic Events' komplexere ‚Composite Events' zusammen. Ist eine entsprechende Variablenbelegung gefunden, so wird diese zurück an die ECA-Engine geschickt. Diese ruft nun mit den gefundenen Variablenbelegungen die **Condition-Komponente** auf. Von dieser können weitere Variablenbelegungen bestimmt werden. Schließlich wird die **Action-Komponente** mit allen bisher bestimmten Variablen aufgerufen. Als Aktion werden hier wiederum Änderungen an den Daten vorgenommen. In der in dieser Arbeit entwickelten Komponente sind dies Änderungen an der RDF-Datenbasis. Die hier beschriebenen Komponenten setzen globale ECA-Regeln um. Global bedeutet dabei, dass sie auf verschiedenen Servern arbeiten. Die Aktion kann zum Beispiel auf einem anderen Server ausgeführt werden als dem Server, von dem aus das Event ursprünglich ausgelöst wurde. Die Trigger stellen dagegen spezielle lokale ECA-Regeln dar, die innerhalb eines Datenbankservers wirken.

2.2 SOAP

SOAP stand eigentlich für **S**imple **O**bject **A**ccess **P**rotokoll, jedoch dient es heute nicht mehr nur für den Zugriff auf Objekte. Die Bedeutung von SOAP hat sich mit der Zeit jedoch geändert, so dass SOAP nun als feststehender Begriff verwendet wird. Mittels SOAP werden Daten zwischen verschiede-

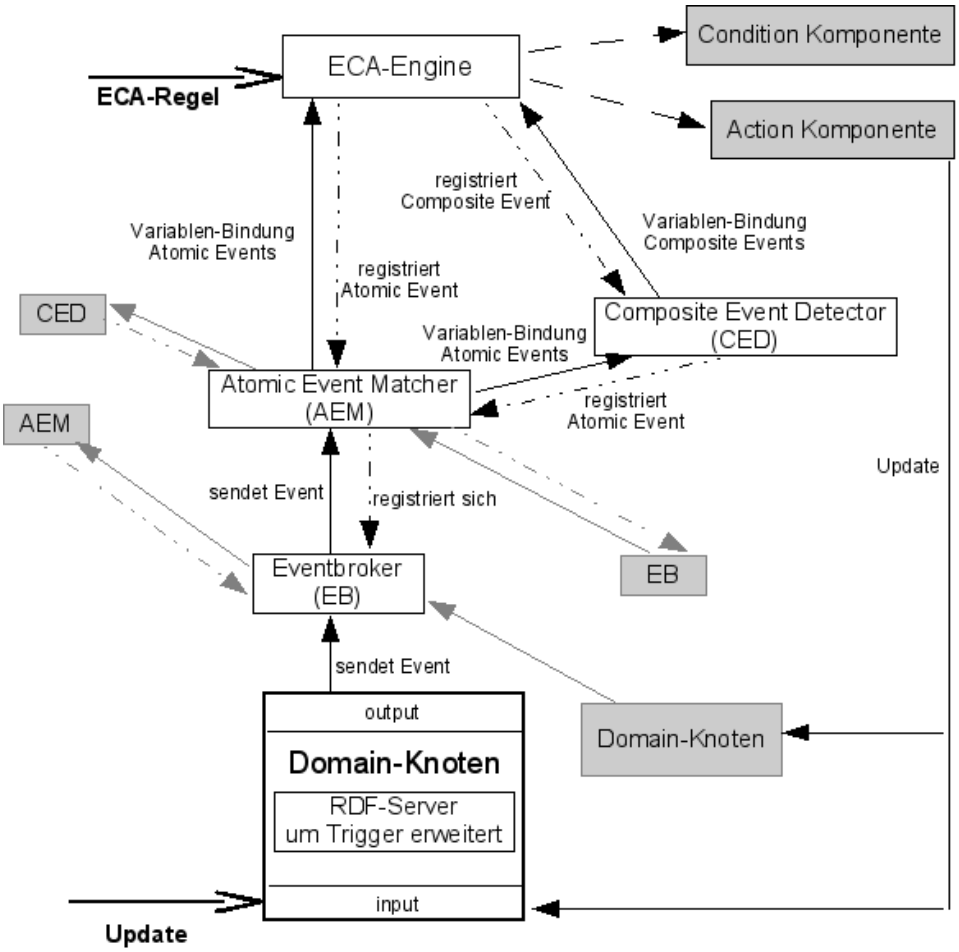


Abbildung 2.1: Umsetzung der ECA-Regeln

nen Systemen übertragen. Das wichtigste Ziel bei der Entwicklung war die Systemunabhängigkeit. SOAP-Nachrichten werden über das http-Protokoll übertragen. Es existieren nicht nur für verschiedene Systeme, sondern auch für etliche Programmiersprachen Implementierungen von SOAP. Die grundlegende Syntax basiert auf XML. Jede SOAP-Nachricht sieht dabei wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.
    xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-
         instance"
     <soapenv:Header>
          Header
     </soapenv:Header>
     <soapenv:Body>
          Inhalt der Nachricht
     <soapenv:Body>
</soapenv:Envelope>
```

Das XML-Tag *soapenv:Header* ist optional. Innerhalb von *soapenv:Body* ist der Inhalt der Nachricht wiederum in XML verfasst. Im einfachsten Fall ist das ein Text. Es können jedoch weitere XML-Elemente folgen. Auch eine Verschachtelung ist möglich. Wichtig ist dabei, dass jedes Tag, das geöffnet wird, auch wieder geschlossen wird. Außerdem ist es möglich ein Attachment an die Nachricht anzuhängen. Das Attachment kann jedes mögliche Format haben, welches im zugehörigen Header beschrieben wird. Dadurch weiß der Empfänger, wie er die Daten interpretieren muss. So ist es zum Beispiel möglich, Bilder zu verschicken.
Bei der Kommunikation über SOAP gibt es prinzipiell zwei Möglichkeiten: Die Nachricht synchron oder asynchron zu versenden. Beim synchronen Versand schickt der Empfänger nach Erhalt der Nachricht eine Antwort zurück. Bei einem asynchronen Versand erhält der Sender keine Informationen darüber, ob die Nachricht den Empfänger erreicht hat.

2.3 Web-Service

Eine Definition des Wortes Web-Services gibt es leider nicht. Die Beschreibungen in diversen Quellen unterscheiden sich in einigen Punkten. Jedoch ist allen gemeinsam, dass es sich dabei um ein Software-Programm handelt, auf das über das Internet zugegriffen wird. Aus dem Wort „Service“ lässt sich auch ableiten, dass hier ein Dienst zur Verfügung gestellt wird. Man

kann nach dieser Definition zum Beispiel das Online-Banking oder die Fahrplanauskunft im Internet als Web-Service ansehen.
Einige Quellen fordern, dass bei einem Web-Service ein XML-Format für das Versenden von Nachrichten genutzt wird, während andere dies völlig offen lassen.

2.4 Trigger

Das Wort „Trigger“ wird in vielen verschiedenen Zusammenhängen benutzt, zum Beispiel in der Elektronik aber auch in der Medizin. Es hat immer die Bedeutung eines Auslösers oder Schalters. Auch in der Datenbankwelt sind Trigger verbreitet und auch hier erfüllen sie eine Schaltfunktion.
Ein Trigger löst eine Aktion immer dann aus, wenn ein definiertes Ereignis eingetreten ist, normalerweise das Einfügen, Löschen oder Ändern von Daten. Trigger sind eine spezielle Form von ECA-Regeln, die lokal arbeiten. Die Trigger sind auf dem gleichen Server zu finden wie die Daten, auf deren Änderung sie reagieren. Als resultierende Aktion können Trigger entweder selbst Änderungen an den lokalen Daten vornehmen oder z.B. Fehlermeldungen ausgeben.
Bei relationalen Datenbanken bilden Trigger die einzige Möglichkeit, dass Änderungen an einer Relation Auswirkungen auf eine andere Relation haben.
Als Beispiel kann eine Firmendatenbank betrachtet werden, die zur Speicherung von Kundendaten dient. In einer weiteren Relation werden die offenen Bestellungen gespeichert. Wird nun ein neuer Kunde eingetragen, so kann ein Trigger dafür sorgen, dass für den neuen Kunden automatisch bei den offenen Bestellungen ein Begrüßungsgeschenk eingetragen wird.
Die Datenbank enthält dabei die Relation „benutzerdaten“ und die Relation „bestellungen“. In „benutzerdaten“ ist neben den persönlichen Daten des Kunden auch eine Kundennummer gespeichert, die diesen Kunden identifiziert. Die Relation „bestellungen“ beinhaltet neben einer Bestellnummer die Kundennummer, die Nummer des zu versendenden Artikels, die Anzahl, in der der bestellte Artikel versandt werden soll und das Datum, an dem die Bestellung die Firma verlassen hat. Dieses Datum kann natürlich noch nicht gesetzt werden, wenn die Bestellung eingegeben wird.
In SQL wird der entsprechende Trigger dabei wie folgt formuliert:

```
CREATE TRIGGER begruessungs_geschenk
AFTER INSERT ON benutzerdaten
FOR EACH ROW
DECLARE
bestellnummer Number;
BEGIN
    SELECT nextval(bestellungen_bestellnummer) INTO
```

```
        bestellnummer;
    INSERT INTO bestellungen(bestellnummer,
        kundennummer, artikelnr, anzahl, versanddatum)
    VALUES (bestellnr, :new.kundennummer, '001', '1',
        null);
END;
```

Der hier definierte Trigger mit dem Namen „begruessungs_geschenk“ reagiert dabei auf jede Zeile, die neu in die Tabelle „benutzerdaten“ eingetragen wird. Dann wird die nächste mögliche Bestellnummer bestimmt und in der Variable *bestellnummer* gespeichert. Mit dieser Bestellnummer wird eine neue Zeile in die Relation „bestellungen“ eingetragen. Dabei ist die Kundennummer die Nummer des neu eingetragenen Kunden, auf die mit *:new.kundennummer* zugegriffen werden kann. Alle anderen Werte sind fest vorgegeben. Dabei soll „001“ die Artikelnummer des Begrüßungsgeschenks sein. Die Angabe „null“ beim Versanddatum bedeutet, dass es für dieses Attribut keinen Wert gibt. Die genaue Syntax eines Triggers in SQL ist bei den verschiedenen Datenbanksystemen leider unterschiedlich. Die oben verwendete Syntax entspricht der von Oracle.
Das hier vorgestellte Prinzip sollte auf RDF-Daten übertragen werden. Natürlich kann die Syntax dabei nicht genauso aussehen wie in SQL, jedoch bleiben die wesentlichen Aspekte erhalten.

Kapitel 3

RDF

RDF steht für Resource Description Framework und wurde 1999 vom World Wide Web Consortium (W3C) erstellt. Es handelt sich dabei um ein Modell zur Darstellung von Metadaten. Heute dient RDF als eine der Grundlagen des Semantic Webs.
Mit RDF können nicht nur typische Metadaten ausgedrückt werden, wie der Autor einer Webseite oder das Datum der letzen Änderung. Ein Onlineshop kann mit RDF beispielsweise seine angebotenen Artikel beschreiben, was Kunden das Suchen nach einem bestimmten Artikel erheblich erleichtern kann. Die Daten werden dabei in einem bestimmten Format gespeichert, so dass es möglich ist, ihre Bedeutung (Semantik) zu erfassen.
Diese Arbeit basiert auf RDF. Durch den Server werden Daten im RDF-Format bereitgestellt, welche durch Updates ergänzt, verändert oder gelöscht werden können. Die Trigger reagieren dabei nicht auf die formale Aktion „es wurde etwas eingefügt“, sondern z.B. auf das Einfügen einer neuen Instanz einer bestimmten Klasse. Daher wird an dieser Stelle RDF mit den wichtigsten Aspekten der Syntax und der Semantik vorgestellt.

3.1 URI

Wie der Name schon vermuten lässt, sind Ressourcen im Zusammenhang mit RDF ein wichtiger Begriff. Eine Ressource ist alles, was durch eine URI (Abkürzung für Uniform Resource Identifier) identifiziert werden kann. Dies kann eine Webseite sein, aber auch ein einzelnes Element innerhalb eines XML-Dokumentes oder eine E-Mail-Adresse.
Die URI kann sich auf verschiedene Bereiche beziehen. Am Anfang steht daher immer ein Schema, welches den Zusammenhang und die Syntax festlegt. Es gibt eine Vielzahl von Schemata, so dass es unmöglich ist alle hier aufzuzählen. Zudem ist die Syntax einer URI nicht festgelegt, so dass ständig neue entstehen können. Einige Schemata sind jedoch sehr verbreitet: „http“ (Hypertext Transfer Protocol) wird für die Darstellung von Webseiten ver-

wendet, „ftp“ (File Transfer Protocol) dient zur Übertragung von Dateien, meist zwischen einem Client und einem Server, „mailto“ dient zur Beschreibung von E-Mail-Adressen und „news“ kennzeichnet Newsgroups. Es gibt auch die Möglichkeit, „URN“s einzusetzen. Dies ist die Abkürzung für Uniform Resource Names, welche Objekte dauerhaft und ortsunabhängig identifizieren. Ein Beispiel hierfür ist die Angabe der ISBN zur Identifikation eines Buches (urn:isbn:3-8273-7038-8). Die im WWW gebräuchlichen URLs sind nur ein Teil möglicher URIs.

3.2 Ontologie

In der Informatik beschreibt der Begriff Ontologie eine Repräsentation der grundlegenden Begriffe und Zusammenhänge eines Wissensgebiets. Ein menschlicher Leser besitzt Hintergrundwissen und hat Zugang zu Nachschlagewerken, mit denen er das erworbene Wissen einordnen kann. Wird das Wissen einer Maschine übergeben, so muss dieses Wissen über die Hintergründe anderweitig erworben werden. Wird einem menschlichen Leser zum Beispiel mitgeteilt, dass Niedersachsen in Deutschland liegt und dass Clausthal in Niedersachsen liegt, so ist für ihn sofort ersichtlich, dass Clausthal in Deutschland liegt. Eine Maschine könnte diesen Schluss jedoch nicht ziehen. Für die Bereitstellung dieses Wissens nutzt man Ontologien. Dabei werden formal Begriffe und ihre Zusammenhänge definiert. Zur Darstellung wird eine formale Repräsentation verwendet, im Bereich des Semantic Webs ist dies in den meisten Fällen RDF bzw. OWL. Eine Ontologie beschreibt immer nur einen Bereich, jedoch können verschiedene Bereiche durch weitere Ontologiedaten miteinander verknüpft werden.

3.3 RDF-Syntax

Die RDF-Syntax basiert auf Tripeln, welche auch oft als Statements bezeichnet werden. Die Bezeichnung Statement (engl. für: Satz) symbolisiert den Aufbau dieser Tripel. Ein Statement besteht jeweils aus den drei Bestandteilen Subjekt, Prädikat und Objekt. Das Subjekt und das Objekt sind hierbei Ressourcen, während das Prädikat die Verbindung zwischen den beiden Ressourcen beschreibt. Dies ist im Normalfall eine Eigenschaft des Subjekts, weshalb das Prädikat häufig auch als „Property“ bezeichnet wird. Alle Subjekte und Prädikate werden dabei durch URIs repräsentiert. Das Objekt kann entweder eine URI oder ein konstanter Wert sein.

Mit Hilfe dieser Tripel wird nun ein sogenanntes „Modell“ aufgebaut. Dabei handelt es sich um einen Graphen, bei dem Subjekt und Objekt eines Statements Knoten darstellen, während das Prädikat eine Kante zwischen ihnen bildet. Dabei werden natürlich die gleichen URIs auf einen Knoten abgebildet, so dass ein Knoten mit mehreren Kanten versehen sein kann.

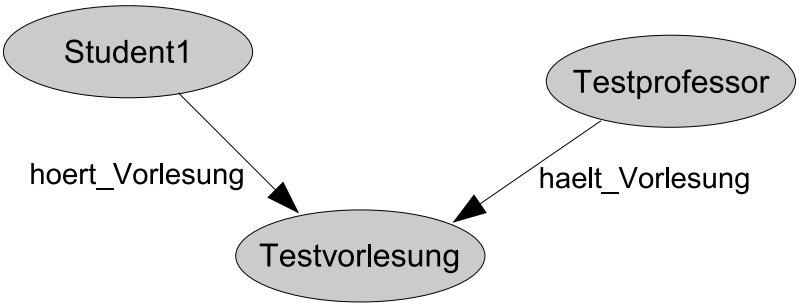


Abbildung 3.1: Beispiel für ein RDF-Modell

Abbildung 3.1 zeigt ein Beispiel für einen aus RDF-Daten erstellten Graphen. Hier sind die Ressourcen Student1, Testvorlesung und Testprofessor zu sehen. Die zugehörigen Statements sind [*Student1, hoert_Vorlesung, Testvorlesung*] und [*Testprofessor, haelt_Vorlesung, Testvorlesung*].
Ein ausführlicheres Beispiel findet sich in Abschnitt 3.7.

Zur Angabe von RDF-Daten reicht es aus, alle Statements anzugeben. Dies entspricht der Menge aller Kanten des Graphen mit ihren Eckpunkten. Auch wenn diese Darstellung nicht so anschaulich ist wie ein Graph, so ist sie doch einfacher zu schreiben und zu verarbeiten. In dieser Diplomarbeit werden daher RDF-Daten nur als Statements angegeben.
In einem Graphen wird eine Menge von Statements zusammengefasst, die ein Konzept beschreiben. Da die Darstellung jedoch in den meisten Fällen nicht durch einen Graphen geschieht, ist für die Beschreibung dieser zusammenhängenden Statements ein anderer Begriff nötig. Im englisch-sprachigen Raum, in dem RDF entwickelt wurde, wird dies als „Model“ bezeichnet. Dieser Name wird auch für die entsprechenden Klassen in dem für diese Arbeit verwendeten Jena-Framework benutzt. Für die Beschreibung bietet sich daher der deutsche Begriff „Modell“an.

3.4 RDF/XML

Um RDF-Dokumente abzuspeichern, ist ein Baum eher unüblich. Die Statements als Tripel abzuspeichern, ist für einen Leser relativ unübersichtlich. Hierfür ist die Darstellung durch eine XML-Syntax gebräuchlich, RDF/XML. Um einen RDF-Graphen in ein XML-Dokument umzuwandeln, wird der Graph von einem Knoten aus durchlaufen. Die Beschreibung des Knotens bildet ein XML-Element. In diesem wird das Prädikat als Element angeordnet und darin der Objektknoten. Ist der Objektknoten über weitere Kanten mit anderen Knoten verbunden, werden diese wiederum innerhalb des Objektknotens angeordnet.

Da RDF als Graph dargestellt wird, dieser jedoch kein Baum sein muss,

kann der Graph nicht immer direkt in RDF/XML-Form übersetzt werden. Es kann zum Beispiel ein Statement die Beziehung [A, Beziehung1, B] beschreiben und ein anderes die Beziehung [B, Beziehung2, A]. Werden dieses Statements als Graph dargestellt, so gibt es eine Kante von A nach B und eine von B nach A. In die andere Richtung ist diese Übersetzung jedoch kein Problem.
Ein Beispiel zu RDF/XML findet sich in Abschnitt 3.7.

In XML-Dokumenten ist es üblich Namespaces zu verwenden. Ein Namespace ermöglicht es, in einem Dokument verschiedene Elemente mit gleichem Namen zu verwenden. Sie verkürzen außerdem die Schreibweise und erleichtern die Lesbarkeit. Hierbei wird der lokale Name eines Elementes durch ein Präfix ergänzt, um Eindeutigkeit zu gewährleisten. Zu Beginn des Dokumentes wird deklariert, für welche URI das Präfix stehen soll.
Namespaces treten auch bei der Verwendung von RDF/XML auf. Nach der Festlegung der URIs werden im Dokument dann nur noch die kurzen lokalen Namen verwendet.
Anstelle von:

`http://www.w3.org/1999/02/22-rdf-syntax-ns#type`

Kann nach einmaliger Festlegung von:

`xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"`

das Präfix verwendet werden. Damit lässt sich die gleiche URI schreiben als

`rdf:type.`

3.5 RDFSchema

Während RDF die eigentlichen Daten zur Verfügung stellt, sorgt RDFSchema (kurz: RDFS) für deren Einordnung. Obwohl die Daten die im RDF-Format gespeichert werden bereits Metadaten sind, gibt es mit RDFS noch eine Ebene darüber. Für diese Daten werden wiederum Metadaten erstellt. Dabei werden Klassen definiert und Beziehungen zwischen diesen Klassen. Die eigentlichen RDF-Daten werden diesen Klassen untergeordnet. Sie werden hier als Individuuen bezeichnet. Mit Hilfe der Metadaten können einerseits die Daten konsistent gehalten werden, andererseits ist es möglich diese Metadaten zu interpretieren und so neue Daten aus den Beziehungen abzuleiten. Für das Ableiten ist dabei ein sogenannter „Reasoner“ verantwortlich.
Durch RDFS werden Abhängigkeiten wie die Zugehörigkeit eines Individuums zu einer Klasse beschrieben. Auch die Beziehung, dass eine Klasse eine Subklasse einer anderen ist kann hiermit ausgedrückt werden. Alle Individuuen der untergeordneten Klasse gehören dann auch der übergeordneten

Klasse an. Mit RDFS kann zum Beispiel auch der Wertebereich eines Properties kann angegeben werden.
RDFS ist kein anderes Format als RDF, sondern lediglich ein festgelegter Satz an URIs für die Beschreibung der Semantik. Sie werden innerhalb der RDF-Statements benutzt. Erst ein „Reasoner“ kann die Semantik dieser speziellen URIs erfassen und dadurch weitere Statements hinzufügen.
Mit Hilfe von RDFS werden Klassen, ihre Beziehungen untereinander und ihre Eigenschaften beschrieben. Die eigentlichen Daten werden als Instanzen dieser Klassen angelegt.

Typische RDFSchema-Elemente sind:

- rdfs:subClassOf – zur Beschreibung von Subklassen-Beziehungen
- rdfs:range – legt fest, dass der Wert eines Properties nur eine Instanz von den hier festgelegten Klassen sein kann.
- rdfs:domain – nur Instanzen dieser Klasse dürfen dieses Property aufweisen
- rdfs:subPropertyOf – das Äquivalent zu SubClassOf für Properties
- rdfs:Container – zur Zusammenfassung einer Menge von Klassen

Um die einfachsten Strukturen zu beschreiben, sind noch weitere Elemente nötig. Obwohl diese Elemente nicht zum RDFS-Namespace gehören, werden sie trotzdem RDFS zugerechnet.

- rdf:type – Gibt den Typ einer Ressource an. Dies kann die Identifikation als Klasse sein, aber auch die Zugehörigkeit einer Instanz zu einer Klasse
- rdf:value – Gibt den Wert eines Properties für eine Klasse an

Eine vollständige Auflistung aller RDF und RDFS-Elemente mit Erklärungen zur Benutzung findet sich in [RDFS1].

3.6 OWL

OWL steht für Web Ontology Language. OWL erweitert RDF und RDF-Schema um weitere Sprachkonstrukte. Hierdurch können zum Beispiel Kardinalitäten ausgedrückt werden oder Charakteristiken von Properties wie Symmetrie. Wie auch RDFS liefert OWL lediglich einen Satz an festgelegten URIs die in RDF verwendet werden. Dabei tritt OWL in drei verschiedenen Versionen auf: OWL Lite, OWL DL und OWL Full. Der Unterschied zwischen den Varianten ist damit zu erklären, dass OWL Full nicht mehr entscheidbar ist und somit für den eigentlichen Zweck, dem Ableiten von Daten

durch Maschinen, nicht immer geeignet ist. Es wurde nötig den Sprachumfang einzuschränken. Bei OWL DL steht das DL für Description Logic. Es wird hiermit eine entscheidbare Untermenge von OWL zur Verfügung gestellt. Dabei wird nicht die Anzahl der Elemente eingeschränkt, sondern die Art, in der sie benutzt werden dürfen. OWL DL verlangt eine strikte Trennung von Klassen, Properties und Individuen. OWL Lite schränkt dagegen tatsächlich die Menge an verwendbaren Konstrukten ein. Die Trennung der einzelnen Ressourcen bleibt wie bei OWL DL erhalten.
Die Zeit, die für die Verarbeitung der Daten benötigt wird, hängt entscheidend von der OWL-Variante ab. Während ein OWL-Lite-Reasoner relativ schnell arbeitet, kann die Verarbeitung bei OWL Full leicht exponentiell zur Datenmenge anwachsen. In dieser Arbeit wurde OWL DL verwendet, da dies eine gute Balance zwischen der Menge an möglichen OWL-Elementen und der Verarbeitungszeit darstellt. Alle in dieser Arbeit über OWL getroffenen Aussagen beziehen sich daher auf diese OWL-Version.
OWL bietet nicht nur einfache Möglichkeiten zur Definition von Klassen und Instanzen. Es können auch komplizierte Sachverhalte, wie die Tatsache, dass zwei Klassen disjunkt sind, in OWL ausgedrückt werden. Bei den in dieser Arbeit benutzten Ontologien treten die folgenden OWL-Elemente häufig auf.

- owl:Class – zum Anlegen einer Klasse
- owl:Restriction – zur Definition spezieller Eigenschaften für die Klasse
- owl:onProperty – zur Definition von Eigenschaften für eine gesamte Klasse
- owl:allValuesFrom – zum Festlegen einer Wertemenge für eine Klasseneigenschaft
- owl:complementOf – eine Klasse bildet das Komplement einer anderen
- owl:intersectionOf – der Durchschnitt von zwei Klassen
- owl:oneOf – ein Element aus der folgenden Menge
- owl:equivalentClass – zwei Klassen sind äquivalent und erhalten die gleichen Instanzen und Properties
- owl:disjointWith – eine Instanz der einen Klasse kann keine Instanz der anderen sein

Eine vollständige Auflistung aller in OWL möglichen Elemente und deren Verwendung findet sich in [OWL3].

In RDF/OWL wird zwischen der Ontologie und den eigentlichen Daten, den sogenannten Individuen, unterschieden. Man spricht dabei auch von der

T-Box (für *terminology*) und der A-Box (für *assertional knowledge*). Das Modell setzt sich aus A-Box und T-Box zusammen, jedoch können diese in unterschiedlichen Dateien gespeichert werden.

Dass die Abkürzung für die Web Ontologie Language OWL und nicht WOL heißt, wird in [wiki] durch eine Anleihe beim Buch „Pu der Bär“ (OT: „Winnie-the-Pooh“) erklärt. Hierin gibt es eine Eule (im Original: Owl) die als einziges Tier ihren Namen schreiben kann - allerdings mit einem Buchstabendreher, so dass sie immer Wol schreibt. Für die Abkürzung der Web Ontologie Language wurde der gleiche Buchstabendreher in die andere Richtung gemacht.

3.7 Beispiel

Um die Möglichkeiten von RDF, RDFSchema und OWL zu verdeutlichen, werden diese an einem etwas größeren Modell vorgestellt. Auf dieses Modell wird auch später bei Beispielen zu den Update-Funktionen (Abschnitt 7.2) und den Triggern (Abschnitt 8.2) eingegangen.

Abbildung 3.2 zeigt das hier betrachtete Modell als Graph. In diesem Modell existieren die vier Klassen Person, Student, Professor und Vorlesung. Die Klassen Student und Professor sind Subklassen von der Klasse Person. Außerdem sind in diesem Modell zwei Individuen zu sehen, Testvorlesung von der Klasse Vorlesung und Testprofessor von der Klasse Professor. Zwischen den Individuen wurde das Statement [*Testprofessor, haelt_Vorlesung, Testvorlesung*] visualisiert.

Dieses Modell liegt ebenfalls als RDF/XML-Datei vor. Diese befindet sich unter dem Namen testontologie.owl auf der dieser Arbeit beiliegenden CD. In RDF/XML sieht das Modell wie folgt aus:

Listing 3.1: Die Ontologie in RDF/XML

```
<rdf:RDF
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns
    #"
xmlns:owl="http://www.w3.org/2002/07/owl#"
>

    <owl:Class rdf:ID="Person"/>

    <owl:Class rdf:ID="Student">
        <rdfs:subClassOf rdf:resource="#Person"/>
    </owl:Class>
```

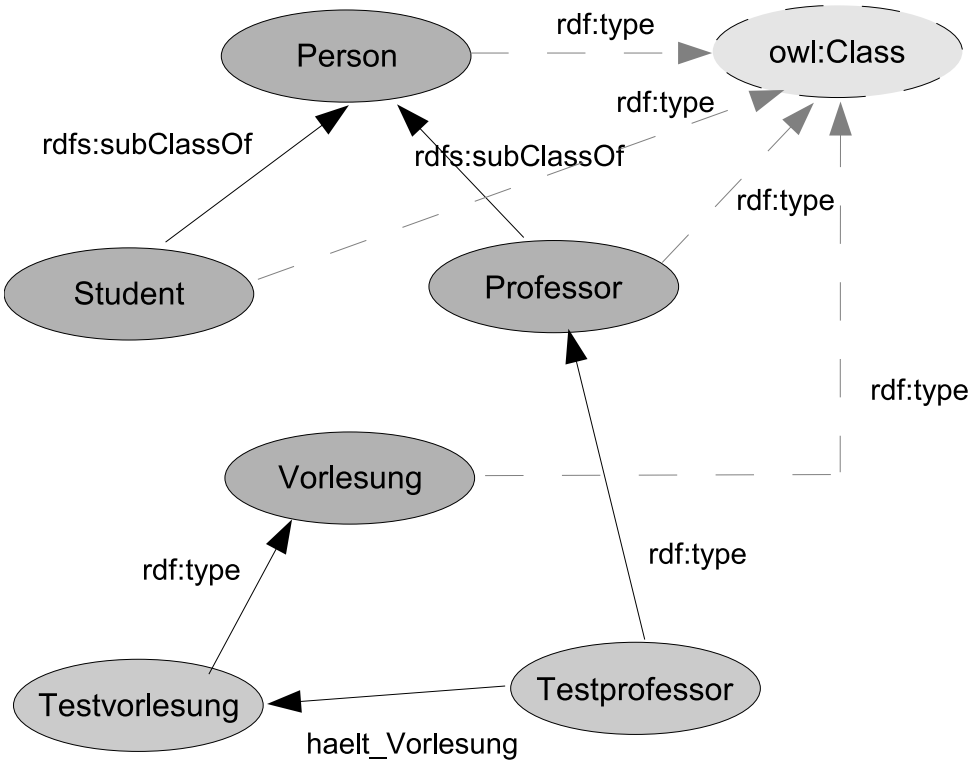


Abbildung 3.2: RDF-Modell mit Klassen und Individuen

```
    <owl:Class rdf:ID="Professor">
        <rdfs:subClassOf rdf:resource="#Person"/>
    </owl:Class>

    <owl:Class rdf:ID="Vorlesung"/>

    <owl:Thing rdf:ID="Testvorlesung">
        <rdf:type rdf:resource="#Vorlesung"/>
    </owl:Thing>

    <owl:Thing rdf:ID="Testprofessor">
        <rdf:type rdf:resource="#Professor"/>
        <liest_Vorlesung rdf:resource="#Testvorlesung
            "/>
    </owl:Thing>

</rdf:RDF>
```

Von einem Reasoner werden bei der Auswertung des Modells weitere Statements hinzugefügt, die über die Semantik geschlussfolgert werden können. Ist zum Beispiel eine Klasse Subklasse einer anderen, so kann man schließen, dass die Individuen der Unterklasse auch Individuen der Oberklasse sind. Das OWL-Element „owl:equivalentClass“ wertet der Reasoner aus, indem alle Klassen und Properties, die für eine Klasse definiert sind, auch für die andere eingefügt werden. Abbildung 3.3 zeigt das Modell nach Einsatz eines Reasoners, hier wird die Subklassen-Beziehung durch ein weiteres Tripel [Testprofessor, rdf:type, Person] berücksichtigt.

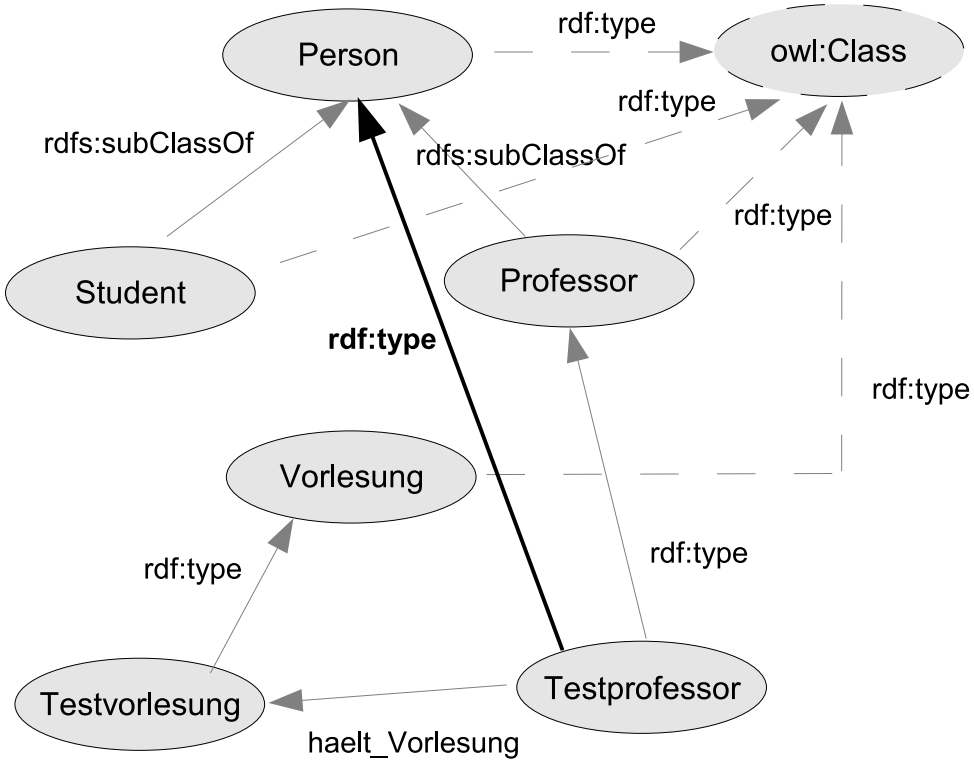


Abbildung 3.3: Das Modell nach dem Reasoning

3.8 RDQL

RDQL ist eine Anfragesprache für RDF. Dabei werden für jede auftretende Variable alle möglichen Belegungen bestimmt. In der Syntax ist RDQL an SQL angelehnt.
Jede RDQL-Anfrage beginnt mit dem Schlüsselwort „SELECT“. Danach folgen die Variablen, die ausgewertet werden sollen. Nur die Variablen, die

hier stehen, werden auch ausgegeben, es können jedoch weitere Variablen verwendet werden. Eine Variable besteht immer aus einem Wort, welches durch ein Fragezeichen eingeleitet wird.
In der RDQL-Anfrage kann dann eine optionale „FROM“-Klausel folgen, mit der die Quelle für die RDF-Daten festgelegt wird. Dann folgt die „WHERE“-Klausel. Damit wird ein Muster aus Statements festgelegt, das erfüllt werden muss. An einigen Stellen treten dabei Variablen auf, wobei nicht alle in der „SELECT“-Klausel auftauchen müssen. Danach sind noch Konditionen möglich, die diese Variablen betreffen. Der Konditionsteil wird dabei mit „AND“ eingeleitet. Schließlich kann noch eine „USING“-Klausel auftreten, womit Namespaces für die URIs der Statements festgelegt werden. Gibt es keine solche Klausel, müssen alle URIs ausgeschrieben werden.

Ein einfaches Beispiel kann wie folgt aussehen:

Listing 3.2: Eine einfache RDQL-Anfrage

```
SELECT ?instanz, ?klasse
WHERE (?instanz, <http://www.w3.org/1999/02/22-rdf-
    syntax-ns#type>, ?klasse)
```

Durch diese Anfrage werden alle Instanzen mit den dazugehörigen Klassen ausgewertet. Im vorangegangenen Beispiel wäre eine der möglichen Variablenbelegungen:
?instanz = Testvorlesung, ?klasse=Vorlesung.

RDQL-Anfragen können innerhalb der festgelegten Syntax auch deutlich komplizierter werden. In [RDQL] findet sich eine Auflistung der gesamten möglichen Syntax. Dabei wird unter anderem diese Anfrage als Beispiel angegeben:

Listing 3.3: Eine komplexere RDQL-Anfrage

```
SELECT ?resource
FROM   <http://example.org/someWebPage>
WHERE  (?resource, info:age, ?age)
AND    ?age >= 24
USING  info FOR <http://example.org/peopleInfo#>
```

Kapitel 4

Jena

In diesem Kapitel wird ein Überblick über das Jena-Framework gegeben, welches für die Erstellung des Programms genutzt wurde. Dies kann die Einarbeitung in das Framework erleichtern, falls es für ähnliche Zwecke eingesetzt wird.

4.1 Einführung

Jena ist der Name eines auf Java basierenden Frameworks für das Semantic Web. Dieses Open-Source-Projekt ist im Internet unter *http://jena.sourceforge.net* zu finden.

Jena kann nicht eigenständig verwendet werden, es liefert nur einige Werkzeuge. Diese können für verschiedene Anwendungen des Semantic Webs benutzt werden, jedoch muss der Anwender selbst festlegen, welche Werkzeuge er wie einsetzen will. Hierzu werden die entsprechenden Bibliotheken von Jena in ein eigenes Java-Programm eingebunden. Ein wesentliches Hilfsmittel dabei ist, wie bei allen Java-Bibliotheken, das zugehörige Javadoc-Dokument.

Für diese Arbeit wurde die bei Beginn aktuelle Version 2.2 benutzt, inzwischen existiert aber bereits die Version 2.3. Jena wird ständig weiterentwickelt, und liefert daher mit der Zeit weitere Möglichkeiten. Auch die Paketstruktur kann sich ändern oder erweitert werden. Daher sind die Aussagen, die hier über Jena getroffen werden, nur im Zusammenhang mit der Version 2.2 zu sehen.

Das gesamte Framework besteht aus 25 Paketen, von denen für diese Arbeit jedoch nur 7 eingesetzt wurden. Jena enthält viele verschiedene Möglichkeiten für die Verarbeitung von RDF-Daten, allerdings ist es trotz der Dokumentation nicht sehr übersichtlich. Die Einarbeitung und die Suche nach den benötigten Funktionen ist deshalb sehr zeitaufwändig.

4.2 Model

Das wichtigste Element zur Verwaltung von RDF-Daten in Jena ist das `Model`. Verschiedene Javaklassen erfüllen die Funktion des RDF-Modells, jedoch sind alle von einem gemeinsamen Interface abgeleitet. Innerhalb des `Model`s werden die Daten als Statements dargestellt. Intern wird das `Model` als Graph behandelt, jedoch kann auf diesen normalerweise nicht zugegriffen werden. Um die Daten des `Model`s zu verändern, stellt Jena die Methoden *add(Statement)* und *delete(Statement)* zur Verfügung. In einem `OntModel`, einem Modell, das weite Teile einer Ontologie unterstützt, gibt es zusätzliche Methoden um Veränderungen vorzunehmen. Dazu gehört zum Beispiel die Methode *createClass(String)*. Diese Methoden führen jedoch intern wieder auf die add- und delete-Methoden zurück.
Dabei können Objekte des Typs `Model` auch kombiniert werden. `ModelRDB` bietet beispielsweise die Möglichkeit, die Daten automatisch in einer Datenbank zu speichern, während es mit einem `OntModel` möglich ist, einen Reasoner für die Verarbeitung der Semantik einzusetzen. Es kann jedoch auch ein `OntModel` erzeugt werden, das seine Basisdaten von einem Modell bezieht, welches mit einer Datenbank verbunden ist. Dabei kann mit dem kombinierten Modell wie mit jedem anderen `OntModel` gearbeitet werden. Die Übergabe der Daten an das BasisModell und damit die Speicherung in der Datenbank werden dabei von Jena bei jeder Änderung ausgeführt.

4.3 Datenbank Unterstützung

Jena bietet die Möglichkeit, ein Modell in einer Datenbank zu speichern und es bei Bedarf wieder zu laden. Wird ein Modell in einer Datenbank gespeichert, so müssen die Daten nicht bei jedem Neustart des Programms neu eingelesen werden. Sofern die Daten aus einer RDF/XML-Datei stammen, sind sie noch relativ leicht zu rekonstruieren. Werden sie jedoch von verschiedenen Quellen zusammengetragen oder einzeln als Statements eingegeben, so ist die Speicherung sehr wichtig. Dabei kann das mit dem folgendem Code erzeugte Objekt der Klasse `ModelRDB` wie jedes andere `Model` in Jena verwendet werden. Die Speicherung geschieht automatisch bei jedem Aufruf von *add()* oder *delete()*. Für die Speicherung eines Modells muss lediglich eine leere Datenbank bereitgestellt werden, das Datenbankschema wird automatisch angelegt. Jena unterstützt zur Zeit für das Erstellen persistenter Modelle die Datenbanksysteme MySQL, Oracle und PostgreSQL. Für die vorliegende Arbeit wurde PostgreSQL verwendet. Vor der ersten Operation mit der Datenbank muss ein entsprechender JDBC-Treiber geladen werden, der die Kommunikation zwischen der Datenbank und Java ermöglicht.

Ein Datenbank-Modell wird dabei wie folgt angelegt:

Listing 4.1: Anlegen eines neuen Datenbankmodells

```
// Create database connection
IDBConnection conn = new DBConnection(DB_URL, DB_USER,
    DB_PASSWD, DB);
// Create a model in the database
ModelMaker maker = ModelFactory.createModelRDBMaker(
    conn);
ModelRDB model = (ModelRDB) maker.createModel("
    winemodel");
```

Hier wird speziell das Modell mit dem Namen „winemodel“ angelegt. Ein solches Modell enthält zunächst noch keine Daten. Die Daten können beispielsweise aus einer Datei im RDF/XML-Format eingelesen werden:

Listing 4.2: Anlegen eines neuen Datenbankmodells

```
//read Model from file
try
{
    model.read(new FileInputStream(filename),"");
}
catch(FileNotFoundException fe)
{
    System.out.println(fe);
}
```

Hierbei ist „filename“ der Name der einzulesenden Datei.

Es ist natürlich auch erforderlich, ein in der Datenbank gespeichertes Modell wieder auszulesen. Dies kann mit dem folgenden Code-Fragment erfolgen.

Listing 4.3: Auslesen eines bestehenden Modells aus der Datenbank

```
// Create database connection
connection = new DBConnection(DB_URL, DB_USER,
    DB_PASSWD, DB);
//reading out model with existing connection
ModelRDB modeldb = ModelRDB.open(connection,
    Configuration.MODEL_NAME);
```

4.4 Listener

Jena stellt verschiedene Listener zur Verfügung, die auf Änderungen im Modell reagieren. Damit kann beispielsweise jedes Mal, wenn ein Statement zum

Modell hinzugefügt wird, eine Nachricht ausgegeben wird, die alle Benutzer über diese Änderungen informiert.
Alle Listener in Jena sind vom Interface `ModelChangedListener` abgeleitet. Um einen Listener einzusetzen, ist es erforderlich, eine eigene Listenerklasse zu schreiben, die entweder von dem Interface oder einer der anderen Listenerklassen abgeleitet ist.

Die Benutzung ist dabei relativ unkompliziert, da der Aufruf automatisch erfolgt. Wird der Listener für ein Modell registriert, so wird bei jeder Add-Operation eine entsprechende *addedStatement*-Methode aufgerufen, entsprechendes gilt für Delete-Operationen. Beim `ModelChangedListener` werden hier als Argumente für die *addedStatement*-Methoden die Argumente der *add*-Methode übernommen. Möglich ist hier ein einzelnes Statement oder ein Array aus Statements, die Statements können auch als Liste oder als StmtIterator (ein Iterator über Statements) übergeben werden. Es ist auch möglich, ein Modell zu übergeben. Dann werden alle im Modell enthaltenen Statements übernommen.
Der `NullListener` ist als Basisklasse gedacht, jede Methode des `ModelChangedListener`s ist hier durch einen leeren Body implementiert. Hierbei müssen bei einer abgeleiteten Klasse lediglich die Methoden überschrieben werden, die wirklich benutzt werden sollen.
Der `ObjectListener` enthält zusätzlich die Methoden *added(Object)* und *removed(Object)*. Hier muss, falls erforderlich, bei der Benutzung noch unterschieden werden, welchen Typ das Argument hat.
Beim `ChangedListener` gibt es lediglich eine zusätzliche Methode *hasChanged()*, mit der festgestellt werden kann, ob seit dem letzten Aufruf der Methode eine Änderung stattgefunden hat.

Der in dieser Arbeit verwendete `StatementListener` zerlegt die Argumente der Add- und Delete-Methoden in einzelne Statements. So müssen hier lediglich die Methoden *addedStatement(Statement)* und *deletedStatement(Statement)* überschrieben werden.

4.5 Reasoner

Ein Reasoner verarbeitet Daten über die Semantik, wie sie von RDFS und OWL bereitgestellt werden. Hier werden Beziehungen zwischen Klassen wie Subklassen ausgewertet und dadurch neu entstehende Tripel eingefügt. Ohne Reasoner stehen zwar Metadaten zur Verfügung, jedoch können diese nicht weiter verarbeitet werden. RDFS und OWL stellen zusätzliche Informationen bereit, durch die Schlüsse gezogen werden können. Dieses Schlussfolgern übernimmt dabei der Reasoner.
Zum Beispiel kann in einer Beschreibung der Geographie Europas beschrieben werden, dass Deutschland in Europa liegt. Weiterhin kann man damit festlegen, dass Niedersachsen in Deutschland liegt. Ist das Property, das die

Beziehung beschreibt, mit dem OWL-Element owl:TransitiveProperty versehen, so kann der Reasoner dadurch auch schließen, dass Niedersachsen in Europa liegt, ohne dass diese Beziehung explizit als Statement definiert werden muss. Erweitert man dieses Beispiel auf alle Länder Europas und vielleicht noch sämtliche Regierungsbezirke, so werden vom Reasoner sehr viele derartige Statements erzeugt, die für eine Verarbeitung sehr hilfreich sind.
Es gibt in Jena zwei verschiedene Möglichkeiten Reasoner einzusetzen. Einerseits enthält Jena eingebaute Reasoner, andererseits bietet es auch ein Interface an, um externe Reasoner einzusetzen.

4.5.1 In Jena eingebaute Reasoner

Jena enthält verschiedene eingebaute Reasoner, die relativ einfach erzeugt und benutzt werden können. Hierzu gehören der „Transitiv Reasoner“, der „RDFS Rule Reasoner“, der „DAML Micro Reasoner“ und der „Generic Rule Reasoner“. Außerdem gibt es noch „OWL“, „OWL Mini“ und „OWL Micro Reasoner“, die auf einer unvollständigen Teilmenge von OWL-Full arbeiten.

Hier wird der OWL-Reasoner verwendet, jedoch erfolgt das Einbinden in ein Modell bei den anderen Reasonern ebenso.

Listing 4.4: Einbinden des in Jena eingebauten OWL-Reasoners

```
Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
InfModel infmodel = ModelFactory.createInfModel(
    reasoner, dbmodel);
```

Die zugrundeliegenden Daten sind hierbei in *dbmodel* enthalten. Dieses sorgt außerdem für die Speicherung in der Datenbank.
Mit einem solchen Modell kann genauso gearbeitet werden wie mit einem Modell ohne Reasoner. Einige Methoden können erst mit einem Modell mit Reasoner korrekt ausgeführt werden. Der Aufruf des Reasoners geschieht dabei durch Jena automatisch.

Leider ist der Funktionsumfang des internen Reasoners gegenüber einem externen Reasoner sehr eingeschränkt. Es werden viele OWL-Elemente nicht korrekt interpretiert und es kommt hierbei zu Fehlern und Abstürzen. Im Gegensatz zum externen Reasoner benötigt der interne Reasoner leider oft viel Zeit für die Auswertung eines Modells. So dauert etwa das Bestimmen der Klassen um ein Vielfaches länger.
Andererseits löst der interne Reasoner im Gegensatz zum externen Reasoner auch für abgeleitete Tripel den Listener aus. Sofern kein großer Funktionsumfang benötigt wird und auf kleinen Modellen gearbeitet wird, kann der Einsatz eines internen Reasoners durchaus sinnvoll sein.

4.5.2 Externe Reasoner

In Jena können auch externe Reasoner, die auf einer Beschreibungslogik (Description Logic) basieren, eingesetzt werden. Dafür stellt das Framework ein sogenanntes DIG-Interface bereit. DIG steht dabei für Description Logic Interest Group. Auch wenn der Ablauf des Reasonings komplex sein mag, so ist der Umgang mit dem Reasoner in Jena relativ einfach. Es muss zunächst ein Modell erstellt werden, welches die ursprünglichen Daten enthält. Nun wird mit diesem Modell und einem DIGReasoner-Objekt ein neues Modell (speziell ein `OntModel`) erstellt. Dieses Modell kann wie jedes andere Modell in Jena benutzt werden. Der Aufruf des Reasoners erfolgt dabei automatisch bei jedem Einfügen und Auslesen der Daten, ohne dass der Reasoner explizit aufgerufen werden muss.

Formal geschieht das Aufsetzen eines mit einem Reasoner verbundenen Modells wie folgt:

Listing 4.5: Einbinden eines externen Reasoners

```
//Setting up the external Reasoner
Model cModel = ModelFactory.createDefaultModel();
Resource conf = cModel.createResource();
conf.addProperty(ReasonerVocabulary.EXT_REASONER_URL,
    cModel.createResource(Configuration.REASONER_URL));
DIGReasonerFactory drf = (DIGReasonerFactory)
    ReasonerRegistry.theRegistry().getFactory(
    DIGReasonerFactory.URI);
DIGReasoner reasoner = (DIGReasoner) drf.create(conf);
OntModelSpec spec = new OntModelSpec(OntModelSpec.
    OWL_DL_MEM);
spec.setReasoner(reasoner);
OntModel ontmodel = ModelFactory.createOntologyModel(
    spec, dbmodel);
```

Hier wird zunächst ein `DefaultModel` erzeugt, mit dessen Hilfe dann ein RDF-Tripel gebildet wird. Dieses enthält als Subjekt eine anonyme Resource und als Prädikat `ReasonerVocabulary`.EXT_REASONER_URL, eine Konstante, die angibt, dass es sich beim Objekt um die URL des externen Reasoners handelt. Das Objekt ist die aus der Konfigurationsdatei eingelesene URL des Reasoners.

Als nächstes wird eine `DIGReasonerFactory` erstellt und damit ein `DIGReasoner`-Objekt mit der zuvor festgelegten URL. Mit Hilfe eines Objektes der Klasse `OntModelSpec` kann schließlich ein `OntModel` erstellt werden. Die Daten stammen dabei aus dem Modell *dbmodel*, welches diese auch in der Datenbank speichert.

Ein Problem bei dem externen Reasoner ist, dass die vom Reasoner erzeugten Daten nicht im Modell gespeichert (materialisiert) sind. Sie werden

erst bestimmt, wenn sie durch einen Methodenaufruf angefordert werden. Dies erschwert den Einsatz von Triggern. Das Einfügen und Löschen von RDF-Tripeln löst normalerweise den Listener aus, jedoch gilt das nicht für die von einem externen Reasoner erzeugten Tripel. (Siehe dazu auch Abschnitt 8.5.2)

Der externe Reasoner ist im Vergleich zum internen wesentlich schneller, so dass sich der Einsatz hinsichtlich der Performance durchaus lohnen kann.

Es gibt auf dem Markt verschiedene externe Reasoner. Für diese Arbeit wurde zunächst Racer in der Version 1.7 verwendet. Das DIG-Interface von Jena wurde laut der Dokumentation hauptsächlich mit diesem Reasoner getestet. Leider wurde Racer nach dieser Version jedoch kommerzialisiert, so dass keine neueren Versionen zur Verfügung standen.
Ein weiterer frei benutzbarer externer Reasoner ist Pellet. Dieser wurde in der Version 1.3 eingesetzt. Bei der Interpretation von Modellen gibt es Unterschiede zwischen den einzelnen Reasonern. In einem Test konnte ein Modell von Pellet verifiziert werden, während Racer hier Fehlermeldungen gab. Racer ging dabei davon aus, dass bereits alle Statements bekannt waren, was nicht das gewünschte Verhalten ist. Man spricht dabei von „closed world“ (für das Ableiten ist alles bekannt) und „open world“ (es können noch weitere Statements hinzugefügt werden, wodurch das Modell konsistent wird).

4.6 Die Paketstruktur von Jena

Abschließend soll hier noch ein Überblick über die in der Arbeit verwendeten Pakete von Jena gegeben werden.

com.hp.hpl.jena.rdf.model Dieses Paket ist der Ausgangspunkt aller Objekte des Typs `Model`. Jede Repräsentation eines RDF-Modells ist nicht nur von dem Interface `Model` dieses Paketes abgeleitet, auch die Erzeugung aller Modelle erfolgt über die Klasse `ModelFactory`. Eine weitere wichtige Klasse dieses Paketes ist `ResourceFactory`. Hiermit ist es möglich, Instanzen der Klassen `Statement` und deren Bestandteile `Resource` und `Property` zu bilden.

com.hp.hpl.jena.db Hier finden sich alle Klassen und Methoden, die für ein persistentes, also in einer Datenbank gespeichertes Modell benötigt werden. In der Arbeit wurde vor allem die Klasse `ModelRDB` verwendet, welche ein mit einer Datenbank verbundenes Modell bildet. Die Änderungen müssen nicht explizit an die Datenbank weitergeleitet werden, sondern werden automatisch bei Änderung des Modells übernommen.

com.hp.hpl.jena.ontology Der für diese Arbeit wichtigste Bestandteil dieses Paketes ist das `OntModel`. Mit Hilfe dieses Model-Objekts können

auch Ontologiedaten verarbeitet werden. Mit der Methode *listClasses()* lassen sich zum Beispiel alle Klassen des Models ausgeben. Außerdem befinden sich die Repräsentationen von Klassen (`OntClass`) und Individuen (`Individual`) in diesem Paket.

com.hp.hpl.jena.rdql Mit diesem Paket ist es möglich, RDQL-Anfragen an das vorliegende Modell zu stellen und die Antwort auszuwerten. Die wichtigsten Klassen hierbei sind `Query` und die nach Ausführung der Anfrage zurückgegebenen `QueryResults`.

com.hp.hpl.jena.rdf.listeners Alle vom `ModelChangedListener` abgeleiteten Listener stammen aus diesem Paket. Wird ein Listener mittels `Model`.*register*(`Listener`) für das Modell registriert, löst jeder *add()*- oder *delete()*-Aufruf die Methode *addedStatement()* bzw. *deletedStatement()* aus. Leider gilt das nicht für die vom Reasoner erzeugten oder entfernten Statements (siehe auch Abschnitt 8.5.2). Interessanterweise findet man den `ModelChangedListener` selbst im Paket *com.hp.hpl.jena.rdf.model*.

com.hp.hpl.jena.reasoner In diesem Paket sind die Elemente enthalten die benötigt werden, um einen der in Jena enthaltenen Reasoner einzusetzen. Es ist auch für den Betrieb eines externen Reasoners notwendig.

com.hp.hpl.jena.reasoner.dig Dieses Paket enthält Klassen, die zusätzlich zu den im Paket *com.hp.hpl.jena.reasoner* enthaltenen Klassen notwendig sind, um einen externen Reasoner in Jena einzusetzen.

Kapitel 5

Programmierung

Ziel dieser Arbeit ist die Entwicklung eines RDF-Servers mit Trigger-Funktionalität. Die Trigger reagieren auf Änderungen in den Daten wie dem Einfügen neuer RDF-Tripel. Der Server muss dazu zunächst RDF-Daten bereitstellen und die Möglichkeit bieten, diese zu verändern. Schließlich wird noch ein Client benötigt, von dem aus die Änderungen der Daten bewirkt werden können. Client und Server sind beides in Java implementierte Programme, die über SOAP miteinander kommunizieren.
Die Elemente von Jena und von Java wurden auf englisch entwickelt. REWERSE ist zudem ein internationales Projekt, dessen Veröffentlichungen ebenfalls in englischer Sprache verfasst sind. Daher habe ich mich entschlossen, mein Programm dem anzupassen. Die Namen der Klassen und Methoden sowie alle Kommentare sind in englisch. Dies führt leider zu sprachlichen Schwierigkeiten bei der Beschreibung des Programms. Ich hoffe die folgenden Kapitel trotzdem verständlich formuliert zu haben.

5.1 Technische Voraussetzungen

Um die in dieser Arbeit entwickelten Programme zu betreiben, ist Java in der derzeitig aktuellen Version 5.0 nötig. Für den Einsatz des Servers braucht man zusätzlich einen Servlet Container wie Jakarta Tomcat.
Außerdem wird ein externer DL-Reasoner benötigt. Bei der Entwicklung der Arbeit wurde hierfür Racer in der Version 1.7 benutzt. Eine PostgreSQL-Datenbank der Version 7 oder 8 wird für die Abspeicherung der Trigger und des RDF-Models eingesetzt. Es ist auch möglich, eine höhere Version von Postgres zu verwenden, jedoch ist nicht sichergestellt, dass der in der .war-Datei enthaltene JDBC-Treiber dies unterstützt. Gegebenenfalls muss noch ein passender Treiber zur Verfügung gestellt werden. Es ist wichtig, dass das Modell, welches benutzt werden soll, bereits in der Datenbank gespeichert ist. Der Name des Modells, sowie der Datenbank-Benutzer und das entsprechende Passwort wird in der Konfigurationsdatei abgelegt und

beim Start des Servers ausgelesen. Die Konfigurationsdatei muss dabei im „webapps“-Verzeichnis des Tomcat Servlet Containers abgespeichert werden. Wird anstelle von Tomcat ein anderer Servlet Container verwendet, so sollte diese Datei im Home-Verzeichnis des Benutzers liegen. Zur Speicherung eines Modells in der Datenbank kann das ebenfalls in dieser Arbeit erstellte Programm „ModelLoader“ benutzt werden, welches sich wie alle anderen Programmteile auf der beiliegenden CD befindet.

5.2 Servlets

Sowohl der Server als auch das zusätzlich entwickelte Programm TriggerServer (siehe Abschnitt 9.2) sind als Servlet implementiert. Diese spezielle Art von Java-Programmen möchte ich an dieser Stelle einführen, da sie für das Verständnis des Programmaufbaus von Bedeutung sind.

Ein Servlet ist ein Java-Programm, welches speziell an die Aufgaben eines Webservers angepasst ist. Der Begriff Servlet gilt als eine Verschmelzung der Wörter Server und Applet. Mit einem Servlet wird eine Art Server-seitiges Applet zur Verfügung gestellt. Wie auch bei einem normalen Applet, ist dies ein Programm, welches nur innerhalb eines Containers laufen kann. Bei der Entwicklung des Programms wurde hierfür Jakarta Tomcat von der Apache Software Foundation verwendet. Dies ist vermutlich der gebräuchlichste Servlet Container. Ein weiterer freier und häufig benutzter Servlet Container ist Jetty (http://jetty.mortbay.org/jetty/). Es gibt jedoch noch etliche andere freie und kommerzielle Container.
Allgemein erfüllt ein Servlet die Funktion, Daten von einem Client entgegenzunehmen, diese auszuwerten und Daten zurückzusenden. Am häufigsten wird dabei die Klasse `HttpServlet` verwendet. Hiermit lassen sich besonders leicht dynamische Webseiten entwickeln. Die Kommunikation erfolgt über das http-Protokoll, welches bei Webseiten verwendet wird. Es gibt zwei unterschiedliche Formen, Daten von einer Webseite aus an den Server zu senden und wieder zu empfangen: „get“ und „post“.

Für diese Arbeit wurde die Klasse `JAXMServlet` benutzt. Diese stammt aus der **JAXM**-Bibliothek, welche ebenfalls von SUN zur Verfügung gestellt wird, jedoch leider nicht Teil der J2EE ist. Das `JAXMServlet` ist vom `HttpServlet` abgeleitet. Es bietet die Möglichkeit, bequem SOAP-Nachrichten zu empfangen und eine Antwort zu senden. Dabei setzt dieses Servlet auf der „post“-Schnittstelle auf und nutzt sie für die Erzeugung von SOAP-Nachrichten. Die entsprechende Methode *doPost()* darf deshalb keinesfalls überschrieben werden.

Bei einem `JAXMServlet`, das gleichzeitig den `ReqRespListener` aus der **JAXM**-Bibliothek implementiert, gibt es zwei wichtige Methoden. Die Methode *init()* wird bei der Initialisierung des Servlets durch den Servlet-

Container aufgerufen. Dies geschieht immer dann, wenn der Container das erste Mal nach dem Start eine Anfrage über „post“ oder „get“ für das Servlet erhält.
Die Methode *onMessage()* wird immer dann aufgerufen, wenn eine SOAP-Nachricht an das Servlet geschickt wurde. Diese Methode erhält als Parameter die neue Nachricht. Der Rückgabewert ist ebenfalls eine SOAP-Nachricht, die als Antwort an den Sender zurückgeschickt wird.
Um Servlets zu betreiben und auch anderen zur Verfügung zu stellen, werden diese in eine Datei mit der Endung .war exportiert. Diese Datei enthält neben den .class-Dateien mit dem kompilierten Programm auch Bibliotheken, die das Servlet verwenden kann. In der Datei web.xml innerhalb der war-Datei kann zusätzlich festgelegt werden, auf welche URL das Servlet reagiert. Es ist auch möglich, verschiedene Servlets in einer war-Datei zu speichern. Die Datei web.xml legt dann fest, welches Servlet auf welche URL wie reagiert.

5.3 Programmstruktur

Das Programm, das im Rahmen dieser Arbeit entwickelt wurde, gliedert sich in zwei größere Teile, den Client und den Server. Hinzu kommt noch ein kleinerer Teil, der TriggerServer, der nur ein Dummy für die im Einsatz vorhandenen anderen Server darstellt. Alle Programmteile wurden in Java geschrieben, so dass diese nicht an ein bestimmtes Betriebssystem gebunden sind. Der Server ist als Servlet implementiert, während der Client eine „normale“ Java-Applikation ist.
Der Server sorgt in diesem Programm für die gesamte Logik. Dies umfasst die Bereitstellung von RDF-Daten, die Ermöglichung von Updates sowie die Bereitstellung von Triggern.
Der Client bietet eine graphische Oberfläche zur Betrachtung der vorhandenen Daten, dem Auslösen von Updates und dem Anlegen und Löschen von Triggern. Die Kommunikation zwischen Client und Server erfolgt über SOAP. Die eingehenden Nachrichten müssen dabei einem festen Format genügen. Es ist durchaus möglich, ein anderes Programm für die oben genannten Aufgaben einzusetzen als den hier entworfenen Client. Da SOAP als systemunabhängiges Protokoll entworfen wurde, ist es auch denkbar, dieses Programm in einer anderen Programmiersprache zu entwickeln.

5.4 Probleme

Die Entwicklung des vorliegenden Programms hat einige Monate gedauert. Dies lag zum einen daran, dass ich mich in Techniken wie Servlets und das Jena-Framework erst einarbeiten musste. Zum anderen bin ich auf etliche programmiertechnische Hürden gestoßen. Die Probleme, welche mehrere

Programmteile betreffen, werden an dieser Stelle angesprochen, andere werden an der Stelle ihres Auftretens im Programm aufgeführt.

5.4.1 Servlets und SOAP

Für die Kommunikation zwischen dem Server mit allen anderen Komponenten wird SOAP benutzt. Um dies in Java zu realisieren, können fertige Bibliotheken benutzt werden. SUN bietet mit der SAAJ-Bibliothek (SOAP with Attachment API for Java) eine bequeme Möglichkeit SOAP-Nachrichten zu senden, doch gibt es dabei keine Möglichkeit, SOAP-Nachrichten zu empfangen und die Antwort an die Quelle zurückzusenden. Für diesen Zweck stellt SUN eine weitere Bibliothek zu Verfügung: JAXM (Java API for XML Messaging). Die beiden wichtigsten Elemente dieser Bibliothek sind das `JAXM-Servlet` und der `ReqRespListener`. ReqResp steht dabei für Request-Response und bedeutet, dass das Servlet eine Anfrage erhält und auf diese antwortet. Dieser Listener bietet mit der Methode *onMessage()* die Möglichkeit, das Versenden und Empfangen der Nachricht vollständig der Java-Bibliothek zu überlassen. Dabei werden die Nachrichten in `SOAPMessage`-Objekte der SAAJ-Bibliothek verwandelt. Allerdings ist JAXM nicht Teil der Java Enterprise Edition (J2EE), die unter anderem auch SAAJ und Servlets umfasst, in der sich auch leider kein Verweis auf JAXM findet. JAXM muss daher getrennt von der SUN-Webseite heruntergeladen werden.

Allerdings gibt es Probleme mit der Kombination von JAXM und SAAJ. Während das normale Empfangen und Zurücksenden von Nachrichten funktioniert, tritt bei der Versendung von Attachments immer eine Fehlermeldung im Servlet auf. Auch bei der Verwendung des Beispielcodes aus dem SUN-Tutorial verursachte dieser die gleiche Fehlermeldung. Wird dieser Code statt in einem Servlet in einer Java-Applikation verwendet, gibt es keine Fehler beim Versenden. Jedoch erzeugt dann das empfangende JAXM-Servlet eine Fehlermeldung. Es ist daher nicht möglich, bei Verwendung von SAAJ und JAXM Attachments einzusetzen.

5.4.2 SOAP

Ein weiteres Problem stellen die unterschiedlichen Implementationen von SOAP in Java dar. Die in dieser Arbeit verwendete SAAJ-Bibliothek von SUN reagiert leider teilweise anders als die von Apache unter dem Namen Axis bereitgestellte Bibliothek. Werden die verschiedenen Bibliotheken auf den verschiedenen Seiten der Kommunikation eingesetzt, führt das teilweise zu Inkompatibilitäten. Eine mittels SAAJ erstellte SOAP-Nachricht erzeugte unter Axis beim Empfangen eine Fehlermeldung (“Missing Content-Type“), während sie von einer SAAJ-Umgebung fehlerfrei verarbeitet wurde. Sowohl im Server als auch im Client wird SAAJ verwendet, so dass es bei der Be-

nutzung beider Komponenten nicht zu derartigen Problemen kommen sollte. Außerdem wurde darauf geachtet, dass der optionale Header beim Senden der Nachrichten richtig gesetzt ist. Dann kann die Nachricht fehlerfrei empfangen werden, auch wenn die Gegenseite mit Axis arbeitet.

5.4.3 Konfigurationsdatei

Der Server benötigt für den Betrieb Zugriff auf eine Datenbank und dazu die entsprechenden URL, das Passwort und den Usernamen. Hinzu kommen der Name des gespeicherten Modells, die URL des externen Reasoners und die URL des Eventbrokers. Diese Konfiguration kann nicht fest in den Programmcode integriert werden. Diese Daten müssen vom Benutzer angepasst und bei Bedarf auch wieder geändert werden können.
In diesem Programm werden alle Daten, die nicht fest vorgegeben werden können, in einer Konfigurationsdatei abgelegt, die dann vom Servlet bei der Initialisierung eingelesen wird. Falls die Umgebungsvariable 'CATALINA_HOME' gesetzt ist, wird die Konfigurationsdatei dort im webapps-Verzeichnis gesucht. 'CATALINA_HOME' bezeichnet das Tomcat-Verzeichnis und wird von Tomcat automatisch gesetzt. Die Datei sollte daher im selben Verzeichnis wie die war-Datei zu finden sein. Die Java-Methode, mit der die Umgebungsvariable gesucht wird, arbeitet dabei plattformunabhängig, so dass die einzige Voraussetzung ist, dass als Servlet-Container Tomcat benutzt wird. Ist diese Umgebungsvariable nicht gesetzt, so wird die Konfigurationsdatei im Home-Verzeichnis des Users gesucht. In diesem Fall wird das Verzeichnis, aus dem die Datei geladen werden soll, im Log vermerkt.
Ich habe diesen Weg gewählt, da es nicht möglich ist, eine Datei innerhalb einer war-Archives, in das das Servlet später exportiert wird, zu editieren. Die Konfigurationsdatei muss deshalb außerhalb des Archivs positioniert werden. Für den Zugriff auf diese Datei ist es nicht möglich, einen absoluten Pfad zu verwenden, da dies die Portierbarkeit zerstören würde. Das Laden der Konfigurationsdatei ist auch mit einem relativen Pfad von einem Servlet aus nicht möglich. Beim Schreiben einer Datei von einem Programm aus, wird das gleiche Verzeichnis verwendet wie beim lesenden Zugriff. Wird eine Datei ohne feste Pfadangabe von einem Servlet aus angelegt, so ist das Verzeichnis in das diese Datei geschrieben wird leider bei verschiedenden Konfigurationen und Rechnern nicht das gleiche. In einem Test landete die Datei einem im Binäry-Verzeichnis des Tomcat, einmal im Home-Verzeichnis des Benutzers und beim Starten von der Programmierumgebung Eclipse aus sogar im Binäry-Verzeichnis von Eclipse. Auch über die interen Konfigurationsmöglichkeiten eines Servlets lassen sich keine externen Dateien laden. Die Methode *getResource()* funktioniert zwar für Dateien innerhalb des Archivs, außerhalb davon erzeugt sie jedoch jedes mal einen Zugriffsfehler. Es bleibt daher nur die im Programm genutzte Variante, Umgebungsvariablen zu verwenden.

5.4.4 RDQL-Anfragen bei inkonsistenten Modells

Im vorliegenden Programm können von den Triggern RDQL-Anfragen gestellt werden, die entweder als Bedingung oder zur Bindung von weiteren Variablen dienen. Diese Anfragen beziehen sich dabei auf das Modell bevor dieses durch die auslösende Änderungsoperation oder andere Trigger verändert wurde. Es ist hier leider nicht möglich, das Modell zu verwenden, bei dem die bisherigen Änderungen bereits vollzogen sind. Die Trigger dienen auch zur Konsistenzerhaltung (siehe Abschnitt 8.4), daher kann ein Modell bei der Ausführung eines Triggers inkonsistent sein. Die Trigger selbst sorgen erst dafür, dass das Modell wieder konsistent wird. Stellt man die Anfrage an das aktuelle Modell, wenn dieses inkonsistent ist, erhält man beim Zugriff auf das Ergebnis, das die Jena-Methode *Query.exec* erzeugt, eine Fehlermeldung. Diese Fehlermeldung stammt vom Reasoner, der Probleme mit dem inkonsistenen Modell hat. Leider tritt der Fehler innerhalb der Jena-Methoden auf, so dass er sich vom Anwender nicht abfangen lässt. Dies führt automatisch zum Absturz des Servlets. Eine Anfrage sollte daher nur an ein Modell gestellt werden, bei dem die Konsistenz garantiert ist.

5.5 Benutzerfreundlichkeit

Bei der Entwicklung von Client und Server wurde auf die Benutzerfreundlichkeit geachtet. Einige Funktionen werden nur in der Zusammenarbeit von Client und Server deutlich. Die Funktionen sind eigentlich auf Serverseite implementiert, jedoch werden sie nur bei der Eingabe auf Clientseite deutlich, weshalb sie an dieser Stelle vorstellt werden.

5.5.1 Erkennung von Schlüsselwörtern

Beim Testen erzeugte ich häufig versehentlich Fehler beim Parsen, weil eines der Schlüsselwörter klein geschrieben war, während beim Parsen nach groß geschriebenen Schlüsselwörtern gesucht wurde. Solche Fehler können auch im normalen Betrieb häufig auftreten und der Fehler ist nicht immer schnell zu sehen. Daher wurde die Eingabe von Update-Operationen und Triggern „case insensitive“ gestaltet. Das bedeutet, dass Schlüsselwörter erkannt werden, unabhängig davon, ob sie groß oder klein geschrieben sind.
Hierzu wird jeder Eingabestring kopiert und mit der Methode *toUpperCase()* alle Zeichen in Großbuchstaben umgewandelt. Der Vergleich mit bestimmten Schlüsselwörtern wird immer auf der Variante aus Großbuchstaben durchgeführt, während der Rest mit diesen Informationen aus dem originalen String extrahiert wird.

5.5.2 Namespaces

Die Metadaten beruhen teilweise auf Namespaces wie „rdf“, „rdfs“ und „owl“. Dies betrifft insbesondere die Ontologiedaten, jedoch werden auch Individuen einer Klasse über das Property „rdf:type“ definiert. Die URL für diese Namespaces kann sich jedoch je nach Alter und Quelle der Daten unterscheiden.
Beim Start des Servers werden die Namespaces des Modells ermittelt und in der Klasse `Configuration` gespeichert. Wird nun zum Beispiel ein Trigger angelegt, der auf das Erzeugen eines Individuums reagieren soll, so wird automatisch der richtige Namespace für das Property gewählt. Außerdem werden die Namespaces bei jeder Eingabe automatisch ersetzt. Dadurch ist die Eingabe kürzer und weniger fehleranfällig.

5.5.3 Darstellung der Klassen

In der ersten Version des Programms wurden lediglich die vorhandenen Statements des Modells ausgelesen und diese als Liste im Client dargestellt. Obwohl so alle Informationen vorhanden waren, war es sehr schwer diese nachzuvollziehen. In der vorliegenden Version des Programms werden alle Klassen des Modells ermittelt, dann werden die Properties und die Instanzen, die dieser Klasse angehören, hinzugefügt. Schließlich werden noch die Properties der einzelnen Instanzen bestimmt. Aus diesen Elementen wird mit den zusätzlichen Erklärungsteilen „Class:“, „Restriction:“, „Instance:“, „Property:“ und „Value:“ ein String zusammengestellt und dem Client gesendet. Dieser formt aus dem erhaltenen String einen Baum, in dem einer Klasse die zugehörigen Properties und Instanzen untergeordnet sind. Die Instanzen enthalten wiederum die Properties und Werte. Für die Sortierung ist dabei einzig der vorangestellte String und die Reihenfolge wichtig. Mit Hilfe dieses Baumes ist es sehr leicht möglich zu sehen, welche Instanzen zu welcher Klasse oder Restriction gehören und welche Werte sie für die einzelnen Properties besitzen.

Kapitel 6

Client

Der Client dient dazu, die RDF-Daten, die auf dem Server vorhanden sind, anzuzeigen, Anfragen an diese zu stellen und Änderungen vorzunehmen. Außerdem werden die vorhandenen Trigger angezeigt, es können neue Trigger angelegt werden und von hier aus gelöscht werden. Hierbei stand die übersichtliche Präsentation und die einfache Bedienbarkeit an oberster Stelle. Die gesamte Logik für die Verarbeitung von RDF-Daten bleibt dabei dem Server überlassen.

6.1 Graphische Oberfläche

Dieses Programm stellt eine graphische Oberfläche zur Verfügung, mit deren Hilfe der Benutzer die vom Server aufbereiteten Daten betrachten kann.
Beim Start des Clients erscheint ein Fenster für die Eingabe der URL des Servers. An diese URL werden alle SOAP-Nachrichten gesendet. Als Eingabehilfe ist hier bereits die URL für den Server angegeben, falls dieser auf demselben Rechner (localhost) läuft. Diese URL kann natürlich bei Bedarf angepasst werden. Abbildung 6.1 zeigt das Fenster für die Eingabe der Server-URL.

Abbildung 6.1: Zu Beginn des Programms muss die URL des SOAP-Servers festgelegt werden.

Ist die URL eingegeben, so sendet der Client eine SOAP-Nachricht an den Server um die vorhandenen Daten zu erhalten. Mit diesen Daten wird dann das Hauptfenster aufgebaut.

Abbildung 6.2 zeigt einen Screenshot bei der Arbeit mit dem Beispiel aus Abschnitt 8.2. Die Anzeige ist dabei horizontal geteilt. In der unteren Hälfte werden alle zur Zeit im Server aktiven Trigger dargestellt. Der obere Teil dient zur Darstellung der RDF-Daten. Obwohl die Daten im Server nur als Tripel verwaltet werden, habe ich mich entschlossen, sie im semantischen Sinn als Klassen mit Individuen und Properties zu präsentieren. So ist es für den Benutzer leichter zu sehen, welche Daten wirklich vorhanden sind. Die Filterung nach Klassen, Properties und Instanzen übernimmt dabei der Server, während der Client die Daten mit Hilfe der vorangestellten Informationen in Baumform bringt.
Auf der rechten Seiten des Hauptfensters finden sich verschiedene Bedienelemente. Mit der Schaltfläche ‚RDQL-Request' kann dabei eine RDQL-Anfrage an den Server gestellt werden. Über die Schaltflächen ‚Insert Statement', ‚Delete Statement', ‚Modify Statement', ‚Rename ', ‚Delete Resource', und ‚Assert Not' werden Update-Operationen durchgeführt. Außerdem kann mit ‚Add Trigger' ein Trigger auf dem Server hinzugefügt und mit ‚Delete Trigger' wieder gelöscht werden. Der letzte Button trägt die Beschriftung ‚Refresh View' und dient dazu, die Klassen und Instanzenbeschreibung sowie die auf dem Server zur Zeit vorhandenen Trigger neu zu laden. Da dies eine relativ zeitaufwendige Operation ist, habe ich mich entschlossen, diese nicht nach jeder Änderung automatisch auszuführen. So kann der Benutzer dieses Update selbst bei Bedarf auslösen.
Das Größenverhältnis der Anzeigen für die RDF-Daten und die Trigger lässt sich je nach Bedarf anpassen. In der Situation, die auf dem Screenshot festgehalten wurde, sind nur einige der Baumelemente ausgeklappt. Durch die Baumform bleibt die Menge der Daten beherrschbar, da nicht alles auf einmal angezeigt wird. Interessiert sich der Benutzer für alle Instanzen einer Klasse, so kann dieses Element des Baumes mit der Maus geöffnet werden.
Die Syntax der RDQL-Anfragen und Update-Funktionen ist relativ komplex, so dass es sehr leicht zu Eingabefehlern kommt. Um dem Benutzer eine Hilfe zu geben, wurde jeder Dialog mit einer Beispieleingabe versehen. Um die Benutzung weiter zu erleichtern, wurden zusätzlich zu den entsprechenden Buttons verschiedene Tastenkürzel eingeführt. Nach der Eingabe der URL des SOAP-Servers reicht es, auf Enter zu drücken um diese zu speichern. Bei den Anfragen und Update-Möglichkeiten hätte ein derartiges Verhalten dazu geführt, dass keine mehrzeilige Eingaben möglich wären, was hier jedoch durchaus sinnvoll ist. Hier können stattdessen mit der Kombination 'ALT'+'s' die Eingaben zum Server gesendet werden.
Während eine SOAPMessage zum Server gesendet wird und die erhaltene Antwort verarbeitet wird, kann einige Zeit vergehen. Um zu symbolisieren,

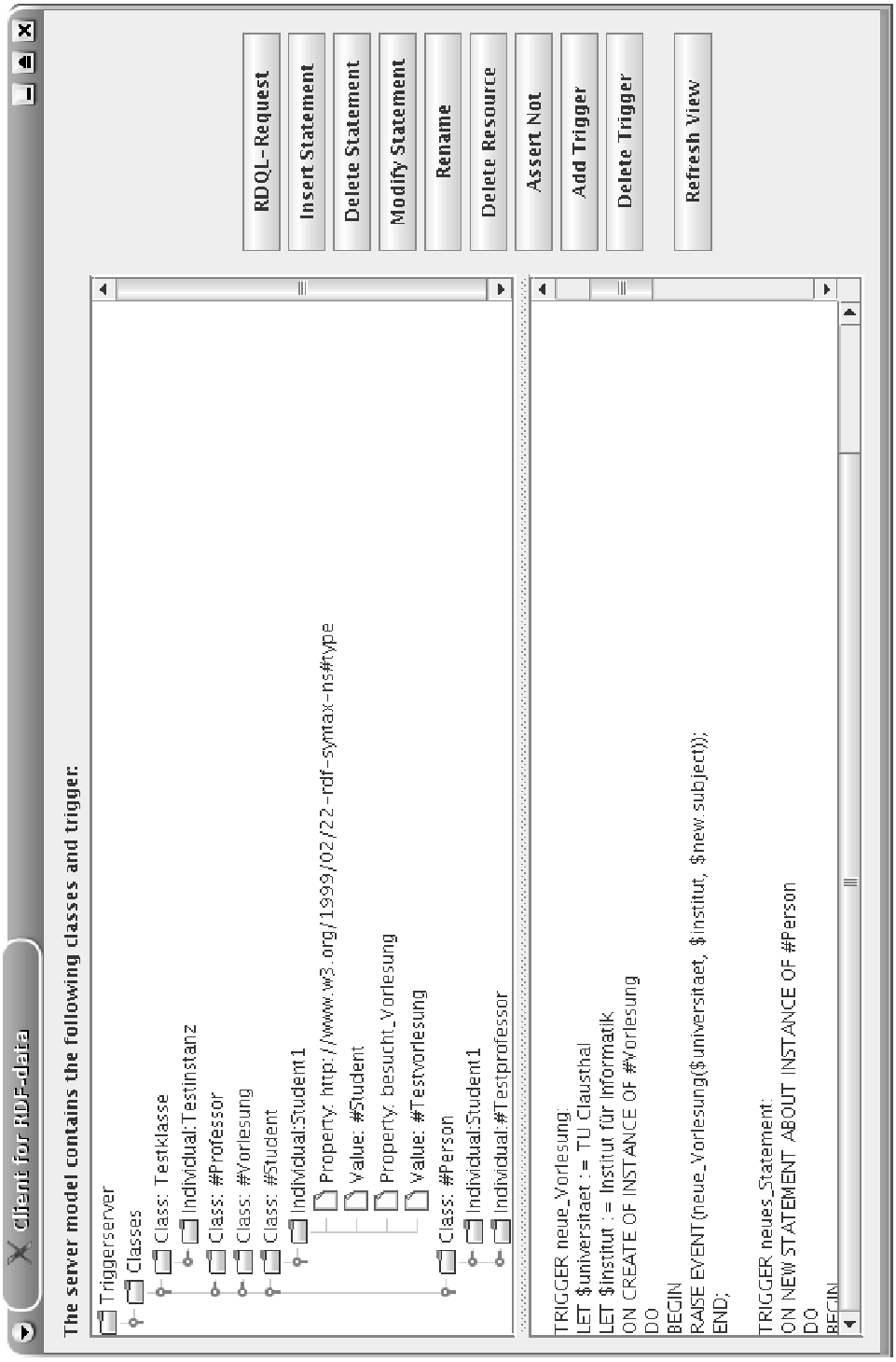


Abbildung 6.2: Die graphische Oberfläche des Clients

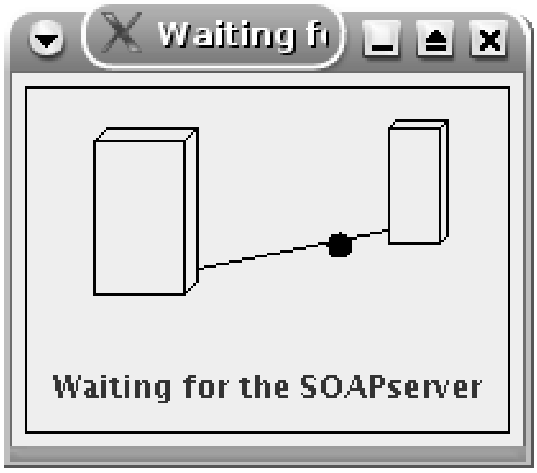


Abbildung 6.3: Der Wartedialog bei der Übertragung und Auswertung von SOAP-Nachrichten

dass der Client die Anfrage noch bearbeitet, wurde ein Wartedialog eingebaut. Hierbei wird in einem eigenen Thread eine kleine Animation gezeigt. Abbildung 6.3 zeigt den Wartedialog, der bei jeder Übertragung erscheint, bis die erhaltene SOAP-Nachricht ausgewertet wurde.

6.2 Funktion

Der Client ermöglicht es, Anfragen in der RDF-Anfragesprache RDQL zu stellen. Dabei gilt die in Abschnitt 3.8 definierte Syntax. Als Antwort wird ein Fenster geöffnet, welches die möglichen Variablenbelegungen enthält.

Der Client bietet außerdem die Möglichkeit, die Daten mit Hilfe der Update-Funktionen zu ändern. Mit ‚Insert Statement' kann dabei ein beliebiges Statement dem Modell hinzugefügt werden. ‚Delete Statement' entfernt dagegen das eingegebene Statement aus dem Modell, sofern es vorhanden ist, sonst erscheint eine Fehlermeldung. Mit ‚Modify Statement' ist es möglich, einen Teil (Subjekt, Prädikat oder Objekt) eines Statements zu ändern. Die Schaltfläche ‚Rename' ermöglicht es schließlich jedes Vorkommen einer spezifizierten Ressource durch eine andere Ressource zu ersetzen. Mit „Rename Property of Class" lässt sich die Ersetzung auf Properties einer bestimmten Klasse einschränken. Die Metadaten der Properties werden dann nicht ersetzt, da hier die Einschränkung auf diese Klasse nicht sichergestellt werden kann. Über die Schaltfläche ‚Delete Resource' können alle Statements über eine Ressource gelöscht werden, unabhängig davon, ob die Ressource hier als Subjekt, Prädikat oder Objekt auftritt. ‚Assert Not' bietet eine ähnliche Funktionalität wie ‚Delete'. Allerdings wird hier nach der Aktion festgestellt,

ob das entsprechende Statement tatsächlich gelöscht ist und nicht vom Reasoner auf Grund von anderen Daten wieder rekonstruiert wurde. Sollte dies der Fall sein, so werden die Änderungen wieder rückgängig gemacht und ein entsprechende Meldung zurückgegeben. Die Syntax der Update-Operationen muss dabei den in Abschnitt 7.1 festgelegten Regeln folgen. Ansonsten erhält der Benutzer eine Fehlermeldung. Bei der Eingabe ist es möglich, die Namespaces „rdf“, „rdfs“ und „owl“ zu nutzen.
Über die Buttons ‚Add Trigger‘ und ‚Delete Trigger‘ können Trigger erstellt und wieder gelöscht werden. Beim Erstellen eines Triggers muss die in Abschnitt 8.1 festgelegte Syntax eingehalten werden. Auch hier kann auf die Namespaces „rdf“, „rdfs“ und „owl“ zurückgegriffen werden.
Die Eingaben werden vom Client ausgelesen und je nach aufgerufener Funktion mit einem bestimmten XML-Tag versehen und als SOAP-Nachricht an den Server geschickt. Danach wird die Antwort, die der Server ebenfalls als SOAP-Nachricht schickt, verarbeitet und dem Benutzer präsentiert. Dies kann eine Bestätigung einer Änderung sein, aber auch das Ergebnis einer Anfrage. Der Client dient dabei nur als Vermittler zwischen dem Benutzer und dem Server.

6.3 Programmstruktur

Abbildung 6.4 zeigt das Klassendiagramm des Clients. In diesem Klassendiagramm sind nur Attribute und Methoden zu sehen, die als „public“ oder „protected“ definiert sind. Die Angabe aller Attribute und Methoden würde das Diagramm zu unübersichtlich machen. Der Client ist eine Java-Applikation und besteht nur aus einem Paket: *client*.
Um eine saubere Trennung zwischen Funktionalität und graphischer Oberfläche zu erreichen, habe ich mich entschieden, diese in unterschiedliche Klassen zu verlegen. In der Klasse `MainFrameGUI` beispielsweise wird nur die Oberfläche mit einer leeren Baumansicht und den Buttons definiert. Die Funktion der Buttons und das Setzen der Elemente des Baumes erfolgt dagegen erst in der von `MainFrameGUI` abgeleiteten Klasse `MainFrame`.
Das Senden und Empfangen von SOAP-Nachrichten, sowie das Parsen der Antwort, um daraus die Elemente zu bestimmen, die angezeigt werden, erfolgt komplett in der Klasse `SOAPMessages`. Hierzu stehen Methoden zur Verfügung, die Nachrichten mit passenden XML-Tags erzeugen, diese an den Server senden und die Antwort wieder aus der Nachricht extrahieren.
Technisch gesehen läuft das Programm im Client wie folgt ab: Zunächst wird von der *main*-Methode der Klasse `ClientGraphical` eine Instanz der Klasse `SetServer` aufgerufen. Nachdem hier die URL des SOAP-Servers eingegeben wurde, wird diese in der Klasse `SOAPMessages` gesetzt. Danach wird eine Instanz der Klasse `MainFrame` erzeugt und das aktuelle Fenster geschlossen. Bei der Erzeugung des `MainFrame` wird zunächst die Methode *getAllState-*

ments() der Klasse `SOAPMessages` aufgerufen. Diese sendet eine SOAP-Nachricht mit dem XML-Tag 'statements' zum Server und erhält eine Liste der verschiedenen Klassen (beginnend mit ‚Class:') mit den zugehörigen Klassenproperties (beginnend mit ‚Property:') und deren Werten, soweit sie vorhanden sind (beginnend mit ‚Value:'). Hinzu kommen noch die der Klasse angehörenden Individuuen mit den jeweiligen Properties und Werten. Nun werden diese Daten in eine Baumstruktur eingefügt. Ist ein solcher Baum erzeugt worden, wird das Hauptfenster mit dieser Ansicht der Klassen angezeigt.
Nun wird mit einer SOAP-Nachricht mit dem Inhalt „ListTrigger" eine Liste der zur Zeit gespeicherten Trigger angefordert. Die erhaltenen Trigger werden im unteren Teil der Anzeige eingetragen.
Während eine SOAP-Nachricht gesendet und die Antwort empfangen und entpackt wird, kann einige Zeit vergehen. Diese wird von der Antwortzeit des Servers, der Größe der Nachricht, der Qualität der Verbindung und der Geschwindigkeit des Rechners beeinflusst. Vor dem Senden wird daher ein Fenster der Klasse `WaitingBox` erzeugt um die Wartezeit zu überbrücken. Dieses läuft als eigener Thread und zeigt dem Benutzer eine kleine Animation. Ist die SOAP-Nachricht ausgewertet, wird die `Waitingbox` wieder deaktiviert.
Wenn die Anzeige aufgebaut ist, reagiert das Programm auf die Buttons auf der rechten Seite. Wird der Button RDQL-Anfrage benutzt, wird ein Dialog der Klasse `Request` erzeugt. Nachdem der Benutzer die Anfrage eingegeben und den 'send'-Button betätigt hat, wird diese aus dem Textfeld extrahiert und mit der Methode *sendRequest()* an den Server gesendet. Dieser erhält die SOAP-Nachricht mit dem XML-Tag 'Request' und verarbeitet die Anfrage. Die Antwort wird mit Hilfe der Klasse `Result` in einem neuen Fenster präsentiert, während das Request-Fenster geschlossen wird.
Die Funktionsweise bei den anderen Buttons ist sehr ähnlich. Es wird jeweils eine Instanz einer von `DialogGUI` abgeleiteten Klasse angezeigt. Diese unterscheiden sich nur im Text, der über dem Fenster angezeigt wird, sowie in der Methode der Klasse `SOAPMessages`, die aufgerufen wird. Namentlich sind dies die Klassen `Insert`, `Update`, `Delete`, `Rename`, `DeleteResource`, `AssertNot` sowie, für die Erstellung und das Löschen von Triggern, die Klassen `Trigger` und `Triggerdelete`. Der Ablauf ist bei allen gleich.

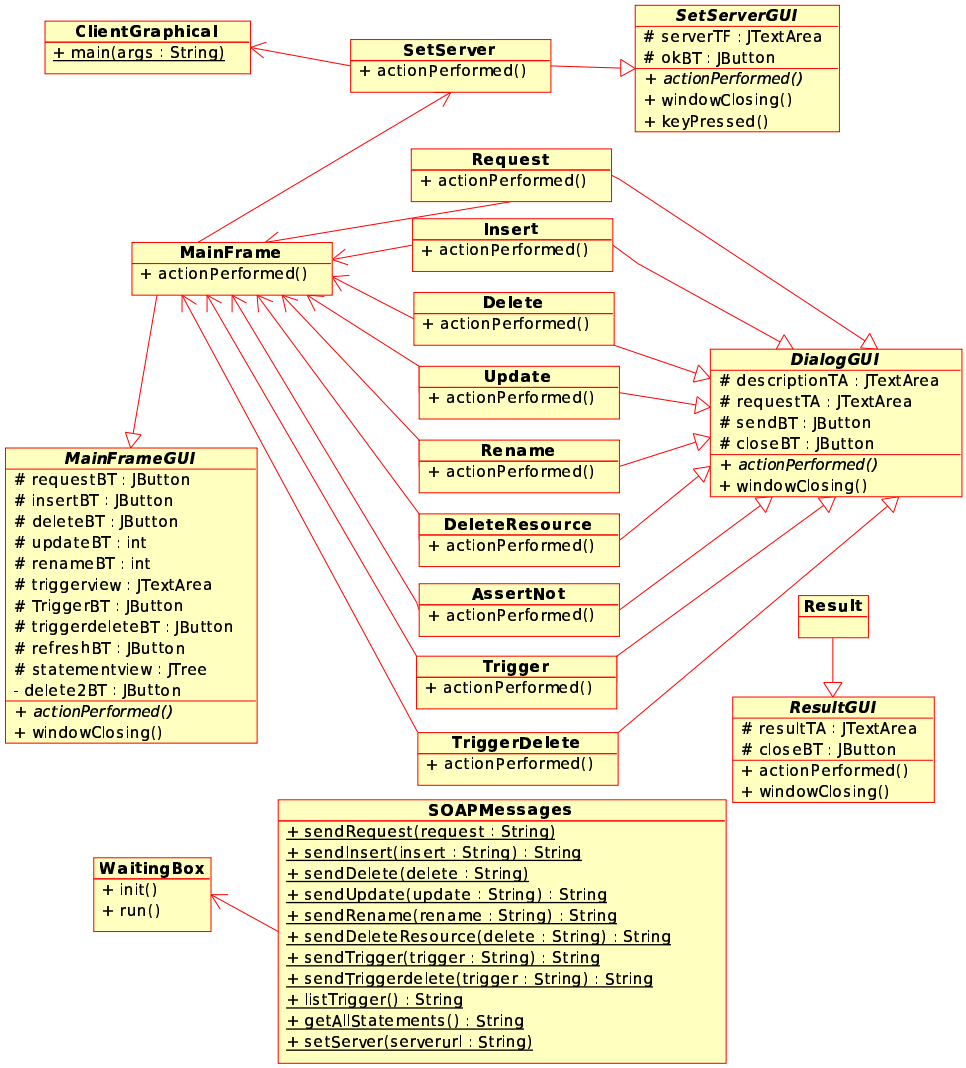


Abbildung 6.4: Klassendiagramm des Client

Kapitel 7

Server

Dieses Kapitel beschreibt den Aufbau und die Funktion des Servers. Obwohl die Trigger eigentlich Teil des Servers sind, werden sie in einem eigenständigen Kapitel betrachtet. Die Trigger bilden das Kernstück dieser Arbeit, jedoch sollen zunächst die anderen Möglichkeiten des Servers wie die Update-Funktionen beschrieben werden.

7.1 Update-Funktionen

Mit Hilfe der Update-Funktionen kann ein Benutzer Änderungen am Modell vornehmen. Hierzu wird eine SOAP-Nachricht eines bestimmten Formats an den Server gesendet. Der Inhalt wird nach dem XML-Tag unterschieden, welches direkt innerhalb des SOAP-Bodies steht. Danach wird der in diesem Tag enthaltene String an die ausführende Methode übergeben. Diese liefert entweder einen Fehler oder eine Bestätigung zurück, die dann als Antwort via SOAP zurück an den Client geschickt wird.

Dabei sind folgende Formate möglich:

<INSERT>
INSERT (*Subjekt* , *Prädikat* , *Objekt*)
</INSERT>

Hierdurch wird das RDF-Tripel (*Subjekt*, *Prädikat*, *Objekt*) zum Modell hinzugefügt.

<DELETE>
DELETE (*Subjekt* , *Prädikat* , *Objekt*)
</DELETE>

Hierdurch wird ein vorhandenes RDF-Tripel aus dem Modell entfernt. Ist das spezifizierte Tripel nicht vorhanden, gibt der Server eine Fehlermeldung zurück. Dabei kann auch kein Statement gelöscht werden, das nicht in der Datenbank gespeichert ist, sondern nur durch den Reasoner erzeugt wird.

Hier besteht jedoch die Möglichkeit, dass durch den Aufruf von Delete für dieses Statement einen direkten Trigger auslöst (siehe Abschnitt 8.3). Wird das Model jedoch nicht durch direkte Trigger verändert, so wird eine Meldung zurückgegeben, die besagt, dass das Statement nicht gelöscht werden kann.

<MODIFY>
MODIFY SUBJECT (*altes_Subjekt* , *Prädikat* , *Objekt* , *neues_Subjekt*)
oder
MODIFY PREDICATE (*Subjekt* , *altes_Prädikat* , *Objekt* , *neues_Prädikat*)
oder
MODIFY OBJECT (*Subjekt* , *Prädikat* , *altes_Objekt* , *neues_Objekt*)
</MODIFY>

Diese Funktionen verändern das Subjekt, das Prädikat oder das Objekt des spezifizierten Tripels, so dass es als neuen Wert das vierte Argument annimmt. Ist kein derartiges Tripel im Modell vorhanden, so gibt der Server eine Fehlermeldung zurück. Für durch den Reasoner erzeugte Statements gilt hier das gleiche wie beim Löschen. Diese Statements werden nicht verändert, hier können nur direkte Trigger reagieren. Wenn das Model nicht verändert wurde, wird eine entsprechende Meldung zurückgegeben.

<RENAME >
RENAME (*alte_Ressource* , *neue_Ressource*)
oder
RENAME PROPERTY OF CLASS (*altes_Property* , *Klasse* , *neues_Property*)
</RENAME >

Durch diese Funktion werden alle Ressourcen, die der URI des Argumentes „alte_Ressource“ entsprechen, durch die URI der neuen Ressource ersetzt. Ohne den Zusatz „PROPERTY OF CLASS“ geschieht diese Ersetzung im ganzen Modell, mit dem Zusatz kann dies auf Properties einer definierten Klasse eingeschränkt werden. Dabei wird davon ausgegangen, dass bei einer Einschränkung auf eine Klasse noch weitere Properties dieses Namens existieren. Die Metadaten eines solchen Properties können daher nicht geändert werden. Wenn ein Benutzer dies trotzdem wünscht, so muss er die Umbenennung für jedes zu ändernde Statement über die Update-Funktion „Modify Statement“ ausführen.

<DELETERESOURCE >
DELETE RESOURCE (*Ressource*)
</DELETERESOURCE >

Diese Funktion dient dazu, sämtliche Statements über die spezifizierte Ressource zu löschen. Dies geschieht unabhängig davon, ob die Ressource dabei

als Subjekt, Prädikat oder als Objekt auftritt.

<ASSERTNOT>
ASSERT NOT (*Subjekt*, *Prädikat*, *Objekt*)
</ASSERTNOT>

Wie auch mit der Funktion „Delete“ können auch mit „Assert Not“ Statements gelöscht werden. Jedoch wird bei dieser Funktion zusätzlich sichergestellt, dass das Statement danach nicht mehr im Modell enthalten ist und nicht durch den Reasoner aus anderen Daten rekonstruiert wird. Sollte dies der Fall sein, so werden alle Änderungen rückgängig gemacht und eine entsprechende Meldung zurückgegeben.

7.2 Beispiel

Die Arbeit mit den Update-Funktionen möchte ich an einem Beispiel verdeutlichen und dazu auf das Beispiel aus Abschnitt 3.7 zurückgreifen.
Die Ontologie testontologie.owl wird mit Hilfe des ModelLoaders geladen und unter dem Namen „Testmodell“ in der Datenbank gespeichert. In der Konfigurationsdatei wird nun der Variablen MODEL_NAME dieser Name zugewiesen. Je nach Konfiguration des Systems müssen an der Konfigurationsdatei noch weitere Variablen wie die URL der Datenbank angepasst werden.
In der Klassenansicht des Clients sind die vier Klassen #Person, #Student, #Professor und #Vorlesung zu sehen, außerdem die Individuen #Testprofessor und #Testvorlesung. Über den Button „Insert Statement“ ist es nun möglich, weitere Daten anzulegen. Dazu wird zunächst ein Individuum Student1 der Klasse #Student erschaffen:

```
INSERT (Student1, rdf:type, #Student).
```

Mit

```
INSERT (Person1, rdf:type, #Person)
```

wird dann das Individuum Person1 der Klasse #Person angelegt.

```
INSERT (Vorlesung2, rdf:type, #Vorlesung)
```

erzeugt das Individuum Vorlesung2 der Klasse #Vorlesung.

Um zu kennzeichnen, das beide Individuen die Vorlesung Testvorlesung bzw. Vorlesung2 besuchen, werden die folgenden Statements hinzugefügt:

```
INSERT (Student1, hoert_Vorlesung, #Testvorlesung)
INSERT (Person1, hoert_Vorlesung, Vorlesung2).
```

Über den Button „Rename“ ist es möglich, Ressourcen global oder nur innerhalb einer Klasse umzubenennen. Durch Eingabe von

```
RENAME PROPERTY OF CLASS(hoert_Vorlesung, #Student,
    besucht_Vorlesung)
```

werden in allen Statements, bei denen das Subjekt der Klasse #Student angehört und das Prädikat ‚hoert_Vorlesung' lautet, die Prädikate durch ‚besucht_Vorlesung' ersetzt. In diesem Beispiel betrifft dies das Individuum Student1, während in dem Statement über Person1 weiterhin das Property ‚hoert_Vorlesung' gültig bleibt.

Über den Button „Modify Statement" ist es möglich, ein Element eines Statements zu ändern. Um das Individuum Person1 statt der Klasse #Person der Klasse #Student zuzuordnen, wird dafür

```
MODIFY OBJECT (Person1, rdf:type, #Person, #Student)
```

eingegeben.

Über den Button „Delete Statement" können einzelne Statements wieder gelöscht werden.

```
DELETE (Person1, rdf:type, #Student)
```

Hiermit wird das Statement gelöscht, welches Person1 als Individuum der Klasse #Student auszeichnet, die anderen Statements über Person1 bleiben dabei jedoch erhalten. Den gleichen Effekt hätte hier

```
ASSERT NOT (Person1, rdf:type, #Student).
```

Mit dem Button „Delete Ressource" können schließlich ganze Ressourcen entfernt werden.

```
DELETE RESOURCE (Vorlesung2)
```

löscht nicht nur die Definition von Vorlesung2 als Individuum der Klasse #Vorlesung, sondern auch das Statement, welches angibt, dass Person1 diese Vorlesung hört.

7.3 Andere Nachrichten

Neben den schon beschriebenen Update-Funktionen reagiert der Server noch auf andere Nachrichten.

<REQUEST>
RDQL-Anfrage
</REQUEST>

Die hier enthaltene RDQL-Anfrage wird vom Server ausgewertet. Die möglichen Variablenbelegungen werden dann in der Antwort-Nachricht aufgeführt. Zum Beispiel könnte eine RDQL-Anfrage für das Beispiel aus Abschnitt 7.2 so aussehen:

```
SELECT ?property, ?class
WHERE (<#Testprofessor>, ?property, ?class).
```

Darauf wird der Server eine SOAP-Nachricht mit folgendem Inhalt zurücksenden.

```
<variable-bindings>
    <tuple>
        <variable name="property">
            http://www.w3.org/1999/02/22-rdf-syntax-ns
                #type
        </variable>
        <variable name="class">
            #Professor
        </variable>
    </tuple>
    <tuple>
        <variable name="property">
            http://www.w3.org/1999/02/22-rdf-syntax-ns
                #type
        </variable>
        <variable name="class">
            http://www.w3.org/2002/07/owl#Thing
        </variable>
    </tuple>
    <tuple>
        <variable name="property">
            liest_Vorlesung
        </variable>
        <variable name="class">
            #Testvorlesung
        </variable>
    </tuple>
</variable-bindings>
```

Die Syntax entspricht dabei der in [i5-d4] vorgeschlagenen XML-Syntax für Variablen-Bindungen. Diese Funktion kann auch für den Condition-Teil einer globalen ECA-Regel genutzt werden.

Um einen Trigger einzugeben, muss eine SOAP-Nachricht mit folgendem Format gesendet werden:

```
<ADDTRIGGER>
   Triggerdefinition (siehe Abschnitt 8.1)
</ADDTRIGGER> .
```

Um einen Trigger wieder zu löschen, muss eine Nachricht der folgenden Form gesendet werden:

<DELETETRIGGER>
DELETE TRIGGER *Triggername*
</DELETETRIGGER> .

Erhält der Server eine SOAP-Nachricht mit dem XML-Tag

<STATEMENTS/>

so ermittelt er alle Klassen mit zugehörigen Instanzen und Properties. Diese werden als Inhalt der Antwort-Nachricht zurück geschickt. Das Parsen und Darstellen der Klassen und Instanzen bleibt dabei dem Client überlassen.

Auf die Nachricht

<LISTTRIGGER/>

reagiert der Server mit einer Liste aller im System aktiven Trigger.

7.4 Technische Realisierung

Den Aufbau des Servers verdeutlicht das UML-Diagramm in Abbildung 7.1. Ich habe darin die meisten privaten Attribute und Methoden weggelassen, da diese das Diagramm zu groß und unübersichtlich machen. Es sind daher nur die für das Verständnis wichtigen privaten Attribute zu sehen.

Der Server besteht aus drei Paketen *server*, *trigger* und *util*. Das Paket *server* ist dabei für den Betrieb am wichtigsten. Alle ankommenden SOAP-Nachrichten werden hier von der Klasse `SOAPServer` angenommen. Je nach umgebendem XML-Tag werden diese dann an die entsprechenden Methoden in den Klassen `ModelServer` oder `TriggerParser` übergeben.
Alle Update-Aktionen werden von der Klasse `ModelServer` dieses Paketes ausgeführt. Für die Speicherung der RDF-Daten werden hier zwei verschiedene Objekte vom Typ `Model` eingesetzt. Das Datenbank-Model enthält alle Daten, die zu Beginn aus einer RDF/XML-Datei eingelesen wurden oder vom Benutzer als Statements eingetragen wurden. Diese Daten werden in einer Datenbank gespeichert, die in der Konfigurationsdatei festgelegt ist. Das zweite Model ist eine Instanz der Klasse `OntModel`, welches zusätzlich die Möglichkeit bietet, die Ontologie von einem Reasoner auswerten zu lassen. Dieser ergänzt die Daten aus dem Datenbank-Model um weitere abgeleitete Statements. Durch die in dieser Klasse implementierten Methoden wird dabei die Datenbasis des `OntModel`s geändert.
Bei einem „INSERT“ wird das entsprechende Statement hinzugefügt, während bei einer „DELETE“-Operation das Statement entfernt wird. Für beide Operationen stellt das Jena-Framework fertige Methoden bereit, jedoch findet sich hier keine Methode um ein Statement zu ändern. Daher sind die

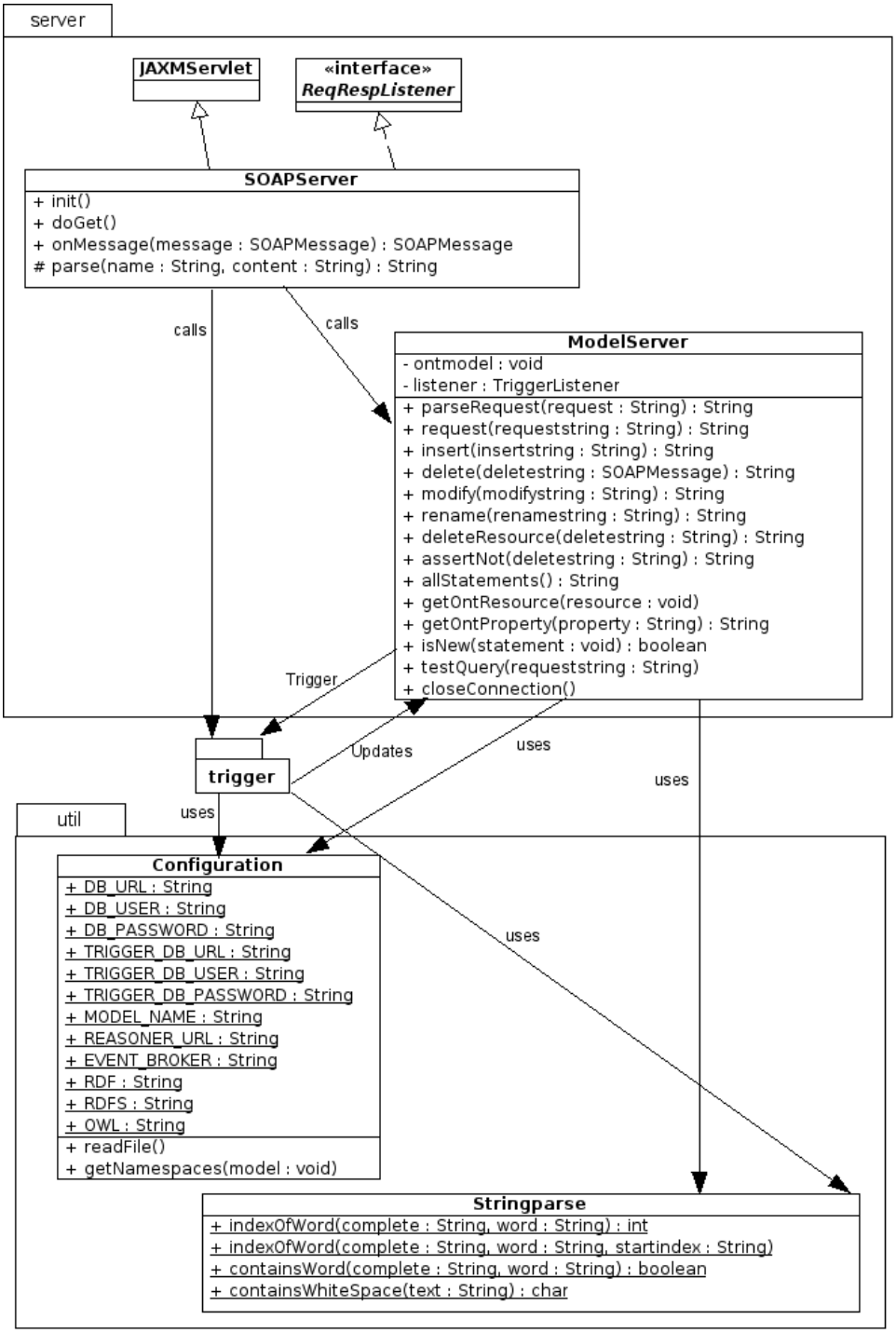


Abbildung 7.1: Die Paketstruktur des Servers

Methoden für „MODIFY“ und „RENAME“ etwas aufwändiger. Bei „MODIFY“ werden hierfür jeweils das alte und das neue Statement erstellt. Ist das alte Statement im Modell enthalten, so wird dieses entfernt und das neue dafür eingefügt. Bei „RENAME“ ist das Vorgehen sehr ähnlich, jedoch findet das Entfernen und Einfügen auf einer Menge von Statements statt, die die festgelegte Ressource aufweisen und ggf. der festgelegten Klasse angehören. Auch bei „DELETE RESSOURCE“ wird zunächst die Menge von Statements mit der betreffenden Ressource bestimmt. Dann werden alle diese Statements gelöscht. „ASSERT NOT“ arbeitet zunächst genau wie „DELETE“. Danach wird jedoch noch einmal explizit überprüft, ob das zu löschende Statement noch im Modell enthalten ist. Ist dies der Fall, so werden die Änderungen wieder rückgängig gemacht, indem genau die gegenteiligen *delete*- und *add*-Operationen ausgeführt werden.
Der aufwändigste Teil dieser Methoden ist die Extraktion der einzelnen Elemente und das Abfangen möglicher Eingabefehler.
Im Paket *trigger* sind alle für die Funktion der Trigger wesentlichen Klassen enthalten. Die Trigger werden in Kapitel 8 betrachtet, für die Funktion des restlichen Servers spielt dieses Paket keine Rolle.
Das Paket *util* stellt lediglich einige Hilfsmethoden zur Verfügung, die sowohl von den im Paket *server* als auch im Paket *trigger* enthaltenen Klassen genutzt werden. Dazu gehört die Klasse `Configuration`, die beim Start des Servers die Daten aus der Konfigurationsdatei ausliest, die nicht fest vorgegeben werden können. Hierzu gehört die URL der Datenbank, wie auch der Benutzername und das Passwort. In der Klasse `Stringparse` sind Methoden enthalten, die für das korrekte Parsen von Triggern und Update-Funktionen nötig sind. Sie alle erweitern die Möglichkeiten der Klasse String. Dies ermöglicht es zum Beispiel, ein Schlüsselwort zu detektieren, ohne dass Teile von anderen Wörtern berücksichtigt werden.

Kapitel 8

Trigger

Die Trigger stellen das Kernstück dieser Arbeit dar. Sie reagieren auf Änderungen an den RDF-Daten mit eigenen Änderungen oder mit dem Versenden von Nachrichten. Zunächst werden die Trigger beschrieben, die im normalen Gebrauch angewendet werden, danach wird noch auf einen Spezialfall eingegangen.

8.1 Syntax

Hier soll zunächst die Syntax der Trigger einführt werden. Dabei ist zu beachten, dass alle in diesem Abschnitt behandelten Trigger auf Änderungen am Modell reagieren und nicht auf den Aufruf der Update-Funktion selbst. Soll der Aufruf der Update-Funktion als Auslöser dienen, so können hierfür die unter 8.3 aufgeführten „direkten“ Trigger verwendet werden. Da es sehr vielfältige Möglichkeiten für Trigger gibt, habe ich eine kompakte Darstellung durch einen regulären Ausdruck gewählt.

CREATE TRIGGER *Name*
[**LET** *Variablenname* := *Variablenwert*]*
ON [[[INSERTION | DELETION | MODIFICATION] OF *Property* [OF *Class*]?]
| [[CREATE | DELETE | UPDATE] OF INSTANCE OF *Class*]
| [NEW CLASS] | [NEW PROPERTY [OF *Class*]?]
| [NEW PROPERTY OF INSTANCE OF *Class*]
| [NEW STATEMENT ABOUT INSTANCE OF *Class*]]
[**WHEN** *RDQL-Anfrage*]?
DO
BEGIN
[[*Update Action* | SEND(*Adresse*, *Nachricht*) | RAISE EVENT(*Eventname*(*Variable1*, *Variable2*, ...))];]*
END;

Update Action kann dabei jede der unter 7.1 aufgeführten Update-Operationen („Insert“, „Delete“, „Modify“, „Rename“ oder „Delete Resource“) sein, einzige Ausnahme ist hier die Funktion „Assert Not“, die hier nicht verwendet werden kann, da ein Trigger die Rückmeldung nicht interpretieren könnte.

Bei der Festlegung der Syntax habe ich mich weitgehend an die Definition im Paper [i5-d4] gehalten. Jedoch ist die im Paper vorgeschlagene Syntax bei Triggern, die auf Löschen oder Ändern von Klassen reagieren, nicht eindeutig. Die Syntax wäre hier die gleiche wie beim Löschen oder Ändern eines Properties. Ich habe daher die Worte „OF INSTANCE“ eingefügt, so dass der Auslöser nun eindeutig zugeordnet werden kann.

Die Triggersyntax kann dabei in verschiedene Abschnitte unterteilt werden. Jede Triggerdefinition beginnt mit den Worten „CREATE TRIGGER“ gefolgt von dem eindeutigen Namen des Triggers. Unter diesem Namen wird der Trigger in der Datenbank gespeichert. Er ist auch der Schlüssel, um Trigger wieder löschen zu können. Mit „ON“ wird der Auslöser eingeleitet. Bei „ON INSERTION OF *Property*“ reagiert der Trigger wenn ein Statement mit dem definierten Property eingefügt wurde. Stehen im Auslöser zusätzlich die Worte „OF *Class*“, so muss das Subjekt des Statements der festgelegten Klasse angehören, damit der Trigger ausgeführt wird. Entsprechend reagiert der Trigger auf Lösch- und Update-Operationen, wenn statt „INSERTION“ „DELETION“ oder „MODIFICATION“ gewählt wird. Trigger, bei denen im Event-Teil die Worte „ON CREATE OF INSTANCE OF *Class*“ stehen, reagieren auf das Einfügen einer neuen Instanz der Klasse *Class*. Bei „DELETE“ reagiert der Trigger auf das Löschen einer Instanz dieser Klasse. „UPDATE“ bedeutet hier eine Änderung an einer Instanz dieser Klasse, so dass diese Instanz nicht länger der Klasse angehört. „ON NEW CLASS“ bezeichnet das Event, dass eine neue Klasse eingefügt wurde. Bei „NEW PROPERTY OF INSTANCE OF *Class*“ wurde einer Instanz der festgelegten Klasse ein neues Property zugewiesen. Der Auslöser „NEW STATEMENT ABOUT INSTANCE OF *Class*“ wirkt prinzipiell genau so, nur reagiert dieser auch, wenn es dieses Property für die Instanz bereits gibt und nur ein neuer Wert eingetragen wurde. Damit ein Trigger mit dem Auslöser „NEW PROPERTY“ ausgeführt wird, muss ein neues Property auf Metadatenebene definiert werden. Dies geschieht entweder durch Einfügen eines Statements [*property*, rdf:type, owl:ObjectProperty] oder zum Beispiel durch Angabe des Definitions- oder Wertebereichs eines Properties (rdfs:domain, rdfs:range). Mit den Worten „OF *Class*“ kann hier zusätzlich der Definitionsbereich (engl.: Domain) eines Properties festgelegt werden. Der Trigger löst in diesem Fall nur aus, wenn ein Property hinzugefügt wurde und die festgelegte Klasse in der Domain des neuen Properties enthalten ist. Auf Grund der Implementierung in Jena kann die Domain eines Properties nur erkannt werden, wenn das Property zuvor als ObjectProperty definiert wurde.

Tritt das definierte Ereignis ein, so wird zunächst die RDQL-Anfrage im „WHEN“-Teil, sofern vorhanden, ausgewertet. Diese Anfrage wird immer auf einer Kopie des Modells ausgewertet, dass vor der ersten Änderung gemacht wurde. Dieses Modell ist immer konsistent und die Anfragen sind damit unabhängig von der Reihenfolge in der die Trigger ausgewertet werden. Ist bei der RDQL-Anfrage eine Variablenbindung zustande gekommen, so wird der Trigger weiter ausgeführt. Das Schlüsselwort „DO“ leitet den Aktionsteil ein. Hier kann entweder ein lokales Update durchgeführt werden oder es wird eine SOAP-Nachricht gesendet. Dabei können zwischen „BEGIN“ und „END;“ beliebig viele Aktionen stehen. Es ist damit auch möglich, sowohl ein lokales Update auszuführen, als auch eine Nachricht zu versenden oder Nachrichten an verschiedene Server zu verschicken. Es gibt zwei verschiedene Typen von Nachrichten. Mit „SEND“ wird eine selbstdefinierte Nachricht an einen beliebigen Server gesendet. „RAISE EVENT“ versendet eine Nachricht im XML-Format an den in der Konfigurationsdatei festgelegten Eventbroker. Dabei werden XML-Tags mit den Variablennamen gebildet, der Inhalt ist der jeweilig Wert.

Die Variablendefinitionen im LET-Teil sind optional. Bei „RAISE EVENT“ können hiermit XML-Elemente in der Form

```
<Variablenname>
  Variablenwert
</Variablenname>
```

eingefügt werden. Auch bei den anderen möglichen Aktionen können solche Variablen definiert werden. Jedoch stellen sie dann nur eine Abkürzung dar, die bei der Ausführung durch den definierten Wert ersetzt wird.
Auch die WHEN-Klausel ist optional. Die dabei gebundenen Variablen können jedoch durch Voranstellen eines Dollarzeichens ($) anstelle des bei RDQL benutzten Fragezeichens auch im Aktionsteil als Variablen verwendet werden. In diesem Fall wird jede Aktion für jede Variablenbindung einmal ausgeführt. Einzige Ausnahme bildet hier die Aktion „RAISE EVENT“. Hier wird, falls keine Variablenbindung gefunden wurde, eine alternative Nachricht ohne die entsprechenden Variablen gesendet. Werden die Variablen in der Aktion nicht weiter verwendet, wirkt der WHEN-Teil wie eine Bedingung: Gibt es eine Variablenbindung, werden alle Aktionen einmal ausgeführt, gibt es keine, gilt dies als „false“ und der Trigger wird nicht weiter ausgeführt.

Es gibt noch eine dritte Art von Variablen, die sowohl in der RDQL-Anfrage, als auch in den Update-Operationen oder den SOAP-Nachrichten verwendet werden können. Dies sind die Elemente des auslösenden Events und das Event selbst. Hierzu werden feste Variablen verwendet:

Beim Einfügen von Tripeln (INSERT) gibt es die Variablen: $new, $new.subject, $new.property und $new.object. Falls der Trigger auf die Erzeugung neuer Klassen reagiert, gibt es zusätzlich die Variable $new.class.

Bei Triggern mit dem Auslöser „ON NEW PROPERTY“ stehen die Variablen $new.property und $new.domain zur Verfügung. Beim Löschen von Tripeln (DELETE) gibt es die entsprechenden Variablen: $old, $old.subject, $old.property und $old.object. Bei einem Update stehen sowohl das alte als auch das neue Tripel zur Verfügung. Diese können über die $old- und die $new-Elemente angesprochen werden.
Wird beispielsweise ein Triple [http://BeispielInstanz, http://hatFarbe, http://Farbe#rot] eingefügt, so kann in der Nachricht durch den Alias $new.object auf das Element http://Farbe#rot zugegriffen werden. Die Ersetzung der Variablen erfolgt immer bei Auslösung eines Triggers.

8.2 Beispiel

An dieser Stelle soll das Beispiel aus Abschnitt 7.2 fortgesetzt werden, so dass die Funktion der Trigger an Beispielfällen nachvollzogen werden kann.

Hierzu werden zunächst über den Button „Add Trigger“ die folgenden Trigger eingegeben.

```
CREATE TRIGGER Prof_zu_Vorlesung
ON INSERTION OF liest_Vorlesung OF #Professor
WHEN SELECT ?type WHERE (<$new.object>, ?type, <#
    Vorlesung>)
DO
BEGIN
SEND(http://localhost:8080/triggerserver/triggerserver
    , Der Professor fuer Vorlesung $new.object steht
    nun fest: $new.subject);
END;

CREATE TRIGGER neue_Vorlesung
LET $universitaet := TU Clausthal
LET $institut := Institut fuer Informatik
ON CREATE OF INSTANCE OF #Vorlesung
DO
BEGIN
RAISE EVENT(neue_Vorlesung($universitaet, $institut,
    $new.subject));
END;

CREATE TRIGGER neues_Statement
ON NEW STATEMENT ABOUT INSTANCE OF #Person
DO
BEGIN
SEND(http://localhost:8080/triggerserver/triggerserver
```

```
    , Der Person $new.subject wurde die Eigenschaft
    $new.property zugewiesen!);
END;

CREATE TRIGGER neue_Klasse
ON NEW CLASS
DO
BEGIN
INSERT (Testinstanz, rdf:type, $new.class);
RAISE EVENT (neue_Klasse($new.class));
END;

CREATE TRIGGER neues_Property
ON NEW PROPERTY
DO
BEGIN
SEND(http://localhost:8080/triggerserver/triggerserver
    , Es wurde ein neues Property  $new.property
eingefuegt);
END;

CREATE TRIGGER Property_Vorlesung
ON NEW PROPERTY OF #Vorlesung
DO
BEGIN
RAISE EVENT(neue_vorlesung($new.property,    $new.
    domain));
END;

CREATE TRIGGER neues_Property_fuer_Instanz
ON NEW PROPERTY OF INSTANCE OF #Vorlesung
DO
BEGIN
RAISE EVENT (neues_Property($new.subject,  $new.
    property, $new.object));
END;
```

Danach werden nacheinander mittels „Insert Statement“ Statements eingefügt:

```
INSERT (Fortsetzungsvorlesung, rdf:type, #Vorlesung)
```

Hierdurch wird eine Instanz der Klasse #Vorlesung erstellt, worauf der Trigger ‚neue Vorlesung‘ reagiert. Es wird nun eine SOAP-Nachricht an den Eventbroker gesendet:

```
<neue_Vorlesung>
    <universitaet>
        TU Clausthal
    </universitaet>
    <institut>
        Institut fuer Informatik
    </institut>
    <newsubject>
        Fortsetzungsvorlesung
    </newsubject>
</neue_Vorlesung>
```

Außerdem reagiert hier der Trigger ‚neues_Property_fuer_Instanz' auf die Erzeugung des Properties „rdf:type". Dadurch erhält der Eventbroker auch das Event:

```
<neues_Property>
  <newsubject>
    Fortsetzungsvorlesung
  </newsubject>
  <newproperty>
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
  </newproperty>
  <newobject>
    #Vorlesung
  </newobject>
</neues_Property>
```

```
INSERT (#Testprofessor, liest_Vorlesung,
    Fortsetzungsvorlesung)
```

Da #Testprofessor eine Instanz der Klasse #Professor ist, wird hierdurch der Trigger ‚Prof_zu_Vorlesung' ausgelöst. Durch die Subklassen-Beziehung von #Professor und #Person wird hier eine neue Eigenschaft bei einer Person eingefügt und somit auch der Trigger ‚neue_Eigenschaft' ausgelöst. Dabei empfängt der Triggerserver die folgenden Nachrichten:

```
<Trigger>
    Der Professor fuer Vorlesung Fortsetzungsvorlesung
        steht nun fest: #Testprofessor
</Trigger>
```

und

```
<Trigger>
    Der Person #Testprofessor wurde die Eigenschaft
       liest_Vorlesung zugewiesen!
</Trigger>.
```

Durch

```
INSERT (Testklasse, rdf:type, owl:Class)
```

wird der Trigger ‚neue_Klasse' ausgelöst und eine Instanz Testinstanz der Klasse Testklasse erzeugt. Als zweite Aktion wird von diesem Trigger die folgende Nachricht gesendet:

```
<neue_Klasse>
  <newclass>
    Testklasse
  </newclass>
</neue_Klasse>
```

Mit

```
INSERT (findet_statt_in, rdf:type, owl:ObjectProperty)
```

wird das neue Property findet_statt_in definiert. Dies löst den Trigger neues_Property aus:

```
<Trigger>
  Es wurde ein neues Property  findet_statt_in
     eingefuegt
</Trigger>.
```

Für dieses Property wird nun eine Domain festgelegt:

```
INSERT (findet_statt_in, rdfs:domain, #Vorlesung).
```

Hier zeigt sich der Nachteil der zwei verschiedenen Auslöser für einen Trigger ON NEW PROPERTY. Der Trigger wird durch das Einfügen dieses Statements ein zweites Mal ausgelöst.

```
<Trigger>
  Es wurde ein neues Property  findet_statt_in
     eingefuegt
</Trigger>
```

Dadurch das jetzt zu dem ObjectProperty auch eine Domain festgelegt wurde, reagiert in diesem Fall auch der Trigger Property_Vorlesung.

```
<neue_vorlesung>
  <newproperty>
    findet_statt_in
  </newproperty>
  <domain>
    #Vorlesung
  </domain>
</neue_vorlesung>
```

Das neue Property wird schließlich bei einer Instanz der Klasse Vorlesung angewendet.

```
INSERT (Fortsetzungsvorlesung, findet_statt_in,
    HoersaalA).
```

Dies löst den Trigger ‚neues_Property_fuer_Instanz' aus, der die folgende Nachricht sendet:

```
<neue_Eigenschaft>
  <newsubject>
    Fortsetzungsvorlesung
  </newsubject>
  <newproperty>
    findet_statt_in
  </newproperty>
  <newobject>
    HoersaalA
  </newobject>
</neue_Eigenschaft>.
```

Ein weiteres Beispiel zur Funktion der Trigger findet sich im Anhang.

8.3 Direkte Trigger

Neben den Triggern, die auf Änderungen am Modell reagieren, erlaubt der Server noch eine zweite Art von Triggern. Dabei reagieren die Trigger nicht auf tatsächliche Änderungen, sondern auf den Aufruf der Update-Funktion. In einem Modell ist es möglich, dass ein Statement das gelöscht werden soll durch den Reasoner aus anderen Daten wieder rekonsturiert wird. Dies ist zum Beispiel der Fall, wenn ein Property mit dem OWL-Element ‚owl:inverseOf' als inverses Property eines anderen Properties definiert wird. So kann zum Beispiel einerseits definiert werden, dass Berlin die Hauptstadt von Deutschland ist und über das inverse Property das Deutschland die Hauptstadt Berlin hat. Wird eines der beiden Statements eingegeben, so gibt es auch automatisch das komplementäre Statement im Modell. Sind in dem Modell Statements mit beiden Properties vorhanden, so wird bei der Löschung eines Statements dieses automatisch aus dem anderen Statement und den Informationen über das Property wieder hergestellt. Auch ein „normaler" Trigger kann hier nicht eingreifen und zusätzlich das inverse Statement löschen, da das Statement ja nicht wirklich aus dem Modell gelöscht wurde. An dieser Stelle sind die direkten Trigger sinnvoll. Ein solcher Trigger reagiert schon auf den Aufruf, dass das besagte Statement gelöscht werden soll und sorgt gleich für die Löschung des inversen Statements. Dadurch kann dieses Statement wirklich aus dem Modell gelöscht werden.

Gerade bei den direkten Triggern muss bei der Definition darauf geachtet werden, dass hierdurch keine Endlosschleifen entstehen. Eine Einfügung oder Löschung bewirkt hier direkt, das die Trigger ausgelöst werden, unabhängig davon, ob ein solches Statement wirklich hinterher im Modell vorhanden ist. Diese Trigger haben fast die gleich Syntax, wie Trigger, die auf Änderungen am Modell reagieren. Jedoch sind die Auslöser hier anders definiert. Möglich sind hier nur die Auslöser:

ON [INSERT | DELETE | UPDATE] OF *Property*

Auch die Aktionen sind bei diesen Triggern eingeschränkt. Hier können nur die Update-Funktionen verwendet werden, die in Abschnitt 7.1 beschrieben wurden. Wie auch bei den anderen Triggern, ist die Funktion ‚Assert Not' jedoch nicht als Trigger Aktion möglich. Beliebige Nachrichten oder Events zu versenden ist von diesen Triggern aus nicht möglich.
Ein Beispiel zur Verwendung der direkten Trigger findet sich im nächsten Abschnitt.

8.4 Konsistenzerhaltung

Der Reasoner wertet ein Modell aus und leitet über die OWL- und RDFS-Elemente Statement ab. Dabei kann es zu Inkonsistenzen kommen. Zum Beispiel werden einem Individuuen zwei Klassen zugeordnet, die als disjunkt definiert sind. Treten solche Inkonsistenzen im Modell auf, so ist es wichtig, dass diese gleich abgefangen werden, ansonsten würde der Reasoner nicht mehr arbeiten können.
Im vorliegenden Programm erhält der Benutzer die Möglichkeit auftretende Inkonsistenzen mit Triggern abzufangen und so wieder ein konsistentes Modell herzustellen. Gelingt dies jedoch nicht und das Modell ist nach Ausführung aller Trigger inkonsistent, so wird das Modell wieder zurückgesetzt und das auslösende Update zurückgewiesen. Beide vorgestellten Typen von Triggern können für die Konsistenzerhaltung im Modell eingesetzt werden. Damit jedoch alle Operationen rückgängig gemacht werden können wenn sich das Modell als inkonsistent herausstellt, dürfen zunächst nur die Update-Aktionen der Trigger ausgeführt werden. Alle SOAP-Nachrichten (beliebige Nachrichten und Events) werden erst versendet, wenn nach Ausführung aller Trigger das Modell positiv auf Konsistenz getestet wurde.
Das Verhalten bei Inkonsistenz kann an einem Beispiel nachvollzogen werden. Hierzu wird die Ontologie universitaet.owl benutzt.

Listing 8.1: Die Ontologie universitaet.owl

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE rdf:RDF [
  <!ENTITY owl     "http://www.w3.org/2002/07/owl#">
```

```
]>
<rdf:RDF
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns
    #"
 xmlns:owl="&owl;"
 xmlns:test="http://localhost/test.owl#"
 xml:base="http://localhost/test.owl#"
>
  <owl:Class rdf:ID="Person"/>
  <owl:Class rdf:ID="Universitaet"/>
  <owl:Class rdf:ID="Stadt"/>

  <owl:ObjectProperty rdf:ID="geborenIn">
    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
    <rdfs:domain rdf:resource="#Person" />
    <rdfs:range  rdf:resource="#Stadt" />
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="hatPraesident">
    <rdf:type rdf:resource="&owl;FunctionalProperty"/>
    <rdfs:domain rdf:resource="#Universitaet"/>
    <rdfs:range  rdf:resource="#Person" />
  </owl:ObjectProperty>

  <owl:AllDifferent>
    <owl:distinctMembers rdf:parseType="Collection">
      <test:Stadt rdf:about="#Berlin"/>
      <test:Stadt rdf:about="#Clausthal"/>
      <test:Stadt rdf:about="#Dortmund"/>
       </owl:distinctMembers>
  </owl:AllDifferent>

  <owl:Thing rdf:ID="Person1">
    <rdf:type rdf:resource="#Person"/>
    <test:geborenIn rdf:resource="#Dortmund"/>
  </owl:Thing>

  <owl:Thing rdf:ID="Person2">
    <rdf:type rdf:resource="#Person"/>
    <test:geborenIn rdf:resource="#Berlin"/>
  </owl:Thing>

  <owl:Thing rdf:ID="UniversitaetA">
```

```
    <rdf:type rdf:resource="#Universitaet"/>
    <test:hatPraesident rdf:resource="#Person1"/>
  </owl:Thing>
</rdf:RDF>
```

Obwohl in der Ontologie Namespaces verwendet werden, werde ich diese bei der Beschreibung weglassen, da sie diese unnötig lang machen. In dieser Ontologie werden die Klassen „Person“, „Universitaet“ und „Stadt“ definiert. Es werden hier außerdem zwei Properties definiert, das Property „geborenIn“ und „hatPraesident“. Für beide Properties werden mit Range und Domain die Klassen des Subjekts und des Objekts für jedes Statement festgelegt, in dem dieses Property verwendet wird. Beide Properties sind zudem als „owl:FunctionalProperty“ definiert. Das bedeutet, dass dieses Property für jedes Individuum nur einmal vergeben werden kann. In der Ontologie werden drei Individuen der Klasse „Stadt“ beschrieben, die durch das OWL-Element „owl:AllDifferent alle verschieden sind. Wird dies nicht definiert, so könnten diese vom Reasoner auf die selbe Instanz abgebildet werden, um das Modell erfüllbar zu machen. In dieser Ontologie werden drei weitere Individuuen definiert. Für die Individuen „Person1“ und „Person2“ werden zusätzlich die Städte festgelegt, in denen diese Personen geboren sind. Es wird außerdem definiert, dass „Person1“ der Präsident des Individuums „UniversitaetA“ ist. Dieses Modell ist konsistent. Wird diesem Modell jedoch das folgende Statement hinzugefügt, so würde das Modell inkonsistent werden.

```
INSERT (http://localhost/test.owl#UniversitaetA,
    http://localhost/test.owl#hatPraesident,
    http://localhost/test.owl#Person2)
```

Die Operation wird deshalb mit der Meldung

```
The model would become inconsistent by this operation.
```

zurückgewiesen. Wenn es jedoch möglich sein soll, einer Universität einen neuen Präsidenten zuzuweisen, ohne den alten Präsidenten zu löschen, so muss dafür ein Trigger definiert werden.
Für diesen Zweck wird hier ein direkter Trigger gewählt:

```
CREATE TRIGGER konsistenztest
ON INSERT OF http://localhost/test.owl#hatPraesident
WHEN SELECT ?praesident WHERE (<$new.subject>,
            <$new.property>, ?praesident)
DO
BEGIN
DELETE ($new.subject, $new.property, $praesident);
END;
```

Dieser Trigger sorgt dafür, dass alle im Modell vorhandenen Zuweisungen eines Präsidenten bei der Eingabe eines neuen Präsidenten für die gleiche

Universität gelöscht werden.
Wird wieder das oben beschriebene Statement eingegeben, so wird dieses nun akzeptiert. Dem Individuum „UniversitaetA“ ist nun „Person2“ als Präsident zugewiesen.

8.5 Technische Realisierung

Bei der Realisierung von Triggern gibt es drei wesentlich Bereiche. Erstens muss eine Möglichkeit geschaffen werden, um die Trigger eingeben und abspeichern zu können. Zweitens müssen die auslösenden Update-Aktionen detektiert werden. Dies können nicht nur die vom Benutzer eingegebenen Statements sein, sondern auch Änderungen in den durch den Reasoner erzeugten Statements. Drittens müssen die Trigger dann unabhängig von ihrem tatsächlichen Inhalt ausgeführt werden.

Für die Umsetzung der Trigger wurden verschiedene Klassen erstellt. Das entsprechende Klassendiagramm ist in Abbildung 8.1 zu sehen. Für die Einbindung der Trigger in die Paketstruktur des Servers möchte ich noch einmal auf das Klassendiagramm in Abbildung 7.1 verweisen. Die Klasse `TriggerObject` realisiert die eigentlichen Trigger. Die Klasse `TriggerListener` wird bei Änderungen aufgerufen und führt dann die einzelnen `TriggerObject`s aus. Die Klasse `TriggerParser` sorgt für das Einlesen und Löschen der Trigger. Die Datenbank-Kommunikation wird vollständig über die Klasse `TriggerDB` abgewickelt. Über die Klasse `SOAPmessage` können von den Triggern SOAP-Nachrichten verschickt werden.

8.5.1 Triggereingabe

Die Klasse `TriggerParser` wird vom Server aufgerufen, wenn dieser einen neuen Trigger erhält. In einer recht aufwändigen Methode wird der Trigger nun in seine Bestandteile zerlegt und auf syntaktische Korrektheit überprüft. Sind alle nötigen Bestandteile vorhanden und fehlerfrei, werden diese an die Klasse `TriggerDB` übergeben. Diese sorgt dafür, dass der Trigger so in der Datenbank gespeichert wird, dass eine einfache Rekonstruktion beim Neustart des Servers möglich ist. Wenn der Trigger in die Datenbank eingetragen werden konnte, wird ein `TriggerObject` konstruiert und dieses dem `TriggerListener` übergeben. Erzeugt das Speichern in der Datenbank einen Fehler, zum Beispiel weil schon ein Trigger mit diesem Namen eingetragen ist, so bricht die Triggererzeugung ab und der Fehler wird dem Benutzer gemeldet.
Der `TriggerParser` sorgt außerdem dafür, dass Trigger wieder aus der Datenbank und dem System gelöscht werden können. Dazu wird der Name des Triggers extrahiert und an Methoden der Klassen `TriggerDB` und `TriggerListener` übergeben.

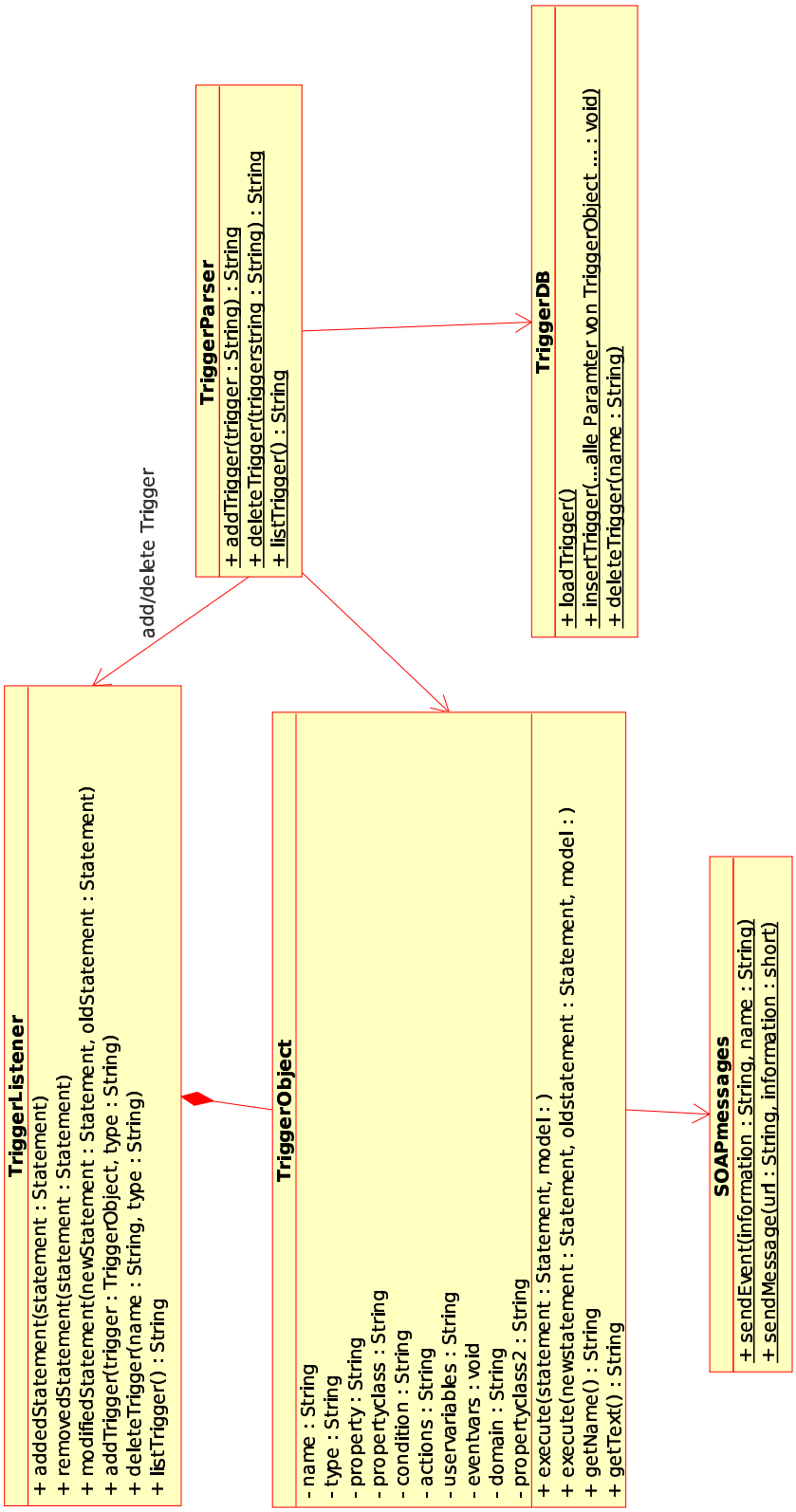


Abbildung 8.1: Klassendiagramm des Pakets *trigger*

8.5.2 Auslöser

Ein Trigger muss zunächst davon Kenntnis erhalten, dass eine Update-Aktion stattgefunden hat. Dafür wird von Jena eine Listener-Klasse bereitgestellt, der `ModelChangedListener`. Dieser bietet mit den Methoden *addedStatement* und *deletedStatement* die Möglichkeit, auf Änderungen zu reagieren. Sofern der entsprechende Listener für das Modell mit `Model`.*register*(`Listener`) registriert ist, löst jedes *add(*Statement*)* und jedes *delete(*Statement*)* den Listener aus und führt die entsprechende Methode aus. Wie bei Java üblich, müssen die Listener in einer eigenen Klasse implementiert werden, die von der Listenerklasse abgeleitet ist. Die Methoden, die benutzt werden sollen, müssen überschrieben werden. Von der Klasse `ModelChangedListener` abgeleitet, gibt es weitere in Jena implementierte Listener-Klassen. Besonders günstig für meine Zwecke erwies sich dabei der `StatementListener`, da hier nur zwei Methoden überschrieben werden müssen. Auch wenn durch eine Methode mehrere Statements hinzugefügt oder gelöscht werden, wird für jedes einzelne Statement die entsprechende Methode aufgerufen.

Dieses Konzept schien mir für die Trigger gut geeignet zu sein. Daher habe ich die Klasse `TriggerListener` von der Klasse `StatementListener` abgeleitet. Innerhalb der beiden Methoden *addedStatement* und *deletedStatement* werden dann die vorhandenen Trigger aufgerufen. Hier wird jeweils überprüft, ob das der Methode übergebene Statement den Triggerbedingungen entspricht und falls das der Fall ist, wird der Aktions-Teil ausgeführt. Dieses Vorgehen funktionierte zunächst sehr gut, bis externe Reasoner angewendet wurden. Der Reasoner hat die Aufgabe, weitere Statements zu erzeugen. Bestehen zum Beispiel Subklassen-Beziehungen, muss eine Instanz nicht nur für eine Klasse eingefügt werden, sondern auch für alle Klassen, von denen diese Klasse eine Unterklasse ist. Der Reasoner erzeugt diese Statements auch automatisch, allerdings lösen diese den Listener nicht aus. Wird ein neuer Student im Modell angelegt, so ist es dadurch leider nicht möglich, einen Trigger, der auf neu eingefügte Personen reagiert, auszulösen. Daher habe ich schließlich den Weg über die Listener verworfen.
Werden vor und nach einer Update-Aktion die vorhandenen Statements des Modells ausgelesen, so unterscheiden sich diese in dem durch die Update-Aktion veränderten Statement sowie allen durch den Reasoner geänderten. Auf dieser Grundlage basieren nun die Trigger. Zwischen zwei Update-Aktionen wird das Modell nicht verändert. Dadurch kann das Modell, das nach der letzten Update-Operation erstellt wurde, auch für den Vergleich bei der nächsten Operation benutzt werden. Die im Modell vorhandenen Statements müssen daher für eine Operation nur einmal bestimmt werden.
Nachdem die geänderten Statements bestimmt wurden, werden diese in einer Schleife durchlaufen und die jeweilige `TriggerListener`-Methode aufgerufen. Dabei werden zunächst nur die Update-Aktionen der Trigger ausgeführt.

Gesendete SOAP-Nachrichten könnten bei einer Zurücksetzung des Modells durch Inkonsistenz nicht wieder rückgängig gemacht werden. Die geänderten Statements werden deshalb für jede Operation („Insert“, „Delete“, „Update“) mitgeschrieben und die Trigger nach erfolgreicher Validierung noch einmal ausgelöst, wobei dann nur die SOAP-Nachrichten gesendet werden. Um die Validierung des Models erst nach Ausführung aller Trigger durchführen zu können, wird bei jeder Methode ein Boolscher Wert übergeben, der angibt ob die Methode von einer externen Nachricht ausgelöst wird oder von einem internen Trigger aufgerufen wird. Prinzipiell ist der Ablauf bei beiden Varianten der gleiche, jedoch wird bei der internen Methode auf die Validierung verzichtet. Auch die Ausführung der Trigger-Aktionen bei denen Nachrichten versendet werden, wird bei den internen Methoden nicht durchgeführt. Stattdessen werden die auslösenden Statements in drei Vektoren gesammelt (für „Insert“, „Delete“ und „Update“). Wird nun die auslösende externe Methode erfolgreich validiert, so werden die Trigger noch einmal für alle in den Vektoren gesammelten Statements aufgerufen.
Handelt es sich bei der entsprechenden Aktion um eine Insert-Operation, so werden nur die Statements untersucht, die zu dem Modell dazugekommen sind. Bei der derzeitigen Konfiguration sind hierbei keine Löschoperationen möglich, so dass diese nicht berücksichtigt werden müssen (siehe dazu Kapitel 10). Die Statements die durch direkte Trigger gelöscht werden, werden bereits zuvor in den entsprechenden Operationen berücksichtig. Sind bei einer Insert-Operation die Hinzugekommenen Statements bestimmt, so wird für jedes dieser Statements die Methode *addedStatement* aufgerufen und damit die normalen Trigger ausgeführt. Entsprechend werden bei einer Delete-Operation nur die aus dem Model entfernten Statements betrachtet. Komplizierter wird das Vorgehen bei einer Update-Operation. Zunächst werden die aus dem Modell entfernten und die neu hinzugekommenen Statements bestimmt. Nun kommt es auf die Art der Operation an. Handelt es sich zum Beispiel um die Operation ‚Update Subject‘, werden alle Statements, die aus dem Modell entfernt wurden, mit den neuen Statements verglichen. Gibt es dabei ein Paar, welches sich nur im Subjekt unterscheidet, so wird für dieses die Methode *updatedStatement* aufgerufen. Dabei kommt es vor, dass nicht für jedes Statement ein Partner gefunden wird. Wird beispielsweise ein Individuum statt der momentanen Klasse einer Subklasse dieser Klasse zugeordnet, so erzeugt der Reasoner nur ein Statement, welches die Zugehörigkeit des Individuums zur Subklasse angibt. Das Statement, welches die Zugehörigkeit zur ursprünglichen Klasse beschreibt, bleibt durch die Subklassen-Beziehung bestehen. Aus dem gleichen Grund kommt es auch vor, dass es Statements gibt, die nur gelöscht werden. Für diese überzähligen Statements werden die Listener-Methoden noch einmal getrennt aufgerufen. Diese Art Trigger auszulösen, ist gegenüber dem Listener natürlich viel aufwändiger. Es muss hier nach jeder Operation die gesamte Menge der Statements bestimmt werden. Je nach Größe des Modells sind deutlich längere

Antwortzeiten zu erwarten. Da aber auch die vom Reasoner abgeleiteten Statements berücksichtigt werden müssen, ist dies leider nicht vermeidbar. Die Berechnung der geänderten Statements durch einen Vergleich erzeugt allerdings noch einen weiteren Unterschied zur Listener-Methode. Der Listener reagiert auf jeden Aufruf von *add* oder *delete*, unabhängig davon, ob ein Statement tatsächlich eingefügt oder geändert wird, weil vorher nicht überprüft wird, ob das Statement im Modell vorhanden ist. Dagegen reagiert der Trigger beim Modellvergleich nur auf tatsächlich geänderte Statements. Jedoch können auch hier die Methodenaufrufe durch die direkten Trigger abgefangen werden. Für einen sinnvollen Einsatz der Trigger ist dies ein Vorteil.

Ich habe vor diesem Hintergrund den in Jena eingebauten Reasoner noch einmal untersucht. Zwar lösen beim Einsatz des internen Reasoners auch abgeleitete Tripel den Listener aus, jedoch gab es dabei zwei Probleme. Zum einen ist der eingebaute Reasoner um ein Vielfaches langsamer, so dass die Antwortzeit selbst bei einer relativ kleinen Datenmenge nicht tolerabel ist. Zum anderen ist der enthaltene Funktionsumfang sehr eingeschränkt, so dass viele häufig verwendete Teile von OWL nicht erkannt werden.
Nachdem das Programm mit dem internen Reasoner gestartet wurde, traten schon bei der Bestimmung der Klassen und ihrer Instanzen Probleme auf. Beim Einsatz des externen Reasoners benötigte das Programm für die Bestimmung wenige Sekunden. Nach etwa 10 Minuten wurde das Programm mit dem internen Reasoner mit folgender Fehlermeldung beendet: „Cannot convert node http://www.w3.org/2002/07/owl#equivalentClass to Individual“. Auch nach Entfernung dieser OWL-Elemente aus der Ontologie kam es zu ähnlichen Fehlern. Dadurch scheint mir der interne Reasoner für das Programm ungeeignet.

Bei jeder Änderungsoperation werden die ermittelten geänderten Statements in eine Log-Datei geschrieben. Diese Datei trägt den Namen „updatedStatements.log“ und befindet sich im selben Verzeichnis wie die Konfigurationsdatei. Da in diese Datei auch die durch den Reasoner geänderten Statements geschrieben werden, kann sie gut für die Überprüfung der Triggerfunktion verwendet werden.

Um die Konsistenz des Modells zu gewährleisten, wird in jeder Änderungsmethode das Modell auf Konsistenz überprüft. Zuvor können jedoch schon Trigger reagieren, dabei werden jedoch nur Update Aktionen ausgeführt. Nachrichten werden nach der Konsistenzprüfung versandt. Um dies zu realisieren wird bei Methoden aufruf ein Wert übergeben, der angibt, ob die Methode von außen oder von einem Trigger aufgerufen wird. Wird die Methode von einem Trigger aufgerufen, so wird hier keine Konsistenzprüfung vorgenommen und auch das Senden von Nachrichten wird zunächst verscho-

ben. So können die Trigger für die Konsistenzerhaltung im Modell sorgen, ohne dass Nachrichten über Änderungen versandt werden, die später wieder rückgängig gemacht werden. Sind alle Änderungen erfolgt und das Modell ist konsistent, so werden die Trigger noch einmal mit den geänderten Nachrichten aufgerufen, diesmal werden jedoch nur Nachrichten versandt.
Ich möchte den Modellvergleich, den ich bei jeder Änderungsoperation durchgeführt wird, am Beispiel der Insert-Methode beschreiben. In dieser Methode wird zunächst das Statement, das hinzugefügt werden soll, aus dem String extrahiert und in ein entsprechendes Objekt aus dem Jena-Framework umgewandelt. Handelt es sich um einen Aufruf durch eine entsprechende SOAP-Nachricht, so folgt darauf dieser Quellcode.

```
//reset the log
logstatement.removeAllElements();
logoperation.removeAllElements();
beforemodel = aftermodel;
aftermodel = ModelFactory.createOntologyModel();
//Catching errors due to inconsistent model
try
{
  //executing direct triggers
  internalaftermodel = beforemodel;
  listener.directInserted(statement);

  //the actual operation
  if (!ontmodel.contains(statement))
  {
    ontmodel.add(statement);
    logstatement.add(statement);
    logoperation.add(new Boolean(true));
  }

  aftermodel.add(ontmodel);

  Model differencemodel = aftermodel.difference(
     internalaftermodel);
  Vector<Statement> differences = new Vector<Statement
     >();
  //triggering the internal updates
  internalaftermodel = aftermodel;
  /*the internal operations are logged to execute the
     event
   * triggers after the validation*/
  internaladd.removeAllElements();
```

```
internaldelete.removeAllElements();
internalmodify.removeAllElements();

/*------Comparing the diffences-----*/
StmtIterator iter = differencemodel.listStatements()
    ;
while (iter.hasNext())
{
  Statement addedstatement = (Statement)iter.next();
  differences.add(addedstatement);
  //Trigger with update actions
  listener.addedStatement(addedstatement, false);
}
aftermodel = internalaftermodel;
//Test if the model is consistent
ValidityReport report = ontmodel.validate();
if (!report.isValid())
{
  //undo all changes
  for (int j=logstatement.size()-1; j>=0; j--)
  {
    if (logoperation.get(j).booleanValue())
    {
      ontmodel.remove(logstatement.get(j));
    }
    else
    {
      ontmodel.add(logstatement.get(j));
    }
  }
  return "The model would become inconsistent by
     this operation.";
}
//Trigger sending messages and events
for (int i=0; i<differences.size(); i++)
{
  Statement addedstatement = differences.get(i);
  try
  {
    writer.write("added: "+addedstatement+"\n");
  }
  catch(IOException ie)
  {
    System.out.println(ie.getStackTrace());
```

```
    }
    listener.addedStatement(addedstatement, true);
  }
  executeInternalTriggers();
}
catch(DIGReasonerException digerr)
{
  System.out.println("The reasoner is not available or
      the model is inconsistent!");
  //undo all changes
  for (int j=logstatement.size()-1; j>=0; j--)
  {
    if (logoperation.get(j).booleanValue())
    {
      ontmodel.remove(logstatement.get(j));
    }
    else
    {
      ontmodel.add(logstatement.get(j));
    }
  }
  return("The model would become inconsistent by this
     operation.");
}
```

In den Vectoren „logstatement“ und „logoperation“ wird jede vorgenommene Änderungsoperation gespeichert. Dies ist nötig, damit bei Inkonsistenz das Modell komplett zurückgesetzt werden kann, bei Relationalen Datenbanken spricht man hier von einem „Rollback“. Dabei dürfen nur Operationen gespeichert werden, die tatsächlich eine Änderung bewirken. Ist ein Statement bereits im Modell vorhanden, so wird eine add-Operation hier nichts bewirken. Wenn diese Operation aber in den Log-Dateien gespeichert wird und das Modell später mit diesen Informationen wieder zurückgesetzt wird, wird dieses Statement aus dem Modell gelöscht obwohl dies nicht Teil des ursprünglichen Zustands ist. Zu Beginn dieses Codeausschnittes werden die „log“-Dateien zurückgesetzt. Dann wird hier das Modell „aftermodel“ das bei der letzten Operation erstellt wurde der Variablen „beforemodel“ zugewiesen. Der Variablen „aftermodel“ wird ein leeres Modell zugewiesen. Zunächst werden dann die direkten Trigger ausgeführt. Bei den hier gemachten Änderungen wird ebenfalls ein Modellvergleich durchgeführt, so dass das „internalaftermodel“dem Modell nach allen Änderungen durch die direkten Trigger entspricht. Deshalb erfolgt der Vergleich auch nicht mit dem „beforemodel“sondern mit diesem Modell. Dann wird mit dem Befehl ontmodel.*add(statement)* die eigentliche Änderungsoperation ausgeführt und dies

in den „log“-Dateien vermerkt. Mit der Jena-Methode *add* werden dann alle Statements des mit dem Reasoner verbundenen „ontmodel“s zum leeren „aftermodel“ hinzugefügt.
Mit dem Befehl aftermodel.*difference(internalaftermodel)* wird ein „differencemodel“ erzeugt. Dieses enthält genau die Statements, die im „aftermodel“ enthalten sind, nicht aber im „internalaftermodel“. In den Vectoren „internaladd“, „internaldelete“ und „internalmodify“ werden die durch Trigger veränderten Statements eingetragen um für diese später die Trigger auslösen zu können, die als Aktion Nachrichten senden.
Die Methode *listStatements()* liefert einen Iterator mit allen Statements des Differenzmodells zurück. Diese werden jeweils in die Log-Datei eingetragen und dann an die Methode *addedStatement* des `TriggerListener`s übergeben. Diese führt für alle Trigger sofern sie ausgelöst werden nur Updates aus. Das gleiche wird für die Statements aus dem Modell durchgeführt, dass die nun gelöschten Statements enthält.
ontmodel.validate() überprüft die Konsistenz des Modells. Ist das Modell inkonsistent, so werden alle in den „log“-Dateien gespeicherten Operationen rückgängig gemacht, indem die gegenteiligen Operationen in umgekehrter Reihenfolge ausgeführt werden. Ist der Test dagegen erfolgreich, so werden für alle bei der Differenzbildung erhaltenen Statements die Trigger noch einmal aufgerufen und die entsprechenden Nachrichten versandt. Die Methode *executeInternalTriggers()* sorgt dafür, dass dies auch für die Statements geschieht, die von den Triggern durch Update-Aktionen geändert wurden.

8.5.3 Triggerausführung

Der eigentliche Trigger wurde mit Hilfe eines `TriggerObject`s realisiert. Dieses Objekt enthält alle für den Trigger nötigen Elemente als Variablen. Besonders wichtig für das Abspeichern und spätere Löschen des Triggers ist das Attribut *name*, welches den Trigger eindeutig identifiziert. Mit *type* wird der Typ des Triggers festgelegt, möglich sind hier ‚INSERTION‘, ‚MODIFICATION‘, ‚DELETION‘, ‚CREATE‘, ‚UPDATE INSTANCE‘, ‚DELETE INSTANCE‘, ‚NEW CLASS‘, ‚NEW PROPERTY‘,‚NEW PROPERTY OF INSTANCE‘ und ‚NEW STATEMENT ABOUT INSTANCE‘. Für direkt auf Änderungsoperationen reagierende Trigger gibt eh hier noch die Belegungen ‚INSERT‘, ‚UPDATE‘ und ‚DELETE‘. *Property* definiert das Prädikat des Statements, auf das der Trigger reagieren soll. Falls die Variable *propertyclass* gesetzt ist, so definiert sie im Fall von ‚INSERTION‘, ‚MODIFICATION‘, ‚DELETION‘, ‚NEW PROPERTY OF INSTANCE‘ und ‚NEW STATEMENT ABOUT INSTANCE‘ die Klasse, der das Subjekt des geänderten Statements angehören muss, um den Trigger auszulösen. Sonst beschreibt *propertyclass* das Objekt des auslösenden Statements. Mit *condition* wird, falls vorhanden, eine RDQL-Anfrage als Bedingung gespeichert. Die für den Aktionsteil nötigen Variablen werden in dem zweidimensionalen

Array *actions* gespeichert. Für jede Aktion des Triggers enthält der Array dabei fünf Elemente. Die erst Variable gibt den Modus der Aktion an, die ausgeführt wird, wenn der Trigger ausgelöst wird. Mögliche Belegungen hier sind ‚insert', ‚delete', ‚modify', ‚rename', ‚delete resource', ‚message' und ‚event'. Im zweiten Array-Element ist die eigentliche Updateoperation gespeichert bzw. bei ‚message' und ‚event' der Inhalt der Nachricht. Falls als Aktion eine beliebige SOAP-Nachricht zu senden ist - das erste Element hat dann den Wert ‚message'- enthält die Variable an fünfter Position die URL des Empfängers.
Für den Fall ‚event' enthält der Array *actions* weitere Elemente, die nur hier eine Bedeutung haben. Dies ist zum einen das Attribut *eventname*, dessen Wert an dritter Stelle gespeichert wird, welches den Namen des äußeren XML-Tags des Events speichert. Dieser wird für das Erzeugen der SOAP-Nachricht benötigt. Außerdem wird an der vierten Stelle des Arrays eine alternative Nachricht gespeichert, die keine Variablen aus der RDQL-Anfrage enthält. Sie wird gesendet, falls die RDQL-Anfrage keine Variablenbindung zurückgegeben hat. Schließlich wird unter dem Namen *eventvars* ein Array mit jeweils einer Liste mit den Variablen, die in der Event-Nachricht vorkommen, gespeichert. Diese sind nur für die Ausgabe des Triggers wichtig. Es wäre zu aufwändig, diese Variablen jedes Mal wieder aus der Nachricht zu extrahieren.
Wurden mittels „LET" vom Benutzer Variablen definiert, so werden diese in der Variable *uservariables* gespeichert. Dies ist vor allem für Trigger mit einer Aktion „RAISE EVENT" interessant, in allen anderen Fällen ist dies nur als abkürzende Schreibweise zu sehen.
Die Variablen *domain* und *propertyclass2* sind für den Auslöser ‚NEW PROPERTY' nötig. *propertyclass2* definiert hier das zweite mögliche Objekt eines auslösenden Statements. Lautet das Event ‚ON NEW PROPERTY OF *Class*' , wird die Domain *Class* des neuen Properties in der Variable *domain* gespeichert.

Jeder Trigger besitzt zwei *execute*-Methoden. Hier wird getestet, ob das übergebene Statement die Triggerbedingungen erfüllt. Gegebenenfalls wird die RDQL-Anfrage ausgewertet und der Aktionsteil ausgeführt. Die beiden Methoden sind nahezu identisch, jedoch werden bei der zweiten Version zwei Statements übergeben. Sie ist für die Auswertung von Update-Operationen gedacht, bei denen immer das alte und das neue Statement übergeben werden.

Der Aufruf der `TriggerObject`s erfolgt durch den `TriggerListener`. Hier sind drei Vektoren mit `TriggerObject`s für die Operationen *insert*, *update* und *delete* vorhanden, drei weitere Vektoren enthalten die TriggerObjects für die entsprechenden direkten Trigger. Wird nun die Methode *addedStatement* aufgerufen, so wird für jedes `TriggerObject` des Vektors

insert die Methode *execute* mit dem übergebenen Statement ausgeführt. In dieser Methode wird zunächst überprüft, ob das Statement dem definierten Auslöser des Triggers entspricht. Falls ein ‚WHEN'-Teil vorhanden ist, so wird dieser nun ausgewertet. Zuvor werden dabei die vorhandenen Variablen ersetzt. Gibt es mindestens eine mögliche Variablenbelegung, wird mit dem Aktionsteil fortgefahren. Wurde keine Belegung gefunden, wird im Fall „RAISE EVENT" die alternative Nachricht gesendet. Andernfalls wird die Ausführung des Triggers abgebrochen. Werden die im „WHEN"-Teil bestimmten Variablen in der Aktion verwendet, wird diese für jede Variablenbindung einmal ausgeführt. Sonst erfolgt die Aktion nur einmal.

8.5.4 TriggerDatenbank

Damit beim Neustart des Servers nicht alle Trigger wieder neu eingegeben werden müssen, war es nötig, die Trigger in einer Datenbank zu speichern. In dem vorliegenden Programm wird hierfür eine PostgreSQL-Datenbank verwendet. Zur Speicherung sind die Relationen **trigger** und **triggeractions** nötig. Diese basieren auf dem folgende Datenbankschema:

```
CREATE TABLE „trigger"
(
name varchar(200) NOT NULL,
„type" varchar(30),
property varchar(200),
propertyclass varchar(200),
condition varchar(300),
uservariables varchar(400),
domain varchar(200),
propertyclass2 varchar(200),
CONSTRAINT trigger_pkey PRIMARY KEY (name)
)

CREATE TABLE triggeractions
(
name varchar(200) NOT NULL references trigger(name),
actionmode varchar(15),
„action" varchar(300),
number integer NOT NULL,
eventname varchar(200),
altmessage varchar(300),
messageurl varchar(200),
CONSTRAINT actions_pkey PRIMARY KEY (name, number)
)
```

Die Attribute der Relation **trigger** entsprechen den Variablen des `TriggerObject`s. Die Aktionen werden dabei in der Relation **triggeractions** gespeichert, deren Attribut ‚name' als Fremdschlüssel auf die Relation „trigger" definiert ist. Mit dem Attribut ‚number' wird die Reihenfolge der Aktionen festgelegt. Die restlichen Attribute dieser Relation bilden die Elemente des Arrays „actions" des `TriggerObject`s.
Die Kommunikation mit der Datenbank wird ausschließlich über die Klasse `TriggerDB` abgewickelt. Sie enthält eine Methode, um Trigger in der Datenbank zu speichern. Außerdem gibt es eine Methode, um alle vorhandenen Trigger auszulesen und diese in `TriggerObject`s zu verwandeln. Diese Methode wird bei der Initialisierung des Servers aufgerufen. Schließlich gibt es noch die Möglichkeit, einzelne Trigger zu löschen. Diese werden dabei durch ihren Namen identifiziert.
Die Parameter für den Datenbankzugriff werden in der Konfigurationsdatei festgelegt. Bei der Initialisierung des Servers werden diese durch die Klasse `Configuration` geladen.

8.5.5 SOAPmessage

Die Klasse `SOAPmessage` bietet zwei verschiedene Methoden zum Versenden von SOAP-Nachrichten. Eine Methode ist für Aktionen des Typs ‚message' bestimmt. Hierdurch wird die bei der Triggereingabe festgelegte Nachricht an die festgelegte URL gesendet. Die andere Methode ist für die Aktion „RAISE EVENT" vorgesehen. Die URL, an die die Nachricht verschickt wird, ist dabei immer die des in der Konfigurationsdatei festgelegten Eventbrokers. Der Inhalt der Nachricht wird schon bei der Eingabe des Triggers zusammengesetzt, jedoch mit den Variablen anstelle der Werte. Bei Auslösen des Triggers werden alle Werte ermittelt und die Variablen dadurch ersetzt. Dann wird eine SOAP-Nachricht mit dem im Trigger definierten Namen und diesem Inhalt gesendet.
Das Erstellen der Nachricht ist recht aufwändig, da nicht bekannt, ist in welcher Reihenfolge die Variablen auftreten. Es gibt zudem drei mögliche Gruppen von Variablen. Das sind zum einen die Variablen des aktuellen Statements wie ‚$new.subject'. Es können jedoch auch Variablen aus der RDQL-Anfrage oder vom Benutzer definierte Variablen aus dem „LET"-Teil sein. Daher habe ich mich entschlossen, die Erstellung der XML-Tags nur bei der Eingabe eines Triggers durchzuführen. Wird der Trigger ausgeführt, müssen jetzt nur einfache String-Ersetzungen ausgeführt werden.

Kapitel 9

Zusätzliche Programme

Der in dieser Arbeit entwickelte Web-Server soll in dem in Kapitel 2 vorgestellten REWERSE-Projekt mit anderen Komponenten zusammenarbeiten. Diese stehen jedoch zur Zeit noch nicht zur Verfügung. Daher wurden andere Programme entwickelt, die diese Funktion simulieren.

9.1 ModelLoader

Der ModelLoader ist ein kleines Java-Programm, welches dazu dient, RDF-Daten aus Dateien zu lesen und als Modell in einer Datenbank zu speichern. Da der Server das Modell nur aus der Datenbank ausliest, ist diese Funktion für den Einsatz des Web-Services sehr wichtig. Das Programm bietet außerdem die Möglichkeit, ein Modell aus der Datenbank zu laden und sich die vorhandenen Statements anzeigen zu lassen. Es ist auch möglich, ein vorhandenes Modell aus der Datenbank zu löschen.
Weiterhin kann der Benutzer mit dem ModelLoader sehen, welche Modelle in der Datenbank vorhanden sind. Darüber hinaus kann ein Modell mit Hilfe eines externen Reasoners validiert werden. Dies ist besonders hilfreich, da bei einem inkonsistenten Modell der Reasoner nicht arbeitet.
Auch dieses Programm basiert auf dem Jena-Framework. Die benötigten Daten wie URL der Datenbank und der Benutzername werden zur Laufzeit erfragt. Dabei muss auch eine Datei angegeben werden, in der die einzulesenden Daten im RDF/XML-Format zur Verfügung stehen. Für die Validierung benötigt das Programm einen externen Reasoner.

9.2 TriggerServer

Im normalen Betrieb können von den Triggern zwei Arten von SOAP-Nachrichten verschickt werden. Zum einen sind dies Nachrichten an eine festzulegende URL, zum anderen Nachrichten an einen sogenannten Eventbroker.

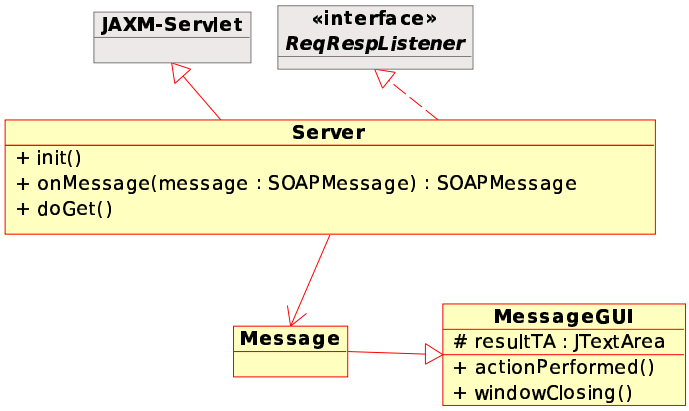


Abbildung 9.1: Klassendiagramm des TriggerServers

Dieser empfängt die Nachrichten, die bei „RAISE EVENT“ gesendet werden, und schickt sie an die hier eingetragenen URLs weiter.
Da die SOAP-Nachrichten synchron gesendet werden, ist es nötig, dass die empfangende URL existiert und auch eine Antwort zurücksendet. Außerdem war es für die Kontrolle der Funktion wichtig zu sehen, wann welche Nachricht empfangen wird.
Daher wurde ein weiteres Servlet entwickelt, welches aus allen eingehenden Nachrichten den Inhalt extrahiert und in einem Fenster darstellt. Zudem sendet dieses Servlet eine Antwort-SOAP-Nachricht zurück.
Die Klassen des TriggerServers befinden sich im Paket *triggerserver* in derselben war-Datei wie der Server. Angesprochen wird der TriggerServer, sofern sich die war-Datei auf dem gleichen Rechner befindet mit:
http://localhost:8080/triggerserver/triggerserver.
Abbildung 9.1 zeigt das Klassendiagramm des TriggerServers.

Im selben Paket befindet sich ein zweites Servlet, welches die gleiche Aufgabe erfüllt und dem ersten Servlet sehr ähnlich ist. Jedoch wird der Inhalt der Nachricht hier nicht in einem Fenster präsentiert, sondern in die Log-Datei geschrieben. Ein solches Servlet wird benötigt, falls das Servlet keine Berechtigung für die graphische Oberfläche des Servers besitzt. Dieses Servlet wird mit der URL
http://localhost:8080/triggerserver/logserver angesprochen.

Kapitel 10

Auswertung

In dem vorliegenden Programm werden bei jeder Update-Operation die Statements des Modells abgespeichert und mit denen vor der Operation verglichen. Dies kann bei großen Modellen aufwendig sein und ist daher keine ideale Lösung. Es stellt sich die Frage, ob es eine Möglichkeit gibt, die vom Reasoner abgeleiteten Statements an die Trigger weiterzuleiten, ohne diesen Vergleich durchführen zu müssen.
Es wäre wünschenswert, die Änderungen, die der Reasoner erzeugt, mit dem Listener abfangen zu können. Bei internen Reasonern funktioniert dies, da die Statements materialisiert werden. Das bedeutet, dass jedes Statement in einem Modell-Graphen gespeichert wird. Das Speichern löst dann den Listener aus. Ein externer Reasoner materialisiert die Statements nicht. Er reagiert nur auf Anfragen und ermittelt die aktuellen Statements durch Description Logic und den Daten aus dem Programm. Die Statements werden daher bei der Änderungsoperation nicht komplett geändert, lediglich die Datenbasis ändert sich. Erst durch die Abfrage aller Statements werden die neuen Statements ermittelt. Aus diesem Grund kann der Listener nicht direkt auf Änderungen an den durch den Reasoner zusätzlich erzeugten Statements reagieren. Auch mögliche Änderungen der Schnittstelle zwischen Jena und dem externen Reasoner hätten keinen Erfolg.
Eine Möglichkeit für die Auslösung des Listeners ohne Modellvergleich wäre, schon vorher zu ermitteln, welche Statements betroffen sein werden. Für diesen Zweck könnten beispielsweise Tabellen oder Dictionaries mitgeführt werden, in denen die Superklassen einer Klasse gespeichert sind. Wird nun an dieser Klasse etwas geändert, so finden die Änderungen auch an den Superklassen statt. Welche Statements bei einer Änderungsoperation durch den Reasoner zusätzlich geändert werden, hängt einerseits davon ab, um was für eine Art Operation es sich handelt. Bei einer Insert-Operation sollten auch vom Reasoner Statements eingefügt werden, während es bei einer Delete-Operation zu weiteren Löschungen kommen wird. Weiterhin ist es jedoch von Interesse, was für eine Art von Statement geändert wird. Beim

Einfügen eines Individuums sind ganz andere inferierte Statements möglich als beim Einfügen einer neuen Klasse. Das größte Problem ist dabei, dass vorher nicht bekannt ist, was für eine Art von Statement geändert wird. Dadurch dass jedes syntaktisch korrekte Statement eingegeben werden kann, ist eine einfache Fallunterscheidung ausgeschlossen. Wird beispielsweise eine neue Subklassen-Beziehung eingefügt, müssen für alle Individuen der Unterklasse Statements erstellt werden, die ausdrücken, dass diese auch der übergeordneten Klasse angehören.
Es müsste im Programm der Reasoner nachgebildet werden, um alle Änderungen erfassen zu können. Dies ist extrem aufwändig und sicherlich nicht der richtige Weg.

Ein weitere Möglichkeit auf den Modellvergleich zu verzichten beinhaltet eine Umstellung der Programmstruktur. Dabei werden die Auslöser der Trigger bei der Eingabe in RDQL-Anfragen umgewandelt. Diese werden vor und nach jeder Änderung an das Modell gestellt. Die Differenz der Ergebnisse der Anfragen bilden genau die Statements, die die Trigger auslösen würden. Dabei werden die Auslöser der Trigger wie folgt in RDQL-Anfragen umgesetzt:
Aus dem Auslöser ON INSERTION OF *property* wird die RDQL-Anfrage

```
SELECT ?subject, ?object
WHERE (?subject, <property>, ?object).
```

Die gleiche Anfrage wird auch bei DELETION oder MODIFICATION anstelle von INSERTION gewählt. Nur der Vergleich der Ergebnisse muss dabei geändert werden. Soll das Subjekt zusätzlich einer bestimmten Klasse angehören, so wird dem Auslöser OF *class* hinzugefügt. Dadurch ändert sich die Anfrage in

```
SELECT ?subject, ?object
WHERE (?subject, <property>, ?object)
  (?subject, <rdf:type>, <class>).
```

Der Auslöser ON [CREATE | DELETE | UPDATE] OF INSTANCE OF *class* wird zu der Anfrage

```
SELECT ?instance
WHERE (?instance, <rdf:type>, <class>).
```

Trigger die auf die Erzeugung neuen Klassen reagieren (ON NEW CLASS) können mit der folgenden Anfrage erfasst werden:

```
SELECT ?class
WHERE (?class, <rdf:type>, <owl:class>).
```

ON NEW STATEMENT ABOUT INSTANCE OF *class* wird durch diese Anfrage ersetzt:

```
SELECT ?subject, ?property, ?object
WHERE (?subject, ?property, ?object)
  (?subject, <rdf:type>, <class>).
```

Die gleiche Anfrage kann auch für Auslöser der Form ON NEW PROPERTY OF INSTANCE OF *class* genutzt werden. Hier muss jedoch sichergestellt werden, dasss vorher kein Statement mit dem gleichen Subjekt und Objekt im Modell vorhanden war. Dies kann mit Hilfe der Ergebniss aus der Anfrage vor der Änderung geschehen.
Auslöser der Form ON NEW PROPERTY können mit den folgenden Anfragen erfasst werden.

```
SELECT ?property
WHERE (?property, rdf:type, owl:ObjectProperty)
```

und

```
SELECT ?property
WHERE (?property, rdf:type, rdf:Property)
```

Da dieser Trigger zwei mögliche Auslöser hat, müssen beide Anfragen getrennt ausgewertet werden.
Wird dabei zusätzlich die Domain des Properties festgelegt, so wird aus der ersten Anfrage:

```
SELECT ?property, ?domain
WHERE (?property, <rdf:type>, owl:ObjectProperty)
(?property, <rdfs:domain>, ?class)
(?domain, <rdfs:subclassOf>, ?class).
```

Entsprechendes gilt für die zweite Anfrage, die hier wieder getrennt ausgewertet werden muss.
Die direkten Trigger kommen hier ohne RDQL-Anfrage aus.

Durch die völlig andere Struktur ist ein Vergleich der beiden Möglichkeiten für die Triggerauslösung schwierig. Bei der in dem vorliegendem Programm implementierten Variante werden nach jeder Änderung am Modell alle Statements mit einer von Jena zur Verfügung gestellten Methode in einem anderen Modell gespeichert. Danach wird mit einer ebenfalls aus Jena stammenden Methode die Differenz der zwischen diesem Modell und dem bei der letzten Änderung erhaltenen Modell gebildet. Für die Triggerauslösung werden nur noch die Statements betrachtet, die nach der Differenzbildung vorhanden sind. Für die Differenzstatements werden nur die TriggerObjekte ausgeführt, die zu dieser Änderung gehören. Dabei wird nach Insert, Delete und Update-Operationen unterschieden, wobei fünf der neun möglichen Auslöserformate der Trigger auf Insert-Operationen reagieren. Durchschnittlich sollten hierbei daher nur maximal 55% aller Trigger überhaupt angesprochen werden.

Der Vergleich, ob das übergebene Statement einen Trigger auslöst, ist relativ einfach, da die Elemente des übergebenen Statements mit festen Werten verglichen werden. Nur im Fall von Auslösern der Form [INSERTION | DELETION | MODIFICATION] OF *Property* OF *Class*, NEW PROPERTY OF INSTANCE OF *Class* und NEW STATEMENT ABOUT INSTANCE OF *Class* sind teilweise mehr Vergleiche nötig, da hier alle möglichen Klassen des Subjekts des übergebenen Statements überprüft werden müssen. Wird die passende Klasse gefunden, bricht der Vergleich ab. Im Normalfall ist die Menge an Klassen, denen eine Instanz angehört, nicht sehr groß, so dass diese Vergleiche überschaubar bleiben.
Bei der Methode, die Trigger durch einen Vergleich von RDQL-Anfragen auszulösen, unterscheidet sich der Ablauf deutlich von dem eben beschriebenen. Hier wird vor und nach jeder Änderung für jeden Trigger eine RDQL-Anfrage gestellt. Die Ergebnisse der einzelnen Anfragen müssen gespeichert werden und mit den späteren Ergebnissen verglichen werden. Für die in der Differenz enthaltenen Statements wird nun jeweils der entsprechende Trigger ausgeführt.
Die Anzahl der bei einer Operation geänderten Statements ist normalerweise relativ klein. Die eigentliche Triggerausführung für diese Statements ist bei beiden Varianten gleich und durch die wenigen Statements und die bei der Eingabe aufbereiteten Aktionen nicht sehr aufwändig. Bei dem Modellvergleich bleibt dabei als aufwändigster Teil das Speichern des Modells und die Berechnung der Differenz. Dies erfolgt jedoch unabhängig davon, wie viele Trigger definiert sind, nur einmal. Durch die Nutzung von Jena-internen Methoden sollte dies auch speicher- und zeitoptimiert implementiert sein. Die Umsetzung durch RDQL-Anfragen erfordert die Auswertung von einer Menge von RDQL-Anfragen. Wenn viele Trigger im Server aktiv sind, ist dies sicherlich auch aufwändig. Dabei kann nicht nur das Auswerten der RDQL-Anfragen selbst bei großen Modellen recht zeitintensiv sein. Wenn eine solche Anfrage viele Variablenbindungen zurückliefert, ist auch der Vergleich mit den Variablenbindungen nach der Auswertung langwierig. Die Auswertung einer einzelnen RDQL-Anfrage mit Vergleich der Variablenbindungen ist wahrscheinlich schneller als die Speicherung und der Vergleich der Statements des Modells. Bei einer Menge an RDQL-Anfragen kann dies jedoch sehr leicht anders aussehen.
Um hier einen genauen Vergleich machen zu können, wäre eine Testreihe mit unterschiedlich großen Modellen und einer unterschiedlichen Menge an Triggern nötig. Nach diesen Überlegungen erscheint mir die vorgestellte Möglichkeit der Triggerauslösung mit RDQL-Anfragen ein möglicher, jedoch nicht unbedingt ein effektiverer Weg zu sein. Andere Möglichkeiten für die Auslösung von Triggern sind mir nicht bekannt.

Im vorliegenden Programm werden bei einer Insert-Operation nur die neu hinzugefügten Statements berücksichtigt. Da die direkten Trigger bereits vorher der eigentlich Operation ausgeführt wurden und der Vergleich mit

dem Modell erfolgt, das bei der Auswertung dieser Trigger entsteht, können alle Änderungen nur direkt oder indirekt durch den Reasoner von dieser Operation stammen. Unter der „closed world“-Annahme ist es möglich, dass bei einer Insert-Operation auch Statements gelöscht werden. „closed world“ bedeutet dabei, dass für die Auswertung des Modells alle Statements bekannt sind. Ist ein Statement nicht im Modell vorhanden, so bedeutet das, dass dieses Statement auch nicht erfüllt ist. Das Gegenteil hierzu ist die „open world“-Annahme, bei der vorausgesetzt wird, dass noch weitere Statements in das Modell eingefügt werden können, die auch ein unter der „closed world“-Annahme inkonsistentes Modell wieder konsistent machen können.
Wird in einem Modell definiert, dass eine Klasse das Komplement einer anderen Klasse ist, so kann unter der „closed world“-Annahme bei der Löschung eines Individuums der ersten Klasse geschlossen werden, dass diese nun der zweiten Klasse angehört. Unter der „open world“-Annahme ist dieser Schluss nicht möglich. Hier könnte zum Beispiel ein Statement eingefügt werden, dass dieses Individuum mit einem anderend Individuum dieser Klasse gleichsetzt. Derartige Bedingungen dienen daher bei „open world“ nur zur Sicherung der Konsistenz, sie verhindern, dass ein Individuum gleichzeitig beiden Klassen angehört. Beim Einfügen von neuen Statements können hierdurch jedoch niemals andere Statements gelöscht werden und bei Löschungen keine Einfügungen stattfinden.
In diesem Programm wird mit der „open world“-Annahme gearbeitet. Daher ist die Untersuchung von neu eingefügten Statements bei Insert-Operationen vollkommen ausreichend, das gleiche gilt für Delete-Operationen. Sollte die Konfiguration im Betrieb soweit verändert werden, dass die Auswertung des Modells unter der „closed world“-Annahme erfolgt, so müssten diese Programmteile entsprechend angepasst werden. Zur Zeit ist eine Arbeit mit der „closed world“-Annahme aus technischen Gründen noch nicht möglich.

Kapitel 11

Zusammenfassung und Ausblick

Im Laufe dieser Arbeit wurde ein Web-Service entwickelt, der RDF-Daten bereitstellt. Mit Hilfe eines ebenfalls entwickelten Clients ist es möglich, diese Daten in einer aufbereiteten Form zu betrachten und RDQL-Anfragen an sie zu stellen. Über verschiedene Update-Funktionen können Änderungen an den Daten vorgenommen werden. Es können außerdem Trigger definiert werden, die auf diese Änderungen entweder mit weiteren lokalen Updates oder mit Nachrichten an andere Server reagieren.
Obwohl die Funktionalität des Programms gewährleistet ist, können noch einige Verbesserungen am Programm vorgenommen werden. Das Interface für Update-Funktionen könnte noch um weitere Funktionen erweitert werden. Es könnte z.B. eine Funktion geben, die automatisch eine neue RDF-Klasse erzeugt, ohne dass der Benutzer die RDF- bzw. OWL-Elemente dafür kennen muss. Da in RDF jedoch alles als Statement gesehen wird, erweitert dies lediglich die Benutzerfreundlichkeit, nicht aber die Funktionalität. In diesem Zusammenhang wäre es sicherlich in einigen Fällen auch wünschenswert, die Möglichkeit zu haben, RDF/XML-Dateien direkt einzulesen und dem Modell hinzuzufügen. Zusätzlich zu den bereits vorhandenen Beispieleingaben können „Hilfe“-Buttons im Client eingeführt werden. Diese zeigen dem Benutzer dann die gesamte hier mögliche Syntax sowie verschiedene Beispiele für die Benutzung.
Die Anzeige aller Klassen mit Individuen und Werten könnte so erweitert werden, dass Subklassen-Beziehungen und äquivalente Klassen leicht zu erkennen sind.
Ein weiterer Aspekt für mögliche Verbesserungen an dem Programm betrifft die Sicherheit. Zur Zeit werden die Benutzernamen und Passwörter im Klartext in die Konfigurationsdatei geschrieben. Für die Entwicklung und zum einfachen Testen ist dies sehr bequem, jedoch kann diese Datei auch von anderen Personen gelesen werden, sofern sie Zugriff auf den Server erlangen.

Eine mögliche Verbesserung wäre, die Passwörter und Usernamen bei jedem Neustart des Servers abzufragen, jedoch muss hierzu dann immer ein Benutzer zur Verfügung stehen. Eine bessere Lösung wäre es sicher, diese Daten vor dem Speichern zu verschlüsseln, so dass nur das Programm selbst die Daten nach dem Einlesen wieder entschlüsseln kann. Hierfür ist jedoch ein weiteres Programm nötig, mit dem eine verschlüsselte Konfiguration angelegt wird und bei Bedarf auch wieder geändert werden kann.

Um festzustellen ob die hier betrachtete Möglichkeit der Triggerauslösung unter den bestehenden Bedingungen optimal ist, müsste das in Kapitel 10 beschriebene Verfahren mit RDQL-Anfragen implementiert werden. Dann sollten für die beiden Implementierungen ausführliche Tests durchgeführt werden, wobei verschiedene Modelle und Trigger betrachtet werden müssen. Entscheidend wird dabei nicht nur die Menge der im Modell vorhandenen Statements und Trigger sein, sondern auch die Art der Trigger und die Menge der durch den Reasoner erzeugten Statements.

Anhang A

Beispiel zur Funktion der Trigger

Zum Abschluss möchte ich noch ein weiteres, umfangreicheres Beispiel angeben. Dieses basiert auf der Vegetarierontologie, die unter [ont1] zum Download zur Verfügung steht. Unter dem Namen vegetarian.owl ist diese Ontologie auch auf der CD enthalten, die dieser Arbeit beigefügt ist. Die Ontologie sieht wie folgt aus:

Listing A.1: Die Ontologie vegetarian.owl

```
<rdf:RDF
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns
    #"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns="http://www.mindswap.org/2003/vegetarian.owl#"
xmlns:un="http://www.ksl.stanford.edu/projects/DAML/
    UNSPSC.daml#"
>

<owl:Class rdf:ID="Omnivore"/>

<owl:Class rdf:ID="Vegetarian">
    <rdfs:subClassOf rdf:resource="http://www.isi.edu/
        webscripter/person.o.daml#Person"/>
    <rdfs:subClassOf rdf:resource="http://xmlns.com/
        foaf/0.1/Person"/>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#eats"/>
          <owl:allValuesFrom rdf:resource="#
              VegetarianFood"/>
```

```
        </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="http://xmlns.com/foaf
      /0.1/#Person"/>
 </owl:Class>



<owl:Class rdf:ID="Vegan">
    <rdfs:subClassOf rdf:resource="#Vegetarian"/>
    <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#eats"/>
          <owl:allValuesFrom rdf:resource="#VeganFood
              "/>
        </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>


<owl:Class rdf:ID="Ovo-LactoVegetarian">
    <rdfs:subClassOf rdf:resource="#Vegetarian"/>
    <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#eats"/>
          <owl:allValuesFrom rdf:resource="#
              OvoLactoVegetarianFood"/>
        </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="OvoVegetarian">
    <rdfs:subClassOf rdf:resource="#Ovo-
       LactoVegetarian"/>
    <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#eats"/>
          <owl:allValuesFrom rdf:resource="#
              OvoVegetarianFood"/>
        </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="LactoVegetarian">
```

```
    <rdfs:subClassOf rdf:resource="#Ovo-
        LactoVegetarian"/>
    <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#eats"/>
          <owl:allValuesFrom rdf:resource="#
              LactoVegetarianFood"/>
        </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>


<owl:ObjectProperty rdf:ID="eats">
    <rdfs:domain rdf:resource="#Vegetarian"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="VegetarianFood">
    <rdfs:subClassOf>
        <owl:Class>
            <owl:intersectionOf rdf:parseType="
                Collection">
                <owl:Class>
                    <rdfs:subClassOf rdf:resource="
                        http://www.ksl.stanford.edu/
                        projects/DAML/UNSPSC.daml#Food-
                        Beverage-and-Tobacco-Products
                        "/>
                </owl:Class>
                <owl:Class>
                    <owl:complementOf>
                        <owl:Class>
                            <rdfs:subClassOf rdf:
                                resource="#Meat"/>
                        </owl:Class>
                    </owl:complementOf>
                </owl:Class>
            </owl:intersectionOf>
        </owl:Class>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="VeganFood">
    <rdfs:subClassOf>
        <owl:Class>
```

```
            <owl:intersectionOf rdf:parseType="
                Collection">
                <owl:Class>
                    <owl:equivalentClass rdf:resource
                        ="http://www.ksl.stanford.edu/
                        projects/DAML/UNSPSC.daml#Food-
                        Beverage-and-Tobacco-Products
                        "/>
                </owl:Class>
                <owl:Class>
                    <owl:complementOf>
                        <owl:Class>
                            <owl:unionOf rdf:parseType
                                ="Collection">
                                <owl:Class><rdfs:
                                    subClassOf rdf:resource
                                    ="#Meat"/></owl:Class>
                                <owl:Class><rdfs:
                                    subClassOf rdf:resource
                                    ="http://www.ksl.
                                    stanford.edu/projects/
                                    DAML/UNSPSC.daml#Dairy-
                                    products-and-eggs"/></
                                    owl:Class>
                            </owl:unionOf>
                        </owl:Class>
                    </owl:complementOf>
                </owl:Class>
            </owl:intersectionOf>
        </owl:Class>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="OvoLactoVegetarianFood">
    <owl:equivalentClass rdf:resource="#VegetarianFood
        "/>
</owl:Class>


<owl:Class rdf:ID="OvoVegetarianFood">
    <rdfs:subClassOf>
        <owl:Class>
            <owl:intersectionOf rdf:parseType="
                Collection">
```

```
                <owl:Class>
                    <owl:equivalentClass rdf:resource
                        ="http://www.ksl.stanford.edu/
                        projects/DAML/UNSPSC.daml#Food-
                        Beverage-and-Tobacco-Products
                        "/>
                </owl:Class>
                <owl:Class>
                    <owl:complementOf>
                        <owl:Class>
                            <owl:unionOf rdf:parseType
                                ="Collection">
                             <owl:Class><rdfs:
                                subClassOf rdf:resource
                                ="#Meat"/></owl:Class>
                             <owl:Class><rdfs:
                                subClassOf rdf:resource
                                ="http://www.ksl.
                                stanford.edu/projects/
                                DAML/UNSPSC.daml#Dairy
                                "/></owl:Class>
                            </owl:unionOf>
                        </owl:Class>
                    </owl:complementOf>
                </owl:Class>
            </owl:intersectionOf>
        </owl:Class>
    </rdfs:subClassOf>
</owl:Class>


<owl:Class rdf:ID="LactoVegetarianFood">
    <rdfs:subClassOf>
        <owl:Class>
            <owl:intersectionOf rdf:parseType="
                Collection">
                <owl:Class>
                    <owl:equivalentClass rdf:resource
                        ="http://www.ksl.stanford.edu/
                        projects/DAML/UNSPSC.daml#Food-
                        Beverage-and-Tobacco-Products
                        "/>
                </owl:Class>
                <owl:Class>
```

```
                        <owl:complementOf>
                                <owl:Class>
                                    <owl:unionOf rdf:
                                        parseType="
                                        Collection">
                                     <owl:Class><rdfs:
                                         subClassOf rdf:
                                         resource="#Meat
                                         "/></owl:Class>
                                     <owl:Class><rdfs:
                                         subClassOf rdf:
                                         resource="http://
                                         www.ksl.stanford.
                                         edu/projects/DAML/
                                         UNSPSC.daml#Eggs
                                         "/></owl:Class>
                                    </owl:unionOf>
                                </owl:Class>
                        </owl:complementOf>
                    </owl:Class>
                </owl:intersectionOf>
            </owl:Class>
        </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Meat">
<rdfs:label> Meat, including seafood</rdfs:label>
<owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="http://www.ksl.stanford.edu/
        projects/DAML/UNSPSC.daml#Meat"/>
    <owl:Class rdf:about="http://www.ksl.stanford.edu/
        projects/DAML/UNSPSC.daml#Seafood"/>
</owl:unionOf>
</owl:Class>
</rdf:RDF>
```

Diese Ontologie bietet eine Kategorisierung von Vegetariern. Sie kann mit Hilfe des ModelLoaders in eine Datenbank eingelesen werden.

Zunächst werden verschiedene Trigger definiert, damit nach Möglichkeit alle Typen vorgestellt werden können. Dazu werden im Client über den Button „Add Trigger“ nacheinander die folgenden Trigger eingegeben.

```
CREATE TRIGGER test1
ON CREATE OF INSTANCE OF #Vegetarian
```

```
DO
BEGIN
SEND (http://localhost:8080/triggerserver/
    triggerserver, Es wurde ein neuer Vegetarier
    eingefuegt: $new.subject!);
END;


CREATE TRIGGER test2
ON MODIFICATION OF http://eats OF #Ovo-LactoVegetarian
WHEN SELECT ?food WHERE(?vegetarian, <http://eats>, ?
    food)
DO BEGIN INSERT (http://Vegetarianfood, http://
    Vegetarianfood#contains, $food);
END;


CREATE TRIGGER test3
LET $triggerserver:=http://139.174.247.64:8080/
    triggerserver/server
ON NEW CLASS
WHEN SELECT ?andere_klassen WHERE (?
    aequivalente_klasse, <rdf:type>, <owl:Class>), (?
    aequivalente_klasse, <OWL:equivalentClass>, <$new.
    class>)
DO BEGIN
RAISE EVENT(neue_Klasse($triggerserver, $new.class,
    $aequivalente_klasse));
END;


CREATE TRIGGER test4
ON NEW PROPERTY OF INSTANCE OF #Vegetarian
DO
BEGIN
SEND (http://localhost:8080/triggerserver/
    triggerserver, Dem Vegetarier $new.subject wurde
    die neue Eigenschaft $new.property zugewiesen!);
END;
```

Um die Trigger zu testen, werden folgende Änderungen an dem Modell vorgenommen:

Über den Button „Insert Statement“ des Clients wird folgendes eingegeben:

```
INSERT (http://personen#testperson, rdf:type, #Ovo-
    LactoVegetarian).
```

→ Hierdurch wird das Statement [*http://personen#testperson, rdf:type, #Ovo-LactoVegetarian*] dem Modell hinzugefügt.
Da LactoVegetarian als Subklasse von Vegetarian definiert ist, sollte hierauf der Trigger „test1“ reagieren und die folgende Nachricht senden:

Es wurde ein neuer Vegetarier eingefuegt: http://personen#testperson!

Außerdem wird hier mehrfach der Trigger „test4“ ausgelöst, da vom Reasoner auch für die übergeordneten Klassen Statements mit dem „neuen“ Property „rdf:type“ eingefügt werden.

Danach wird mit „Insert Statement“

```
INSERT (http://personen#testperson, http://eats, http
    ://food#milk)
```

eingegeben.

→ Da die Testperson auch der Klasse Vegetarier angehört und für sie ein neues Property angelegt wurde, sollte der Trigger „test4“ hierauf reagieren. Es wird folgende Nachricht an den Triggerserver verschickt:
Dem Vegetarier http://personen#testperson wurde die neue Eigenschaft http://eats zugewiesen!

Über den Button „Modify Statement“ wird danach folgende Änderung vorgenommen:

```
MODIFY OBJECT (http://personen#testperson, http://eats
    , http://food#milk, http://food#cheese)
```

→ Das Statement [*http://personen#testperson, http://eats, http://food#milk*] wird aus dem Modell entfernt und statt dessen das Statement [*http://personen#testperson, http://eats, http://food#cheese*] hinzugefügt.
Auf dieses Update sollte Trigger „test2“ reagieren, indem das folgende Statement dem Modell hinzugefügt wird:
(http://Vegetarianfood, http://Vegetarianfood#contains, http://food#cheese)

Dies kann nur durch eine RDQL-Anfrage festgestellt werden.

```
Select ?praedikat, ?objekt where (<http://
    Vegetarianfood>, ?praedikat, ?objekt)
```

Schließlich wird noch folgendes Statement eingefügt:

```
INSERT (http://testklasse#Scheinvegetarier, rdf:type,
    owl:Class)
```

→ Hiermit wird die Klasse Scheinvegetarier angelegt. Der Trigger „test3“ wird in diesem Fall die folgende SOAP-Nachricht an den in der Konfigurationsdatei festgelegten Eventbroker senden:

```
<neue_Klasse>
    <triggerserver>
        http://139.174.247.64:8080/
            triggerserver/server
    </triggerserver>
    <newclass>
        http://testklasse#Scheinvegetarier
    </newclass>
<neue_Klasse>
```

Die Variable $aequivalente_klasse gibt es in diesem Fall nicht.

Anhang B

Kommunikationsschnittstellen

Die Kommunikation zwischen Server und Client und auch die Kommunikation zwischen dem in dieser Arbeit entwickelten und anderen Servern erfolgt über SOAP. Über die gleiche Schnittstelle kann der Server auch von anderen Servern kontaktiert werden. Die möglichen Nachrichten-Formate und ihre Bedeutung werden in den Abschnitten 7.1 und 7.3 beschrieben.

Da die im Laufe dieser Arbeit entwickelten Programme im Rahmen eines größeren Projektes eingesetzt werden sollen (siehe Abschnitt 2.1), ist es möglich, dass die Kommunikationsschnittstellen geändert werden müssen. Aufgrund der Struktur der einzelnen Programmteile ist dies relativ leicht möglich. Die Kommunikation ist jeweils in eine eigene Klasse ausgelagert. Im Client ist dies die Klasse `SOAPMessages`. Die Methoden dieser Klasse senden jeweils eine SOAP-Nachricht mit einem bestimmten XML-Tag an den Server.
Im Server werden die SOAP-Nachrichten von der Klasse `SOAPServer` aus dem Paket *server* empfangen. Dabei wird in der Methode *onMessage()* der Inhalt der Nachricht extrahiert und an die Methode *parse()* weitergegeben. Diese ruft die Methoden für die Bearbeitung auf und liefert das Ergebnis als String zurück. Mit diesem Inhalt wird nun die Antwort-Nachricht zurück zum Sender geschickt.
Im Server gibt es noch eine zweite Kommunikationsschnittstelle. Von den Triggern können SOAP-Nachrichten an andere Server gesendet werden. Hierfür ist die Klasse `SOAPmessages` des Pakets *trigger* zuständig. Hier gibt es die Möglichkeit, eine Nachricht mit beliebigem Inhalt und dem festen XML-Markup „Trigger“ an eine beliebige URL zu senden. Außerdem kann eine Nachricht mit dem in Abschnitt 8.1 festgelegten Format an den in der Konfigurationsdatei festgelegten Eventbroker gesendet werden. Bei beiden Methoden wird die Nachricht als String übergeben.
Bei einer Umstellung des Kommunikationsprotokolls muss auch der Trigger-Server geändert werden, sofern er für den Betrieb noch nötig ist. Hier sorgt

die Klasse `Server` für den Empfang der SOAP-Nachrichten und das Senden einer Antwort.
Der Inhalt der Antwort-Nachrichten bei RDQL-Anfragen und Updates wird direkt bei der Bearbeitung durch die Methoden des `ModelServer`s erzeugt. Soll zum Beispiel die Ausgabe der Variablenbindungen einer RDQL-Anfrage geändert werden, müssen diese Änderungen in der Methode *request()* dieser Klasse geschehen.

Anhang C

Installation

Zur Installation der Programme wird eine Java-Laufzeit-Umgebung und eine PostgreSQL-Datenbank benötigt, in der das RDF-Modell und die Trigger abgespeichert werden. Für die Abspeicherung des Modells muss dabei eine leere Datenbank zur Verfügung gestellt werden, das Datenbankschema wird automatisch festgelegt. Für die Abspeicherung der Trigger sind zwei weitere Relationen „trigger“ und „triggeractions“ nötig. Das zugehörige Datenbankschema findet sich unter dem Namen `triggerdb` auf der beiliegenden CD. Die Trigger können auch in einer anderen Datenbank abgespeichert werden als das RDF-Modell.
Für die Speicherung eines Modells in einer Datenbank kann das ebenfalls in dieser Arbeit entwickelte Programm ModelLoader verwendet werden. Dieses Programm steht als jar-Datei zur Verfügung. Es kann mit dem Kommando
java -jar `modelloader.jar`
aufgerufen werden. Dabei wird eine RDF/XML-Datei eingelesen und das Modell in der Datenbank abgelegt.
Für die Installation des Servlets muss die Datei `triggerserver.war` in das „webapps“-Verzeichnis des Servlet-Containers kopiert werden. Wird hierfür der Servlet-Container Jakarta Tomcat verwendet, wird die Konfigurationsdatei `configuration.txt` ebenfalls in diesem Verzeichnis abgelegt. Andernfalls wird dafür das Home-Verzeichnis des Users gewählt, der den Servlet-Container startet.
Für den Server muss ein externer Reasoner wie Racer oder Pellet zur Verfügung gestellt werden. Die URL des Reasoners, wie auch die URL der Datenbanken und die entsprechenden Benutzernamen müssen in der Konfigurationsdatei angepasst werden.
Der Client kann mit dem Kommando
java -jar `client.jar`
gestartet werden. Bei Windows-Systemen reicht auch ein Doppelklick auf die Datei.

Anhang D

Inhalt der beiliegenden CD

Auf der CD sind alle für die Installation nötigen Dateien im Verzeichnis „Installation“ zu finden. Im einzelnen sind das:

- configuration.txt - die Konfigurationsdatei
- client.jar - der Client als jar-Datei
- modelloader.jar - das Programm ModelLoader
- triggerdb - das Datenbankschema zur Abspeicherung der Trigger
- triggerserver.war - Diese Datei enthält die Servlets für den Betrieb des Servers und des „Dummys“TriggerServer

Außerdem sind im Verzeichnis „Beispiele“ die in dieser Arbeit beschriebenen Beispiele gespeichert.

- testontologie.owl - die in Beispiel 3.7 erstellte Ontologie
- Beispielupdates.txt - die Updates und Trigger aus den Beispielen 7.2 und 8.2
- universitaet.owl - die Ontologie aus dem Beispiel aus Kapitel 8.4
- Konsistenzerhaltung.txt - die in diesem Beispiel verwendeten Updates und Trigger
- vegetarian.owl - die Ontologie aus Anhang A
- Triggerbeispiele.txt - die verwendeten Trigger und Updates dieses Beispiels

Im Verzeichnis „source“ liegen die Quellcodes aller im Laufe dieser Arbeit erstellten Programme. Beschreibungen der einzelnen Pakete und Klassen finden sich in den Abschnitten 6.3, 7.4, 8.5 und in Kapitel 9.

Alle in dieser Arbeit eingebundenen Bilder befinden sich in einem etwas größeren Format noch einmal im Verzeichnis „Bilder“.

Die Javadoc-Dokumente vom Client und vom Server inklusive des TriggerServers befinden sich im Verzeichnis „doc“.

Das Verzeichnis „lib“ enthält für die Programme verwendeten Bibliotheken.

Literaturverzeichnis

[i5-d1] José Júlio Alferes, James Bailey, Mikael Berndtsson, François Bry, Jens Dietrich, Alexander Kozlenkov, Wolfgang May, Paula-Lavinia Pătrânjan, Alexandre Pinto, Michael Schröder, and Gerd Wagner. State-of-the-art on evolution and reactivity. Technical Report Deliverable I5-D1, REWERSE EU FP6 NoE, 2004. Available at http://www.rewerse.net.

[i5-d2] José Júlio Alferes, Mikael Berndtsson, François Bry, Michael Eckert, Wolfgang May , Paula Lavinia Pătrânjan, and Michael Schröder. Use cases in evolution and reactivity. Technical Report Deliverable I5-D2, REWERSE EU FP6 NoE, 2005. Available at http://www.rewerse.net.

[i5-d4] José Júlio Alferes, Ricardo Amador, Erik Behrends, Mikael Berndtsson, François Bry, Gihan Dawelbait, Andreas Doms, Michael Eckert, Oliver Fritzen, Wolfgang May, Paula Lavinia Pătrânjan, Loic Royer, Franz Schenk, and Michael Schröder. Specification of a model, language and architecture for evolution and reactivity. Technical Report Deliverable I5-D4, REWERSE EU FP6 NoE, 2005. Available at http://www.rewerse.net.

Informationen über RDF:

[RDF1] http://www.xml.com/pub/a/98/06/rdf.html.

[RDF2] http://www.xml.com/pub/a/2001/01/24/rdf.html.

[RDF3] http://www.w3.org/tr/rdf-primer/.

[RDF4] http://www.w3.org/tr/rdf-syntax-grammar/.

Informationen über RDFS:

[RDFS1] http://www.w3.org/tr/rdf-schema/.

[RDFS2] http://www.computerbase.de/lexikon/rdf-schema.

Informationen über OWL:

[OWL1] http://www.w3.org/tr/owl-features/.

[OWL2] http://www.computerbase.de/lexikon/web_ontology_language/.

[OWL3] http://www.w3.org/tr/owl-ref/.

Weiter Internetquellen:

[wiki] *Das Internet-Lexikon Wikipedia*
http://www.wikipedia.de

[jena] *Die Dokumentation zum Jena-Framework*
http://jena.sourceforge.net/documentation.html.

[dig] http://dl.kr.org/dig/.

[ont1] http://www.schemaweb.info/.

[ontdef] http://www.gi-ev.de/service/informatiklexikon/informatiklexikon-detailansicht/meldung/57/.

[namesp] http://www.sql-und-xml.de/xml-lernen/namespace-xml-document.html.

[RDQL] http://www.w3.org/submission/2004/subm-rdql-20040109/.

[semweb] http://www.w3.org/2001/sw/.