Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer:	Dipl.-Inform. Christoph Mangold
begonnen am:	10. September 2001
beendet am:	08. März 2002
CR-Klassifikation:	H.2.3, H.2.1

Diplomarbeit Nr. 1963

Anfragesprache für ein semantisches Netz

Björn Stadler

Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Inhalt der Arbeit . . . 2

1.2 Gliederung . . . 2

2 Grundbegriffe **3**

2.1 Föderierte Datenbanksysteme . . . 3

2.1.1 Klassifizierung föderierter Datenbanksysteme . . . 4

2.1.2 Architektur föderierter Datenbanken . . . 6

2.2 Datenmodelle . . . 8

2.2.1 Strukturierte Datenmodelle . . . 8

2.2.2 Modelle für semistrukturierte Daten . . . 11

3 Anforderungen an die Sprache **19**

3.1 Allgemeine Anforderungen an eine Anfragesprache . . . 19

3.2 Anforderungen im Projekt Föderal . . . 21

4 Anfragesprachen **25**

4.1 OQL (Object Query Language) . . . 25

4.1.1 Bewertung . . . 27

4.2 Lorel (Lore Language) . . . 29

4.2.1 Pfadausdrücke . . . 29

4.2.2 Pfad - und Objektvariablen . . . 31

4.2.3 Manipulation von Daten . . . 32

4.2.4 Bewertung . . . 34

4.3 XPath (XML Path Language) . . . 35

4.3.1 Bewertung . . . 37

4.4 XQL (XML Query Language) . . . 40

4.4.1 Bewertung . . . 41

4.5 Quilt . . . 42

4.5.1 Anfragen der Form FLWR . . . 43

4.5.2 Filter . . . 44
4.5.3 Bewertung . . . 45
4.6 Zusammenfassung . . . 45

5 Entwurf einer Anfragesprache **47**
5.1 Pfadausdrücke . . . 47
5.2 Auswahlanfragen . . . 48
5.3 Änderungsanweisungen . . . 51
5.3.1 Einfügen von Knoten und Relationen . . . 51
5.3.2 Löschen von Knoten und Relationen . . . 54
5.3.3 Änderung des Werts von Knoten . . . 56
5.3.4 Erzeugen von Knoten- und Relationstypen . . . 56
5.4 Kopieren von Teilnetzen . . . 57
5.5 Unterschiede von SNQL gegenüber Lorel . . . 58

6 Implementierung **61**
6.1 Realisierungsalternativen . . . 61
6.2 Erzeugung eines Parsers . . . 62
6.3 Implementierung der Sprachelemente . . . 64
6.3.1 Pfadausdrücke . . . 64
6.3.2 Auswahlanfragen . . . 67
6.3.3 Änderungsanweisungen . . . 69
6.3.4 Kopieren von Teilnetzen . . . 71
6.4 Test des Interpreters . . . 72

7 Zusammenfassung **79**

A Abkürzungsverzeichnis **81**

B EBNF der Anfragesprache **83**

C Konfigurationsdatei für die Erzeugung des Parsers **85**

Literaturverzeichnis **91**

Abbildungsverzeichnis

2.1 Grobaufbau eines föderierten Datenbanksystems [Con97] 4
2.2 3-Ebenen-Architektur nach ANSI/X3/SPARC 6
2.3 5-Ebenen-Architektur föderierter Datenbanksysteme [SL90] 7
2.4 Begriffe im relationalen Datenmodell an einem Beispiel 9
2.5 Begriffe im objektorientierten Datenmodell anhand eines Beispiels 10
2.6 Semistrukturierte und strukturierte Daten 11
2.7 Einordnung semistrukturierter Daten 12
2.8 Beispiel eines XML Dokumentes 13
2.9 OEM Datenmodell 15
2.10 Datenmodell des Projektes Föderal 16
2.11 Schichtenarchitektur im Föderalprojekt 18

3.1 Pfade im semantischen Netz 22
3.2 Löschen eines Knotens im semantischen Netz 23
3.3 Kopieren von Teilnetzen 23

4.1 OQL-Schema eines Beispiels 27
4.2 Verhalten von OQL beim Kopieren von Objekten 28
4.3 Beispiel eines OEM Graphen [AQM+97] 30
4.4 Ausschnitt aus einem OEM Graphen 32
4.5 Beispiel eines komplexen OEM - Objektes 33
4.6 Ausschnitt aus einem OEM - Graphen 34
4.7 Achsen bei XPath [BM00] 38

5.1 Beispiel eines semantischen Netzes 50
5.2 Einfügen einer Relation in ein semantisches Netz 52
5.3 Einfügen eines Knotens und zweier Relationen 53
5.4 Einfügen einer Menge von Relationen 54
5.5 Einfügen einer Relationen auf einer Relation 55
5.6 Löschen einer einzelnen Relation 56
5.7 Kopieren eines Teilnetzes 58

6.1 Realisierungsalternativen . . . 62
6.2 Arbeitsweise eines Parsers . . . 63
6.3 Hierarchie des Interfaces `IStatement` . . . 64
6.4 Aufbau der Pfadtabelle an einem Beispiel . . . 66
6.5 Klassenschema für Pfadausdrücke . . . 67
6.6 Schematischer Ablauf der Auswertung einer Auswahlanfrage . . . 68
6.7 Semantisches Netz . . . 69
6.8 Klassenschema der Auswahlanfrage . . . 70
6.9 Semantisches Netz zu Testzwecken . . . 73
6.10 Semantisches Netz nach einer Kopieroperation . . . 76
6.11 Semantisches Netz nach Entfernen von zwei Relationen vom Typ `rel1` . . . 76
6.12 Semantisches Netz nach Entfernen des Knotens `k4` . . . 77

Tabellenverzeichnis

4.1 Achsen bei XPath [BM00] 37
4.2 Beispiele für XPath - Ausdrücke 39
4.3 Beispiele für Ausdrücke in XQL 41
4.4 Pfadoperatoren in Quilt 42
4.5 Tabellarische Zusammenfassung 46

5.1 Syntax von Pfadausdrücken 48
5.2 Syntax von Auswahlanfragen 49
5.3 Syntax von Änderungsanweisungen 51

Kapitel 1

Einleitung

Die Produktentwicklung in der Industrie ist heute von großem Zeitdruck geprägt. Der weltweite Wettbewerb verlangt von den Unternehmen nicht nur eine stetige Verkürzung der Entwicklungszeit, sondern oft auch eine drastische Reduzierung der Entwicklungskosten. Daher entwickeln viele Firmen ihre Produkte nach einem Baukastenprinzip, bei dem vorhandene Komponenten wiederverwendet werden. Um die Wiederverwendung zu ermöglichen, wird die Verwaltung von Produktdaten zunehmend ein wichtiger Bestandteil der betrieblichen Informationsverarbeitung.
Komponenten einzelner Produkte werden im allgemeinen von unterschiedlichen Abteilungen unter Verwendung spezialisierter Werkzeuge, wie CAD- oder Dokumentationssystemen, entwickelt. So sind relevante Daten über viele Systeme verteilt und oft redundant vorhanden. Daraus resultiert ein hoher Anteil an manueller Datenintegration bei der Wiederverwendung, was wiederum hohe Kosten verursacht. Die Verteilung der Daten hat zudem zur Folge, dass Informationen über die Verbindung der Daten nicht modelliert werden und nur in den Köpfen der einzelnen Mitarbeiter vorhanden sind. Diese Probleme sind schwerwiegend und mit bestehenden Systemen nicht zu lösen.
Die geschilderte Situation trifft insbesondere auf Unternehmen des Anlagen- und Maschinenbaus zu. Daher haben sich Firmen dieser Branche mit einem IT-Dienstleister und zwei Forschungseinrichtungen zum Projekt Föderal zusammengeschlossen. Das Projekt hat sich zum Zeil gesetzt, eine föderale Informationsarchitektur für Ingenieursanwendungen zu entwerfen und umzusetzen. Dabei liegt der Schwerpunkt der Projekts auf der Modellierung der Verbindungsinformationen und der Intergration vorhandener Entwicklungswerkzeuge. Bei der Architektur handelt es sich technisch um ein föderiertes System, bei dem die Verbindungsinformationen als Netz modelliert sind und die Einzelsysteme miteinander verknüpfen.

1.1 Inhalt der Arbeit

Im Projekt Föderal soll eine föderale Datenbankarchitektur erstellt werden, bei der das globale Schema durch ein semantisches Netz gebildet wird. Im Rahmen dieser Arbeit wird eine Anfragesprache für das semantische Netz im Projekt Föderal entwickelt.
Dazu werden zuerst die Anforderungen des Projekts an eine solche Anfragesprache erhoben und formalisiert. In einem zweiten Schritt werden bereits definierte Anfragesprachen an strukturierte und semistrukturierte Datenmodelle auf diese Anforderungen hin untersucht. Auf diesem Ergebnis aufbauend wird eine an das semantische Netz angepasste Sprache entwickelt. Anschließend wird unter Zuhilfenahme von Generatoren ein Interpreter implementiert, der Ausdrücke der Sprache auf die prozedurale Schnittstelle des Semantic Net Interfaces (*SNI*) übersetzt.

1.2 Gliederung

Nach diesem Einleitungskapitel folgt eine Einführung in die für diese Arbeit wichtigen Bereiche der föderierten Systeme. Dabei wird nicht nur die Architektur föderaler Systeme vorgestellt, sondern auch eine Klassifikation vorgenommen. Den Abschluss des Kapitels zwei bildet eine Einführung in strukturierte und semistrukturierte Datenmodelle. Unter anderem wird hier das Datenmodell für das semantische Netz in der Föderalen Informationsarchitektur vorgestellt.
In Kapitel drei werden die Anforderungen an eine Anfragesprache für das globale Schema der Föderalen Informationsarchitektur beschrieben. Das Kapitel gliedert sich in zwei Teile. Während im ersten Teil auf allgemeine Anforderungen eingegangen wird, beschreibt der zweite Teil die spezifischen Anforderungen aus dem Projekt Föderal.
Kapitel vier untersucht verschiedene Anfragesprachen, wie OQL und Quilt in Hinblick auf die festgestellten Anforderungen. Dabei werden Sprachen für unterschiedliche Datenmodelle betrachtet. Eine Zusammenfassung der Ergebnisse bildet den Abschluss des Kapitels.
Darauf folgt in Kapitel fünf der Entwurf einer Anfragesprache für das semantische Netz. Die Syntax und Funktionalität der Sprache wird in diesem Kapitel anhand von Beispielen erläutert.
Der im Rahmen dieser Arbeit implementierte Interpreter für die neu entworfene Anfragesprache wird in Kapitel sechs vorgestellt. Die erste Hälfte des Kapitel erläutert die Implementierung des Interpreters, während der Rest des Kapitels eine Auswertung des Interpreters enthält.
Den Abschluss dieser Arbeit bildet eine Zusammenfassung.

Kapitel 2

Grundbegriffe

In diesem Kapitel soll zu Anfang eine Einführung in den für diese Diplomarbeit wichtigen Bereich föderierter Systeme gegeben werden. Dazu gehören die Definition eines föderierten Systems, die Darstellung der Architektur föderierter Systeme, sowie eine Klassifikation.
Im zweiten Teil des Kapitels werden verschiedene Datenmodelle vorgestellt, wobei die wesentlichen Unterscheidungsmerkmale strukturierter und semistrukturierter Modellen diskutiert werden.

2.1 Föderierte Datenbanksysteme

Der Begriff der föderierten Systeme in der Informatik stammt eigentlich aus dem politischen Bereich. Dort bezeichnet eine Föderation einen vertraglichen Zusammenschluss von teilweise autonomen Staaten oder Teilstaaten. Beispiele solcher Föderationen sind die Bundesrepublik Deutschland wie auch die Europäische Union.
Analog dazu wird in der Informatik ein Datenbanksystem, das aus mehreren autonomen und heterogenen Einzel- oder Komponentendatenbanken besteht, als föderiertes Datenbanksystem bezeichnet, sofern die einzelnen Komponenten eine gewisse Autonomie besitzen.
Der Begriff Autonomie lässt sich nach [BKLW99, Con97] weiter in Entwurfsautonomie (*design autonomy*), Kommunikationsautonomie (*communication autonomy*) und Ausführungsautonomie (*execution autonomy*) unterscheiden.
Entwurfsautonomie ist gegeben, wenn die Komponentendatenbanken unabhängig voneinander entworfen wurden und erst anschließend in die Föderation eingebracht werden.
Kommunikationsautonomie bezeichnet die Möglichkeit, selbständig zu entscheiden, mit welchen anderen Systemen kommuniziert werden soll. Bei föderierten Systemen bedeutet dies, dass die Komponenten selbst entscheiden, ob und wann sie einem föderierten System beitreten, beziehungsweise dieses wieder verlassen. Selbständige Entscheidung durch das System bedeutet hier, dass ein Datenbankadministrator diese Entscheidung für das System trifft.

Falls ein System selbst entscheidet, welche Anwendungsprogramme, Anfragen und Änderungsoperationen durchgeführt werden, sowie die Reihenfolge der Operationen festlegt, spricht man von Ausführungsautonomie.
Im Allgemeinen wird ein System, bei dem mehrere unabhängige oder autonome Datenbanken durch einen Föderierungsdienst miteinander gekoppelt werden, als föderiertes oder auch föderatives [Rah94] Datenbanksystem bezeichnet. Der Grobaufbau eines solchen Systems ist in Abbildung 2.1 dargestellt. In der Abbildung sind n unabhängige Datenbanken durch einen

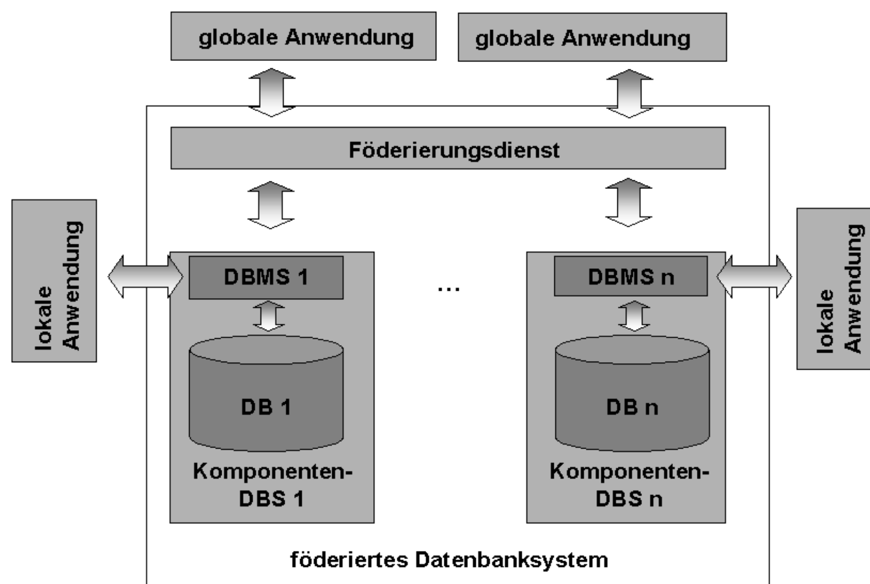


Abbildung 2.1: Grobaufbau eines föderierten Datenbanksystems [Con97]

Föderierungsdienst miteinander verbunden. Der Föderierungsdienst stellt dabei eine Schnittstelle zur Verfügung, über die die Daten der einzelnen Komponentendatenbanken abgefragt werden können. Bereits vorhandene lokale Anwendungen, das heißt, Anwendungen, die nur die Daten einer Komponente benötigen, können unverändert übernommen werden. Die Funktion des Föderierungsdienstes wird im folgenden Abschnitt weiter verfeinert.

2.1.1 Klassifizierung föderierter Datenbanksysteme

Föderierte Datenbanksysteme lassen sich nach [BKLW99] nach folgenden Kriterien klassifizieren.

Kopplung

Man unterscheidet zwischen *enger Föderation* und *loser Föderation*. Bei der engen Föderation existiert ein einheitliches föderiertes Schema für den Zugriff auf die Föderation. Eine lose

Föderation dagegen besitzt kein einheitliches Schema. Dadurch sind die Anwendungen selbst für die Art der Datenintegration verantwortlich.

Datenmodell

Für die Integration der Komponentendatenbanken liegt dem Föderierungsdienst ein Datenmodell zugrunde. Dabei lassen sich strukturierte und semistrukturierte Modelle unterscheiden. Als strukturierte Modelle werden heute im Allgemeinen relationale ([Dat95]) oder objektorientierte ([Cat94a]) Modelle verwendet.

Transparenz

Föderierte Systeme versuchen, vollständige Transparenz zu erreichen, das bedeutet, alle Aspekte, die mit der Föderation unabhängiger Datenquellen zusammenhängen, sollen dem Benutzer verborgen bleiben. So soll transparent sein, in welcher Komponente die Daten liegen (*Ortstransparenz, location transparency*), welches Datenmodell dem System zugrunde liegt (*Schematransparenz, schema transparency*), sowie auf welche Art auf die Daten zugegriffen werden kann (*Sprachtransparenz, language transparency*).

Semantische Integration

Bei der Integration von Daten in ein föderiertes System lassen sich mehrere verschiedene Möglichkeiten unterscheiden. Eine Möglichkeit besteht darin, die Daten der Komponentendatenbanken ohne Abgleich äquivalenter Objekte aus unterschiedlichen Quellen oder Fehlerkorrektur zu übernehmen. Dieser Vorgang wird *Sammeln (collection)* genannt.
Dagegen wird beim *Verschmelzen (fusion)* der Versuch unternommen, äquivalente Objekte zu identifizieren und eine konsistente Repräsentation der Objekte zu erreichen.
Bei der *Abstraktion (abstraction)* werden die Daten in der Komponenten dem Abstraktionsniveau des föderierten Schemas angepasst.
Falls die Daten im föderierten Schema durch zusätzliche Informationen erweitert werden, spricht man von *Ergänzung (supplementation).*

Systemaufbau

Im Allgemeinen unterscheidet man die *top-down* und *bottom-up* Vorgehensweise. Bei der top-down Technik geht man bei der Schemaintegration der Komponenten vom bereits modellierten föderierten Schema aus, wohingegen bei bottom-up die Schemata der Komponenten die Grundlage der Schemaintegration bilden.

Zugriff

Es existieren Systeme, die über den Föderierungsdienst nur lesenden Zugriff auf die Komponenten erlauben und damit das Problem, Änderungen vom Föderierungsdienst aus an die betroffenen Komponenten weiterzugeben, umgehen, sowie Systeme, die sowohl lesenden als auch schreibenden Zugriff zulassen.

Zugriffsmethoden

Es gibt verschiedenen Möglichkeiten, auf die Daten in Datenbanken zuzugreifen. Erstens ist es möglich, den Zugriff über eine *Anfragesprache*, wie SQL oder OQL zu regeln, zweitens kann der Zugriff über eine *prozedurale Schnittstelle* ermöglicht werden, bei der die Anfragen bis auf Parameter festgelegt sind. Außerdem besteht die Möglichkeit durch *Browsen* auf die Daten zuzugreifen.

Einordnung der Föderal-Architektur

Bei der Föderalen Informationsarchitektur (*FIA*) im Projekt Föderal handelt es sich um ein eng gekoppeltes föderiertes System. Das föderale Schema ist als semistrukturiertes Datenmodell in Form des semantischen Netzes modelliert. Dabei besteht für Anwendungen, die über das semantische Netz auf die Daten der Föderation zugreifen volle Transparenz, das bedeutet, es ist den Anwendungen nicht bekannt, wo die Daten gespeichert sind, in welcher Art von Datenmodell sie modelliert sind und wie auf die sie zugegriffen wird. Die FIA erlaubt sowohl schreibenden als auch lesenden Zugriff auf die Daten. Der Zugriff wird über eine prozedurale Schnittstelle durchgeführt, die Anfragesprache für das semantische Netz ist Gegenstand dieser Arbeit. Bei der Integration der Teilsysteme werden die Daten durch Informationen über die semantischen Zusammenhänge ergänzt, das heisst, die semantische Integration erfolgt über Ergänzung.

2.1.2 Architektur föderierter Datenbanken

Bevor hier die allgemein anerkannte Architekur föderierter Datenbanksysteme vorgestellt wird, soll im Folgenden zuerst kurz auf die Architektur zentralisierter Datenbanksysteme eingegangen werden.

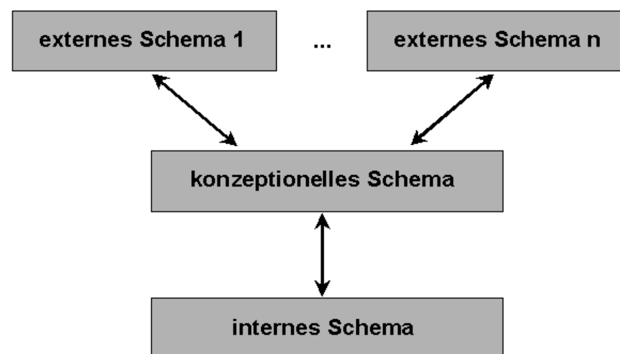


Abbildung 2.2: 3-Ebenen-Architektur nach ANSI/X3/SPARC

In Abbildung 2.2 ist die standardisierte Architektur zentralisierter Datenbanksysteme dargestellt. Sie unterscheidet drei Arten von Schemata. Die *externen Schemata* stellen die verschiedenen Sichten der Anwendungsprogramme und Benutzer auf die Datenbank dar. Die Gesamtmenge der vom Datenbanksystem verwalteten Daten ist im *konzeptionellen Schema* systemunabhängig und damit abstrakt beschrieben. Die interne, systemeigene Darstellung des konzeptionellen Schemas schließlich ist im *internen Schema* enthalten. Das interne Schema beschreibt, wie die Daten physisch gespeichert werden [HR99].
Die in Abbildung 2.3 dargestellte Architektur föderierter Datenbanksysteme wurde in [SL90] vorgestellt.

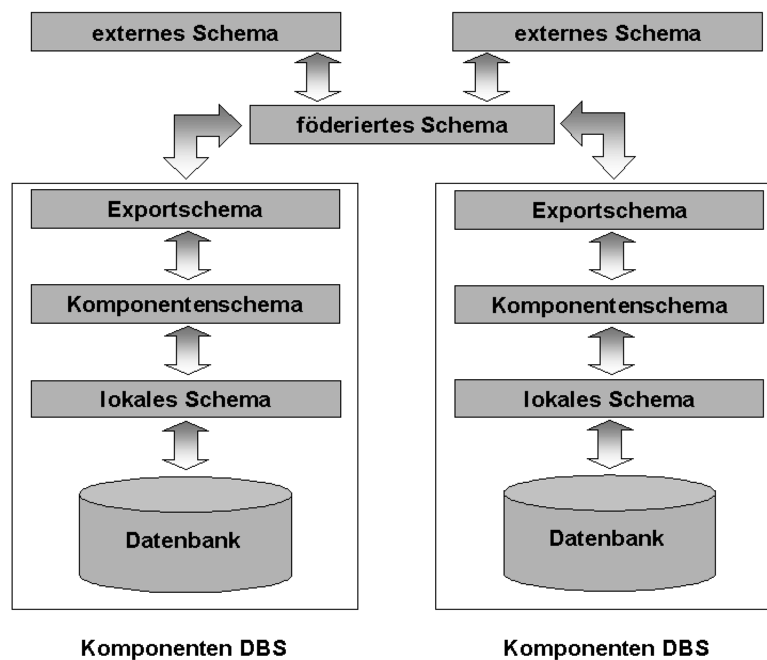


Abbildung 2.3: 5-Ebenen-Architektur föderierter Datenbanksysteme [SL90]

Das *lokale Schema* der Architektur entspricht einem konzeptionellen Schema aus der 3-Ebenen-Architektur (siehe Abbildung 2.2) für eine Komponentendatenbank des föderierten Systems. Hier werden die Daten der Komponente implementierungsunabhängig beschrieben.
Da die Komponenten im Allgemeinen auf unterschiedlichen Datenmodellen basieren, werden die lokalen Schemata in ein *Komponentenschema* überführt. Das Komponentenschema enthält dieselben Informationen wie das lokale Schema, ist aber in einem für alle Komponenten einheitlichen Datenmodell beschrieben.
Das *Exportschema* beschreibt den Ausschnitt des Komponentenschemas, der in die Föderation eingebracht werden soll. Das Exportschema ist im selben Modell beschrieben wie das Komponentenschema und das föderierte Schema, um weitere Schematransformationen zu vermeiden.
Die Gesamtheit der in die Föderation einfließenden Daten werden im *föderierten Schema*, auch

globales Schema genannt, beschrieben. Es stellt die Vereinigung der Exportschemata dar. Bei der Integration können diverse Probleme entstehen, siehe dazu zum Beispiel [Con97].
Die *externen Schemata* schließlich beschreiben Sichten verschiedener Anwendungen auf das föderierte Schema. Die Funktion der externen Schemata in der föderierten Architektur entspricht der externer Schemata im zentralisierten Fall.
In [Con97] werden weitere Referenzarchitekturen beschrieben, allerdings erlaubt die 5-Ebenen-Architektur als einzige der dort beschriebenen Architekturen volle Datenbankfunktionalität durch die Verwendung eines globalen föderierten Schemas. Dies schließt Transaktionsverwaltung auf globaler Ebene ein.

2.2 Datenmodelle

Im Rahmen dieser Arbeit soll eine Anfragesprache für ein Datenmodell im globalen Schema einer föderierten Datenbankarchitektur entwickelt werden. Da Anfragesprachen für ein bestimmtes Datenmodell konzipiert werden, sollen in diesem Abschnitt verbreitete Datenmodelle vorgestellt werden. Man unterscheidet Modelle für strukturierte und semistrukturierte Daten. In der vorgegebenen Architektur wird ein proprietäres Datenmodell eingesetzt, welches sowohl Merkmale strukturierter als auch semistrukturierter Datenmodelle aufweist.
In [Cod80] wird eine Definition für den Begriff *Datenmodell* gegeben. Danach gehört zu einem Datenmodell eine Sammlung von Datenstrukturen, eine Sammlung von Operatoren oder Regeln, mit deren Hilfe die Daten oder Teile davon in beliebiger Kombination bearbeitet werden können, sowie allgemeine Integritätsregeln.

2.2.1 Strukturierte Datenmodelle

Modelle für strukturierte Daten sind durch ein Schema, das den Daten zugrunde liegt, gekennzeichnet. Dieses Schema wird im Voraus definiert. Die Folge dieser Vorgehensweise ist, dass alle Daten, die durch das definierte Modelle repräsentiert werden sollen, mit dem Schema konform sein müssen. Die Daten und das Schema müssen zueinander passen. Änderungen am Schema sind später, wenn überhaupt, nur mit großem Aufwand möglich. Die Schemainformation und die Daten werden in strukturierten Modellen grundsätzlich logisch von einander getrennt.
Im Folgenden werden die bekanntesten Vertreter strukturierter Datenmodelle, das relationale und das objektorientierte Modell, vorgestellt.

Das relationale Datenmodell

Das relationale Datenmodell besteht aus drei Teilen, Datenstruktur, Integrität und Operatoren [Dat95]. Die Datenstruktur des Modells basiert auf *Relationen*, auch Tabellen genannt, das heißt alle Daten werden als Relationen modelliert. Eine Relation besteht aus *Tupeln* und

Attributen. Mit Tupel wird eine Zeile der Tabelle, mit Attribut eine Spalte der Tabelle bezeichnet. Die Attribute sind von einem atomaren Datentyp, zum Beispiel `Integer` oder `String`.
Die Integrität der Daten wird durch das Konzept von *Primär-* und *Fremdschlüsseln* erreicht. Mit Primärschlüssel wird ein Attribut oder eine Kombination von Attributen bezeichnet, durch die jedes Tupel einer Relation eindeutig identifiziert werden kann. Mit anderen Worten ausgedrückt bedeutet dies, dass ein Wert eines Primärschlüssels in einer Relation höchstens einmal vorkommen darf.
Mit Hilfe von Fremdschlüsseln werden Beziehungen zwischen mehreren Relationen ausgedrückt. Ein Attribut oder eine Kombination von Attributen wird als Fremdschlüssel bezeichnet, sofern sie einen Verweis auf den Primärschlüssel einer anderen Relation darstellt. Dabei muss sichergestellt werden, dass zu jedem Fremdschlüsselwert ein Primärschlüsselwert in der referenzierten Relation existiert. Diese Eigenschaft wird referenzielle Integrität genannt. Dies wird in relationalen Datenbanksystemen durch das System selbst gewährleistet.

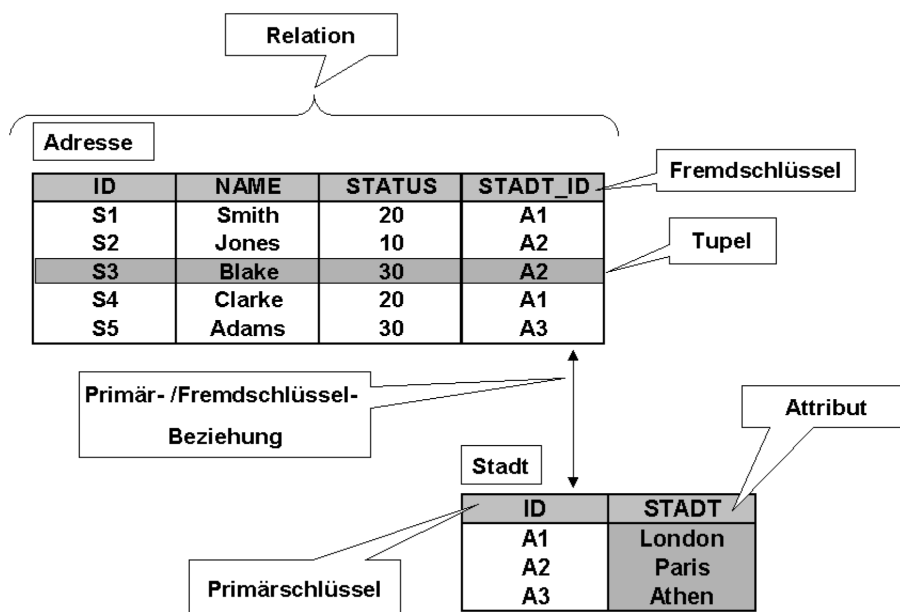


Abbildung 2.4: Begriffe im relationalen Datenmodell an einem Beispiel

In Zusammenhang mit der Integrität ist es zudem wichtig, dass jedes Attribut einen Definitionsbereich (*domain*) besitzt. So ist bekannt, welche Werte für ein beliebiges Attribut erlaubt sind. Ungültige Werte können so auf einfache Art zurückgewiesen werden.
Die Begriffe des relationalen Datenmodells sind in Abbildung 2.4 an einem Beispiel zusammengestellt. Die Abbildung zeigt zwei Tabellen `Adresse` und `Stadt`, die über einen Fremdschlüssel `STADT_ID` verknüpft sind.
Die Operatoren des relationalen Modells stammen aus der Relationenalgebra beziehungsweise aus dem Relationenkalkül. Alle Operatoren arbeiten auf Relationen und liefern wiederum Relationen als Ergebnis. Basisoperationen sind die Restriktion (*restriction*), die Projektion (*pro-*

jection), Produktbildung (*product*), Mengendifferenz (*difference*) und die Vereinigung (*union*) [Dat95].

Das objektorientierte Datenmodell

Die objektorientierte Modellierung basiert auf der Sichtweise, dass die modellierte Realität aus einer Menge von Objekten besteht. Die Objekte besitzen einen Zustand und haben ein objekttypisches Verhalten. Das Verhalten eines Objekts wird in den *Methoden* des Objekts modelliert, über die der Zustand des Objekts abgefragt und verändert werden kann.
Objekte gleichen Typs werden in *Klassen* zusammengefasst. Die Klasse stellt dabei eine abstrakte Objektbeschreibung dar. Ein wichtiges und mächtiges Konzept der objektorientierten Modellierung ist die *Vererbung*. Darunter versteht man die Möglichkeit, Eigenschaften einer Klasse an weitere Klassen weiterzugeben. Dies bietet den Vorteil, dass gemeinsame Eigenschaften nur einmal modelliert werden müssen. Durch die Vererbung zwischen Klassen entsteht eine Klassenhierarchie. Erbende Klassen können durch neue Eigenschaften erweitert werden.
Konkrete Objekte werden durch *Instanziierung* von einer Klassen abgeleitet. Instanziierung bedeutet, dass die abstrakte Beschreibung der Klasse mit konkreten Daten gefüllt wird [BW95, Cat94a, Cat94b].

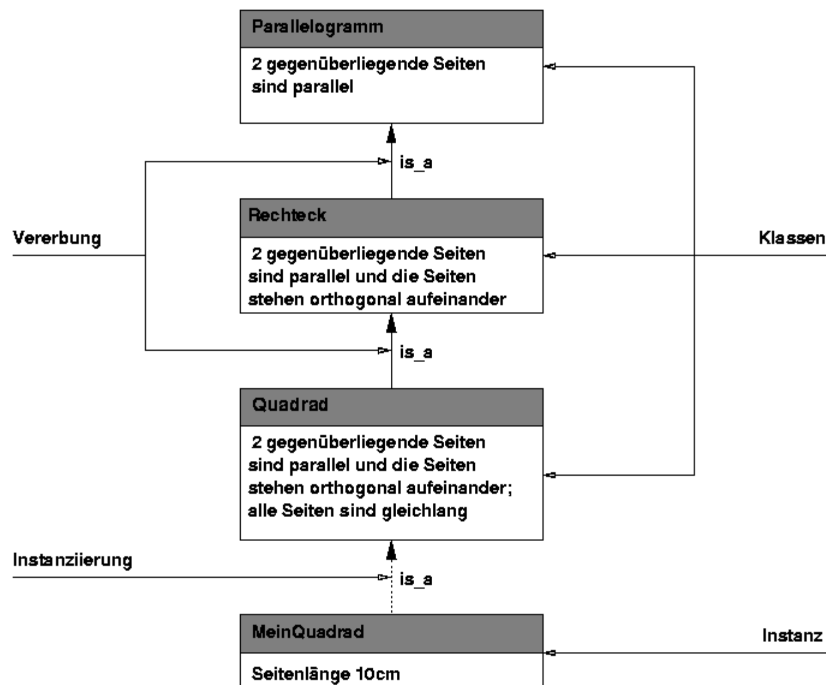


Abbildung 2.5: Begriffe im objektorientierten Datenmodell anhand eines Beispiels

In Abbildung 2.5 sind die Konzepte des objektorientierten Datenmodells an einem Beispiel zusammengestellt. Die Abbildung zeigt eine Klassenhierarchie aus `Parallelogramm`, `Rechteck` und `Quadrat`, sowie eine Instanz `MeinQuadrat` der Klasse `Quadrat`.

2.2.2 Modelle für semistrukturierte Daten

Der Begriff semistrukturierte Daten lässt sich durch Abgrenzung gegenüber strukturierten Daten definieren. Traditionelle Datenbanksysteme erfordern ein im Voraus definiertes einheitliches Schema, nach dem die Daten strukturiert sind. Jeder Datensatz erfüllt dieses Schema ohne Ausnahme. Solche Daten werden als strukturierte Daten bezeichnet.
Semistrukturierte Daten dagegen sind im Allgemeinen durch potenziell unregelmäßige Strukturen gekennzeichnet, das bedeutet, nicht alle Datensätze enthalten dieselben Komponenten oder einzelne Komponenten kommen wiederholt vor. Ein weiteres Kennzeichen ist, dass sich die Struktur semistrukturierter Daten schnell und häufig ändern kann. Semistrukturiert bedeutet jedoch nicht, dass die Daten keinerlei Struktur enthalten, obwohl semistrukturierte Daten auch als unstrukturierte Daten bezeichnet wurden [AQM+97, BKOS01, Bun97, Suc98].

```
<Datenbank>
    <Adresse>
        <Name>Mustermann</Name>
        <Email>Mustermann@Musternet.net</Email>
    </Adresse>
    <Adresse>
        <Name>Ottonormalverbraucher</Name>
        <Email>Ottonormalverbraucher@Musternet.net</Email>
        <URL>http://www.musternet.net/ottonormalverbraucher</URL>
    </Adresse>
    ...
</Datenbank>
```

(a) Semistrukturierte Daten in einem XML-Dokument

```
<Datenbank>
    <Adresse>
        <Name>Mustermann</Name>
        <Email>Mustermann@Musternet.net</Email>
        <URL></URL>
    </Adresse>
    <Adresse>
        <Name>Ottonormalverbraucher</Name>
        <Email>Ottonormalverbraucher@Musternet.net</Email>
        <URL>http://www.musternet.net/ottonormalverbraucher</URL>
    </Adresse>
    ...
</Datenbank>
```

(b) Darstellung strukturierter Daten durch XML

Abbildung 2.6: Semistrukturierte und strukturierte Daten

Der Unterschied zwischen strukturierten und semistrukturierten Daten soll im Folgenden am Beispiel eines in XML beschriebenen Dokumentes verdeutlicht werden.
Abbildung 2.6(a) zeigt einen Ausschnitt aus einem XML-Dokument. Die in dem Dokument enthaltenen Adressen haben kein einheitliches Schema. Die erste Adresse besteht aus zwei Komponenten `Name` und `Email`, wohingegen die folgende Adresse zusätzlich zu diesen eine

Komponente URL enthält. Im Gegensatz dazu zeigt die Abbildung 2.6(b) dieselben Daten als strukturierte Daten. Hier ist in der ersten Adresse ein leeres Element URL notwendig, um einem einheitlichen Schema zu genügen, das besagt, dass Adressen drei Elemente enthalten.
Zusammenfassend lässt sich feststellen, dass semistrukturierte Daten eine irreguläre Struktur besitzen. Mit irregulärer Struktur ist hier gemeint, dass manche Elemente unvollständig, andere dagegen zusätzliche Information enthalten (siehe Abbildung 2.6(a)). Die Struktur der Daten ist bei semistrukturierten Daten oft implizit enthalten, das bedeutet die Strukturinformation und die Daten sind nicht getrennt, sondern eine Aufbereitung der Daten, zum Beispiel durch *parsing* der Daten, ist notwendig, um die Struktur zu extrahieren [Abi97]. Da die Struktur in den Daten mit enthalten ist, spricht man auch von *selbsterklärenden* Daten [BKOS01].

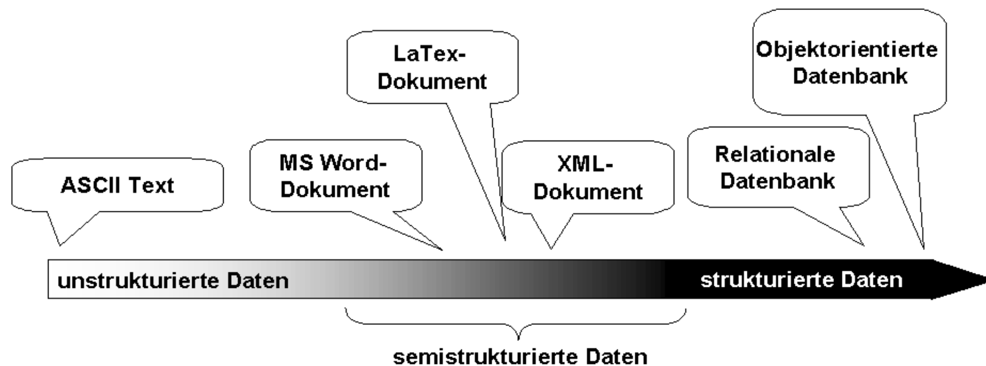


Abbildung 2.7: Einordnung semistrukturierter Daten. Es gibt Daten, die sicher unstrukturiert, wie reine ASCII - Texte, oder sicher strukturiert sind, wie relationale oder objektorientierte Datenbanken. Daten, die sich irgendwo dazwischen einordnen lassen, wie XML-Dokumente, werden als semistrukturiert bezeichnet. Die Übergänge zwischen den Begriffen sind fließend.

Man kann zudem feststellen, dass die Übergänge zwischen unstrukturierten, semistrukturierten und strukturierten Daten fließend sind. Dies soll durch Abbildung 2.7 verdeutlicht werden.

XML (Extensible Markup Language)

Die *Extensible Markup Language (XML)* [BPSM00] wurde vom *World Wide Web Consortium (W3C)* 1996 als Standard für die Verarbeitung von Dokumenten vorgeschlagen. Bei XML handelt es sich um eine Metasprache, mit der Auszeichnungssprachen für verschiedenste Anwendungen definiert werden können. Ein Beispiel ist der Austausch von Katalogdaten. Dies lässt sich im Allgemeinen nicht durch eine einzige Sprache bewerkstelligen, da von Branche zu Branche unterschiedliche Sprachkonstrukte notwendig sind. Reiseveranstalter beispielsweise benötigten andere Konstrukte als Maschinenbauer.
XML basiert auf der *Standard Generalized Markup Language (SGML)*, die 1986 standardisiert wurde. Das Ziel dieser Entwicklung war, eine einfach anwendbare Sprache mit semantischem

Markup, im Gegensatz zum logischen Markup der *Hypertext Markup Language (HTML)* zu definieren, die die unterschiedlichen Anforderungen des Webs, wie Heterogenität oder Erweiterbarkeit, unterstützt. Daneben sollten die mit XML erstellten Dokumente für den Menschen lesbar und schnell zu erstellen sein [BPSM00, Far99].
Ein XML Dokument besteht im Allgemeinen aus zwei Dateien, von denen die erste, die *Document Type Definition (DTD)* die Definition oder Grammatik der Sprache enthält, die zweite Datei enthält dagegen das eigentliche XML Dokument. In der DTD werden die Elemente der Auszeichnungssprache definiert, die später zur Auszeichnung von Daten verwendet werden können. Die Elemente können entweder Terminale der Sprache, in XML „Tags" genannt, definieren oder die Produktionsregeln der Grammatik beschreiben. Eine DTD kann für viele XML-Dokumente verwendet werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT ADRESSBUCH ANSCHRIFT*>
<!ELEMENT ANSCHRIFT (NAME, (STRASSE | POSTFACH)?, ORT)>
<!ELEMENT NAME ANY>
<!ELEMENT STRASSE ANY>
<!ELEMENT POSTFACH ANY>
<!ELEMENT ORT EMPTY>
<!ATTLIST ORT
  PLZ CDATA #REQUIRED
  NAME CDATA #REQUIRED
  LAND CDATA #IMPLIED>
```

(a) Beispiel einer Document Type Definition (anschrift.dtd) [Tol99]

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ADRESSBUCH SYSTEM="anschrift.dtd">
<ADRESSBUCH>
  <ANSCHRIFT>
    <NAME>Robert Tolksdorf</NAME>
    <STRASSE>Franklinstr. 28/29</STRASSE>
    <ORT PLZ="10587" NAME="Berlin"/>
  </ANSCHRIFT>
</ADRESSBUCH>
```

(b) Beispiel eines XML Dokumentes [Tol99]

Abbildung 2.8: Beispiel eines XML Dokumentes

Abbildung 2.8(a) zeigt ein Beispiel für eine DTD, die eine Sprache für die Auszeichnung von Adressen definiert. Ein Adressbuch wird mit `<ADRESSBUCH>` und `</ADRESSBUCH>` geklammert. Es kann eine beliebige Anzahl von Anschriften enthalten. Jede Anschrift ihrerseits ist durch `<ANSCHRIFT>` und `</ANSCHRIFT>` gekennzeichnet und muss einen Namen `<NAME>`, dem ein Element `<STRASSE>` oder `<POSTFACH>` und ein Ort `<ORT>` folgen, enthalten. Das Element Ort hat drei Attribute, `PLZ`, `NAME` und als optionales Attribut `LAND`.
Mit `ANY` wird gekennzeichnet, dass zwischen dem Anfangs- und dem Endtag jede beliebige Zeichenkette, auch eine weitere Auszeichnung, stehen kann. Falls leere Elemente definiert werden, muss dies durch `EMPTY`, wie im Beispiel beim Element `ORT`, markiert werden. Daneben kann man durch `(#PCDATA)` festlegen, dass zwischen dem Anfangs- und dem Endtag eines Elemen-

tes nur allgemeine Zeichenketten und keine weiteren Tags stehen dürfen (PCDATA steht für *parsed Character Data*) [Eck00].
Abbildung 2.8(b) zeigt ein Beispieldokument, das die Definitionen der DTD aus Abbildung 2.8(a) benutzt. Das Element `<!DOCTYPE ...>` gibt an, dass das Dokument die Syntax der DTD anschrift.dtd verwendet, danach folgt erst der eigentliche Inhalt des Dokuments.
XML spielt aufgrund seiner datenorientierten Markierungsmöglichkeit auch eine Rolle im Bereich der Datenbanken, zum Beispiel dann, wenn aus Datenbanken Web-Seiten generiert werden sollen. Viele Datenbankhersteller bieten inzwischen einen Export des Datenbankinhaltes nach XML an [Tol99].

OEM (Object Exchange Model)

Das Object Exchange Model (OEM) wurde im Rahmen des Projektes TSIMMIS (**T**he **S**tanford - **I**BM **M**anager of **M**ultiple **I**nformation **S**ources, [GMPQ+97]) vorgestellt. Weitere Verwendung fand das Modell bei Lore (Lightweight Object Repository, [MAG+97]).
OEM wurde für die Modellierung semistrukturierter Daten entwickelt, die sich bei der Integration heterogener Datenquellen ergeben. Ein Objekt im Sinne von OEM besteht aus vier Feldern, einer Objekt-ID, einem Feld für den Typen des Objekts, einem weiteren für den Objektnamen und schließlich eines für den Wert des Objekts.
Die Objekt-ID dient dazu, Objekte eindeutig zu identifizieren. Objekte können entweder von atomarem oder komplexem Typ sein. Unter atomarem Typ sind zum Beipsiel `integer`, `string` oder `real` zu verstehen. Komplexe Objekte sind vom Typ `set`. Bei komplexen Objekten vom Typ `set` enthält das Feld für den Wert Paare (Name, Objekt-ID), die Verweise auf weitere zum komplexen Objekt gehörenden Objekte darstellen [AQM+97, GMPQ+97, MAG+97, PGMW95, Suc98].
OEM Daten kann man sich als gerichteten Graphen vorstellen, bei dem die Knoten den Objekten entsprechen, die Kanten mit Attributen beschriftet und Blattknoten mit atomaren Werten assoziiert sind. Außerdem besitzt der Graph eine Wurzel, von der aus alle Objekte erreichbar sind [Suc98].

Definition 2.2.1
Formal ist ein OEM - Graph auf einer endlichen Menge von Namen $\mathbf{R}$ *definiert durch ([AQM+97]):*

(i) einen beschrifteten Graphen $(V_a \cup V_c, E)$*, wobei* V_a *und* V_c *disjunkte Mengen von Objekt-IDs sind, die die atomaren beziehungsweise komplexen Objekte beschreiben,* E *bezeichnet die mit Texten beschrifteten Kanten des Graphen,*

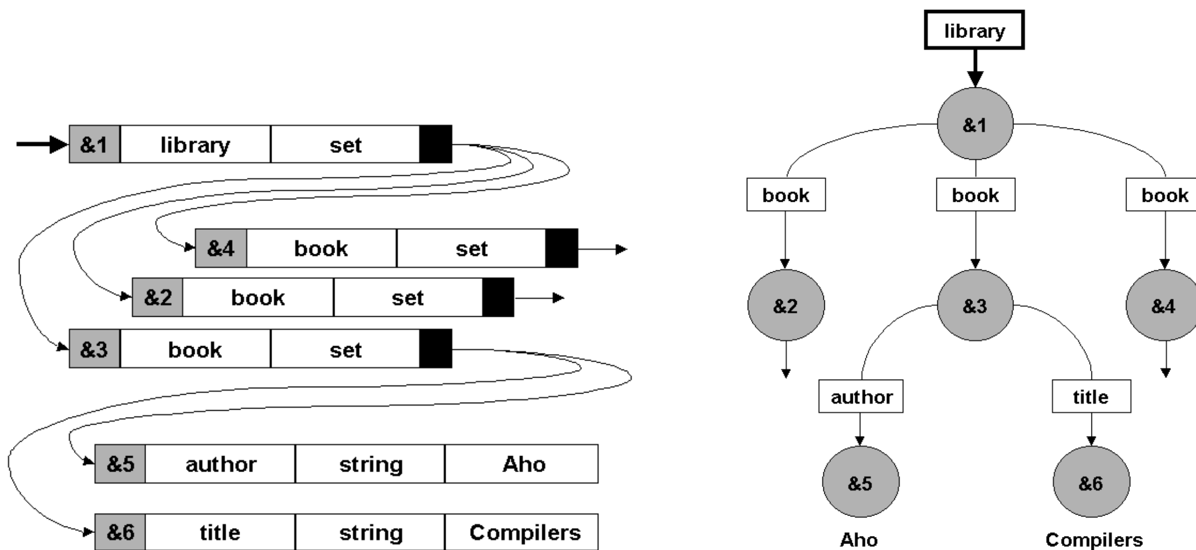
(ii) eine Funktion name *von* $\mathbf{R}$ *nach* $V_a \cup V_c$ *und*

(iii) eine Funktion val*, die die Objekte in* V_a *mit atomaren Werten in Beziehung setzt.*

(iv) *Knoten, die atomare Werte enthalten, haben keine ausgehenden Kanten.*

(v) *Jeder Knoten ist vom Objekt name(N) erreichbar für irgendeinen Namen N in* $\boldsymbol{R}$

Die Struktur von Objekten in OEM ist in Abbildung 2.9(a) dargestellt. Die Abbildung zeigt als Beispiel ein Objekt `library` vom Typ `set` mit der Objekt-ID `&1`. Dieses Objekt besteht aus mehreren Objekten `book`. Objekte, die Teile komplexer Objekte repräsentieren, sind in der Abbildung aus Gründen der Darstellung durch Pfeile mit dem jeweiligen Objekt verbunden. Abbildung 2.9(b) zeigt dieselben Objekte als Baum.



(a) Struktur von OEM-Objekten, nach [GMPQ+97]

(b) OEM-Baum

Abbildung 2.9: OEM Datenmodell

Obwohl OEM auf den ersten Blick objektorientierten Datenmodellen gleicht, ergeben sich doch Unterschiede. OEM verzichtet auf objektorientierte Konstrukte wie die Vererbung, Methoden und Klassen. Die Navigation innerhalb eines OEM - Graphen geschieht mittels der Namen der OEM - Elemente. Dieses Vorgehen bietet dann Vorteile, wenn eine Anwendung im Voraus keine Kenntnis über die Namen oder Strukturen der OEM - Elemente besitzt [PGMW95].

Das Datenmodell im Projekt Föderal

Dieses Modell unterscheidet sich, obwohl es ebenfalls für semistrukturierte Daten gedacht ist, von den bisher in diesem Abschnitt vorgestellten Modellen für semistrukturierte Daten dadurch, dass hier die Beziehungen zwischen Knoten typisiert sind. Die Typen der Beziehungen sind in einer Objekthierarchie angeordnet. Man kann sich das Modell als gerichteten, bipartiten Graphen vorstellen, das heißt, es handelt sich um einen Graphen, der zwei verschiedene

Arten von Knoten enthält, hier sind dies Knoten für Daten und Beziehungen. Bei einem bipartiten Graphen dürfen Kanten nur zwischen Knoten verschiedener Knotenmengen verlaufen.
Abbildung 2.10 zeigt ein Beispiel eines solchen Modells. Die Knoten, die Beziehungen repräsentieren, sind in der Darstellung grau eingefärbt, die Knoten für Daten sind durch Objekt-IDs gekennzeichnet. Die Knoten `Parallelogramm` und `Rechteck`, sowie die Knoten `Rechteck` und `Quadrat` sind durch Beziehungen vom Typ `Ableitung` verbunden, während zwischen den Knoten `Quadrat` und `MeinQuadrat` eine Beziehung vom Typ `Instanziierung` besteht . Das Beispiel folgt damit der Objekthierarchie aus Abbildung 2.5.

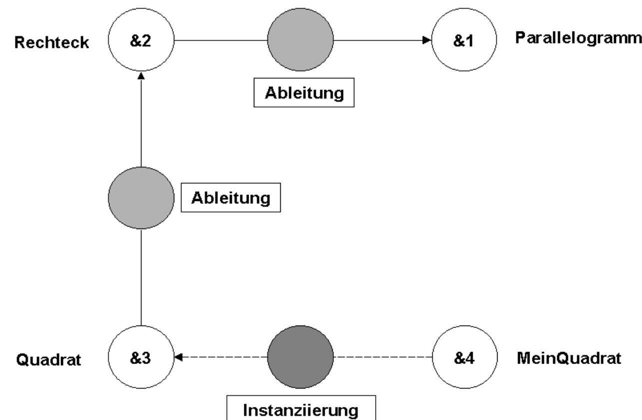


Abbildung 2.10: Datenmodell des Projektes Föderal

Als Besonderheit lässt diese Art der Modellierung Beziehungen auf Beziehungen zu. Ein Graph, der Relationen auf Relationen enthält ist nicht mehr bipartit.
Für den Entwurf einer Anfragesprache für dieses Datenmodell erweist es sich jedoch sinnvoller, die Relationsknoten zu Kanten in dem Graphen zu abstrahieren. Dadurch kann die Navigation zwischen benachbarten Datenknoten gedanklich direkt über eine einzelne Kante ohne dazwischenliegende Relationsknoten erfolgen. Zu beachten ist dabei allerdings, dass die Kanten typisiert sind. Kanten, das heißt die Relationen zwischen einzelnen Knoten besitzen eine Multiplizität. Darunter ist zu verstehen, dass für jede Relation festgelegt ist, wie oft diese zwischen zwei Knoten existieren muss beziehungsweise existieren darf. Die Relationen sind zwar gerichtet, das heißt sie führen von einem Quellknoten zu einem Zielknoten, es besteht aber die Möglichkeit, der Relation in beide Richtungen zu folgen. Es handelt sich also um bidirektionale Relationen.
Das Netz lässt sich in dem Sinn als semantisches Netz bezeichnen, als dass Wissen über den Zusammenhang der Objekte in ihm abgelegt und modelliert ist. Der Begriff semantisches Netz ist etwas irreführend gewählt, da das Netz in diesem Datenmodell nichts mit semantischen Netzen im Sinne der künstlichen Intelligenz (*KI*) zu tun hat. Mit Hilfe dieses Netzes soll hier kein *automated reasoning* durchgeführt werden.
Das Netz ist in zwei Schichten (*layer*) unterteilt. Außer den Daten sind im Netz zusätzlich

Informationen über das Modell und das Metamodell abgelegt. Das Modell stellt, objektorientiert ausgedrückt, die Hierarchie der instanziierbaren Objekte für die Darstellung der Daten zur Verfügung. Die Semantik des Modells wird durch das Metamodell beschrieben. Auf das semantische Netz übertragen heißt dies, im Metamodell ist abgelegt, dass unter anderem verschiedene Varianten von Relationen existieren.
Die Schichtenarchitektur ist in Abbildung 2.11 dargestellt. Das Modell im Beispiel sagt aus, dass ein `Parallelogramm` in einer Vererbungsbeziehung zu einem `Rechteck` und einem `Quadrat` steht. Damit folgt das Modell dem Modell aus Abbildung 2.5. Die Beziehungen zwischen Modell und Metamodell fehlen in der Abbildung aus Gründen der Übersichtlichkeit. Außerdem wurde auf die Darstellung der vierten Schicht, des *Meta - Metamodells* verzichtet, da dieses die Implementierung der anderen drei Schichten beschreibt und für den Entwurf einer Anfragesprache nicht relevant ist.

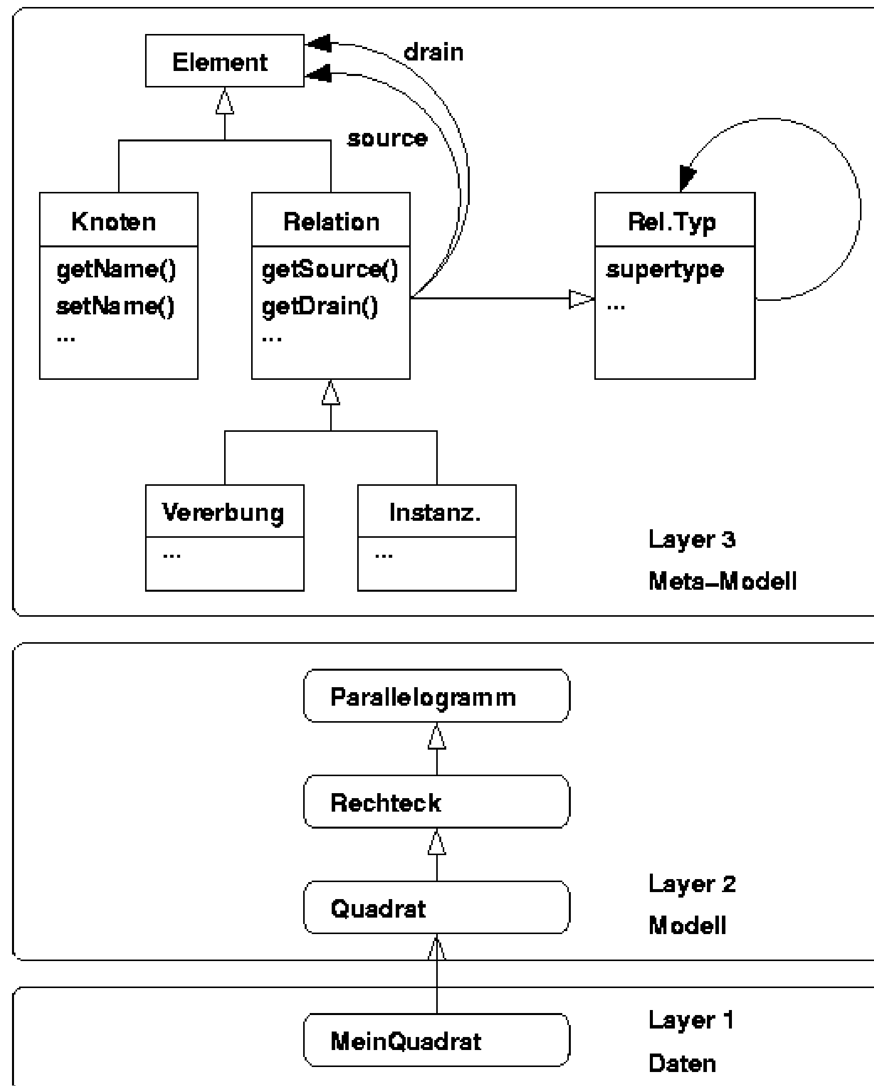


Abbildung 2.11: Schichtenarchitektur im Föderalprojekt. In der Abbildung wurde auf die Darstellung des Meta - Metamodells verzichtet, da dieses für den Entwurf einer Anfragesprache keine Bedeutung hat. Aus Gründen der Übersichtlichkeit wurde auf die Verbindung zwischen Modell und Metamodell verzichtet. Das Modell besteht hier aus einer Vererbungshierarchie aus `Parallelogramm`, `Rechteck` und `Quadrat`. Die Datenschicht enthält nur eine Instanz eines Quadrats mit Namen `MeinQuadrat`. Das Meta-Modell ist in UML-Manier beschrieben.

Kapitel 3

Anforderungen an die Sprache

Dieses Kapitel dient der Darstellung der Anforderungen an eine Anfragesprache an ein semantisches Netz, das zur Datenmodellierung im Rahmen des Projektes Föderal entwickelt wurde. Das Kapitel gliedert sich in zwei Teile. Im ersten Teil werden einige Anforderungen, die für alle Anfragesprachen an semistrukturierte Datenmodelle gelten, beschrieben. Der zweite Teil beschäftigt sich mit den speziellen Anforderungen aus dem Projekt Föderal. Hier wird beschrieben, inwiefern die allgemeinen Anforderungen aus dem ersten Teil des Kapitels auf Anforderungen aus dem Projekt Föderal zutreffen und welche zusätzlichen Anforderungen sich für eine Anfragesprache an das semantische Netz ergeben.

3.1 Allgemeine Anforderungen an eine Anfragesprache

In [Abi97] werden einige allgemeine Anforderungen an Anfragesprachen formuliert Diese werden im folgenden kurz dargestellt.

Flexible Typisierung

Anfragesprachen für semistrukturierte Datenmodelle, wie sie im Abschnitt 2.2.2 beschrieben werden, müssen flexibler sein als Sprachen in traditionellen Datenbanksystemen, vor allem, was die Typisierung angeht. Da die genaue Struktur der Daten im Allgemeinen nicht bekannt ist, muss eine Sprache automatische Typumwandlungen (*coercion*) beherrschen. Dies ist zum Beispiel bei Vergleichen zwischen atomaren Daten verschiedenen Typs, wie `string` und `integer` oder `integer` und `float` notwendig, wird aber auch beim Vergleich von Literalen und Strukturen benötigt.

Rekursion

Die Datenstruktur kann tief verschachtelt oder gar zyklisch aufgebaut sein, so dass eine Anfragesprache für semistrukturierte Daten daher Sprachelemente beinhalten muss, die Rekursion

erlauben. Ein Kalkül oder eine Algebra als Grundlage einer solchen Sprache, wie es beispielsweise bei SQL92 [Dat95] der Fall ist, reicht nicht aus, da sie keine Rekursion beinhalten.

Pfadausdrücke und reguläre Ausdrücke

Es muss mit Mitteln der Sprache möglich sein, die tief verschachtelte Datenstruktur zu traversieren, wie dies bei Graphen oder Hypertext üblich ist. Das kann zum Beispiel durch Pfadausdrücke realisiert werden.

Definition 3.1.1
Unter einem Pfadausdruck sind Ausdrücke zu verstehen, durch die Pfade oder Wege in einem Graphen von einem Knoten x zu einem Knoten y beschrieben werden. Ein Pfad ist eine Liste von Knoten, in der aufeinanderfolgende Knoten durch Kanten im Graph miteinander verbunden sind [Sed92].
Im Allgemeinen werden Pfadausdrücke mit Hilfe der Kanten notiert. Pfadausdrücke sind immer relativ zu einem Bezugsknoten zu verstehen. Aufeinanderfolgende Kanten werden dabei durch Punkte getrennt. Es gilt:

i) jeder Kantenname n im Graphen ist ein Pfadausdruck p, er beschreibt alle Pfade im Graphen, die vom Bezugsknoten aus über eine mit n beschriftete Kante zu anderen Knoten führen.

ii) $p.p$ ist ein Pfadausdruck.

Es ist zu beachten, dass Pfade beliebiger Länge möglich sein sollten. Beispielsweise muss die Suche nach allen Postleitzahlen, egal in welcher Strukturtiefe, in der Datenbank möglich sein. Dies ist durch die Verwendung regulärer Ausdrücke oder ähnlicher Konstrukte innerhalb der Pfadausdrücke möglich [Bun97]. Reguläre Ausdrücke lassen sich wie folgt definieren:

Definition 3.1.2
Die Menge der regulären Ausdrücke über einem Alphabet Σ ist induktiv definiert durch [Sch95]:

i) $\emptyset$ ist ein regulärer Ausdruck, dabei bezeichnet $\emptyset$ die leere Menge,

ii) ϵ ist ein regulärer Ausdruck, mit ϵ wird das leere Wort bezeichnet, das heißt ein Wort mit der Wortlänge 0,

iii) für alle $a \in \Sigma$ ist a ein regulärer Ausdruck,

iv) wenn α und β reguläre Ausdrücke sind, sind auch $\alpha\beta$ (Konkatenation), $(\alpha|\beta)$ (Disjunktion) sowie $(\alpha)*$ reguläre Ausdrücke.

Reguläre Ausdrücke finden auch bei der Suche nach Mustern in Daten Verwendung.

Standardoperationen

Eine Sprache für semistrukturierte Daten soll eine Reihe von Standardoperationen aus dem Datenbankbereich enthalten. Dazu gehören Sortieroperationen, Filter, sowie Operation zum Einfügen, Löschen und Ändern von Daten.

Deklarative Anfragesprache

Eine Anfragesprache soll außerdem deklarativ sein. Durch die Anfrage wird also spezifiziert, wie die gesuchten Daten aussehen und nicht auf welche Art sie gefunden werden können. Anfragen sollen natürlich für den Anwender einfach zu erstellen und zu verstehen sein [BC00].

3.2 Anforderungen im Projekt Föderal

Im Projekt Föderal muss eine Anfragesprache Sprachelemente enthalten, mit denen Teile des semantischen Netzes ausgewählt werden können, dies entspricht der Funktion des Schlüsselworts `select` in SQL (*Structured Query Language*).

Navigation durch Pfadausdrücke

Die Navigation innerhalb des Netzes soll dabei durch eine Art Pfadausdruck möglich sein. Weiter ist eine Möglichkeit vorzusehen, Pfade beliebiger Länge, beispielsweise mit Hilfe von regulären Ausdrücken, zu spezifizieren. Zum Beispiel soll durch einen Ausdruck `root.person.*.adresse` ausgedrückt werden, dass alle Pfade in die Suche einbezogen werden sollen, die mit `root.person` beginnen und mit `adresse` enden. In Abbildung 3.1 sind damit die Pfade `root.person.adresse` und `root.person.firma.adresse` gemeint.
Reguläre Ausdrücke sollen auch bei der Angabe von Kantennamen verwendet werden können. Der Ausdruck `root.person.*name` wird auf den Graphen in Abbildung 3.1 bezogen zu den Pfaden `root.person.name` und `root.person.vorname` expandiert. Dies hat sich als sinnvoll erwiesen, da die genaue Struktur des Netzes oft nicht bekannt ist. Außerdem soll mit einer einzelnen Anweisung in einem Schritt über mehrere verschiedene Relationstypen navigiert werden können.
Bei der Navigation ist zu beachten, dass Relationen zwischen Knoten im Datenmodell des semantischen Netzes typisiert sind. Deshalb soll die Navigation über die Typen der Relationen erfolgen. Außerdem muss die Anfragesprache den Bidirektionalität der Relationen unterstützen.

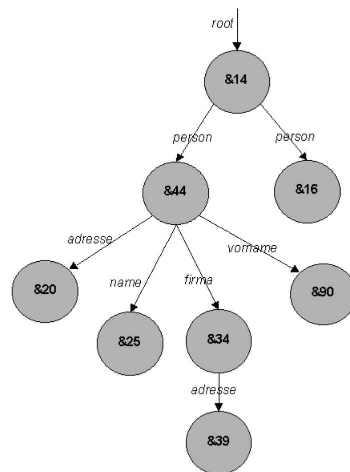


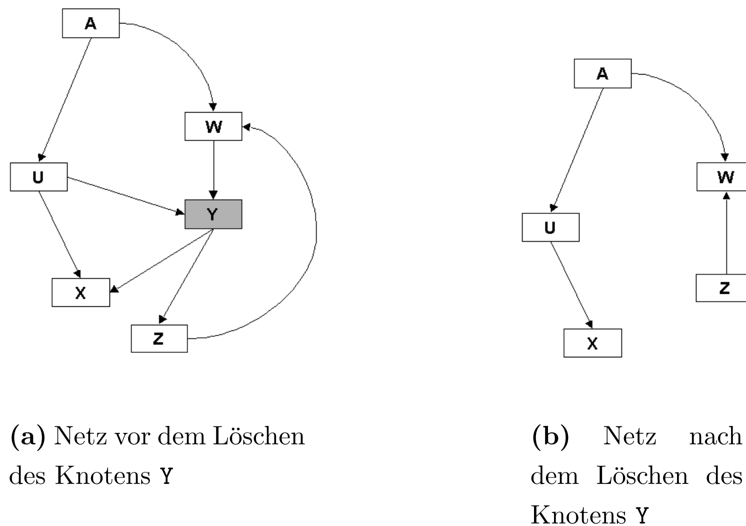
Abbildung 3.1: Pfade im semantischen Netz

Standardoperationen auf Knoten

Es muss möglich sein, das Suchergebnis durch eine Restriktion weiter einzuschränken. Dies kann durch Prädikte, die die Elemente der Ergebnismenge erfüllen müssen, realisiert werden. Weiter ist ein Sprachelement zur Sortierung des Ergebnisses einer Anfrage vorzusehen, wie beispielsweise `order by` in SQL.
Als weitere Primitive der Sprache sind Operationen zur Manipulation des Netzes vorzusehen. Dazu gehören Operationen, mit denen Knoten in das Netz eingefügt oder gelöscht werden können. Beim Löschen eines Knotens sind alle Relationen, in denen der Knoten partizipert, ebenfalls zu löschen. Dies ist in Abbildung 3.2 zu sehen. Der Zustand eines semantischen Netzes vor dem Löschen ist in Abbildung 3.2(a) dargestellt. Im Netz soll der grau eingefärbte Knoten `Y` gelöscht werden. Abbildung 3.2(b) zeigt dasselbe semantische Netz nach Entfernen des Knotens `Y` mitsamt seinen Relationen. Die Einhaltung von Integritätsregeln muss durch Mechanismen des semantischen Netzes gewährleistet werden. Überprüfungen solcher Art sind nicht Bestandteil der Sprache. Knoten müssen durch ein Sprachelement umbenannt werden können.

Standardoperationen auf Relationen

Die Manipulation des Netzes bezieht sich auch auf die Relationen zwischen den Knoten. So müssen Operationen zum Einfügen und Löschen von Relationen vorhanden sein. Auch hier ist die Sprache nicht für die Einhaltung von Integritätsregeln verantwortlich.
Die Hierarchie der Relationen muss erweiterbar sein, das bedeutet, über die Sprache müssen neue Relationstypen definiert werden können.



(a) Netz vor dem Löschen des Knotens `Y`

(b) Netz nach dem Löschen des Knotens `Y`

Abbildung 3.2: Löschen eines Knotens im semantischen Netz. Die Abbildung zeigt den Zustand eines semantischen Netzes vor und nach Entfernen des grau eingefärbten Knotens `Y` mitsamt seinen Relationen zu anderen Knoten.

Kopieren

Erwünscht ist zudem ein Konstrukt zum Kopieren von Teilnetzen. Dies ist notwendig, da häufig nach dem Baukastenprinzip modelliert wird. Dabei müssen Module kopiert werden. Die Operation soll nicht nur Kopien der Knoten anlegen, sondern auch die Beziehungen zwischen den Knoten sollen auf die Kopie übertragen werden. Dieser Mechanismus soll am Beispiel in Abbildung 3.3 erläutert werden.

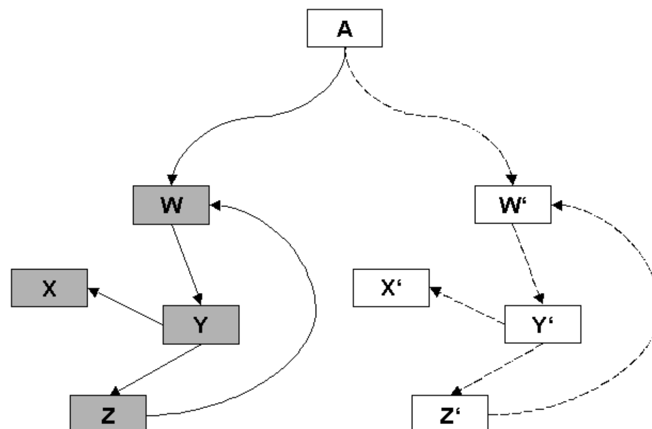


Abbildung 3.3: Kopieren von Teilnetzen - In der Abbildung ist das Kopieren des grau eingefärbten Teilnetzes aus den Knoten `W`, `X`, `Y` und `Z` dargestellt. Es wird von jedem Objekt in der Vorlage eine Kopie erzeugt, sowie die Relationen aus der Vorlage auf die Kopie übertragen.

Im Beispiel soll das durch graue Einfärbung markierte Telnetz aus den Objekten `W`, `X`, `Y` und `Z`

kopiert werden. Von jedem Objekt im markierten Subnetz wird eine Kopie, in der Abbildung durch ein Hochkomma gekennzeichnet, erzeugt. Außerdem werden die Relationen zwischen den Objekten in der Vorlage auf die Kopien der Objekte übertragen. Die neu erzeugten Relationen sind in der Abbildung als gestrichelte Kanten eingezeichnet.
Beim Kopieren sind zwei Varianten vorzusehen. Die erste Alternative soll den oben vorgestellten Kopiermechanismus realisieren, während die zweite Alternative zusätzlich zu den Relationen innerhalb des Teilnetzes die Relationen mitkopiert, die das Teilnetz mit dem Rest des Netzes verbinden.

Kapitel 4

Anfragesprachen

In den folgenden Abschnitten werden verschiedene bereits vorhandene Anfragesprachen kurz vorgestellt und in Hinblick auf die Anforderungen im Projekt Föderal analysiert und bewertet. Bei der Auswahl wurde darauf geachtet, dass zu jedem in Kapitel 3 dieser Arbeit beschriebenen Datenmodell, mit Ausnahme des Modells im Projekt Föderal, mindestens eine Anfragesprache betrachtet wird. Sofern mehrere Sprachen vorhanden sind, wie bei XML, so unterscheiden sich diese in ihrer Syntax oder ihrem Funktionsumfang. Im folgenden werden OQL, Lorel, sowie die XML-Sprachen XPath, XQL und Quilt in Hinblick auf die Anforderungen hin untersucht. Außer den hier beschriebenen Sprachen existieren noch viele weitere, die sich in ihrem Anwendungsgebiet und Funktionsumfang unterscheiden. Hier sei nur auf UnQL [BDHS96], StruQL [FFLS99] und XML-GL [SSE+99], sowie XQuery [W3C01b] und XML-QL [DFF+98] hingewiesen.

SQL (*Structured Query Language*) wurde hier nicht betrachtet, da SQL, nach dem Standard von 1992, weder Rekursion noch Typhierarchien unterstützt. Zudem existiert in SQL kein Sprachelement mit dem es möglich ist, mit Hilfe einer Art von Pfadausdruck über die Daten zu navigieren. Dies ist jedoch bei einem semistrukturierten Datenmodell, wie dem Modell im Projekt Föderal, notwendig. Rekursion und Typhierarchien sind inzwischen Teil des SQL-Standards von 1999. Dieser wird jedoch von Datenbanksystemen bisher nur mit Einschränkungen unterstützt.

4.1 OQL (Object Query Language)

Die *Object Query Language (OQL)* wurde von der *Object Database Management Group (ODMG)* 1993 als Anfragesprache für objektorientierte Datenbanken vorgestellt.
Die Syntax von OQL ähnelt der von *SQL (Structured Query Language)*. Anfragen sind als `select...from...where` Ausdruck aufgebaut. Im `select` - Teil der Anfrage werden diejeni-

gen Attribute angegeben, deren Werte im Anfrageergebnis enthalten sein sollen, im `from` - Teil wird die Menge der Objekte definiert, auf die sich die Anfrage bezieht. Der `where` - Teil der Anfrage dient als Filter auf die Objektmenge. Die Attribute einzelner Objekte werden in der Form `<objekt>.<attribut>` notiert.
Diese grundsätzliche Struktur der Anfragen soll an folgendem Beispiel erläutert werden. Die Datenbank für das Beispiel enthält Objekte vom Typ `Person` mit zwei Attributen, `age` und `name`. Die Menge aller Objekte vom Typ `Person` (*extent*) soll mit `Personen` bezeichnet sein.

```
select X.age
from X in Personen
where X.name = "Pat"
```

Hier wird das Attribut `age` aller Objekte vom Typ `Person` ausgewählt, deren Attribut `name` den Wert `Pat` enthält. Bei der Auswertung wird zunächst jedes Objekt vom Typ `Person` an die Variable `X` gebunden. Anschließend wird für jedes dieser Objekte geprüft, ob das Attribut `name` den Wert `Pat` enthält. Falls dies der Fall ist, wird der Wert des Attributs `age` in das Ergebnis übernommen.
Objekte sind in dem OQL zugrunde liegenden Datenmodell durch Beziehungen (*relationships*) mit einander verbunden. Dadurch ergibt sich die Möglichkeit, über die Beziehungen von einem Objekt zu anderen Objekten zu navigieren. In OQL ist diese Navigation über Beziehungen durch Pfadausdrücke realisiert. Pfade werden in der Form `<objekt>(.<property>)+` notiert, wobei `<property>` sowohl Attribute eines Objektes als auch seine Beziehungen zu anderen Objekten bezeichnet. Das `+`-Zeichen in obiger Definition bedeutet, dass der Klammerausdruck beliebig oft verwendet werden darf, dabei aber mindestens einmal verwendet werden muss. Falls `<property>` eine Beziehung (*relationship*) ist, liefert die Auswertung des Pfades ein Objekt.
Für jeden Objekttyp existiert ein Konstruktor gleichen Namens, mit dem neue Instanzen des Typs erzeugt werden können. Beispielsweise wird mit

```
Person (name: "Pat", birthdate: "3/28/56", salary: 100000)
```

eine Instanz vom Typ `Person` erzeugt. Dabei erhalten die Attribute `name`, `birthdate` und `salary` die angegebenen Werte.
Konstruktoranweisungen müssen im `select` - Teil einer Anfrage immer dann benutzt werden, wenn das Ergebnis mehr als eine Eigenschaft eines Objektes enthalten soll. Neben den Konstruktoren für die Objekttypen stehen Konstruktoren für die eingebauten Typen `set`, `bag`, `list`, `array` und `struct` zur Verfügung. In folgendem Beispiel wird das Ergebnis als Menge von Objekten des Typs `couple` berechnet. Das Beispiel setzt ein Schema mit den drei Objekttypen `Student`, `Course` und `Professor` voraus. Zwischen Objekten vom Typ `Student` und `Course` besteht eine Beziehung `takes`. Außerdem besteht zwischen Objekten vom Typ `Course`

und Professor eine Beziehung taught_by. Die Menge aller Objekte vom Typ Student sei mit Studenten bezeichnet. Das Schema ist in Abbildung 4.1 graphisch dargestellt.

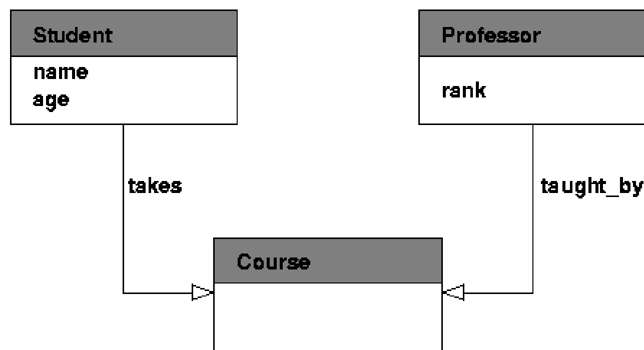


Abbildung 4.1: OQL-Schema eines Beispiels

```
select couple (student: X.name, professor: Z.name)
from X in Studenten,
     Y in X.takes,
     Z in Y.taught_by
where Z.rank = "full professor"
```

Umgekehrt ist es auch möglich, select - Ausdrücke innerhalb von Konstruktoranweisungen zu verwenden. So ist es möglich, Instanzen zu kopieren.
Ergebnisse einer Anfrage können mit dem Operator sort by lexikographisch sortiert werden. Zum Beispiel werden durch sort X in Personen by X.age, X.name alle Personen nach Alter und Name sortiert.
OQL enthält die Möglichkeit, Methoden, die auf bestimmten Objekttypen definiert sind, auszuführen. Der Methodenaufruf hat die Form <objekt>.<methode>(<parameter>). Im Allgemeinen ist der Aufruf von Methoden immer auch an den Stellen möglich, an denen Attributbezeichnungen verwendet werden können [Cat94b].

4.1.1 Bewertung

Da OQL eine Anfragesprache für ein strukturiertes Datenmodell ist, kann man nicht erwarten, dass Anforderungen, die durch ein semistrukturiertes Datenmodell enstehen, erfüllt werden.
OQL erlaubt das Auswählen beliebiger Objektmengen über Ausdrücke der Form select... from...where. Beziehungen zwischen Objekten werden durch Pfadausdrücke formuliert. Dabei wird ein festes, vordefiniertes Schema verwendet, so dass reguläre Ausdrücke innerhalb von Pfaden nicht benötigt werden.
Die Ergebnismenge einer Anfrage wird durch die where - Klausel im OQL - Ausdruck gefiltert. Das Ergebnis kann lexikographisch sortiert werden.

Neue Instanzen von Objekttypen können durch Verwendung von Konstruktoren für die Typen erzeugt werden. Das Löschen und Ändern von Objekten ist dagegen nur durch Methoden, die zusätzlich auf den Objekten definiert werden, möglich. OQL besitzt keine explizite `delete` oder `update` Anweisung.
Das Kopieren von Instanzen ist zum Teil durch die Kombination von Auswahlanfrage und Erzeugung neuer Objekte möglich. Dabei wird eine neue Instanz erzeugt, deren Attribute dieselben Werte erhalten wie die Attribute einer durch die Anfrage ausgewählten, bereits existierenden Instanz. Die Instanzen, mit denen die bereits existierende Instanz, die Kopiervorlage, in Beziehung steht, werden jedoch nicht kopiert. Das bedeutet, die neu erzeugte Instanz steht mit denselben Instanzen in Beziehung wie die Kopiervorlage. Dieses Verhalten ist in objektorientierten Datenbanken erwünscht, um Duplikate zu vermeiden, reicht aber für das semantische Netz im Projekt F:oderal nicht aus.
Das Verhalten von OQL beim Kopieren von Objekten ist in Abbildung 4.2 dargestellt. Hier besitzt ein Objekt `PersonenGruppe` Beziehungen zu den Objekten `Person1` und `Person2`. Im Folgenden soll das Teilnetz aus `PersonenGruppe`, `Person1` und `Person2` kopiert werden. Durch die Anweisung

```
PersonenGruppe (select X
                from X in Personen)
```

wird das Objekt `PersonenGruppe’` erzeugt. Insgesamt ergibt sich die durch durchgezogene Linien dargestellte Situation, das heißt das neu erzeugte Objekt `PersonenGruppe’` besitzt Beziehungen zu den Objekten `Person1` und `Person2`. Erwünscht wäre jedoch die durch gestrichelte Verbindungen angedeutete Situation. Mit dem Objekt `PersonenGruppe` sollen auch die Objekte `Person1` und `Person2` kopiert werden.

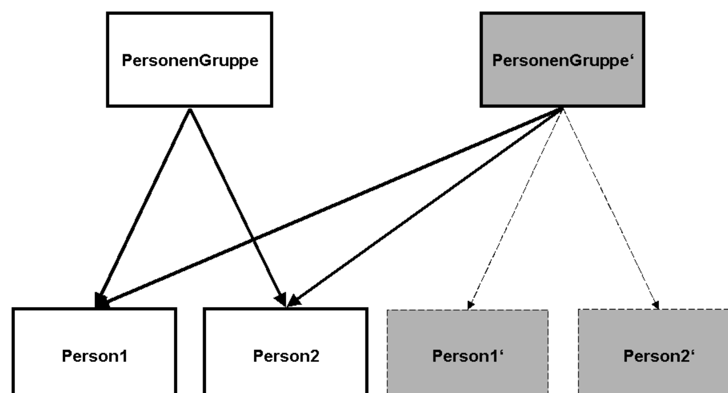


Abbildung 4.2: Verhalten von OQL beim Kopieren von Objekten

OQL unterscheidet Objekte zwar nach ihrem Typ, einer Erweiterung des Typsystems ist jedoch nur durch eine Schemaänderung möglich. Dies ist nur mit einer *Object Definition Language*

(ODL), nicht mit OQL möglich. Beziehungen zwischen Objekten sind nicht typisiert, aber nur für die Objekte verfügbar für die sie im Schema definiert wurden. Auch die verfügbaren Beziehungen lassen sich nur mit Hilfe einer ODL verändern.

4.2 Lorel (Lore Language)

Die *Lore Language (Lorel)*, für *Lore (Ligthweight Object Repository)* entwickelt, ist eine Anfragesprache für das *Object Exchange Model (OEM)*, vergleiche Abschnitt 2.2.2. Bei Lorel handelt es sich um eine Erweiterung der *Object Query Language (OQL)*.
Ein wichtiges Konzept in Lorel ist die automatische Typumwandlung (*coercion*) bei Vergleichen, um die strenge Typisierung von OQL für semistrukturierte Daten abzuschwächen.
Als Erweiterung von OQL übernimmt Lorel die grundsätzliche `select` ... `from` ... `where` Struktur von Anfragen aus OQL und SQL.

4.2.1 Pfadausdrücke

Lorel verwendet zwei Arten von Pfadausdrücken, um Objekte über Erreichbarkeit in einem OEM - Graphen auszuwählen. Es gibt zum einen einfache Pfadausdrücke (*simple path expressions*), sowie generelle Pfadausdrücke *(general path expressions)*, die auf regulären Ausdrücken basieren, siehe Abschnitt 3.1
Pfadausdrücke dürfen im `select` -, `from` -, wie auch im `where`- Teil einer Anfrage verwendet werden. Beide Arten von Pfadausdrücken werden im Folgenden formal definiert, bevor ihre Verwendung an Beispielen gezeigt wird. Die Beispiele im Text beziehen sich auf den OEM - Graphen in Abbildung 4.3.

Definition 4.2.1
Ein einfacher Pfadausdruck (simple path expression) ist eine Folge $Z.l_1 \cdots l_n$, wobei Z für den Namen eines Objektes oder einer Variablen, die ein Objekt bezeichnet, steht und $l_1 \cdots l_n$ die beschrifteten Kanten eines Graphen sind.
Ein Datenpfad ist eine Folge $o_0 l_1 o_1 l_2 \cdots l_n o_n$, wobei Objekte mit o_i und Kanten zwischen zwei Objekten o_{i-1} und o_i mit l_i bezeichnet werden [AQM+97].

Die Ausdrücke `Guide.restaurant.zipcode`, `Guide.restaurant`, sowie `Guide.restaurant.address.zipcode` sind Beispiele für einfache Pfadausdrücke. Der Pfad `Guide.restaurant.zipcode` bezieht sich auf das Objekt mit der ID `&54` in Abbildung 4.3, während durch `Guide.restaurant` die Objekte `&19`, `&35` und `&77` referenziert werden. Der Pfadausdruck `Guide.restaurant.address.zipcode` führt zum Objekt mit der ID `&16`.

Definition 4.2.2
Ein genereller Pfadausdruck (general path expression) besteht aus einem Objektnamen, gefolgt

von mehreren Komponenten. Für die Komponenten gilt:

i) Sofern l ein Bezeichner ist, dann ist $.l$ eine Komponente eines generellen Pfadausdrucks.

ii) Wenn p_1 und p_2 Komponenten eines generellen Pfadausdrucks sind, dann sind auch p_1p_2, $p_1 \mid p_2$, (p_1), $(p_1)?$, $(p_1)+$ und $(p_1)*$ Komponenten eines generellen Pfadausdrucks [AQM+97].

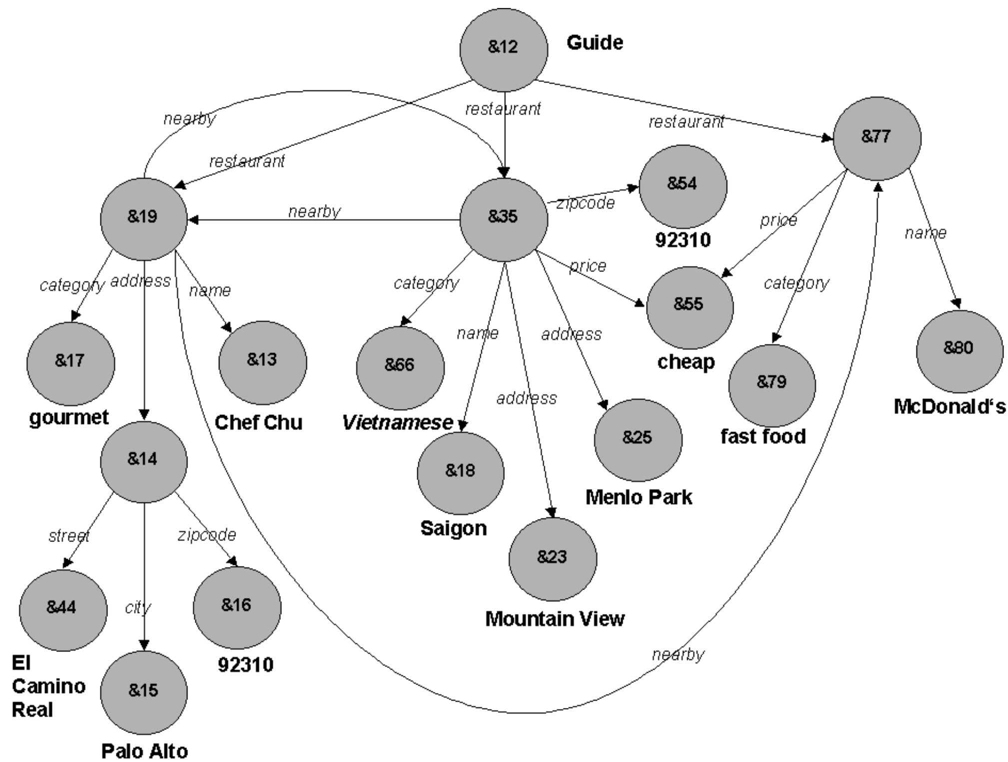


Abbildung 4.3: Beispiel eines OEM Graphen [AQM+97]

Bei dem Pfadausdruck `Guide.restaurant(.address)?.zipcode` handelt es sich um einen generellen Pfadausdruck, der sich durch die einfachen Pfadausdrücken `Guide.restaurant.address.zipcode` und `Guide.restaurant.zipcode` ausdrücken lässt. Er verweist auf die Objekte `&54` und `&16` in Abbildung 4.3.

Bei der Verwendung des Kleene - Operators $*$ bei regulären Ausdrücken in Graphen muss auf ein Problem hingewiesen werden. Die Operation kann als Ergebnis unter Umständen unendlich viele Objekte liefern. Dies ist der Fall, sofern sich im Pfad, der durch den regulären Ausdruck beschrieben wird, ein Zyklus befindet. In Abbildung 4.3 ergibt sich dieses Problem beispielsweise bei dem Pfadausdruck `Guide.restaurant.(nearby)*`. Der Zyklus befindet sich dabei zwischen den Objekten `&19` und `&35`. Lorel löst dieses Problem durch die Beschränkung, dass ein Objekt bei der Auswertung eines generellen Pfadausdrucks höchstens einmal durchlaufen werden darf.

Lorel erlaubt die Verwendung von Platzhaltern (*wildcards*) innerhalb von Pfadausdrücken. Zur Verfügung stehen die Zeichen `%` und `#`. Das Zeichen `%` steht für null oder mehrere Zeichen innerhalb eines Bezeichners, `zip%` beispielsweise passt auf alle Bezeichner mit dem Präfix `zip`. Der Platzhalter `#` dient als Abkürzung für Datenpfade beliebiger Länge. Er lässt sich auch als `(%)*` schreiben. Der Pfadausdruck `Guide.restaurant.#.zipcode` passt in Abbildung 4.3 auf die Pfadausdrücke `Guide.restaurant.zipcode`, `Guide.restaurant.address.zipcode`, `Guide.restaurant.nearby.zipcode` und `Guide.restaurant.nearby.address.zipcode`. Die Platzhalter vereinfachen die Formulierung von Pfadausdrücken besonders dann, wenn die Struktur der Daten nicht exakt bekannt ist, was vor allem bei semistrukturierten Daten häufig der Fall ist.

Um Pfadausdrücke abzukürzen, gibt es in Lorel die Möglichkeit, Objekten Namen zuzuordnen. Die Namen können anschließend als Eintrittspunkte in die Datenbank benutzt werden. Die Zuordnung erfolgt mit name <name> := <expression>, <expression> kann dabei eine Anfrage oder ein neues Objekt sein oder aber den Wert `null` haben. Mit der Anweisung

```
name myFavorite : = element (
     select Guide.restaurant
     where Guide.restaurant.name = "Saigon")
```

beispielsweise wird dem Objekt `&18` der Name `myFavorite` zugeordnet. Die Funktion `element` extrahiert einen einzelnen Wert aus einer Menge mit einem einzelnen Element [AQM+97].

4.2.2 Pfad - und Objektvariablen

In Lorel können Datenpfade an Variablen gebunden werden. Auf diese Weise können Pfade auf Gleichheit oder Ungleichheit getestet werden oder die Struktur des Datengraphs mit Hilfe der Funktion `path-of` untersucht werden. In der Anfrage

```
select distinct path-of(P)
from Guide.#@P.zipcode
```

zum Beispiel werden alle Pfade, die zu einer Kante `zipcode` führen durch die Anweisung `@P` an eine Pfadvariable `P` gebunden und über die Funktion `path-of` als Ergebnis ausgegeben. Im Beispiel aus Abbildung 4.3 erhält man durch diese Anfrage die Pfade `restaurant`, `restaurant.address`, `restaurant.nearby` und `restaurant.nearby.address` als Ergebnis.

Eine zweite Art von Variablen in Lorel sind Objektvariablen. Mit Hilfe dieser Variablen können Pfade unterschieden werden, die sonst als syntaktisch identisch betrachtet werden und somit nicht unterscheidbar wären. Dies soll an folgendem Beispiel deutlich werden.

```
select N
from Guide.restaurant{R}.name N
```

```
where R.address{A1}.city = "Palo Alto" and
      R.address{A2}.city = "Menlo Park"
```

In der Anfrage wird der Pfad `Guide.restaurant` durch `{R}` an eine Variable `R` gebunden. Analog wird der Pfad `Guide.restaurant.address` an die Variablen `A1` und `A2` gebunden. Das Ergebnis der Anfrage sind alle Restaurants, die sowohl in Palo Alto als auch in Menlo Park zu finden sind.

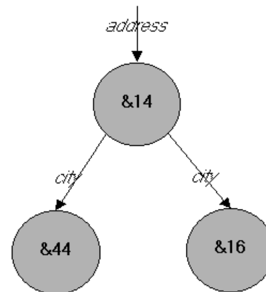


Abbildung 4.4: Ausschnitt aus einem OEM Graphen

Ohne die Bindung an die Variablen `A1` und `A2` wäre in beiden Komponenten des `where` - Teils der Anfrage dieselbe Kante `R.address` im Graphen gemeint. Dies würde nur dann ein nichtleeres Ergebnis liefern, falls im Graphen von einem Objekt, das über eine Kante `Guide.restaurant.address` erreichbar ist, mehr als eine Kante `city` ausgeht, wie in Abbildung 4.4 gezeigt [AQM+97].

4.2.3 Manipulation von Daten

Lorel stellt keine explizite Operation für das Löschen von Objekten zur Verfügung. Stattdessen werden alle Objekte, die durch keinen Pfad im Graphen erreichbar sind, als gelöscht angesehen.

Neue OEM - Objekte können mit der Anweisung `new_oem (val_type, value)` erzeugt werden. Dabei gibt `val_type` den Typ des Objektes an. Objekte können entweder einen atomaren Typ, wie `integer`, `string` oder `real` haben oder sie sind komplexe Objekte. Mit der Anweisung `new_oem (int,5)` wird ein Objekt vom atomaren Typ `integer` mit dem Wert 5 erzeugt. Die folgende Anweisung dagegen erzeugt ein komplexes Objekt.

```
new_oem (complex,
    struct (a:{new_oem (int,5)}, b:{X,Y}))
```

Das neue Objekt besitzt eine mit `a` beschriftete Kante zu einem Objekt vom Typ `integer` mit dem Wert 5 und jeweils eine mit `b` bezeichnete Kante zu den Objekten `X` und `Y`. Die Struktur

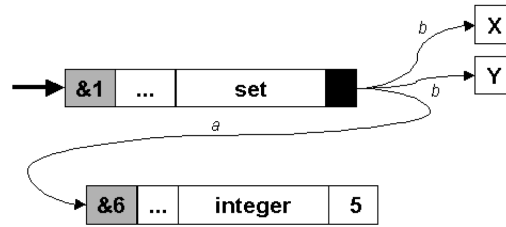


Abbildung 4.5: Beispiel eines komplexen OEM - Objektes

des erzeugten Objekts ist in Abbildung 4.5 dargestellt.
Grundsätzlich kann nach dem Konstruktor auch eine Auswahlanfrage folgen, beispielsweise

```
new_oem (select Guide.restaurant
         where Guide.restaurant.name = "Saigon).
```

Dies entspricht jedoch keiner Kopieroperation im Sinne der Anforderungen aus Kapitel 3, vielmehr wird ein neues Objekt vom Typ `complex` erzeugt, das durch Kanten mit den bereits existierenden Objekten, die durch die Anfrage ausgewählt werden, verbunden wird.
Bei Änderungen an Objekten sind mehrere Fälle zu unterscheiden. Änderungen bei atomaren Objekten beziehen sich auf den im Objekt gespeicherten Wert, wohingegen sich Änderungen bei komplexen Objekten auf die Kanten zwischen den Objekten beziehen. Der Wert eines atomaren Objektes wird durch die Anweisung `update <name> [+|-|:] = <wert>` geändert. Durch `update Price := 7` beispielsweise wird der Wert eines Objekt mit dem Namen `Price` auf 7 geändert, mit der Anweisung `update Price += 1` wird der Wert eines Objektes mit dem Namen `Price` um eins erhöht.
Bei komplexen Objekten beziehen sich die Operatoren `:=`, `+=` und `-=` im Gegensatz zu atomaren Objekten nicht auf Werte sondern auf die von diesem Objekt ausgehenden Kanten. Mit der Anweisung `update MyFavorite.address += "Sunnyvale"` wird eine Kante vom Objekt `MyFavorite` zu dem Objekt mit dem Wert `Sunnyvale` eingefügt.
Analog zu SQL oder OQL können auch in Lorel mehrere Objekte mit einer einzelnen Anweisung verändert werden. Die Syntax für solche Anweisungen lautet folgendermaßen:

```
update P [+|-|:] = <expression>
from <from-clause>
where <where-clause>
```

Dabei bezeichnet `P` eine Variable, die im `from` - Teil der Anweisung gebunden wird, `<expression>` steht stellvertretend für eine Menge von Objekten. Beispielsweise wird mit der Anweisung

```
update X.city += Z
from Guide.restaurant{X}.address Z
where Z = "Menlo Park"
```

eine mit `city` beschriftete Kante zwischen den Objekten `&35` und `&25` aus Abbildung 4.3 eingefügt. Abbildung 4.6 zeigt das Ergebnis der Anweisung.

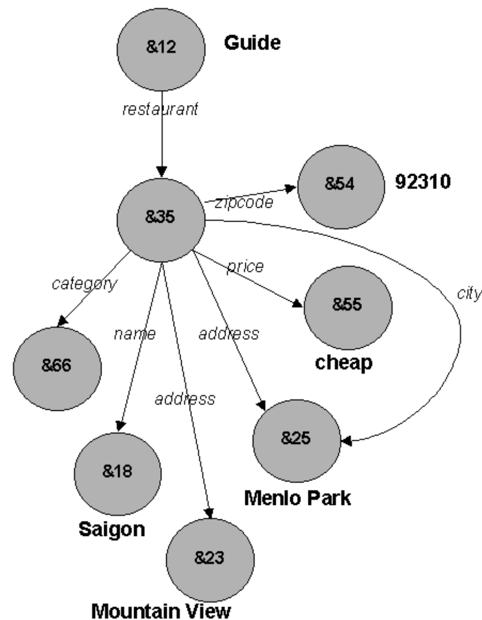


Abbildung 4.6: Ausschnitt aus einem OEM - Graphen

Kanten können gelöscht werden, indem sie auf die leere Menge `{}` gesetzt werden, beispielsweise werden durch die Anweisung `update Guide.restaurant := {}` alle mit `restaurant` beschrifteten Kanten in Abbildung 4.3, die vom Objekt `&12` ausgehen, gelöscht [AQM+97].

4.2.4 Bewertung

Lorel bietet durch die Einführung genereller Pfadausdrücke in die Syntax von OQL ein mächtiges Konstrukt für die Navigation in OEM - Graphen an. Mit Hilfe solcher Pfadausdrücke können einzelne Knoten oder Mengen von Knoten aus einem Graphen ausgewählt werden.
Die Ergebnismenge einer Anfrage kann durch Angabe einer `where` - Klausel analog zu SQL oder OQL eingeschränkt werden. Als Erweiterung von OQL beinhaltet die Syntax von Lorel außerdem einen `sort by` Ausdruck für die Sortierung von Ergebnissen [BC00, BL01]
Zudem stellt Lorel Anweisung zum Einfügen und Ändern von Objekten zur Verfügung. Dabei wird zwischen atomaren und komplexen Objekten unterschieden. Bei atomaren Objekten beziehen sich die Operationen auf die Werte der Objekte, bei komplexen Objekten dagegen auf die Kanten zwischen den Objekten.
Lorel enthält keine explizite Operation für das Löschen von Objekten. Objekte gelten als gelöscht, sobald sie im Graphen nicht mehr erreichbar sind. Das Löschen eines Objektes lässt sich demnach durch das Entfernen aller zum Objekt führenden Kanten erreichen, was bei vielen Kanten umständlicher sein kann, als eine einzelne Anweisung, die das Objekt mit allen

Kanten löscht.
Teile des OEM - Graphen können mit Lorel nicht kopiert werden. Es ist bei der Objekterzeugung zwar möglich, eine Auswahlanweisung anzugeben, dadurch werden jedoch keine Kopien der Objekte angelegt, sondern die Originalobjekte durch Kanten mit einem neuen Objekt verbunden. Daher kann eine Kopieroperation auch nicht durch Einfügen und Auswählen simuliert werden.
Da OEM nur Knoten vom Typ OEM - Objekt als Datenstruktur enthält, kennt Lorel nur diese eine Art von Knoten. Eine Anfragesprache für das semantische Netz im Projekt Föderal muss jedoch verschiedene Typen unterscheiden können.
Außerdem bietet Lorel keine Möglichkeit an, Kanten anhand ihres Typs zu unterscheiden. Die Navigation erfolgt ausschließlich über die Namen der Kanten. Dies hat seine Ursache darin, dass OEM keine typisierten Kanten kennt. Im semantischen Netz dagegen muss die Navigation über die Typen der Kanten erfolgen.

4.3 XPath (XML Path Language)

Die *XML Path Language (XPath)* [BM00, CD99] wurde vom *World Wide Web Consortium (W3C)* im November 1999 als Anfragesprache für XML-Dokumente vorgestellt. Sie soll eine einheitliche Syntax und Semantik für gemeinsame Funktionalität von *Extensible Stylesheet Language Tranformation (XSLT)* [ABC+01] und *XML Pointer Language (XPointer)* [W3C01a] bereitstellen. XSLT dient zur Übersetzung von Dokumenten aus einem XML - Format in ein anderes XML-Format. Mit XPointer sind Verweise zu beliebigen Elementen innerhalb von XML - Dokumenten möglich.
Mit Hilfe von XPath können Teile eines XML - Dokuments adressiert werden. Außerdem stellt XPath Basisfunktionen für die Manipulation von Zeichenketten, Zahlen und bool'schen Werten bereit. XPath navigiert auf der logischen Struktur der Dokumente. Dafür werden Pfadnotationen, ähnlich der *Uniform Ressource Locators (URL)* verwendet.
XPath interpretiert die hierarchische Struktur eines XML-Dokuments als Baum, bei dem die Elemente des Dokumentes den Knoten *(nodes)* des Baums entsprechen. Als weitere Datenstruktur kennt XPath sogenannte Knotensätze *(node-sets)*, darunter sind strukturell zusammenhängende Gruppen von Knoten zu verstehen. XML-Elemente und deren Attribute erscheinen im XML-Baum in derselben Reihenfolge wie im Dokument.
XPath unterscheidet sieben Arten von Knoten. Es gibt in jedem Baum einen *Wurzelknoten (root node)*, der jedoch nicht dem Wurzelelement des XML-Dokuments entspricht. Das Wurzelelement des Dokuments ist ein Nachfolgeknoten des Wurzelknotens. Jedes Element wird in einem Knoten vom Typ *Elementknoten (element node)* repräsentiert. Jeder Elementknoten kann weitere Elementknoten als Kinder haben. Die Blätter des Baumes sind Zeichenketten, die

in *Textknoten (text nodes)* enthalten sind. Attribute der Elemente werden als *Attributknoten (attribute nodes)* dargestellt. Diese sind keine Kinder der Elementknoten, sondern dem jeweiligem Elementknoten zugeordnete Knoten. Außerdem gibt es eigene Typen für *Kommentarknoten (comment nodes)*, *Verarbeitungsanweisungs-Knoten (processing instruction nodes)* und *Namensraumknoten (namespace nodes)*.
Namensräume werden beispielsweise bei XML-Dokumenten benötigt, die mehrere DTDs benutzen. Durch die Verwendung von Namensräumen kann bei Elementen gleichen Namens aus unterschiedlichen DTDs festgestellt werden, welches Element gemeint ist. Jedes Element bekommt durch den Namensraum einen in diesem Zusammenhang eindeutigen Namen.
Das wichtigste syntaktische Konstrukt in XPath ist der Ausdruck (*expression*). Jeder Ausdruck wird zu einer Knotenmenge (*node-set*), einem bool'schen Wert, einer Zahl oder einer Zeichenkette ausgewertet. Die Auswertung erfolgt stets im Kontext eines Knotens (*context node*), einem Paar von positiven ganzen Zahlen ungleich Null (*context position* und *context size*), einer Menge von Variablenbindungen, einer Funktionsbibliothek, sowie einer Menge von Deklarationen für den Namensraum des Ausdrucks. Die Werte für *context position* und *context size* können durch Prädikate im Ausdruck verändert werden. XPointer und XSLT spezifizieren, wie der Kontext bei der jeweiligen Sprache bestimmt wird.
Die Navigation innerhalb eines XML - Dokuments erfolgt über eine spezielle Art von Ausdrücken, *location path* genannt. Man unterscheidet relative und absolute Pfade. Ein relativer Pfad besteht aus einer Reihe von mehreren durch `"/"` getrennten Navigationsschritten. Jeder Schritt liefert eine Menge von Knoten, die anschließend als Kontext für den folgenden Navigationsschritt verwendet werden. Absolute Pfade beginnen mit dem Zeichen `/`, gefolgt von einem relativen Pfad. Sie beziehen sich immer auf den Wurzelknoten des Dokuments. Diese Pfadausdrücke ähneln Pfaden in Dateisystemen.
Ein Navigationsschritt besteht aus folgenden syntaktischen Teilen: einer Achse (*axis*), die die Beziehung zwischen den gesuchten Knoten und dem Knotextkonten im Baum beschreibt, einem Knotentest (*node test*), der den Knotentyp und Namen der gesuchten Knoten spezifiziert, sowie optional einer Anzahl von Prädikaten. Durch die Prädikate kann die Menge der ausgewählten Knoten weiter eingeschränkt werden. Die Achsenbezeichnung und der Knotentest werden durch einen doppelten Doppelpunkt (`::`) voneinander getrennt, Prädikate werden in eckige Klammern (`[]`) eingeschlossen. Im Ausdruck `child::para[position()=1]` beispielsweise bezeichnet `child` die Achse, `para` den Knotentest und `[position()=1]` ein Prädikat. Im Beispiel wird das erste `para`-Element der direkten Nachfolger des Kontextknotens ausgewählt. Tabelle 4.1 enthält die in Xpath definierten Achsen und deren Bedeutung. Die Bedeutung einiger Achsen, bezogen auf einen markierten Knoten, ist in Abbildung 4.7 graphisch dargestellt.
Der Knotentest besteht entweder aus dem Namen eines Elements des XML-Dokuments, dem Platzhalter `*` oder einer Testfunktion für einen bestimmten Knotentyp. Falls der Test aus einem Elementnamen besteht, liefert die Anfrage Elementknoten mit diesem Namen als Ergebnis,

Achse	Bedeutung
`child`	Kindelemente
`parent`	Direkter Vorgänger
`descendant`	Nachfolger
`ancestor`	Vorgänger
`following`	Alle nach dem gegenwärtigen Knoten im Dokument vorhandenen Knoten
`preceding`	Alle vor dem gegenwärtigen Knoten im Dokument vorhandenen Knoten
`following-sibling`	Geschwister des gegenwärtigen Knotens, die noch folgen
`preceding-sibling`	Geschwister, die vor dem gegenwärtigen Knoten angesiedelt sind
`attribute`	Die Attribute eines Knotens (nur bei Elementen)
`namespace`	Namensraumknoten des gegenwärtigen Knotens (nur bei Elementen)
`self`	Der gegenwärtige Knoten
`descendent-or-self`	Wie `descendent`, plus der gegenwärtige Knoten
`ancestor-or-self`	Wie `ancestor`, plus der gegenwärtige Knoten

Tabelle 4.1: Achsen bei XPath [BM00]

falls es sich um den Platzhalter `*` handelt, besteht das Ergebnis aus allen Elementknoten, die in der Achse enthalten sind. Bei Verwendung einer Testfunktion für Knotentypen werden nur Knoten dieses Typs ausgewählt. Beispielsweise liefert der Ausdruck `child::text()` alle Textknoten, die Kindknoten des Kontextknotens sind, das Ergebnis der Anfrage `child::para` dagegen besteht aus allen `para`-Elementen, die Kinder des Kontextknotens sind. Tabelle 4.2 zeigt verschiedene Beispiele für XPath-Ausdrücke und die durch diese ausgewählten Objekte.

4.3.1 Bewertung

XPath erlaubt die Auswahl von Teilmengen von Knoten eines XML - Baumes durch die Verwendung von Pfadausdrücken. Dabei ist es nicht notwendig, die exakte Tiefe eines Elements im Baum zu kennen. Die Suche kann durch die Achsen `descendant` oder `ancestor` zum Beispiel auf alle Nachfolger beziehungsweise Vorgänger ausgeweitet werden. Dies soll an folgendem Beispiel verdeutlicht werden.

```
\renewcommand{\baselinestretch}{1}
\normalfont
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Adressbuch SYSTEM="beispiel.dtd">
<Adressbuch>
```

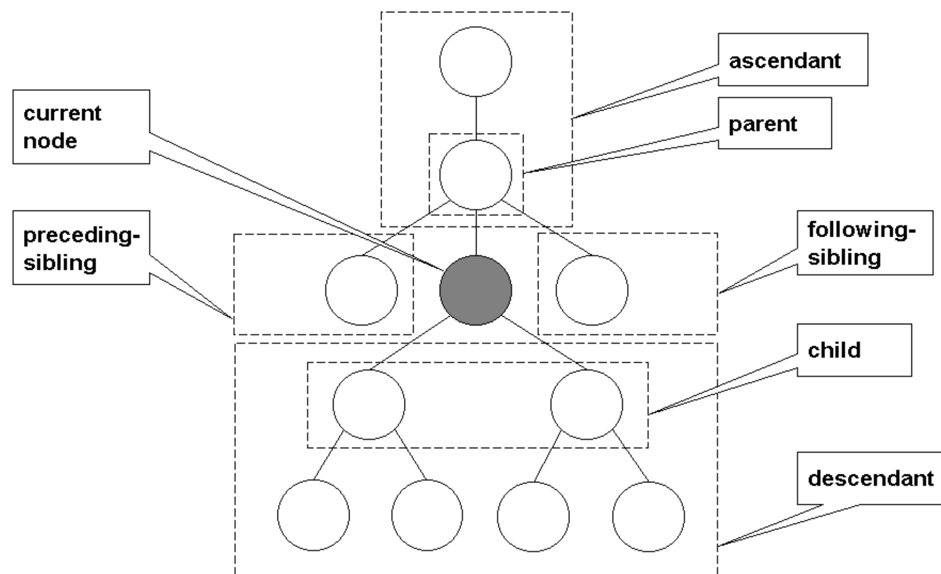


Abbildung 4.7: Achsen bei XPath [BM00]

```
    <Anschrift>
        <Name>Mustermann</Name>
        <Email>Mustermann@Musternet.net</Email>
        <Strasse>Beispielstrasse 7</Strasse>
    </Anschrift>
    <Anschrift>
        <Name>Ottonormalverbraucher</Name>
        <Email>Ottonormalverbraucher@Musternet.net</Email>
        <URL>http://www.musternet.net/ottonormalverbraucher</URL>
        <Adresse>
            <Strasse>Weg 14</Strasse>
        </Adresse>
    </Anschrift>
</Adressbuch>
\renewcommand{\baselinestretch}{1.24}
\normalfont
```

Das Beispieldokument enthält in beiden Anschriften das Element `Strasse`. In der ersten Anschrift ist es als Kindelement von `Anschrift` enthalten, in der zweiten Adresse dagegen befindet es sich eine Ebene tiefer als Kindelement von `Adresse`. Beide Elemente sind im Ergebnis der Anfrage `/descendant::Strasse` enthalten.
Die Ergebnismenge wird durch die Achsen gefiltert. Die Achse `child` sorgt dafür, dass bei der Auswertung nur direkte Nachfolger des Kontextknotens betrachtet werden, während bei Verwendung der Achse `descendant` alle Nachfolger des Kontextknotens betrachtet werden.
Durch die Verwendung von Prädikaten ist eine weitere Einschränkung des Ergebnisses möglich.

Ausdruck	Zugriff auf
`child::*`	Alle Kindelemente des gegenwärtigen Knotens
`attribute::*`	Alle Attribute des gegenwärtigen Knotens
`descendant::absatz`	Alle `absatz` - Nachfolger des gegenwärtigen Knotens
`/`	Ursprung des Dokumentenbaums
`/descendant::absatz`	Alle `absatz` - Elemente des Dokuments
`child::absatz[position()=1]`	Erstes `absatz` - Kindelement des gegenwärtigen Knotens
`child::absatz[position()=last()]`	Letztes `absatz` - Kindelement des gegenwärtigen Knotens
`child::text()`	Alle Textknoten, die Kinder des Kontextknotens sind
`child::node()`	Alle Kindknoten (egal welchen Typs) des Kontextknoten
`child::kapitel/descendant::absatz`	Alle `absatz` - Elemente der `kapitel` - Kindelemente
`child::absatz[position()=last()] [attribute::type="beispiel"]`	Das letzte Kindelement `absatz`, wenn es ein Attribut `type` mit dem Wert `beispiel` enthält

Tabelle 4.2: Beispiele für XPath - Ausdrücke

So werden beispielsweise durch das Prädikat `[attribute::type="beispiel"]` nur Elemente in die Ergebnismenge aufgenommen, die ein Attribut `type` mit dem Wert `beispiel` haben. Achsen und Prädikate können daher als eine Art Filter auf den XML - Baum angesehen werden. Eine Sortierung der Ergebnisse ist dagegen nicht möglich, da die Reihenfolge der Elemente in XML - Dokumenten von Bedeutung ist.

XPath ist eine reine Anfragesprache, das bedeutet, die Sprache enthält keinerlei Konstrukte, mit denen der XML - Baum manipuliert werden kann. So ist es nicht vorgesehen, Elemente in das Dokument einzufügen, Elemente zu löschen oder Elemente zu verändern. Daher sieht die Sprache auch keine Kopieroperation vor.

XPath unterscheidet zwar verschiedene Typen von Knoten, wie Attributknoten oder Elementknoten, die Typhierarchie kann jedoch nicht erweitert werden. Dasselbe gilt für die Beziehungen zwischen verschiedenen Knoten. Beispielsweise besteht zwischen zwei Elementknoten nicht dieselbe Art von Beziehung wie zwischen einem Elementknoten und seinen zugeordneten Attributknoten. XPath stellt eine feste Anzahl von Beziehungen, die Achsen, zur Verfügung,

es ist aber nicht möglich, weitere zu definieren. Beides muss jedoch in einer Anfragesprache für das semantische Netz im Projekt Föderal möglich sein.

4.4 XQL (XML Query Language)

Im September 1998 wurde die *XML Query Language (XQL)* der *XSL Working Group* des *World Wide Web Consotiums (W3C)* als Erweiterung der Systax von *XSL* vorgeschlagen. Ziel dieses Vorschlages war es, eine deklarative, speziell für XML - Dokumente geeignete Anfragesprache zu entwickeln.

Die Ausdrücke in XQL ähneln der Notation von Pfaden in Dateisystemen. Wie in XPath, siehe Abschnitt 4.3, werden XML - Dokumente in XQL als Baum interpretiert.

Ausdrücke werden in XQL immer in Beziehung zu einem Kontext ausgewertet. Dabei gibt es zwei Möglichkeiten. Der Ausdruck kann in Beziehung zur Wurzel des Dokumentes (*root context*) oder im Zusammenhang mit einem aktuellen Knoten (*current context*) ausgewertet werden. Dies entspricht einer Unterscheidung von absoluten und relativen Pfaden. Absolute Pfade beginnen immer mit einem Schrägstrich.

XQL kennt zwei Arten von Pfadoperatoren. Durch `/` werden die direkten Nachfolger eines Elements adressiert, während mit einem doppelten Schrägstrich (`//`) alle Nachfolger eines Elements in die Suche einbezogen werden. Mit Hilfe dieser Pfadoperatoren werden in XQL die Namen von XML - Elementen miteinander verknüpft. Die Namen von Attribute der Elemente werden durch das Zeichen `@` besonders gekennzeichnet. Der Bezeichner eines XML - Elementes kann durch den Platzhalter $*$ ersetzt werden, falls irgendein Element gemeint ist.

So werden durch die Anfrage `bookstore/*/title` alle `title` - Elemente selektiert, die Enkelelemente des Elements `bookstore` sind, während durch `bookstore/science/title` nur diejenigen `title` - Elemente ausgewählt werden, deren Vorgänger ein `science` - Element ist. Alle `title` - Elemente, die Nachfolger eines `bookstore` - Elementes sind, erhält man durch den Ausdruck `bookstore//title`. Die Attribute `style` aller im Dokument enthaltenen Elemente `book` können beispielsweise durch die Anfrage `//book/@style` ausgewählt werden.

Das Ergebnis einer Anfrage kann durch die Verwendung von Filtern weiter eingeschränkt werden. Filter sind in XQL bool'sche Ausdrücke und werden durch `[` und `]` gekennzeichnet. Sie können durch die bool'schen Operatoren `and`, `or` und `not` beliebig kombiniert werden. Zur Unterscheidung von XML - Elementen müssen die Operatoren durch das Zeichen `$` eingeschlossen werden. Durch den Ausdruck `//author[degree $and$ award]` werden zum Beispiel alle Elemente `author`, die sowohl mindestens ein Element `degree` als auch mindestens ein Element `award` enthalten, ausgewählt.

XQL unterscheidet wie auch XPath verschiedene Typen von Knoten. Dazu gehören Typen für Attribute, Elemente oder Kommentare. Der Typ eines Knotens kann durch die Anwendung

Ausdruck	Zugriff auf
author/first-name	Alle first-name - Elemente innerhalb eines author - Elements relativ zum Kontext
//author/first-name	Alle first-name - Elemente innerhalb eines author - Elements
author/*	Alle Element Nachfolger eines author - Elements (relativ zum Kontext)
price/@exchange	Alle Attribute exchange innerhalb eines Elements price bezüglich des Kontexts
//book[@style]	Alle Elemente book, die ein Attribut style enthalten
//author/address[@type='email']	Alle Elemente address innerhalb von author - Elementen, die ein Attribut type mit dem Wert email haben
//author/title[index()=0]	Erstes Element title jedes author - Elements

Tabelle 4.3: Beispiele für Ausdrücke in XQL

der Funktionen `nodeType()` ermittelt werden. Jede Struktur, die ein XML - Dokument enthalten kann, ist einem dieser Typen zugeordnet. Daher besteht keine Notwendigkeit, in XQL einen Mechanismus bereitzustellen, mit dem neue Typen definiert werden können.
Analog zu XPath kann auch in XQL - Anfragen Bezug auf die Reihenfolge der Elemente genommen werden. Dafür stellt XQL die Funktion `index()` zur Verfügung [RLS98, Rob99]. Tabelle 4.3 enthält einige Beispiele für XQL - Anfragen.

4.4.1 Bewertung

Mit Hilfe von XQL können Elementmengen innerhalb von XML - Dokumenten ausgewählt werden. Dies erfolgt durch Pfadausdrücke, die in ihrer Syntax Pfaden in Dateisystemen ähnlich sind. XQL enthält ein Konstrukt, mit dem alle Nachfolger eines Knotens in die Suche mit einbezogen werden können. Dies ist in der Semantik bestimmten regulären Ausdrücken in sofern ähnlich, als dass Pfade beliebiger Länge betrachtet werden.
Die Ergebnismenge einer Anfrage kann durch Filter eingeschränkt werden, eine Sortierung der Ergebnismenge dagegen ist nicht möglich, da die Reihenfolge der Elemente in XML von Bedeutung ist.
XQL ist eine reine Anfragesprache und enthält somit keine Sprachelemente, durch die ein XML - Dokument manipuliert werden kann. Mit anderen Worten: Elemente können mit XQL nicht eingefügt, gelöscht oder geändert werden. Daher ist auch keine Kopieroperation verfügbar.

Analog zu XPath unterscheidet XQL zwar mehrere Arten von Knoten, in XML besteht jedoch keine Notwedigkeit, das Typsystem zu erweitern. XQL unterscheidet hierarchische Relationen zwischen Elementknoten, die Nachfolgerrelation, und die Relation zwischen Elementknoten und Attributknoten. Es sind allerdings keine Sprachelemente in XQL enthalten, mit denen sich weitere Relationen definieren lassen.

4.5 Quilt

Quilt ist eine Anfragesprache für XML - Dokumente, die versucht Erfahrungen mit verschiedenen Sprachen miteinander zu kombinieren. So übernimmt Quilt die Notation für Pfadausdrücke aus XPath, siehe Abschnitt 4.3 und XQL, siehe Abschnitt 4.4, sowie die Kombination von Klauseln aus SQL [BL01].
Quilt benutzt XML - Dokumente, Teile von XML - Dokumenten oder Sammlungen von XML - Dokumenten sowohl als Eingabe als auch als Ergebnis von Anfragen.
Pfadausdrücke werden in Quilt auf die gleiche Art behandelt wie in XPath. So bestehen Pfadausdrücke aus mehreren Schritten, von denen jeder eine Bewegung durch das Dokument repräsentiert. Das Ergebnis eines Schrittes dient als Ausgangspunkt für den darauf folgenden Schritt. In jedem Schritt können Prädikate angewendet werden, um Elemente herauszufiltern, die das Prädikat nicht erfüllen. Die in Quilt zur Verfügung stehenden Pfadoperatoren sind in Tabelle 4.4 zusammengestellt. Durch den Ausdruck

```
document("zoo.xml")/chapter[2]//figure[caption = "Tree Frogs"]
```

zum Beispiel werden alle Abbildungen , Element `figure`, mit dem Titel `Tree Frogs` im zweiten Kapitel des Dokumentes `zoo.xml` selektiert.

Operator	Bedeutung
`.`	aktueller Knoten
`/`	direkte Nachfolger (Kinder) des aktuellen Knoten
`//`	alle Nachfolger des aktueller Knoten, entspricht der Hülle über `/`
`@`	bezeichnet ein Attribut
`[]`	Die Klammern umschließen einen bool'schen Ausdruck, der als Prädikat für eine Navigationsschritt dient
`[n]`	Sofern ein Prädikat aus einer ganzen Zahl besteht, beschreibt es die Ordinalposition eines Elementes innerhalb einer Liste von Elementen.
`document(string)`	bestimmt die Wurzel eines XML - Baums `string`
`->`	Derefernzierungsoperator

Tabelle 4.4: Pfadoperatoren in Quilt

Zusätzlich zu den Operatoren aus XQL definiert Quilt einen Dereferenzierungsoperator `->` für das XML - Sprachelement `IDREF`. Die Verwendung dieses Operators wird in folgendem Beispiel demonstriert.

```
document("zoo.xml")/chapter[title = "Frogs"]//figref/@refid -> /caption
```

Der Ausdruck ermittelt die Titel (`caption`) derjenigen Abbildungen, die durch ein `figref` Element in Kapiteln mit dem Titel `Frogs` referenziert werden.
Quilt enthält Konstruktoren für Elementknoten. Mit Hilfe dieser Konstruktoren können beliebige XML - Elemente erzeugt werden. Ein Konstruktor besteht aus einem Anfangs- und einem Endtag, die eine Liste von Ausdrücken einschließen, welche den Inhalt des Elements spezifizieren. Das Anfangstag eines Elements darf Attribut enthalten. Im folgenden Beispiel wird ein Element `emp` erzeugt, das zwei andere Elemente, `name` und `job` einschließt. Die Zeichenketten `$id`, `$n` und `$j` bezeichnen Variablen, die in anderen Teilen einer Anfrage gebunden werden [CRF00].

```
<emp empid = $id>
    <name> $n </name>
    <job> $j </job>
</emp>
```

4.5.1 Anfragen der Form FLWR

Für allgemeine Anfrage wird in Quilt ein `FOR`, `LET`, `WHERE`, `RETURN`(FLWR) Konstrukt verwendet. Im `FOR` - Teil einer Anfrage werden Variablen als Iteratoren über Listen von Tupeln gebunden. Dies ist mit Schleifen in Programmiersprachen vergleichbar. In der `LET` - Klausel werden Variablen an Ausdrücke gebunden, was eher einer Variablendeklaration in einer Programmiersprache entspricht.
Jedes der durch den `FOR/LET` - Teil einer Anfrage erzeugten Tupel kann durch einen Ausdruck in der `WHERE` - Klausel gefiltert werden. Prädikate in diesem Filter können durch `and`, `or` und `not` miteinander kombiniert werden. Variablen, die im `FOR/LET` - Teil gebunden werden, können zur Formulierung von Prädikaten verwendet werden.
Das Ergebnis einer Anfrage wird durch die `RETURN` - Klausel definiert. Sie wird für jedes Tupel, das das Prädikat in der `WHERE` - Klausel erfüllt, ausgeführt. Die `RETURN`- Klausel kann Konstruktoren enthalten.
Die Bedeutung der einzelnen Klauseln soll an den folgenden Beispielen deutlich werden. Die Beispiele basieren auf einem XML - Dokument `bib.xml`, das eine Liste von `book` - Elementen enthält. Jedes dieser `book` - Elemente enthält ein `title` - Element, ein oder mehrere `author` - Elemente, ein `publisher`- Element, ein `year` - Element, sowie ein `price` - Element.

```
FOR $b IN document("bib.xml")//book
WHERE $b/publisher = "Morgan Kaufmann" and $b/year = "1998"
RETURN $b/title
```

Dieses Beispiel erzeugt eine Liste der Titel derjenigen Bücher, die von `Morgan Kaufmann` 1998 veröffentlicht wurden. Im folgenden Beispiel wird eine Liste aller Verlage mit Durchschnittspreis der Bücher der einzelnen Verlage erzeugt.

```
FOR $p IN distinct (document("bib.xml")//publisher)
LET $a := avg(document("bib.xml")/book[publisher = $p]/price)
RETURN
    <publisher>
        <name> $p/text() </name>
        <avgprice> $a </avgprice>
    <publisher> SORTBY $p/text()
```

Die Ergebnisse einer Anfrage können durch Angabe einer `SORTBY` Klausel nach dem Konstruktor oder Pfadausdruck alpahbetisch sortiert werden [CRF00].

4.5.2 Filter

Mit Hilfe von `FILTER` Ausdrücken ist es möglich, Dokumente auf bestimmte Elemente zu reduzieren, wobei alle unerwünschten Elemente entfernt werden, die hierarchische Struktur der Elemente allerdings erhalten bleibt. Die Funktion des Operators soll an folgendem Beispiel erläutert werden.

```
<toc>
    document("cookbook.xml") FILTER
        //section | //section/title | //section/title/text()
</toc>
```

Der Operator `Filter` hat zwei Operanden. Der erste Operand liefert einen geordneten Wald von Knoten. Im Beispiel liefert `document("cookbook.xml")` alle Elemente im XML - Dokument `cookbook.xml`. Die einzelnen Elemente sind nicht durch eine virtuelle Wurzel miteinander verknüpft. Im zweiten Operanden wird definiert, welche Elemente erhalten blieben sollen. Die einzelnen Elemente werden durch einen senkrechten Strich `|` voneinander getrennt. Im Beispiel bleiben `section` - Elemente, `title` Elemente innerhalb der `section` - Elemente, sowie der Text dieser Elemente erhalten. Durch den Ausdruck im Beispiel wird ein Inhaltsverzeichnis erzeugt.

4.5.3 Bewertung

Quilt enthält durch das `FLWR` - Konstrukt ein Sprachelement, durch das eine Teilmenge eines XML - Dokumenten ausgewählt werden kann. Verwendung finden dabei Pfadausdrücke. Das Ergebnis einer Anfrage kann durch Verwendung von Prädikaten in der `WHERE` - Klausel analog zu SQL gefiltert werden. Zusätzlich zu dieser Möglichkeit, Filter zu definieren, enthält Quilt einen eigenen Filteroperator. Eine Sortierung der Ergebnisse ist ebenfalls möglich.
Mit Hilfe von Konstruktoren können zwar neue XML - Elemente erzeugt werden, diese werden allerdings nicht in das Dokument eingefügt, auf dem die Anfrage basiert, sondern stellen den Inhalt eines neuen Dokuments dar. Es ist mit anderen Worten mit Sprachelementen von Quilt nicht möglich, Elemente in ein bereits existierendes Dokument einzufügen. Quilt stellt keine Operationen für das Löschen oder Ändern von Elementen bereit. Da keine Elemente in eine bereits bestehende Struktur eingefügt werden können, ist auch keine Kopieroperation vorhanden.
Wie XPath und XQL unterscheidet Quilt verschiedene Arten von Knoten im XML - Baum, beispielsweise Elementknoten und Attributknoten. Knoten stehen in verschiedenen Relationen zu einander. Elementknoten stehen in hierarchischen Relationen zueinander, während Attribut in einer Art `attribut_von` - Beziehung zu den Elementknoten stehen, denen sie zuordnet sind. Es ist in Quilt weder möglich, weitere Knoten- noch zusätzliche Relationstypen zu definieren.

4.6 Zusammenfassung

Von den in diesem Kapitel betrachteten Anfragesprachen unterscheiden nur XPath, XQL, Quilt und OQL verschiedene Typen von Knoten, Lorel dagegen kennt nur OEM-Objekte. Es ist jedoch mit keiner dieser Sprachen möglich, weitere Typen zu definieren.
Die Beziehungen zwischen einzelnen Knoten sind bei XPath, XQL, Quilt und OQL mit Einschränkungen typisiert. XPath verwendet dafür vordefinierte Achsen, die aber nur hierarchische Beziehungen ausdrücken, XQL und Quilt verwenden für dieselbe Semantik Pfadausdrücke unterschiedlicher Syntax, OQL schließlich stellt Pfadausdrücke in der üblichen objektorientierten Schreibweise mit dem Punkt als Trennzeichen zur Verfügung. Reguläre Ausdrücke werden in OQL nicht benötigt. Lorel kennt dagegen im Datenmodell keine verschiedenen Arten von Beziehungen zwischen Objekten. Die Beziehungen zwischen Objekten werden in Lorel durch Kanten zwischen den jeweiligen Objekten ausgedrückt. Neue Typen von Beziehungen lassen sich in keiner der betrachteten Sprachen definieren.
Bei XPath, XQL und Quilt handelt es sich um reine Anfragesprachen, die daher auch keinerlei Konstrukte enthalten, mit deren Hilfe Dokumente verändert werden können. Quilt stellt zwar eine Operation zum Einfügen neuer Knoten zur Vefügung, dabei wird jedoch nicht das bearbeitete Dokument verändert, sondern ein neues angelegt. In OQL und Lorel ist es möglich

neue Objekte in bereits vorhandene Datenstrukturen einzufügen. Löschen und Ändern von Objekten ist aber in OQL nur durch Methoden der Objekte möglich, OQL stellt dafür keine eigenen Sprachelemente zur Verfügung. Das Ändern von Objekten ist mit Lorel möglich, dagegen existiert keine explizite Löschoperation. Objekte, die im Graphen nicht mehr erreicht werden können, gelten in Lorel als gelöscht. Keine der betrachteten Sprachen enthält eine Kopieroperation, die eine wirkliche Kopie der ausgewählten Datenstruktur anlegt. Lediglich mit OQL ist es mit Einschränkungen möglich, Kopien von Objekten anzulegen, siehe dazu Abschnitt 4.1.1.

Alle betrachteten Sprachen benutzen für die Formulierung von Anfragen Pfadausdrücke, die Syntax unterscheidet sich dabei von Sprache zu Sprache. Filteroperation stellen ebenfalls alle betrachteten Sprachen zur Verfügung. Eine Sortierung der Ergebnisse ist dagegen nur in Lorel, Quilt und OQL möglich. XQL, XPath und Quilt erlauben innerhalb von Pfadausdrücken eine Art von regulären Ausdrücken, während OQL keine regulären Ausdrücke benutzt. Nur Lorel verwendet wirklich reguläre Ausdrücke.

Die Zusammenfassung ist in Tabelle 4.5 tabellarisch dargestellt. Dabei bedeutet ein ausgefüllter Kreis (•) vollständige Unterstützung, eine Unterstützung mit Einschränkung ist durch einen Kreis (○) gekennzeichnet und keine Unterstützung wird durch einen Strich (-) markiert.

Anforderung	**XPath**	**XQL**	**Quilt**	**OQL**	**Lorel**
Datenmodell					
Unterscheidung von Knotentypen	•	•	•	•	-
Definition neuer Knotentypen	-	-	-	-	-
Beziehungen zwischen Knoten sind typisiert	○	○	○	○	-
Definition neuer Typen von Relationen	-	-	-	-	-
Manipulationsoperationen					
Einfügen	-	-	○	•	•
Löschen	-	-	-	○	○
Änderungen	-	-	-	○	•
Kopieroperation	-	-	-	○	-
Anfragen					
Pfadausdrücke	•	•	•	•	•
Reguläre Ausdrücke in Pfaden	○	○	○	-	•
Filterung möglich	•	•	•	•	•
Sortierung der Ergebnisse	-	-	•	•	•

Tabelle 4.5: Tabellarische Zusammenfassung

Kapitel 5

Entwurf einer Anfragesprache

In diesem Kapitel wird der Entwurf einer Anfragesprache für das semantische Netz im Projekt Föderal vorgestellt. Mit Hilfe der Sprache soll es möglich sein, Anfragen an das Netz zu stellen, sowie das semantische Netz zu manipulieren. Die Syntax der *Semantic Net Query Language (SNQL)* orientiert sich an SQL, beziehungsweise OQL.
Dies hat sich als sinnvoll erwiesen, da die bereits existierende Sprache Lorel [AQM+97] einen großen Teil der in Kapitel 3 aufgestellten Anforderungen erfüllt, so dass syntaktische Konstrukte aus Lorel mit kleinen Änderungen und Erweiterungen übernommen werden konnten. Außerdem ist diese Syntax vielen Anwendern vertraut. Die vollständige Syntax der Sprache ist in Anhang B enthalten.

5.1 Pfadausdrücke

Die Anfragesprache für das semantische Netz verwendet für die Navigation innerhalb der Datenstruktur Pfadausdrücke. Diese sind der Syntax von Lorel entnommen. Die angepaßte Syntax von Pfadausdrücken in SNQL ist in Tabelle 5.1 dargestellt. Im Unterschied zu Lorel bestehen in SNQL Pfadausdrücke, abgesehen von Variablen, ausschließlich aus den Namen von Relationstypen, während in Lorel Pfadausdrücke stets mit dem Namen eines Objektes oder einer Referenz auf ein Objekt beginnen. In Lorel ist der Knoten, auf den sich Pfadausdrücke beziehen Teil, des Ausdrucks, vergleiche dazu Abschnitt 4.2.1.
Die Sprache enthält wie Lorel die Möglichkeit, generelle Pfadausdrücke zu notieren. Dabei werden reguläre Ausdrücke verwendet. Hierbei wird als Alphabet Σ, auf dem die regulären Ausdrücke basieren, die Menge der Namen der Relationstypen verwendet.

Definition 5.1.1
Falls p_1 und p_2 Pfadausdrücke sind, dann sind auch $p_1.p_2$, $p_1|p_2$, $(p_1)?$, $(p_1)*$ und $(p_1)+$ Pfadausdrücke.

Durch $p_1|p_2$ wird die Disjunktion der Pfadausdrücke formuliert, das Fragezeichen kennzeichnet einen optionalen Ausdruck und durch den Stern *, sowie das Pluszeichen + wird das mehrfache Vorkommen eines Ausdrucks bezeichnet.
Innerhalb der Pfadausdrücke dürfen Platzhalter verwendet werden. Beliebige Zeichenketten können durch das Zeichen % abgekürzt werden, wobei der Platzhalter nur am Ende beziehungsweise am Anfang einer Zeichenkette stehen darf. Er steht damit also für Präfix- , oder Suffixausdrücke. Bei der Verwendung von Variablen innerhalb von Pfadausdrücken, muss dem Variablennamen das Zeichen `&` vorausgehen, um eine Unterscheidung zwischen Variablennamen und Namen von Relationstypen zu ermöglichen. Die Verwendung von Variablen wird im folgenden Abschnitt erläutert. Der Platzhalter `#` steht in Pfadausdrücken für einen beliebigen Relationstyp.
Mit Hilfe der Funktion `inverse()` ist es möglich, Kanten entgegen der Pfeilrichtung entlang zu navigieren.
Beispiele für Pfadausdrücke sind den folgenden Abschnitten zu entnehmen.

PathExpression	=	*PathComponent* {„.“ *PathComponent* }.
PathComponent	=	*VariableExpression* | *LabelExpression*
		| *InverseLabelExpression* | *RegularPathExpression.*
VariableExpression	=	„&“ *StringLiteral.*
LabelExpression	=	„#“ | [„%“] *StringLiteral* [„%“].
InverseLabelExpression	=	„inverse (“ *LabelExpression* „)“.
RegularPathExpression	=	„(“ *RegularPathExpression* „|“ *RegularPathExpression* „)“
		| *RegularPathExpression* [„*“| „+“| „?“]
		| „(“ *PathExpression* „)“.

Tabelle 5.1: Syntax von Pfadausdrücken

5.2 Auswahlanfragen

Auswahlanfragen in der Sprache für das semantische Netz folgen der aus SQL und OQL bekannten `select from where` Form. Die `select`-Klausel spezifiziert die auszuwählenden Objekte mit Hilfe eines Pfadausdruckes.
Durch die optionale `from`-Klausel können Variablen an Pfadausdrücke gebunden werden. Diese Variablen können wieder in anderen Pfadausdrücken verwendet werden, z.B. in der `select`- oder `where`-Klausel. Innerhalb von Pfadausdrücken werden Variablen durch das Zeichen `&` vor dem Variablennamen gekennzeichnet, vergleiche Beispiel 5.2.1.
Durch Verwendung des `where`-Teils der Anfrage kann das Anfrageergebnis durch Prädikate eingeschränkt werden. Die `where`-Klausel ist als bool’scher Ausdruck aufgebaut, d.h. die ein-

zelnen Prädikate können durch die bool'chen Operatoren `and` und `or` mit einander verknüpft werden. Die Negation eines Ausdruck ist durch den Operator `not` möglich.
Auf den Wert eines Knoten kann innerhalb eines Prädikats durch des Schlüsselwort `$value`, abgekürzt durch `@` zugegriffen werden. Man beachte, dass das Zeichen `@` in Lorel bei der Variablenbindung verwendet wird. Für Vergleiche zwischen Werten stehen zusätzlich die folgenden Prädikate Verfügung. Dabei überprüft das Prädikat `präfix` ob sein linker Operand ein Präfix des rechten Operanden darstellt, das Prädikat `postfix` überprüft auf Suffixe, das Prädikat `infix` schließlich ist eine Kombination aus beiden. Zeichenketten müssen bei Vergleichen durch das Zeichen `$` gekennzeichnet werden. Vergleiche zwischen Pfadausdrücken, das bedeutet von Objekten, und Werten sind zwar syntakisch möglich, machen aber keinen Sinn, da Objekte eine Sammlung von Werten beinhalten und die Vergleichsoperation zwischen einem Einzelwert und einer Wertemenge undefiniert ist.
Die Syntax, siehe Tabelle 5.2, soll im Folgenden an Beispielen anhand von Abbildung 5.1 erläutert werden. Die Beschriftungen an den Kanten geben in der Abbildung den Namen der Relatationstypen an.

<table>
<tr><td>SfwStatement</td><td>=</td><td>„select“ PathExpression
[„from“ FromClause]
[„where“ WhereClause]
[„order by“ ValueKey].</td></tr>
<tr><td>FromClause</td><td>=</td><td>PathExpression [„as“] StringLiteral
{„,“ PathExpression [„as“] StringLiteral }.</td></tr>
<tr><td>Predicate</td><td>=</td><td>Operand CompareOperation Operand
| „(“ Predicate („and“| „or“) Predicate „)“
| „(“ „not (“ Predicate „))“.</td></tr>
<tr><td>CompareOperation</td><td>=</td><td>„ < “ | „ = “ | „ > “ | „ <= “ | „ >= “ | „ <> “
| „infix“ | „praefix“ | „postfix“.</td></tr>
<tr><td>Operand</td><td>=</td><td>PathExpression [ValueKey] | „$“ AtomicValue.</td></tr>
<tr><td>ValueKey</td><td>=</td><td>„$value“ | „@“.</td></tr>
</table>

Tabelle 5.2: Syntax von Auswahlanfragen

Beispiel 5.2.1
In diesem Beispiel wird das in Abbildung 5.1 durch Einfärbung hervorgehobene Teilnetz ausgewählt.

```
select &R.((instanz) | (instanz.komp))
from root_rel as R
```

Die Bearbeitung der Anfrage beginnt am Einstiegspunkt in das semantische Netz. Von dort aus folgt man der Relation `root_rel`*, sowie den Pfaden* `instanz.komp` *und* `instanz`*. Die*

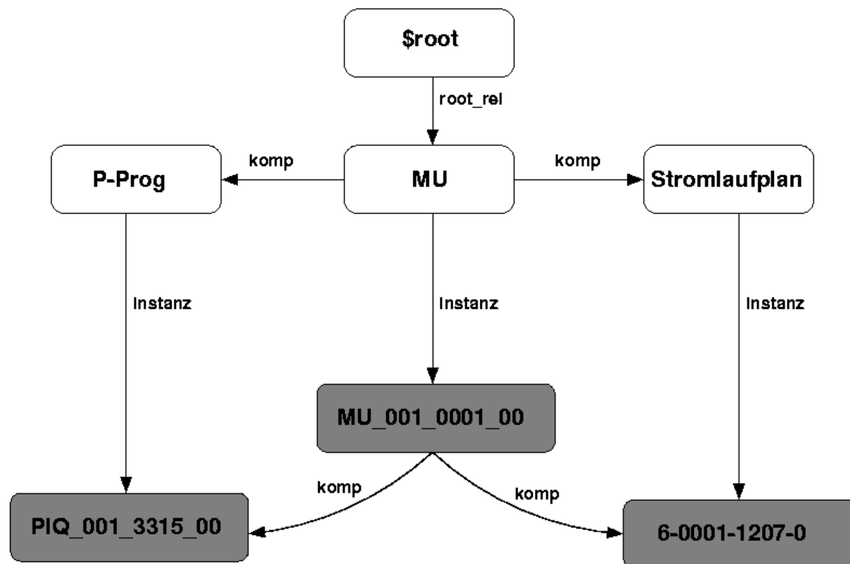


Abbildung 5.1: Beispiel eines semantischen Netzes

Variable `R` *stellt in der Anfrage lediglich eine Abkürzung für den Pfad* `root_rel`*. Die Disjunktion der Pfade führt zur Vereinigung der Ergebnismengen.*
Der Ausdruck `((instanz)|(instanz.komp))` *lässt sich auch äquivalent durch den regulären Ausdruck* `instanz.(komp)?` *formulieren. Dieser wird dann intern zu den Pfaden* `instanz` *und* `instanz.komp` *expandiert.*

```
select &R.instanz.(komp)?
from root_rel as R
```

Beispiel 5.2.2
Die folgende Anweisung selektiert in Abbildung 5.1 den Knoten mit dem Namen `PIQ_001_3315_00`.

```
select &K
from root_rel.komp.instanz as K
where (&K$value = $PIQ_001_3315_00)
```

Dabei dient das Prädikat `&K$value =$PIQ_001-3315_00` *der Überprüfung des Knotenwerts. Dabei stellt der Ausdruck* `$value` *ein Schlüsselwort dar, mit dem auf den Wert eines Knotens zugegriffen werden kann.*

Beispiel 5.2.3
Es ist möglich, das Ergebnis einer Anfrage zu sortieren. In diesem Beispiel werden die Knoten im Ergebnis der Anfrage aus Beispiel 5.2.1 nach dem Wert der Knoten sortiert.

```
select &V
from root_rel.instanz.(komp)? as V
where (&V$value = $PIQ_001_3315_00)
order by &V$value
```

5.3 Änderungsanweisungen

In diesem Abschnitt werden syntaktische Elemente von SNQL vorgestellt, mit denen es möglich ist, die Struktur eines bereits existierenden Netzes zu verändern. Dazu gehören das Einfügen und Entfernen von Knoten und Kanten, im weiteren Relationen genannt, sowie das Anlegen von Relationstypen und Änderungen des Wertes von Knoten. Die Syntax von Änderungsanweisungen ist Tabelle 5.3 zu entnehmen.

SetValueStatement	=	„set value (“ *SfwStatement* „) to“ *StringLiteral*.
InsertStatement	=	„insert (“ *StringLiteral* [„,“ *StringLiteral*] „)“.
DeleteStatement	=	„delete (“ *SfwStatement* „)“.
UpdateStatement	=	„update“ (*SfwStatement* | *InsertStatement* | *RootKey*)
		[„+ =“| „− =“]
		(*SfwStatement* | *InsertStatement* | *RootKey*).
CreateStatement	=	„create relation type“ *StringLiteral*.
RootKey	=	„$root“ | „∼“.

Tabelle 5.3: Syntax von Änderungsanweisungen

5.3.1 Einfügen von Knoten und Relationen

SNQL bietet durch die `insert`-Anweisung die Möglichkeit Knoten in ein bereits vorhandenes semantisches Netz einzufügen. Die Anweisung benötigt den Namen des Knotens und optional den Typen des Knotens als Parameter. Falls der Typ des Knotens nicht angegeben wird, wird ein Knoten vom Standardtyp angelegt.

Beispiel 5.3.1
Die folgende Anweisung erzeugt einen Knoten vom Typ `StringNode` *mit dem Wert* `PIQ_001_3315_12`.

```
insert (PIQ_001_3315-12, StringNode)
```

Relationen können zwischen bestehenden Knoten und Kanten eingefügt werden. Dafür muss die Menge der Anfangs- und die Menge der Endknoten angegeben werden. Dies erfolgt in SNQL entweder durch eine `insert`- oder `select`-Anweisung. Zusätzlich steht das Schlüsselwort

`$root`, abgekürzt durch $\sim$, zur Verfügung, mit dem der Einstiegspunkt in des semantische Netz referenziert werden kann, siehe Beispiel 5.3.2. Dieser Anweisung folgt, durch einen Punkt getrennt, der Name des Typs der neu anzulegenden Relation. Zwischen dem Namen des Relationstyps und der Definition der Menge der Endknoten steht schließlich der Operator `+=`. Sofern die Mengen nicht die Kardinalität eins besitzen, wird von jedem Knoten aus der Menge der Anfangsmenge eine Relation zu jedem Knoten aus der Menge der Endknoten eingefügt, vergleiche Beispiel 5.3.4. Formal ausgedrückt heisst das:

Definition 5.3.1
Sei A die Menge der Anfangsknoten und E die Menge der Endknoten, dann erzeuge für alle $a \in A$ eine Relation von a nach e mit $e \in E$.

Falls eine Relation an einer schon bestehenden Kante enden oder beginnen soll, so muss zusätzlich vermerkt werden, dass in diesem Fall nicht der Knoten am Ende des angegebenen Pfades als Ende der Relation angesehen wird, sondern die letzte Kante des Pfades, siehe Beispiel 5.3.5. Die Syntax in SNQL soll an folgenden Beispielen erläutert werden.

Beispiel 5.3.2
Das folgende Beispiel fügt eine Relation vom Typ `meine_rel` *zwischen dem Einstiegspunkt des Netzes und dem Knoten* `MU_250_0001_00` *ein.*

```
update $root.meine_rel +=
select &M
from root_rel.instanz as M
where (&M$value = $MU_250_0001_00)
```

Abbildung 5.2 zeigt ein semantisches Netz vor und nach dem Einfügen der Relation vom Typ `meine_rel`.

(a) vorher **(b)** nachher

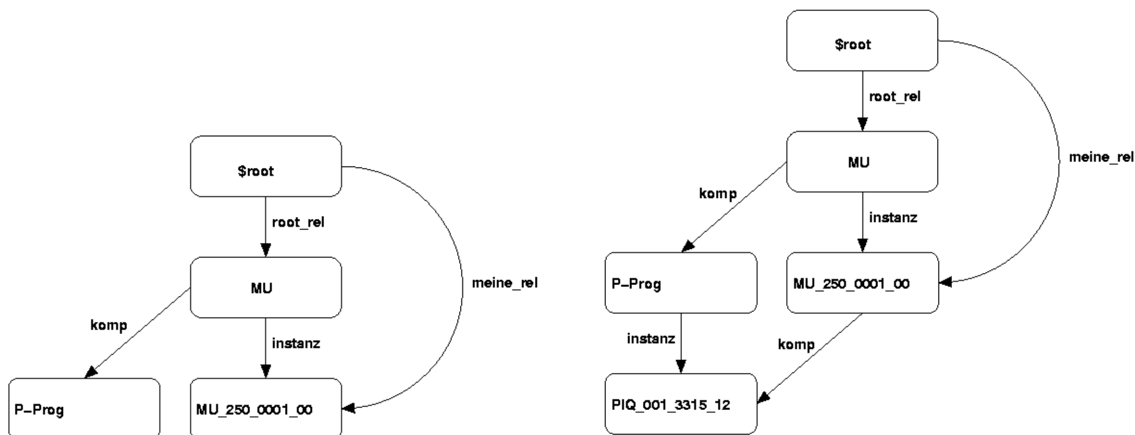
Abbildung 5.2: Einfügen einer Relation in ein semantisches Netz

Beispiel 5.3.3

Hier werden zwei Relationen in die Abbildung 5.3(a) eingefügt. Zuerst wird der Knoten mit dem Wert `PIQ_001_3315_12` *eingefügt und über eine Relation vom Typ* `komp` *mit dem Knoten* `MU_250_0001_00` *verbunden.*
Anschließend wird der neue Knoten durch eine Relation vom Typ `instanz` *mit dem Knoten* `P-Prog` *verbunden. Das daraus resultierende Nezt ist in Abbildung 5.3(b) dargestellt.*

```
update
( select &M
  from root_rel.instanz as M,
  where (&M$value = $MU_250_0001_00) ).komp +=
insert (PIQ_001_3315_12, StringNode)

update
( select &X
  from root_rel.komp as X
  where (&X$value = $P-Prog) ).instanz +=
select &Y
from root_rel.instanz.komp as Y
where (&Y$value= $PIQ_001_3315_12)
```



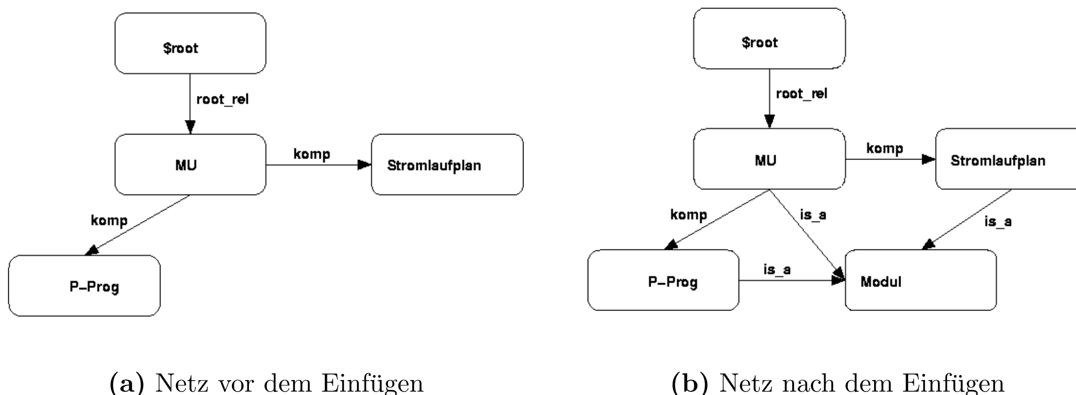
(a) Netz vor dem Einfügen **(b)** Netz nach dem Einfügen

Abbildung 5.3: Einfügen eines Knotens und zweier Relationen

Beispiel 5.3.4

In diesem Beispiel werden in Abbildung 5.4(a) drei Relationen vom Typ `is_a` *zwischen den Knoten* `MU`, `P-Prog`, `Stromlaufplan` *und dem Knoten* `Modul` *angelegt. Das resultierende Netz ist in Abbildung 5.4(b) zu sehen.*

```
update
( select root_rel.(komp)? ).is_a += insert (Modul, StringNode)
```



(a) Netz vor dem Einfügen **(b)** Netz nach dem Einfügen

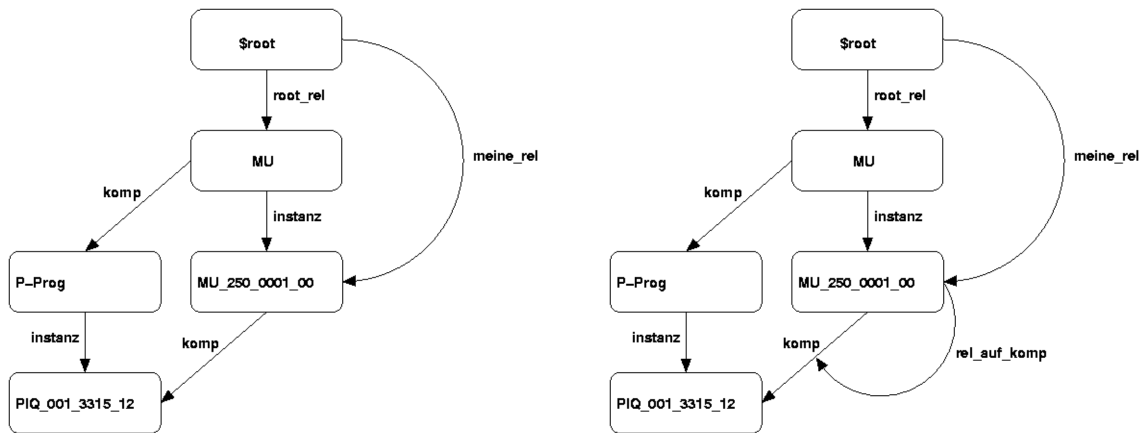
Abbildung 5.4: Einfügen einer Menge von Relationen

Beispiel 5.3.5
In diesem Beispiel wird eine Relation vom Knoten `MU_250_0001_00` aus zu der bereits bestehenden Relationen `komp` angelegt. Die Angabe `(relation)` legt im Pfadausdruck `&Y(relation)` fest, dass nicht der Knoten am Ende des Ausdrucks, sondern die letzte Relation des Pfadausdrucks als Ende der neuen Kante interpretiert wird. Das Beispiel ist in Abbildung 5.5 graphisch dargestellt.

```
update
( select &X
  from root_rel.instanz as X
  where (&X$value = $MU_250_0001_00).rel_auf_komp +=
select &Y(relation)
from root_rel.instanz.komp as Y
where (&Y$value = $PIQ_001_3315_12)
```

5.3.2 Löschen von Knoten und Relationen

Knoten können mit Hilfe der `delete`-Anweisung aus einem semantischen Netz entfernt werden. Beim Löschen eines Knotens werden alle Relationen, an denen dieser Knoten teilnimmt, mit gelöscht. Die Menge der zu löschenden Knoten wird mit Hilfe einer `select`-Anweisung spezifiziert.



(a) Netz vor dem Einfügen

(b) Netz nach dem Einfügen

Abbildung 5.5: Einfügen einer Relationen auf einer Relation

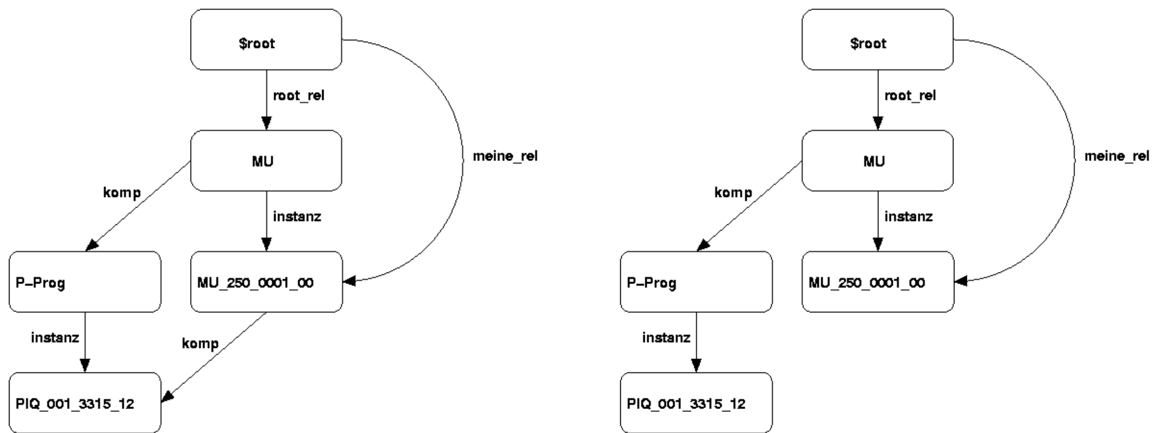
Beispiel 5.3.6
In diesem Beispiel wird der Knoten `PIQ_001_3315_12`*, der im vorigen Abschnitt in das Netz eingefügt wurde, siehe Abbildung 5.3(b), wieder aus dem Netz entfernt. Dabei werden die Relationen vom Typ* `instanz` *und* `komp` *entfernt.*

```
delete ( select &M
         from root_rel.instanz.komp as M,
         where (&M$value = $PIQ_001_3315_12) )
```

Die Anweisung zum Löschen von Relationen hat große Ähnlichkeit mit der Anweisung für das Einfügen von Relationen. Der einzige Unterschied liegt im Operator. Statt des Operators `+=` wird `-=` verwendet.

Beispiel 5.3.7
In der folgenden Anweisung wird die Relation vom Typ `komp` *zwischen den Knoten mit den Werten* `MU_250_0001_00` *und* `PIQ_001_3315_12` *in Abbildung 5.6(a) entfernt.*

```
update
( select &M
  from root_rel.instanz as M
  where (&M$value = $MU_250_0001_00) ).komp -=
select &P
from root_rel.komp.instanz as P
where (&P$value = $PIQ_001_3315_12)
```



(a) Netz vor dem Löschen

(b) Netz nach dem Löschen

Abbildung 5.6: Löschen einer einzelnen Relation

5.3.3 Änderung des Werts von Knoten

Der Wert eines Knoten wird in SNQL mit Hilfe einer Anweisung `setvalue` geändert. Diesem Schlüsselausdruck folgt eine Auswahlanweisung der Form `select from where`, siehe Abschnitt 5.2. Anschließend folgt das Schlüsselwort `to` mit dem neuen Namen.

Beispiel 5.3.8
In diesem Beispiel werden alle in Abbildung 5.1 durch Einfärbung hervorgehobenen Knoten mit dem Wert Teilnetz versehen.

```
set value
( select &R.instanz.(komp)?
  from root_rel as R )
to Teilnetz
```

5.3.4 Erzeugen von Knoten- und Relationstypen

Mit Hilfe von SNQL ist es möglich, neue Relationstypen zu definieren, die anschließend sofort verwendet werden können, um Relationen in ein semantisches Netz einzufügen. Die Anweisung beginnt mit dem Schlüsselwort `create relation type`, gefolgt vom Namen des neuen Relationstyps.

Beispiel 5.3.9
Die folgende Anweisung erzeugt einen neuen Relationstyp mit Namen `mein_neuer_relations_typ`.

```
create relation type mein_neuer_relations_typ
```

Knotentypen werden in der FIA durch Java-Klassen repräsentiert. Für jeden unterstützten Knotentyp existiert eine Klasse. Daraus folgt, dass beim Anlegen eines neuen Knotentyp auch eine passende Java-Klasse erstellt werden muss. Dies durch die Sprache zu unterstützen, erschien daher zu aufwendig. Neu erstellte Klassen müssen außerdem kompiliert und in das der Packagestruktur entsprechende Verzeichnis kopiert werden, bevor sie verwendet werden können.

5.4 Kopieren von Teilnetzen

SNQL enthält zwei Varianten eines Kopierbefehls, mit dem ganze Teilnetze kopiert werden können.
Die erste Variante kopiert alle Knoten sowie alle Relationen die zwischen diesen Knoten bestehen. Die Kopie des Subnetzes hat somit keine Verbindung zum übrigen Teil des semantischen Netzes. Im Gegensatz dazu kopiert die zweite Variante der Anweisung diejenigen Relationen mit, die die Kopiervorlage mit dem übrigen semantischen Netz verknüpft. Die Werte der Knoten werden in der Kopie mit dem Präfix `copy_of` versehen.
Die Funktion der Anweisungen soll an folgenden Beipielen dargestellt werden.

Beispiel 5.4.1
Mit Hilfe der folgenden Anweisung soll in Abbildung 5.7 das Teilnetz aus den Knoten `MU_001_0001_00`, `PIQ_001_3315_00` *sowie* `6-0001-1207-0` *und der Relation vom Typ* `komp` *zwischen diesen Knoten kopiert werden. Die Vorlage ist in Abbildung 5.7 durch Einfärbung hervorgehoben.*

```
copy
( select &M.(komp)?
  from root_rel.instanz as M
  where &M$value = $MU_001_0001_00 )
```

Falls auch alle Relationen, die aus dem Teilnetz herausführen, mitkopiert werden sollen, muss die Anweisung durch das Schlüsselwort `relation` ergänzt werden.

Beispiel 5.4.2
Die folgende Anweisung fügt in Abbildung 5.7 zusätzlich die Relationen zwischen der Kopie des Knotens `MU_001_0001_00` *und dem Knoten Knoten* `MU` *sowie zwischen der Kopie des Knotens* `PIQ_001_3315_00` *und dem Knoten* `P-Prog`*. ein.*

```
copy relation
( select &M.(komp)?
```

```
from root_rel.instanz as M
where &M$value = $MU_001_0001_00 )
```

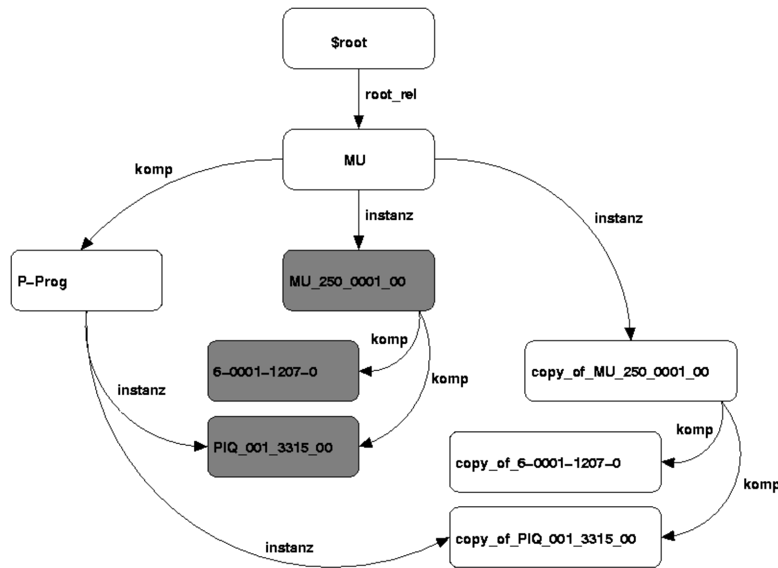


Abbildung 5.7: Beispiel eines semantischen Netzes - Kopieren eines Teilnetzes. Die Kopiervorlage ist grau eingefärbt. Die kopierten Knoten sind am Präfix `copy_of` zu erkennen.

5.5 Unterschiede von SNQL gegenüber Lorel

In diesem Abschnitt sollen die wesentlichen Unterschiede zwischen der in diesem Kapitel vorgestellten Semantic Net Query Language (SNQL) und Lorel diskutiert werden.
Das OEM-Modell, das Lorel zugrunde liegt, kennt im Gegensatz zum Datenmodell in Föderal keine Typisierung der Knoten. Daher muss SNQL Knoten im semantischen Netz nach deren Typ unterscheiden. Dies zeigt sich an der Einfügeoperation in SNQL dadurch, dass der Typ des anzulegenden Knotens als Parameter angegeben werden muss.
Lorel besitzt keine explizite Löschanweisung für Knoten. Ein Knoten gilt als gelöscht, sofern er durch keine Kante mehr mit dem OEM-Graphen verbunden ist. Ein Knoten wird in Lorel also durch Entfernen aller Kanten des Knotens gelöscht. Der Nachteil dabei ist, dass dies sehr aufwendig wird, falls der zu löschende Knoten durch viele Kanten mit anderen Knoten verbunden ist. SNQL versucht dieses Verhalten durch eine explizite Löschoperation anzupassen. So werden beim Löschen eines Knoten alle Relationen entfernt, an denen der Knoten beteiligt ist.
Im Unterschied zu Lorel ist es mit Hilfe von SNQL möglich, Teilgraphen zu kopieren. Dafür besitzt SNQL eine eigene Kopieroperation. Dies ist notwendig, da es in Lorel nicht möglich ist, ein Kopieren durch die Kombination von Auswählen und Einfügen zu ersetzen. Bei einem sol-

chen Vorgehen werden nur die Knoten kopiert. Die Kopieroperation in SNQL dagegen kopiert die Relationen innerhalb des Teilnetzes immer mit.

Kapitel 6

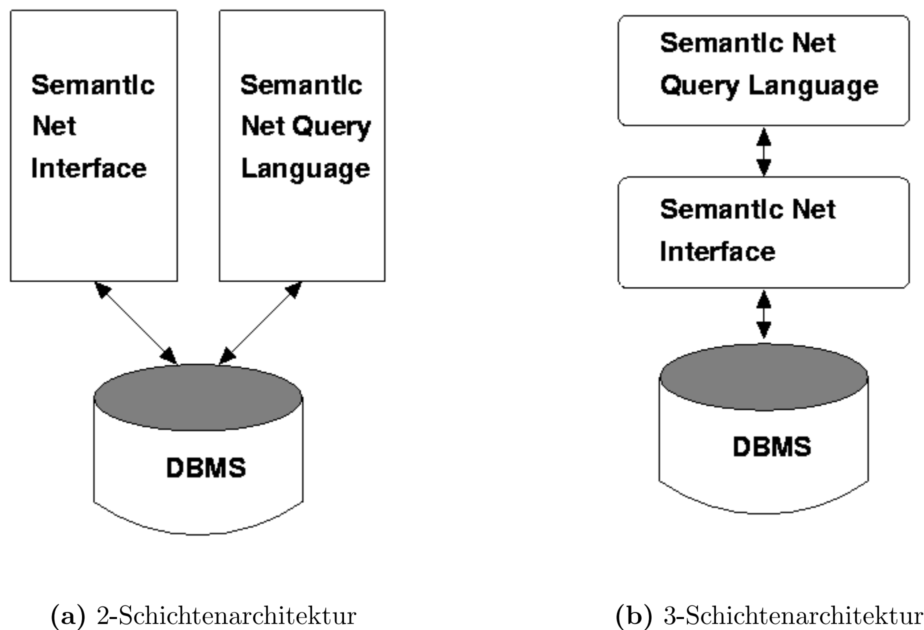
Implementierung

Im Folgenden wird die Realisierung des Sprachentwurfs aus Kapitel 5 beschrieben. Die Darstellung gliedert sich in drei Teile. Zuerst werden verschiedene Realisierungsalternativen diskutiert, bevor die Erzeugung eines geeigneten Parser und schließlich die Umsetzung der einzelnen Sprachelemente dargestellt wird. Der Interpreter wurde in Java implementiert. Das Ende des Kapitels bildet ein Abschnitt über einen erfolgten Test des Interpreters. Der Interpreter unterstützt zwei Formen der Bedienung. Erstens existiert eine Kommandozeilenversion des Interpreters, bei der der Benutzer die SNQL-Anweisungen über eine Befehlszeile eingibt. Außerdem kann der Interpreter über eine Methode von einem Java-Programm aus aufgerufen werden.

6.1 Realisierungsalternativen

Im Allgemeinen ergeben sich zwei Alternativen für die Realisierung einer Datenbankanwendung. Einmal besteht die Möglichkeit, die Anwendung direkt auf der Datenbank arbeiten zu lassen. Die zweite Alternative besteht darin, eine Schnittstelle (*engl.: interface*) zwischen die Anwendung und das Datenbanksystem zu setzen.
Auf das Projekt Föderal übertragen ergeben sich die in Abbildung 6.1 dargestellten Alternativen. Abbildung 6.1(a) zeigt eine Schichtenarchitektur, bei der die Anfragesprache SNQL direkt auf das Datenbanksystem zugreift, möglicherweise existiert parallel zur Sprache eine Schnittstelle für weitere Anwendungen, in der Abbildung als *semantic net interface* bezeichnet. Die alternative Schichtenarchitektur fügt die Schnittstelle als zusätzliche Schicht zwischen Anfragesprache und Datenbank ein, siehe Abbildung 6.1(b).
Die 2-Schichtenarchitektur aus Anwendung und Datenbank bietet einen Laufzeitvorteil gegenüber der 3-Schichtenarchitektur mit Schnittstelle zwischen Datenbank und Anwendung. Der Laufzeitvorteil entsteht einmal dadurch, dass im Vergleich zur alternativen Architektur die Methoden- beziehungsweise Funktionsaufrufe der Schnittstelle eingespart werden können.

Zudem bieten viele Datenbanksysteme einen Optimierer, der Anfragen in Bezug auf die Laufzeit optimiert.
Andererseits kann bei der 3-Schichtenarchitektur das Datenbanksystem ohne Neuimplementierung der Anwendung ausgetauscht werden. Die Schnittstelle muss dann natürlich angepasst werden.
Für die Implementierung der Anfragesprache in dieser Arbeit wurde aus folgenden Gründen eine 3-Schichtenarchitektur gewählt. Erstens stellt das verwendete objektorientierte Datenbanksystem keine Anfragesprache, wie OQL, zur Verfügung, so dass der Laufzeitvorteil einer 2-Schichtenarchitektur entfällt, da die Datenbankzugriffe durch Methodenaufrufe realisiert werden müssen.
Zweitens existiert bereits eine Schnittstelle für das semantische Netz, die auch die Konsistenz des Netztes sicherstellt. So muss die Konsistenz des semantischen Netzes bei der Implementierung der Sprache nicht zusätzlich beachtet werden. Außerdem ist geplant, im weiteren Verlauf des Projektes das objektorientierte durch ein relationales Datenbanksystem zu ersetzen.



(a) 2-Schichtenarchitektur **(b)** 3-Schichtenarchitektur

Abbildung 6.1: Die Abbildung zeigt zwei Realisierungsalternativen für eine Schichtenarchitektur aus Datenbank, Semantic Net Interface und der Semantic Net Language

6.2 Erzeugung eines Parsers

Bevor eine Anweisung der Anfragesprache ausgeführt werden kann, muss sie auf korrekte Syntax hin untersucht werden. Diese Syntaxanalyse oder hierarchische Analyse wird von Parsern

durchgeführt. Parser arbeiten im Allgemeinen nach dem Schema in Abbildung 6.2.
Der Syntaxanalyse geht eine lexikalische Analyse durch einen Scanner voraus. Der Scanner liest die Eingabe und gruppiert sie zu lexikalischen Symbolen, beispielsweise Zeichenketten, Zahlen oder Bezeichner, die er dann an den Parser weiterreicht. Dieser versucht, eine Ableitung aus der Grammatik der Sprache für die Eingabe zu finden. Diese Ableitung wird hierarchisch als Strukturbaum oder auch Parse-Baum, dargestellt.
Man unterscheidet zwischen Top-Down- und Bottom-Up-Syntaxanalyse. Bei der Top-Down-

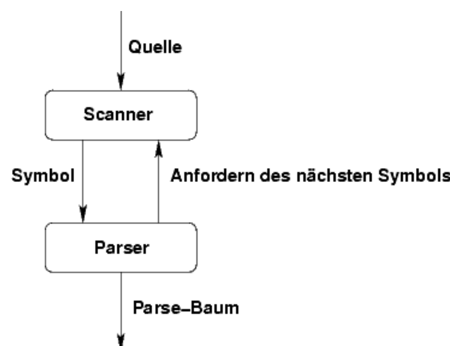


Abbildung 6.2: Arbeitsweise eines Parsers

Analyse wird der Parse-Baum von der Wurzel her erzeugt, während die Bottom-Up-Analyse den Baum von den Blattknoten her aufbaut. Zu beachten ist dabei, dass die Grammatikbeschreibung für Top-Down-Parser keine linksrekursiven Regeln enthalten darf, diese führen bei der Top-Down-Syntaxanalyse zur Endlosrekursion [ASU88].
Scanner und Parser können heute mit Hilfe von Parsergeneratoren, wie *lex* und *yacc*, sowie einer Grammatikbeschreibung erzeugt werden.
In dieser Arbeit wird der Parsergenerator *Java Compiler Compiler* verwendet. Dieser erzeugt aus einer Grammatikbeschreibung sowohl den Scanner als auch den Parser in Java. Der generierte Parser arbeitet als Top-Down-Parser und liefert einen Parse-Baum als Ergebnis.
Die Konfigurationsdatei für den Parsergenerator ist aus drei Teilen aufgebaut. Der erste Teil, gekennzeichnet durch `PARSER_BEGIN` und `PARSER_END`, wird direkt in die erzeugte Parser-Klasse übernommen. Hier kann beispielsweise definiert werden, zu welchem Java-Package die Klasse gehören soll. Der zweite Teil der Datei wird verwendet, um die Symbole der Grammatik zu definieren. Diese Angaben werden für die Erzeugung des Scanners benötigt. Die Produktionen der Grammatik bilden schließlich den letzten Teil der Datei. Der Generator erzeugt für jede Regel der Grammatik eine Java-Klasse, die als Knoten im Parse-Baum verwendet wird. Diese Klassen können anschließend erweitert werden. Die Grammatikdefinition für den Generator ist dem Anhang C zu entnehmen.

6.3 Implementierung der Sprachelemente

Für die Implementierung der Sprachelemente von SNQL wurden die vom Parsergenerator erzeugten Java-Klassen in einer Hierarchie angeordnet. So wurden Klassen, die für Anweisungen stehen, wie die Klassen `AstCopyStatement` oder `AstDeleteStatement`, als Implementierung eines gemeinsamen Java-Interfaces `IStatement` zusammengefasst. Dadurch können im Quellcode des Interpreters Typunterscheidungen und explizite Typumwandlungen eingespart werden. Die Hierarchie des Interfaces `IStatement` ist in Abbildung 6.3 dargestellt. Jede implementierende Klasse enthält eine Methode `execute`, die die entsprechende SNQL-Anweisung ausführt. Die weiteren Klassen und Interfaces, sowie die grundsätzliche Funktionsweise des Interpreters werden in den folgenden Abschnitten vorgestellt. Im Prinzip entspricht die Ausführung einer Anweisung der Abarbeitung des entsprechenden Strukturaums der Anweisung.

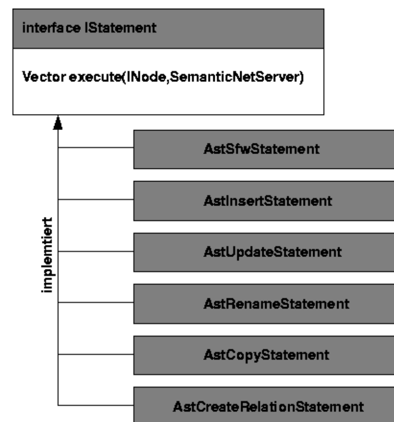


Abbildung 6.3: Hierarchie des Interfaces `IStatement`

6.3.1 Pfadausdrücke

Da Pfadausdrücke in vielen Sprachelementen der Anfragesprache verwendet werden, wird die Implementierung von Pfadausdrücken hier gesondert betrachtet. Für die Ausführung der Anweisungen ist nicht nur die Menge der Knoten, die durch Pfadausdrücke definiert werden, notwendig, sondern es werden zusätzlich Informationen über den Weg von der Wurzel des Netzes zu den einzelnen Knoten benötigt. Beispielweise können sich Bedingungen in der `where`-Klausel einer Auswahlanfrage sowohl auf die einzelnen Knoten, wie auch auf den Pfad, der zu diesem Knoten führt, beziehen. Pfade werden durch die Java-Klasse `Path` realisiert. Ein Pfad enthält Verweise auf die Knoten des Pfades, sowie die Typen der Relationen zwischen den Knoten.
Aufgrund der Mengenorientierung der Sprache werden bei den Anweisungen nicht einzelne Pfade, sondern Mengen von Pfaden betrachtet. Eine Menge von Pfaden wird durch eine Pfadtabelle repräsentiert. Diese wird durch die Klasse `PathTable` implementiert.

Innerhalb von Pfadausdrücken können Variablen verwendet werden. Daher verwendet der Interpreter eine Symboltabelle. Jeder Eintrag in dieser Tabelle besteht aus dem Variablennamen und einem Verweis auf die Definition der Variablen im Parse-Baum. Variablen werden erst dann ausgewertet, wenn der Interpreter bei der Verarbeitung von Pfadausdrücken auf Variablen stößt. Die Auswertung erfolgt dabei relativ zum bisherigen Pfadausdruck. Die Auswertung von Variablen entspricht also der Ersetzung der Variablen durch den Strukturbaum des Pfadausdrucks, für den die Variable steht. Die Symboltabelle wird durch die Java-Klasse SymbolTable realisiert.

Der Strukturbaum eines Pfadausdrucks gliedert sich in die einzelnen Navigationsschritte. Diese Schritte werden in der Syntax durch Punkte voneinander getrennt. Die Auswertung eines Pfadausdrucks entspricht dem Aufbau einer Pfadtabelle. Zu Beginn enthält die Pfadtabelle den Bezugspunkt für die Pfadausdrücke, bei der Kommandozeilenversion des Interpreters ist dies der Einstiegspunkt in das semantische Netz. Das Ergebnis eines jeden Navigationsschrittes ist eine Menge von Pfaden, die in der Pfadtabelle abgelegt werden. Jeder Navigationsschritt verwendet das Ergebnis des voherigen Schrittes als Ausgangsbasis, das heißt, es wird versucht, den Navigationsschritt für jeden Pfad aus der Pfadtabelle auszuführen. In die Ergebnistabelle werden anschließend diejenigen Pfade, um den Navigationsschritt ergänzt, übernommen, für die die Ausführung möglich war.

Der Aufbau der Pfadtabelle kann formal folgendermaßen ausgedrückt werden: für einen Navigationsschritt entlang einer Relation $relation$ enthält die Pfadtabelle P_{neu} alle Pfade p aus der Pfadtabelle P_{alt} des voherigen Schrittes für die gilt, dass ein Pfad $p.relation$ im semantischen Netz existiert. Dies soll anhand des Beispiels in Abbildung 6.4 erläutert werden.

In diesem Beispiel wird der Pfadausdruck relation1.relation2 ausgewertet. In der Darstellung werden sowohl die Namen der Knoten als auch die Typen der Relation durch < und > eingeschlossen. Zu Beginn enthält die Pfadtabelle nur den Einstiegspunkt in das semantische Netz. Der erste Navigationsschritt besteht aus der Navigation entlang der Relationen vom Typ relation1 vom Einstiegspunkt aus zu den Knoten K1 und K2. Am Ende des Schrittes enthält die Tabelle die Pfade <$root><relation1><K1> und <$root><relation1><K2>. Im folgenden Navigationsschritt werden die Pfade aus der jetzt entstandenen Pfadtabelle als Basis für die weitere Navigation entlang der Relationen vom Typ relation2 genommen. So gelangt man vom Knoten am Ende des Pfades <$root><relation1><K1> über die Relation relation2 zu den Knoten K3 und K4. Man erhält als Ergebnis somit die Pfade <$root><relation1><K1> <relation2><K3> und <$root><relation1><K1><relation2><K4>. Da eine weitere Navigation vom Knoten K2 über eine Relation relation2 nicht mehr möglich ist, bilden diese Pfade die Ergebnistabelle.

Reguläre Ausdrücke innerhalb von Pfadausdrücken lassen sich auf die Vereingung von Pfadtabellen abbilden. So führen Ausdrücke der Form $(p_1|p_2)$, wobei p_1 und p_2 Pfadausdrücke sind, zur Vereinigung der Pfadtabellen, die das Ergebnis der Auswertung der Pfadausdrücke p_1 und

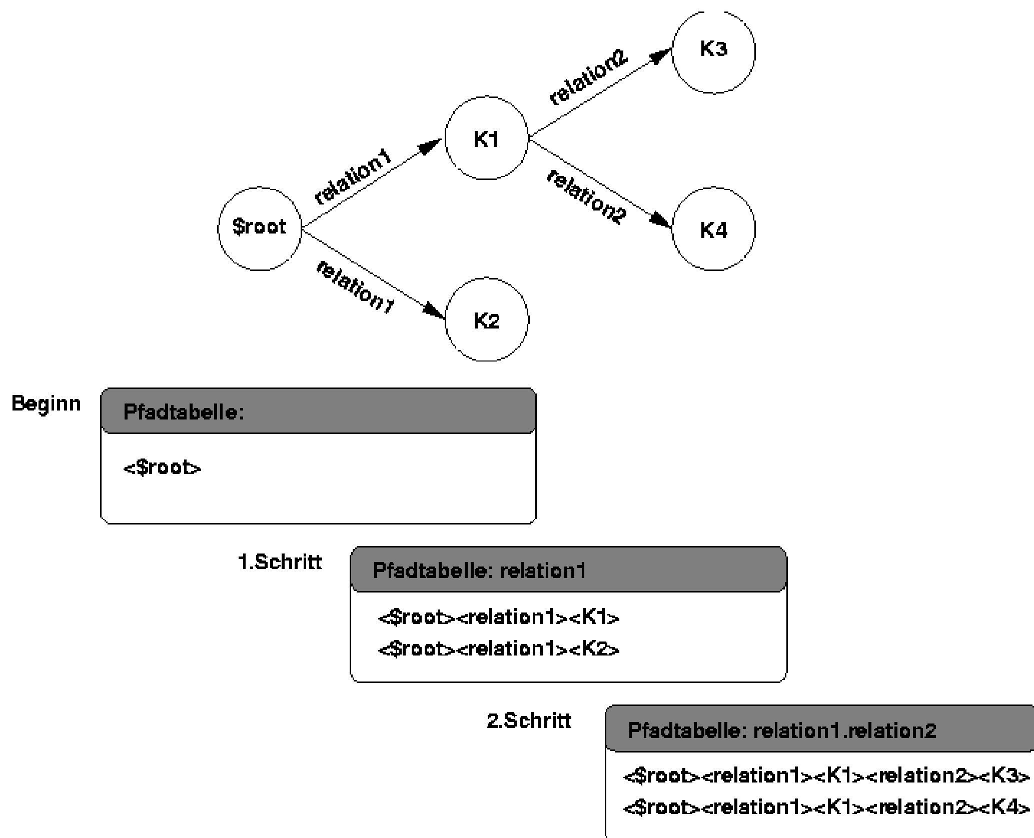
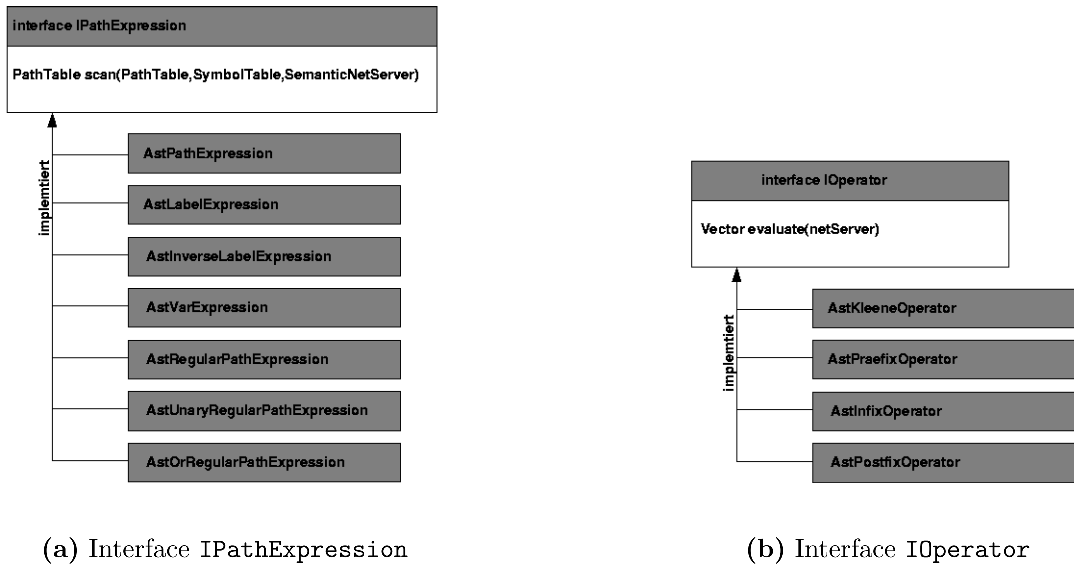


Abbildung 6.4: Aufbau der Pfadtabelle an einem Beispiel

p_2 bilden. Bei Ausdrücken der Form $p_1.(p_2)?$ werden die Pfadtabellen der Pfadausdrücke p_1 und $p_1.p_2$ vereinigt. Reguläre Ausdrücke der Form $p.(p_1)+$ führen zur Vereinigung der Pfadtabellen der Pfadausdrücke $p.p_1$, $p.p_1.p_1$, ... Die Hüllenkonstruktion nach Kleene in der Form $p.(p_1)*$ lässt sich äquivalent durch $(p \,|\, p.(p_1)+)$ ausdrücken und führt damit zur Vereinigung der Pfadtabelle der Pfadausdrücke p und $p.(p_1)+$. Dabei muss beachtet werden, dass solche Ausdrücke bei Netzen mit Zyklen zur unendlichen Vereinigung führen. Daher wird bei der Konstruktion von Pfaden überprüft, ob der Pfad einen Zyklus enthält, das heißt, ob ein Knoten mehrfach im Pfad vorkommt. Falls dies der Fall ist, wird die Konstruktion des Pfades abgebrochen.

Die Platzhalter für die Namen der Relationstypen werden dadurch realisiert, dass dieses Muster mit allen im Netz definierten Relationstypen verglichen wird. Die Navigation wird anschließend mit denjenigen Relationstypen durchgeführt, die auf das Muster passen.

Alle Klassen, die zur Auswertung von Pfadausdrücken benötigt werden, implementieren das Interface `IPathExpression`, so dass die Auswertung von Pfadausdrücken vereinheitlicht werden kann. Die Platzhalter für Namen von Relationstypen implementieren das Interface `IOperator`. Das Klassenschema ist in Abbildung 6.5 dargestellt.



(a) Interface `IPathExpression` **(b)** Interface `IOperator`

Abbildung 6.5: Klassenschema für Pfadausdrücke

6.3.2 Auswahlanfragen

Auswahlanfragen der Form `select<Pfadausdruck>from<Variablenliste>where<Bedingung> orderby<Ausdruck>` werden vom Interpreter der Anfragesprache nach dem Schema aus Abbildung 6.6 abgearbeitet.

Im ersten Schritt wird die `from`-Klausel ausgewertet. Dabei wird die Symboltabelle des Interpreters aufgebaut. Falls die `from`-Klausel fehlt, ist die Symboltabelle leer. Sie enthält Informationen zur Bindung der Variablen und wird im zweiten Schritt beim Auswerten des Pfadausdrucks innerhalb der `select`-Klausel benötigt. Dabei wird der Pfadausdruck auf eine Pfadtabelle abgebildet, zum Mechanismus dieser Abbildung siehe Abschnitt 6.3.1.

Die bei der Abbildung des Pfadausdrucks entstandene Pfadtabelle P ist die Basis für die darauf folgende Auswertung der Bedingung der `where`-Klausel. Die Bedingung ist eine Verknüpfung von Prädikaten durch die bool'schen Operatoren `and`, `or` und `not`. Die Prädikate bestehen aus einem Operator und zwei Operanden. Dabei müssen zwei Arten von Operanden unterschieden werden. Erstens gibt es atomare Werte, wie Zahlen oder Zeichenketten, zweitens kann ein Operand auch durch einen Pfadausdruck repräsentiert werden. Sofern ein Operand ein Pfadausdruck ist, wird dieser Ausdruck ebenfalls auf eine Pfadtabelle P_1 abgebildet.

Die Bedingung der `where`-Klausel wird für jeden Pfad $p \in P$ folgendermaßen ausgewertet. Werte in einem ersten Schritt jedes Prädikat der Bedingung zu `true` oder `false` aus. Falls mindestens ein Operand des Prädikats ein Pfadausdruck ist, dann schneide P_1 mit p. Die Schnittmengenbildung entspricht hier nicht exakt derjenigen aus der Mengenlehre. Die Schnittmenge enthält neben den Pfaden, die gleich p sind, die Pfade, die ein Präfix von p sind oder für die gilt, dass p ein Präfix von ihnen ist. Dies heißt formal: $Q \cap q = \{x \in Q | x = q \vee x.y = q \vee q.y = x\}$,

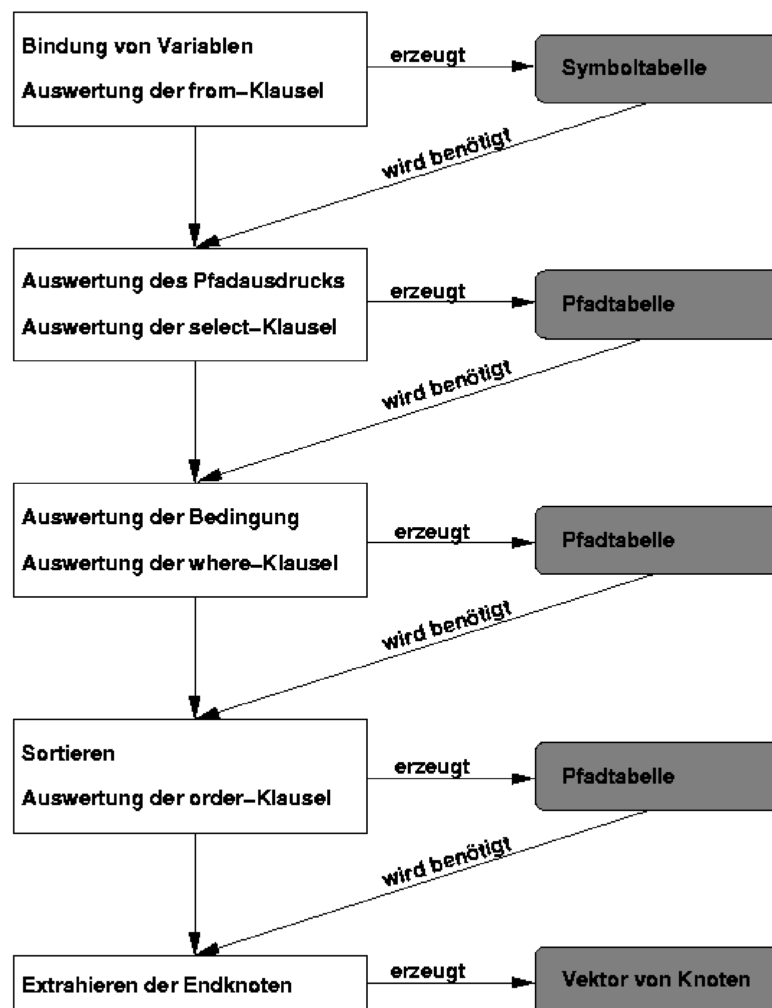


Abbildung 6.6: Schematischer Ablauf der Auswertung einer Auswahlanfrage

wobei Q eine Pfadtabelle und q, x, sowie y Pfade sind. Diese Definition ist notwendig, da sich Pfadausdrücke in der Bedingung nicht nur auf die selektierten Knoten, sondern auch auf Nachfolger oder Vergänger bezüglich des Pfades zu den einzelnen Knoten beziehen können. Dieser Sachverhalt soll anhand des Beispiels aus Abbildung 6.7 erläutert werden. In diesem Beispiel sollen diejenigen Knoten ausgewählt werden, die über einen Pfad `relation1.relation2` von der Wurzel aus erreichbar sind und deren Vorgänger der Knoten mit dem Wert `K1` ist. Dies sind in Abbildung 6.7 die Knoten `K3` und `K4`. In SNQL kann die Auswahl folgendermaßen formuliert werden:

```
select &R.relation2
from relation1 as R
where (&R$value = $K1)
```

Ein Prädikat wird zu `true` ausgewertet, falls ein Pfad $q \in (P_1 \cap p)$ existiert, für den das Prädikat den Wert `true` annimmt. Falls beide Operanden Pfadausdrücke sind, nimmt das Prädikat den

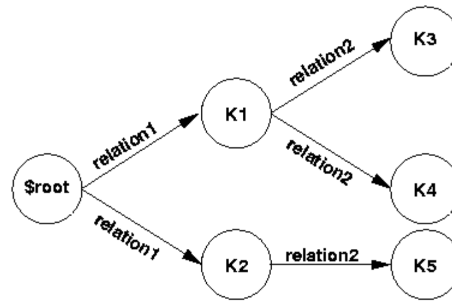
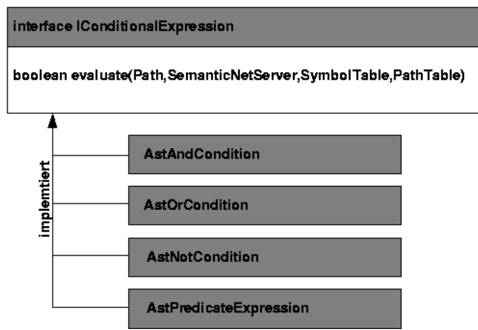


Abbildung 6.7: Semantisches Netz

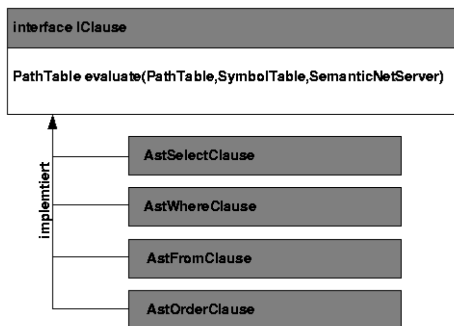
Wert `true` an, falls das Prädikat für ein Paar (q, r) mit $q \in (P_l \cap p)$ und $r \in (P_r \cap p)$, wobei P_l die Pfadtabelle des linken und P_r die Pfadtabelle des rechten Operanden bezeichnet, diesen Wert annimmt. Das Ergebnis von Vergleichen zwischen Knoten und atomaren Werten ist undefiniert und wird zu `false` ausgewertet. Vergleiche zwischen atomaren Werten werden immer als Vergleiche zwischen Zeichenketten durchgeführt, das bedeutet, dass beispielsweise Zahlen vor Vergleichsoperationen in Zeichenketten umgewandelt werden.
Anschließend werden die Ergebnisse der Prädikate entsprechend den bool'schen Operatoren der Bedingung miteinander verknüpt. Das Ergebnis der Auswertung der `where`-Klausel ist eine Pfadtabelle, die alle Pfade $p \in P$ enthält, für die die Bedingung zu `true` evaluiert.
Falls die Auswahlanfrage eine `order-by`-Klausel enthält, wird die Pfadtabelle, die das Ergebnis der bisherigen Auswertung darstellt, bezüglich des Werts der Knoten am Ende der Pfade sortiert. Dies erfolgt in dieser Implementierung nach der Methode des Sortierens durch direktes Einfügen (*Insertion Sort*), siehe dazu [Sed92].
Abschließend wird ein Vektor aus den Knoten am Ende eines jeden Pfades aus der Pfadtabelle gebildet, der das Ergebnis der Anfrage darstellt. Dabei werden Dublikate eliminiert.
Die Klassen der Implementierung basieren auf den Interfaces `IConditionalExpression` für die bool'schen Bedingungen in der `where`-Klausel, `IPredicateOperation` für die verschiedenen Prädikate und `IClause` für die Klauseln der Anweisung. Das Klassenschema ist in Abbildung 6.8 dargestellt.

6.3.3 Änderungsanweisungen

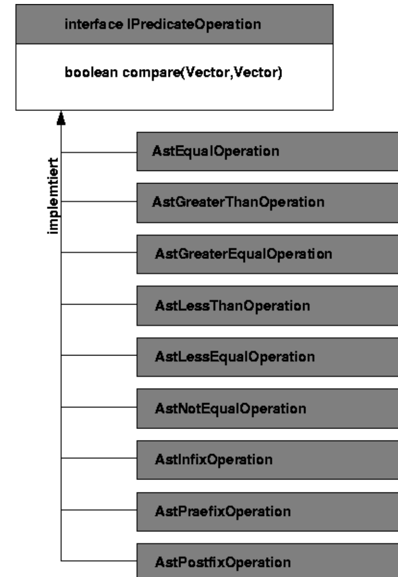
In den folgenden Abschnitten wird die Implementierung von Anweisungen, mit deren Hilfe die Struktur des semantischen Netzes verändert werden kann, vorgestellt. Auf die Darstellung der `create relation type`- und der `insert`-Anweisung wurde verzichtet, da diese direkt von der Schnittstelle zum semantischen Netz unterstützt werden. Der Rückgabewert der `insert`-Anweisung besteht aus einem Vektor, der den neu erzeugten Knoten enthält.
An dieser Stelle muss darauf hingewiesen werden, dass die Implementierung keine Relationen auf Relationen unterstützt. Dies hat seine Ursache darin, dass das Datenmodell des Projek-



(a) Interface `IConditionalExpression`



(b) Interface `IClause`



(c) Interface `IPredicateOperation`

Abbildung 6.8: Klassenschema der Auswahlanfrage

tes Föderal zwischen dem Sprachentwurf und der Implementierung der Anfragesprache dahingehend verändert wurde, dass Relationen auf Relationen nicht mehr möglich sind. Diese Änderung am Datenmodell konnte aus Zeitgründen nicht mehr berücksichtigt werden. Zudem können mit dem hier implementierten Interpreter nur Knoten vom Typ `StringNode` angelegt werden, da die Schnittstelle zum semantischen Netz im Augenblick nur diesen Knotentyp unterstützt.

Einfügen von Relationen

Die `update`-Anweisung enthält zwei Teilanweisungen, die dazu dienen, die Knoten zu definieren, zwischen denen neue Relationen eingefügt werden sollen. Das Ergebnis dieser Teilanweisungen, möglich sind eine Auswahlanfrage, eine Einfügeoperation für Knoten oder das Schlüsselwort für den Einstiegspunkt in das semantische Netz, ist jeweils ein Vektor von Knoten. Das eigentliche Einfügen kann durch eine doppelte Schleife realisiert werden, so dass die Relationen (r, s) vom in der Anfrage spezifizierten Typen für alle $r \in V_{quelle}$ und alle $s \in V_{ziel}$ erzeugt werden. Hier bezeichnen V_{quelle} und V_{ziel} Vektoren von Knoten.

Löschen von Knoten und Relationen

Die Schnittstelle zum semantischen Netz unterstützt das Löschen von Knoten nicht direkt, statt dessen gelten Knoten, die durch keine Relation mit anderen Knoten des Netzes verbunden sind, als gelöscht. Die Knoten, die gelöscht werden sollen, werden durch eine Auswahlanfrage, die einen Vektor dieser Knoten als Ergebnis liefert, spezifiziert. Die Löschoperation nimmt jeden Knoten aus diesem Vektor und entfernt alle Relationen dieses Knotens aus dem Netz. Damit sind die Knoten im Netz nicht mehr erreichbar und gelten somit als gelöscht.
Das Löschen von Relationen ist analog zum Einfügen von Relationen implementiert. Im Unterschied zum Einfügen werden hier die Relationen (r, s), sofern vorhanden, entfernt.

Änderung des Werts von Knoten

Bei der `set value`-Anweisung wird die Menge der Knoten, deren Wert verändert werden soll, durch eine Auswahlanfrage angegeben. Diese Auswahlanfrage gibt die ausgewählten Knoten in Form eines Vektors zurück. Anschließend wird für jeden Knoten dieses Vektors die Methode zum Ändern des Wertes eines Knoten aus der Schnittstelle mit dem neuen Wert als Parameter aufgerufen.

6.3.4 Kopieren von Teilnetzen

Beim Kopieren werden zwei Operationen unterschieden. Die eine Operation kopiert die Knoten mit allen Relationen, die zwischen diesen Knoten bestehen, während die andere auch die Relationen mitkopiert, die das Teilnetz mit dem Rest des semantischen Netzes verbinden.
Beiden Operationen ist gemeinsam, dass das Teilnetz über eine Auswahlanweisung definiert wird. Das Ergebnis einer Auswahlanweisung ist ein Vektor V derjenigen Knoten, die durch den Pfadausdruck der Anweisung spezifiert werden. Im ersten Schritt der Ausführung einer Kopieroperation werden die Knoten dieses Vektors neu angelegt. Auch diese Knoten werden in einem Vektor V_{copy} angeordnet, in dem die Knoten dieselbe Reihenfolge haben wie im Vektor Vektor V, das heißt, $V_{copy}[i]$ ist die Kopie von $V[i]$, wobei $V[i]$ den Knoten an der Position i innerhalb von V bezeichnet. Anschließend muss zwischen den Varianten unterschieden werden.
Die erste Variante des Kopierens betrachtet nur die Relationen, die innerhalb des Teilnetzes verlaufen. Daher werden die von den Knoten des Vektors V ausgehenden Relationen r vom Typ t daraufhin untersucht, ob der Zielknoten der Relation r ebenfalls in dem Vektor V enthalten ist. Ist dies der Fall, so verläuft die Relation r innerhalb des Teilnetzes und zwischen der Kopie des Quell- und des Zielknoten wird eine Relation vom Typ t angelegt. Die Kopien der Knoten des Vektors V können im Vektor V_{copy} auf einfache Art über deren Position innerhalb des Vektors gefunden werden, da beide Vektoren die Knoten in derselben Reihenfolge enthalten.
Bei der zweiten Variante müssen zusätzlich zu den von den Knoten im Vektor V ausgehenden

Relationen, deren Zielknoten ebenfalls im Vektor V enthalten ist, diejenigen Relationen der Knotens kopiert werden, die aus dem Netz herausführen, beziehungsweise in das Teilnetz führen. Dazu müssen alle Relation der Knoten betrachtet werden. Falls der gerade betrachteten Knoten K der Quellknoten der Relation r vom Typ t ist, so erzeuge eine neue Relation vom Typ t zwischen der Kopie des Quellknotens K und der Kopie des Zielknotens. Zusätzlich wird eine neue Relation vom Typ t angelegt, falls K der Zielknoten der Relation r ist und der Quellknoten nicht im Vektor V enthalten ist.

6.4 Test des Interpreters

Der Interpreter wird mit Hilfe einer Testklasse geprüft. Im Test wird zunächst ein semantisches Netz aufgebaut, bevor Anfragen an dieses Netz gestellt werden, sowie das Netz manipuliert wird. In der folgenden Darstellung werden die SNQL-Anweisungen zusammen mit dem jeweiligen Rückgabewert des Interpreters angegeben. Änderungen am semantischen Netz sind durch Abbildungen veranschaulicht, um die Veränderungen nachvollziehen zu können.

Aufbau des semnatischen Netzes

Im ersten Schritt werden die Relationstypen `rel1`, `rel2` und `rel3` angelegt.

```
create relation type rel1
create relation type rel2
create relation type rel3
```

Das semantische Netz zu Testzwecken wird mit Hilfe folgender Anweisungen aufgebaut. Das aufgebaute Netz ist in Abbildung 6.9 dargestellt.

```
update ~.rel1 += insert(k1)
```

Diese Anweisung erzeugt den Knoten `k1` und verbindet ihn durch eine Relation vom Typ `rel1` mit dem Einstiegsknoten des Netzes.

```
update (select rel1).rel2 += insert(k2)
```

In diesem Schritt wird ein Knoten `k2` angelegt und mit einer Relation vom Typ `rel2` mit dem Knoten `k1` verbunden.

```
update (select rel1.rel2).rel2 += insert(k3)
```

Hier wird ein Knoten mit dem Wert `k3` angelegt und durch eine Relation vom Typ `rel2` mit dem Knoten `k2` verknüpft.

```
update (select rel1.rel2.rel2).rel2 += select rel1
```

Durch diese Anweisung wird eine Relation vom Typ `rel2` zwischen den Knoten `k3` und `k1` eingefügt. Auf diese Weise entsteht ein Zyklus im Netz.

```
update ~.rel2 += insert(k4)
update ~.rel1 += insert(k5)
```

Diese Anweisungen erzeugen die Knoten `k4` und `k5` und verknüpfen diese Knoten über eine Relation vom Typ `rel2`, beziehungsweise `rel1` mit dem Einstiegspunkt in das semantische Netz.

```
update (select rel2).rel3 += select rel1.(rel2)+
```

Abschließend wird hier der Knoten `k4` über Relationen vom Typ `rel3` mit den Knoten `k2`, `k3` und `k1` verbunden.

Auswahlanfragen

In diesem Abschnitt werden SNQL-Anweisungen vorgestellt, die Auswahlanfragen an das semantische Netz aus Abbildung 6.9 definieren.

Die folgende Anweisung wählt alle Knoten aus, die durch eine Relation vom Typ `rel1` mit

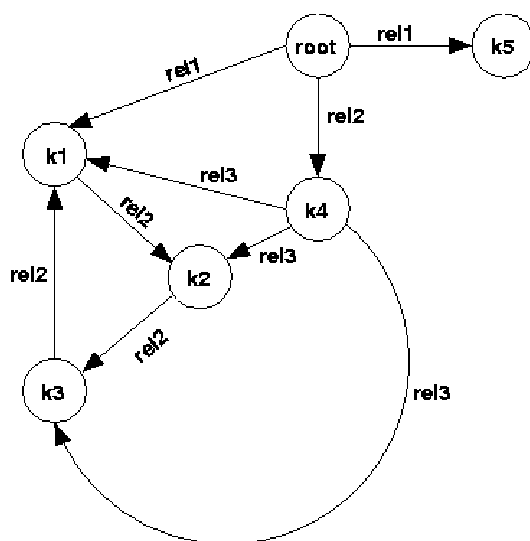


Abbildung 6.9: Semantisches Netz zu Testzwecken

dem Einstiegspunkt in das Netz verbunden sind. In der Abbildung sind dies die Knoten `k1` und `k5`.

```
select rel1
[k5,k1]
```

Durch Verwendung einer `where`-Klausel in der zweiten Anfrage wird das Ergebnis der vorigen Anweisung auf Knoten mit dem Wert `k1` eingeschränkt.

```
select rel1 where (rel1@=$k1)
[k1]
```

Die folgende Anweisung stellt eine Variante der zweiten Anfrage mit Verwendung einer Variablen dar.

```
select &r from rel1 r where (&r@=$k1)
[k1]
```

Mit Hilfe des Platzhalters `#` werden in der nachfolgenden Anfrage alle Knoten ausgewählt, die mit einer Relation beliebigen Typs mit dem Einstiegspunkt des Netzes verknüpft sind. Dies sind die Knoten `k1`, `k4` und `k5`.

```
select #
[k4,k1,k5]
```

Die nächste Anfrage variiert die vorigen Anfrage durch Verwendung einer Variablen und einer `where`-Klausel, welche die Auswahl auf den Knoten `k1` einschränkt.

```
select &x from # x where (&x$value=$k1)
[k1]
```

Anschließend folgen drei Anweisungen, die den Platzhalter `%` innerhalb von Namen von Relationstypen testen. Die erste Anfrage wählt alle Knoten, die durch Relationen, deren Typ mit der Zeichenkette `rel` beginnt, mit dem Einstiegspunkt in das Netz verbunden sind. Das sind in Abbildung 6.9 die Knoten `k1`, `k4` und `k5`. Die zweite Anweisung bindet die Relationstypen, die die Zeichenfolge `el` enthalten, in die Anfrage ein, während die letzte Anfrage nur Relationstypen untersucht, die mit der Zeichenkette `el1` enden. Das Anfrageergebnis ist unterhalb der Anfragen angegeben.

```
select rel%
[k4,k5,k1]
select %el%
[k4,k5,k1]
select %el1
[k5,k1]
```

Die folgenden Anweisungen testen die Implementierung regulärer Ausdrücke innerhalb von Pfadausdrücken. Die zweite Anfrage verwendet eine `order by` -Klausel zum Sortieren des Ergebnisses. Die einzelnen Ergebnisse sind auch hier jeweils unterhalb der Anweisungen angegeben.

```
select rel1.(rel2)*
[k2,k3,k1,k5]
select rel1.(rel2)* order by $value
[k1,k2,k3]
select rel1.(rel2.rel2)*
[k3,k1]
select rel1.(rel2)+
[k2,k3,k1]
select rel1.(rel2)?
[k2,k1]
select rel%.((rel2)|(rel3))
[k2,k1,k3]
```

Kopieren

In diesem Abschnitt wird die Kopieroperation in beiden Varianten getestet. Zu beachten ist hierbei, dass erst die zweite Kopieranweisung Auswirkungen auf das Netz hat, da die erste die Relationen, die das selektierte Teilnetz mit dem übrigen semantischen Netz verbinden, nicht kopiert. Das veränderte Netz ist in Abbildung 6.10 dargestellt. Das kopierte Teilnetz ist durch Einfärbung markiert. Knotenwerte, die mit `copy_of_` beginnen, sind in der Abbildung durch `c.` abgekürzt.

```
copy (select &x.&y from rel1 as x, (rel2)+ as y)
[copy_of_k1,copy_of_k2,copy_k3]
copy relation (select &x.&y from rel1 as x, (rel2)+ as y)
[copy_of_k1,copy_of_k2,copy_k3]
select rel1
[k1,copy_of_k1]
select rel1.(rel2)+
[k1,k2,k3,copy_of_k1,copy_of_k2,copy_k3]
```

Löschen von Relationen

Die nächste Anweisung entfernt im Netz aus Abbildung 6.10 die Relationen vom Typ `rel2` zwischen den Knoten `k1` und `k2` beziehungsweise `c.k1` und `c.k2`. Das dadurch entstehende Netz ist in Abbildung 6.11 dargestellt. Die nachfolgenden Auswahlanfragen zeigen, dass diese Relationen tatsächlich entfernt wurden. So ist der Knoten `k3` noch erreichbar, wenn die Relation vom Typ `rel2` vom Knoten `k1` aus entgegen der Pfeilrichtung benutzt wird.

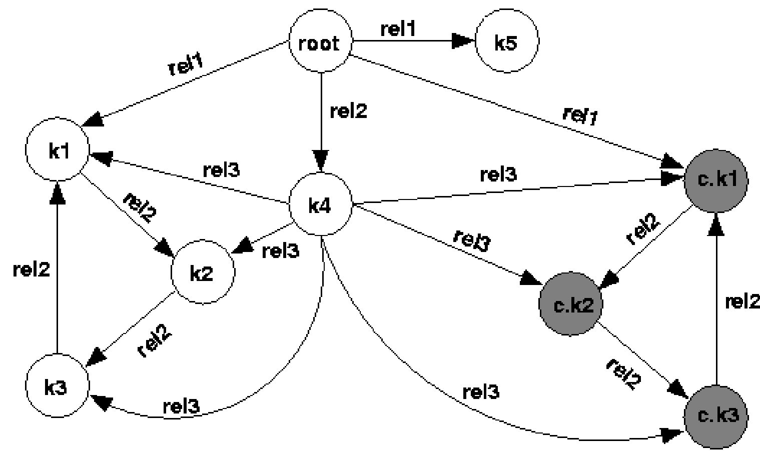


Abbildung 6.10: Semantisches Netz nach einer Kopieroperation. Das kopierte Teilnetz ist grau eingefärbt.

```
update (select rel1).rel2 -= select rel1.rel2
select rel1.(rel2)+
[]
select rel1.(inverse(rel2))+
[k2,copy_of_k2,copy_of_k3,k3]
```

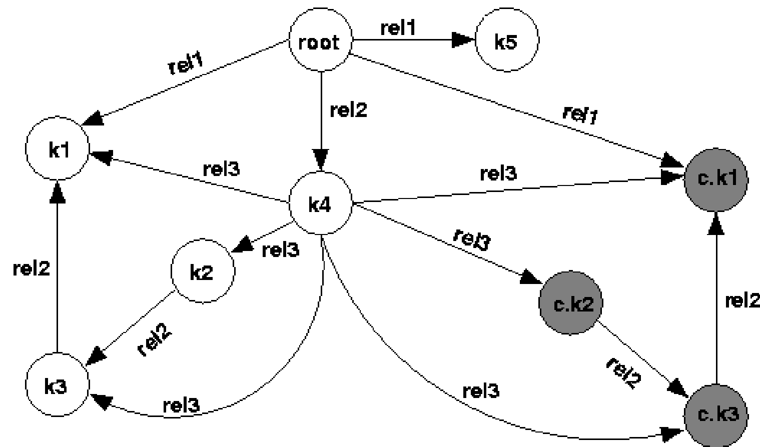


Abbildung 6.11: Semantisches Netz nach Entfernen von zwei Relationen vom Typ `rel1`

Löschen eines Knoten

Als letzter Test wird in der folgenden Anweisung der Knoten `k4` aus dem Netz in Abbildung 6.11 entfernt. Dabei werden alle Relationen des Knotens entfernt. Abbildung 6.12 zeigt das veränderte Netz nach dem Löschen. Die beiden Auswahlanfragen testen die bisher im Netz bestehenden Pfade zum Knoten `k4` und bestätigen durch die leere Ergebnismenge das erfolgreiche Löschen.

```
delete (select rel2)
select rel2
[]
select rel1.inverse(rel3)
[]
```

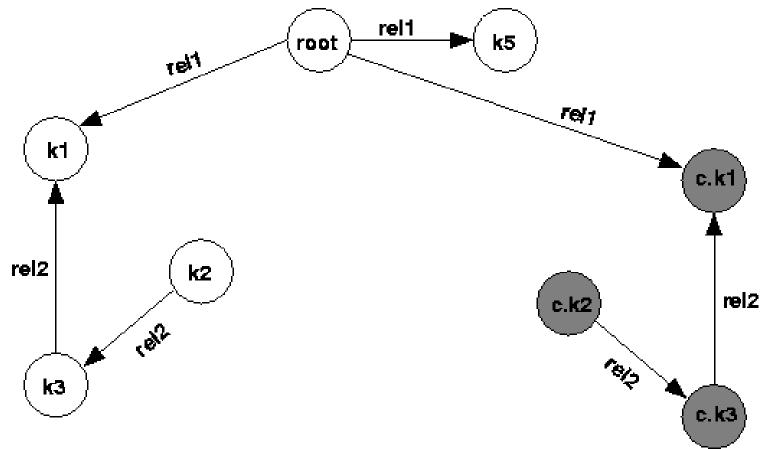


Abbildung 6.12: Semantisches Netz nach Entfernen des Knotens `k4`

Kapitel 7

Zusammenfassung

Im Projekt Föderal verfolgen Unternehmen des Anlagen- und Maschinenbaus, sowie ein IT-Dienstleister und zwei Forschungseinrichtungen das Ziel, eine föderale Architektur für die Verwaltung von Produktdaten zu entwickeln. Dabei wird die Föderierungsschicht der Architektur als semantisches Netz modelliert. Unter einem semantischen Netz ist hier ein Netz zu verstehen, das Informationen über die Zusammenhänge von Daten modelliert. In dieser Arbeit wird eine Anfragesprache für dieses Netz entworfen und implementiert.

In einem ersten Schritt wurden die Anforderungen an eine Sprache für das semantische Netz erhoben. Dabei hat sich gezeigt, dass sich die Anforderungen in allgemeine und projektspezifische Anforderungen unterteilen lassen. So ist es notwendig, dass die Sprache eine flexible Typisierung und Pfadausdrücke enthält. Weiter müssen rekursive Anfragen mit Hilfe der Sprache formuliert werden können. Die Sprache muss Standardoperationen, wie Einfügen und Löschen von Knoten und Kanten oder Kopieren von Teilstrukturen, enthalten. Außerdem soll die Sprache mengenorientiert und deklarativ sein.

Bei der Untersuchung bereits vorhandener Anfragesprache für strukturierte und semistrukturierte Datenmodelle hat sich gezeigt, dass diese Anforderungen von keiner der untersuchten Sprache vollständig abgedeckt werden. So fehlt zum Beispiel in allen Sprachen die Unterstützung für Kopieroperationen.

Daher wurde im Rahmen dieser Arbeit eine Anfragesprache für das semantische Netz auf Basis von Lorel entworfen. Die Sprache übernimmt die von SQL und OQL gewohnte Syntax für Anfragen mit den notwendigen Änderungen für die Unterstützung von Pfadausdrücken. Weitere Änderungen waren notwendig, um die Manipulation des semantischen Netzes zu ermöglichen. So wurden Anweisungen zum Anlegen neuer Typen von Kanten und zum Kopieren von Teilnetzen definiert. Die Kopieroperation wird in zwei Varianten unterstützt, um die Modellierung von Netzen nach dem Baukastenprinzip zu unterstützen.

Im Rahmen der Arbeit wurde ein Interpreter für die Sprache in Java implementiert. Der Interpreter greift in dieser Implementierung über eine prozedurale Schnittstelle auf das semantische

Netz zu, das in einer Datenbank gespeichert ist.
In einem an diese Arbeit anschließenden Schritt wäre es sinnvoll, die Sprache unter praxisnahen Bedingungen zu testen. Dabei sollte untersucht werden, ob die Funktionalität der Sprache für einen Praxiseinsatz ausreichend ist. Weiter sollte die Syntax der Sprache auf ihre Praxistauglichkeit überprüft werden. Dabei sollte auch überprüft werden, ob es sinnvoll ist, Abkürzungen für bestimmte Sprachkonstrukte in die Syntax der Sprache einzubauen. Sinnvoll wäre zudem eine Performance-Evaluierung des Interpreters. Eventuell stellt sich dabei heraus, dass ein relationales Datenbanksystem als Datenbank für das semantische Netz Laufzeitvorteile bietet. Dann kann es sinnvoll sein, den SNQL-Interpreter direkt nach SQL übersetzen zu lassen.

Anhang A

Abkürzungsverzeichnis

DTD	Document Type Definition
EBNF	Extended Backus-Naur-Form
FIA	Föderale Informationsarchitektur
HTML	Hypertext Markup Language
Lore	Lightweight Object Repository
Lorel	Lightweight Object Repository Language
ODMG	Object Database Management Group
ODL	Object Definition Language
OEM	Object Exchange Model
OQl	Object Query Language
SGML	Standard Generalized Markup Language
SNQL	Semantic Net Query Language
SQL	Structured Query Language
TSIMMIS	The Stanford-IBM Manager of Multiple Information Sources
UML	Unified Modelling Language
URL	Uniform Resource Locator
XML	Extensible Markup Language
XPath	XML Path Language
XPointer	XML Pointer Language
XQL	XML Query Language
XSL	XML Stylesheet Language
XSLT	XSL Transformation
W3C	World Wide Web Consortium

Anhang B

EBNF der Anfragesprache

In diesem Abschnitt folgt die vollständige Syntax der Anfragesprache SNQL für das semantische Netz in erweiterter Backus Naur Form (EBNF).

Digit = „0“| „1“| ... | „9“.

Letter = „a“| „b“| ... | „z“| „A“| „B“| ... | „Z“| „_“| „-“.

StringLiteral = *Letter* {(*Digit* | *Letter*) }.

IntegerLiteral = *Digit*.

FloatLiteral = *IntegerLiteral* „.“*IntegerLiteral*.

AtomicValue = *IntegerLiteral* | *FloatLiteral* | *StringLiteral*.

PathExpression = *PathComponent* {„.“ *PathComponent* }.

PathComponent = *VariableExpression* | *LabelExpression*
| *InverseLabelExpression* | *RegularPathExpression*.

VariableExpression = „&“ *StringLiteral*.

LabelExpression = „#“ | [„%“] *StringLiteral* [„%“].

InverseLabelExpression = „inverse (“ *LabelExpression* „)“.

RegularPathExpression = „(“ *RegularPathExpression* „|“ *RegularPathExpression* „)“
| *RegularPathExpression* [„*“| „+“| „?“]
| „(“ *PathExpression* „)“.

Query = *SfwStatement* | *SetValueStatement* | *InsertStatement*
| *UpdateStatement* | *DeleteStatement* | *CopyStatement*
| *CreateStatement*.

SfwStatement = „select“ *PathExpression*
[„from“ *FromClause*]
[„where“ *WhereClause*]
[„order by“ *ValueKey*].

SetValueStatement = „set value (“ *SfwStatement* „) to“ *StringLiteral*.

InsertStatement = „insert (“ *StringLiteral* [„,“ *StringLiteral*] „)“.
DeleteStatement = „delete (“ *SfwStatement* „)“.
CopyStatement = „copy “ [„relation“] „(“ *SfwStatement* „)“.
UpdateStatement = „update“ (*SfwStatement* | *InsertStatement* | *RootKey*)
[„+ =“| „− =“]
(*SfwStatement* | *InsertStatement* | *RootKey*).
CreateStatement = „create relation type“ *StringLiteral*.
FromClause = *PathExpression* [„as“] *StringLiteral*
{„,“ *PathExpression* [„as“] *StringLiteral* }.
Predicate = *Operand* *CompareOperation* *Operand*
| „(“ *Predicate* („and“| „or“) *Predicate* „)“
| „(“ „not (“ *Predicate* „))“.
CompareOperation = „ < “ | „ = “ | „ > “ | „ <= “ | „ >= “ | „ <> “
| „*infix*“ | „*praefix*“ | „*postfix*“.
Operand = *PathExpression* [*ValueKey*] | „$“ *AtomicValue*.
ValueKey = „$value“ | „@“.
RootKey = „$root“ | „∼“.

Anhang C

Konfigurationsdatei für die Erzeugung des Parsers

```
 1  PARSER_BEGIN(Snqlparser)
 2
 3  package snql.parser;
 4
 5  public class Snqlparser {
 6    public void main(String args[]) throws ParseException {
 7      Snqlparser parser = new Snqlparser(System.in);
 8      try {
 9          AstQuery n = parser.Query();
10          System.out.println("SNQL>");
11          n.dump("");
12          System.out.println("Thank␣you.");
13      }
14      catch (Exception e) {
15          System.out.println("Oops.");
16          System.out.println(e.getMessage());
17          e.printStackTrace();
18      }
19    }
20  }
21
22  PARSER_END(Snqlparser)
23
24  SKIP :
25  { "␣" | "\t" | "\n" | "\r" }
26  TOKEN :
27  {
28      <INTEGER: ( ["0"-"9"] )+> | <REAL: (<INTEGER>)?"."(<INTEGER>)+>
29      | <SELECT: "select"> | <FROM: "from">
30      | <WHERE: "where"> | <ORDER_BY: "order␣by"> | <NOT: "not">
31      | <AND: "and"> | <OR: "or"> | <AS: "as">
32      | <EQUAL: "="> | <NOT_EQUAL: "<>"> | <GREATER: ">"> | <LESS: "<">
33      | <PLUSEQUAL: "+="> | <MINUSEQUAL: "-=">
34      | <GREATER_EQUAL: ">="> | <LESS_EQUAL: "<=">
35      | <CREATE: "create"> | <TYPE: "type"> | <INFIX: "infix"> | <PRAEFIX: "praefix">
36      | <POSTFIX: "postfix">
37      | <INVERSE: "inverse"> | <SETVALUE: "set␣value">
38      | <RELATION: "relation"> | <TO: "to">
39      | <INSERT: "insert"> | <UPDATE: "update"> | <COPY: "copy"> | <DELETE: "delete">
40      | <DOLLAR: "$"> | <ROOT: <DOLLAR> "root">
41      | <VALUE: <DOLLAR> "value"> | <QUESTION_MARK: "?"> | <KLEENE: "#">
42      | <LIKE: "%"> | <VAR: "&">
43      | <DOT: "."> | <COMMA: ","> | <ASTERISK: "*"> | <PLUS: "+"> | <LBRACE: "(">
44      | <RBRACE: ")">
45      | <UNION: "|">  | <LBRACK: "["> | <RBRACK: "]"> | <TILDE: "~"> | <AT: "@">
46      | <STRING: ["a"-"z","A"-"Z","_","-"] ( ["a"-"z","A"-"Z","_","-","0"-"9"] )*>
47  }
```

```
48
49  AstQuery Query(): {}
50  {
51      SfwStatement() <EOF> { return jjtThis; }
52      | RenameStatement() <EOF> { return jjtThis; }
53      | InsertStatement() <EOF> { return jjtThis; }
54      | DeleteStatement() <EOF> { return jjtThis; }
55      | CopyStatement() <EOF> { return jjtThis; }
56      | UpdateStatement() <EOF> { return jjtThis; }
57      | CreateRelationStatement() <EOF> { return jjtThis; }
58  }
59
60  void SfwStatement(): {}
61  {
62      SelectClause() [FromClause()] [WhereClause()] [OrderClause()]
63  }
64
65  void SelectClause(): {}
66  {
67      <SELECT> PathExpression()
68  }
69
70  void VarExpression(): {}
71  {
72      <VAR> String()
73  }
74
75  void PathExpression(): {}
76  {
77      (VarExpression() | LabelExpression() | InverseLabelExpression()
78          | RegularPathExpression())
79      ( <DOT>
80      (VarExpression() | LabelExpression() | InverseLabelExpression()
81          | RegularPathExpression()) )*
82  }
83
84  void RegularPathExpression(): {}
85  {
86      LOOKAHEAD(3) <LBRACE> OrRegularPathExpression() <RBRACE>
87      | UnaryRegularPathExpression()
88  }
89
90  void OrRegularPathExpression() #void: {}
91  {
92      (
93          UnaryRegularPathExpression() ( <UNION> UnaryRegularPathExpression() )*
94      )#OrRegularPathExpression(>1)
95  }
96
97  void UnaryRegularPathExpression():{}
98  {
99       <LBRACE> PathExpression() <RBRACE> [RegularOperator() ]
100 }
101
102 void RegularOperator(): {Token t;}
103 {
104   t = <PLUS>  {jjtThis.setValue(t.image);}
105   | t = <ASTERISK> {jjtThis.setValue(t.image);}
106   | t = <QUESTION_MARK> {jjtThis.setValue(t.image);}
107 }
108
109 void LabelExpression(): {}
110 {
111     LabelComponent()
112 }
113
114 void LabelComponent() #void: {}
115 {
116     ( KleeneOperator() | LikeOperator() )
117 }
118
```

```
119  void InverseLabelExpression(): {}
120  {
121      <INVERSE> <LBRACE> LabelComponent() <RBRACE>
122  }
123
124  void PraefixOperator(): {}
125  {
126      String() [ <LIKE> ]
127  }
128
129  void InfixOperator(): {}
130  {
131      <LIKE> String() <LIKE>
132  }
133
134  void PostfixOperator(): {}
135  {
136      <LIKE> String()
137  }
138
139  void LikeOperator() #void: {}
140  {
141      ( PraefixOperator() | LOOKAHEAD(3) InfixOperator() | LOOKAHEAD(3) PostfixOperator())
142  }
143
144  void KleeneOperator(): {}
145  {
146      <KLEENE>
147  }
148
149  void FromClause(): {}
150  {
151      <FROM> VarDeclarationList()
152  }
153
154  void VarDeclarationList()  #void: {}
155  {
156      VarDeclaration() ( <COMMA> VarDeclaration() )*
157  }
158
159  void VarDeclaration(): {}
160  {
161      PathExpression() [ <AS> ] String()
162  }
163
164  void WhereClause(): {}
165  {
166      <WHERE> ConditionalExpression()
167  }
168
169  void ConditionalExpression() #void : {}
170  {
171      OrCondition()
172  }
173
174  void OrCondition() #void : {}
175  {
176    ( AndCondition() ( <OR> AndCondition() )* ) #OrCondition(>1)
177  }
178
179  void AndCondition() #void : {}
180  {
181    ( UnaryExpression() ( <AND> UnaryExpression() )* ) #AndCondition(>1)
182  }
183
184  void NotCondition(): {}
185  {
186      <NOT> UnaryExpression()
187  }
188
189  void UnaryExpression() #void: {}
```

```
190  {
191      <LBRACE> Expression() <RBRACE>
192  }
193
194  void Expression() #void: {}
195  {
196      LOOKAHEAD(3) ConditionalExpression() | PredicateExpression() | NotCondition()
197  }
198
199  void PredicateExpression(): {}
200  {
201      Operand() PredicateOperation() Operand()
202  }
203
204  void PredicateOperation() #void: {}
205  {
206      EqualOperation() | NotEqualOperation() | GreaterThanOperation()
207      | LessThanOperation()
208      | GreaterEqualOperation() | LessEqualOperation() | LikeOperation()
209  }
210
211  void LikeOperation() #void: {}
212  {
213      InfixOperation() | PraefixOperation() | PostfixOperation()
214  }
215
216  void InfixOperation(): {}
217  {
218      <INFIX>
219  }
220
221  void PraefixOperation(): {}
222  {
223      <PRAEFIX>
224  }
225
226  void PostfixOperation(): {}
227  {
228      <POSTFIX>
229  }
230
231  void EqualOperation(): {}
232  {
233      <EQUAL>
234  }
235
236  void NotEqualOperation(): {}
237  {
238      <NOT_EQUAL>
239  }
240
241  void GreaterThanOperation(): {}
242  {
243      <GREATER>
244  }
245
246  void LessThanOperation(): {}
247  {
248      <LESS>
249  }
250
251  void GreaterEqualOperation(): {}
252  {
253      <GREATER_EQUAL>
254  }
255
256  void LessEqualOperation(): {}
257  {
258    <LESS_EQUAL>
259  }
260
```

```
261 void Operand(): {}
262 {
263     LOOKAHEAD(2) PathExpression() [ValueKey()] | LOOKAHEAD(2) Integer()
264     | LOOKAHEAD(2) Real() | <DOLLAR> String()
265 }
266
267 void OrderClause(): {}
268 {
269     <ORDER_BY> ValueKey()
270 }
271
272 void String(): {Token t;}
273 {
274     t = <STRING> {jjtThis.setValue(t.image);}
275 }
276
277 void Integer(): {Token t;}
278 {
279     t = <INTEGER> {jjtThis.setValue(t.image);}
280 }
281
282 void Real(): {Token t;}
283 {
284     t = <REAL> {jjtThis.setValue(t.image);}
285 }
286
287 void RenameStatement(): {}
288 {
289     <SETVALUE> <LBRACE>  SfwStatement() <RBRACE> <TO> String()
290 }
291
292 void InsertStatement(): {}
293 {
294     <INSERT> <LBRACE> String() [ <COMMA> String() ] <RBRACE>
295 }
296
297 void DeleteStatement(): {}
298 {
299     <DELETE> <LBRACE> SfwStatement() <RBRACE>
300 }
301
302 void CopyStatement(): {}
303 {
304     <COPY> [ RelationKey() ] <LBRACE>  SfwStatement() <RBRACE>
305 }
306
307 void AddRelationOperator(): {}
308 {
309     <PLUSEQUAL>
310 }
311
312 void DropRelationOperator(): {}
313 {
314   <MINUSEQUAL>
315 }
316
317 void RelationKey(): {}
318 {
319     <RELATION>
320 }
321
322 void RootKey(): {}
323 {
324     <ROOT> | <TILDE>
325 }
326
327 void ValueKey(): {}
328 {
329     <VALUE> | <AT>
330 }
331
```

```
332 void UpdateStatement(): {}
333 {
334     <UPDATE>
335     (RootKey() | <LBRACE> (InsertStatement() | SfwStatement()) <RBRACE>)
336     <DOT> String() [ <LBRACK> RelationKey() <RBRACK> ]
337     ( AddRelationOperator() | DropRelationOperator() )
338     (RootKey() | InsertStatement() | SfwStatement()) [ RelationKey() ]
339 }
340
341 void CreateRelationStatement(): {}
342 {
343     <CREATE> <RELATION> <TYPE> String()
344 }
```

Literaturverzeichnis

[ABC+01] ADLER, Sharon ; BERGLUND, Anders ; CARUSO, Jeff ; DEACH, Stephen ; GRAHAM, Tony ; GROSSO, Paul ; GUTENTAG, Eduardo ; MILOWSKI, Alex ; PARNELL, Scott ; RICHMAN, Jeremy ; ZILLES, Steve. *Extensible Stylesheet Language (XSL) Version 1.0.* http://www.w3.org/TR/2001/REC-xsl-20011015/. Oktober 2001

[Abi97] ABITEBOUL, Serge: Querying Semi - Structured Data. In: *Proceedings of ICDT* (1997), Januar, S. 1 – 18

[AQM+97] ABITEBOUL, Serge ; QUASS, Dallas ; MCHUGH, Jason ; WIDOM, Jennifer ; WIENER, Janet L.: The Lorel Query Language for Semistructured Data. In: *International Journal on Digital Libraries* 1 (1997), April, Nr. 1, S. 68 – 88

[ASU88] AHO, Alfred ; SETHI, Ravi ; ULLMAN, Jeffrey D.: *Compilerbau.* Addison-Wesley Publishing Company, Inc., 1988. – ISBN 3–89319–150–x

[BC00] BONIFATI, Angela ; CERI, Stefano: Comparative Analysis of Five XML Query Languages. In: *SIGMOD Record* 29 (2000), Nr. 1, S. 68 – 79

[BDHS96] BUNEMANN, Peter ; DAVIDSON, Susan ; HILEBRAND, Gerd ; SUCIU, Dan: A Query Language and Optimization Techniques for Unstructured Data. In: *Proceedings of ACMSIGMOD International Conference on Management of Data*, 1996, S. 505 – 516

[BKLW99] BUSSE, Susanne ; KUTSCHE, Ralf-Detlef ; LESER, Ulf ; WEBER, Herbert: Federated Information Systems: Concepts, Terminology and Architectures / Technische Universität Berlin. 1999. – Forschungsbericht

[BKOS01] BRY, François ; KRAUS, Michael ; OLTEANU, Dan ; SCHAFFERT, Sebastian: Semistrukturierte Daten. In: *Informatik Spektrum* 24 (2001), August, Nr. 4, S. 230 – 233

[BL01] BONIFATI, Angela ; LEE, Dongwon. *Technical Survey of XML Schema and Query Languages.* Januar 2001

[BM00] Behme, Henning ; Mintert, Stefan: *XML in der Praxis*. Addison-Wesley Publishing Company, Inc., 2000. – ISBN 3–8273–1636–7

[BPSM00] Bray, Tim ; Paoli, Jean ; Sperberg-McQueen, C.M. ; Maler, Eve: *Extensible Markup Language (XML) 1.0*. 2. World Wide Web Consortium (W3C), Oktober 2000. – Erhältlich unter http://www.w3.org/XML/xml-V10-2e-errata

[Bun97] Bunemann, Peter: Semistructured Data. In: *16. ACM Symposium on Principles of Database Systems PODS, Tuscon, Arizona* (1997), S. 117 – 121. – Erhältlich unter http://db.cis.upenn.edu/Publications/

[BW95] Böszörményi, László ; Weich, Carsten: *Programmieren mit Modula-3: Eine Einführung in stilvolle Programmierung*. Springer Verlag Berlin Heidelberg, 1995

[Cat94a] Cattell, Roderic Geoffrey G.: *Object Data Management: object-oriented and extended relational database systems*. Addison-Wesley Publishing Company, Inc., 1994

[Cat94b] Cattell, Roderic Geoffrey G. (Hrsg.): *The Object Database Standard: ODMG – 93 Release 1.1*. Morgan Kaufmann Publishers Inc., 1994

[CD99] Clark, James ; DeRose, Steve. *XML Path Language (XPath) Version 1.0*. http://www.w3c.org/TR/xpath. November 1999

[Cod80] Codd, E. F.: Data Models in Database Management. In: Brodie, Michael L. (Hrsg.) ; Zilles, Stephen N. (Hrsg.): *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling, Pingree Park, Colorado, June 23-26, 1980* Bd. 11, ACM Press, 1980. – ISBN 0–89791–031–1, S. 112 – 114

[Con97] Conrad, Stefan: *Föderierte Datenbanksysteme: Konzepte der Datenintegration*. Springer-Verlag Berlin Heidelberg New York, 1997

[CRF00] Chamberlin, Don ; Robie, Jonathan ; Florescu, Daniela: Quilt: An XML Query Language for Heterogeneous Data Sources. In: *International Workshop on the Web and Databases (WebDB), Dallas* (2000), Mai

[Dat95] Date, C. J.: *An Introduction to Database Systems*. 6. Addison-Wesley Bonn Paris Reading, Mass., 1995

[DFF+98] Deutsch, Alin ; Fernandez, Mary ; Florescu, Daniela ; Levy, Alon ; Suciu, Dan. *XML-QL: A Query Language for XML*. http://www.w3.org/TR/1998/NOTE-xml-ql-19980819.html. August 1998

[Eck00] Eckstein, Robert: *XML kurz & gut*. 2. O'Reilly, 2000

[Far99] Farsi, Reza: XML. In: *Informatik Spektrum* 22 (1999), Dezember, Nr. 6, S. 436 – 438

[FFLS99] Fernandez, Mary ; Florescu, Daniela ; Levy, Alon ; Suciu, Dan: A Query Language for a Web-Site Management System. In: *Proceedings of the 8th International World Wide Web Conference*, 1999

[GMPQ+97] Garcia-Molina, Hector ; Papakonstantinou, Yannis ; Quass, Dallan ; Rajaraman, Anand ; Sagiv, Yehoshua ; Ullman, Jeffrey D. ; Vassalos, Vasilis ; Widom, Jennifer: The TSIMMIS Approach to Mediation: Data Models and Languages. In: *Journal of Intelligent Information Systems* 8 (1997), Nr. 2, S. 117 – 132

[HR99] Härder, Theo ; Rahm, Erhard: *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer Verlag Berlin Heidelberg New York, 1999

[MAG+97] McHugh, Jason ; Abiteboul, Serge ; Goldman, Roy ; Quass, Dallan ; Widom, Jennifer: Lore: A Database Management System for Semistructured Data. In: *SIGMOD Record* 26 (1997), Nr. 3, S. 54 – 66

[PGMW95] Papakonstantinou, Yannis ; Garcia-Molina, Hector ; Widom, Jennifer: Object Exchange Across Heterogeneous Information Sources. In: Yu, Philip S. (Hrsg.) ; Chen, Arbee L. P. (Hrsg.): *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, IEEE Computer Society, 1995. – ISBN 0–8186–6910–1, S. 251–260

[Rah94] Rahm, Erhard: *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Addison-Wesley Publishing Company, Inc., 1994

[RLS98] Robie, Jonathan ; Lapp, Joe ; Schach, David. *XML Query Language (XQL*. http://www.w3c.org/TandS/QL/QL98/pp/xql.html. September 1998

[Rob99] Robie, Jonathan. *XQL Tutorial*. http://www.ibiblio.org/xql/xql-tutorial.html. März 1999

[Sch95] Schöning, Uwe: *Theoretische Informatik – kurz gefaßt*. 2. Spektrum Akademischer Verlag Heidelberg, 1995. – ISBN 3–86025–711–0

[Sed92] Sedgewick, Robert: *Algorithmen*. Addison-Wesley Publishing Company, Inc., 1992. – ISBN 3–89319–402–9

[SL90] Sheth, Amit P. ; Larson, James A.: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. In: *ACM Computing Surveys* 22 (1990), September, Nr. 3, S. 183 – 236

[SSE+99] S.Ceri ; S.Comai ; E.Damiani ; P.Fraternali ; S.Paraboschi ; L.Tanca. *XML-GL: A Graphical Language for Querying and Restructuring XML Documents*. 1999

[Suc98] Suciu, Dan: Database Theory Column: An Overvies of Semistructured Data. In: *SIGACT News* 29 (1998), Dezember, Nr. 4, S. 28 – 38. – Erhältlich unter http://www.cs.washington.edu/homes/suciu/

[Tol99] Tolkstorf, R.: XML und darauf basierende Standards: Die neuen Auszeichnungssprachen des Web. In: *Informatik Spektrum* 22 (1999), Dezember, Nr. 6, S. 407 – 421

[W3C01a] DeRose, Steven (Hrsg.) ; Maler, Eve (Hrsg.) ; Ron Daniel Jr. (Hrsg.). *XML Pointer Language (XPointer) Version 1.0*. http://www.w3.org/TR/2001/CR-xptr-20010911/. September 2001

[W3C01b] Chamberlin, Don (Hrsg.) ; Clark, James (Hrsg.) ; Florescu, Daniela (Hrsg.) ; Robie, Jonathan (Hrsg.) ; Siméon, Jérôme (Hrsg.) ; Stephanescu, Mugur (Hrsg.). *XQuery 1.0: An XML Query Language*. http://www.w3.org/TR/2001/CR-xptr-20010911/. Juni 2001

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfasst und nur die
angegebenen Hilfsmittel benutzt zu haben.

(Björn Stadler)