Universität Hamburg
FB Informatik

Diplomarbeit

Wechselseitige Integration von Information Retrieval und Datenbanken

Dezember 1997

Ulrike Steffens

eingereicht bei:

Prof. Dr. Joachim W. Schmidt
Arbeitsbereich Softwaresysteme
Technische Universität Harburg

und

Prof. Dr. Christopher Habel
Arbeitsbereich Wissens- und
Sprachverarbeitung
Universität Hamburg

Danksagung

Ich bedanke mich bei allen Menschen, die diese Arbeit ermöglicht haben.

Mein spezieller Dank gilt Herrn Prof. Dr. Joachim W. Schmidt für die Unterstützung bei der Durchführung der Arbeit und die Schaffung der hervorragenden Arbeitsbedingungen sowie Herrn Prof. Dr. Christopher Habel für die zusätzliche Betreuung dieser Arbeit.

Frau Claudia Niederée danke ich für die zahlreichen Anregungen zum Thema dieser Arbeit und dafür, daß sie trotz vieler anderer Aufgaben immer ein offenes Ohr für meine Fragen und Probleme hatte.

Herrn Gerald Schröder danke ich dafür, daß er mich auch in schwierigen Phasen immer wieder motiviert hat, diese Arbeit in ihre jetzige Form zu bringen.

Herrn Arslan Brömme danke ich dafür, daß er sich stets die Zeit genommen hat, um mir in Diskussionen neue Perspektiven auf das Arbeitsthema zu eröffnen, und dafür, daß er während des letzten Jahres nie die Geduld mit mir verloren hat.

Ein besonderer Dank gilt meiner Familie: Meinen Eltern, die mir meine Ausbildung ermöglicht haben, meinen Großeltern, die stets ein reges Intersse an meiner Arbeit gezeigt und mir damit immer wieder neuen Antrieb gegeben haben, und ganz besonders meiner Mutter, die mir in ihrer ruhigen und geduldigen Art eine ideale Umgebung zur Anfertigung dieser Arbeit geschaffen hat.

Hamburg, 10. Dezember 1997 Ulrike Steffens

Inhaltsverzeichnis

1. Einleitung und Motivation **1**

2. Information Retrieval **3**
2.1 Die semantische Suche in Information Retrieval Systemen 3
2.1.1 Modelle des Information Retrieval 4
2.1.2 Gewichtung von Information Retrieval Eigenschaften 6
2.1.3 Information Retrieval in Hyperlinksystemen 6
2.1.4 Benutzungsschnittstellen für Information Retrieval Systeme 8
2.2 Die Architektur von Information Retrieval Systemen 9
2.2.1 Information Retrieval in verteilten Systemen 10
2.2.2 Datenunabhägigkeit in Information Retrieval Systemen 12
2.2.3 Aktualisierung von Information Retrieval Objekten 13
2.2.4 Information Retrieval im nebenläufigen Mehrbenutzerbetrieb 14
2.3 Typisierung im Information Retrieval 14

3. Datenbanksysteme **16**
3.1 Datenbankfunktionalität 17
3.1.1 Persistenz 17
3.1.2 Mehrbenutzerbetrieb in Datenbanksystemen 17
3.1.3 Datenintegrität 21
3.1.4 Externe Schnittstellen 21
3.2 Die 3-Ebenen-Architektur 22
3.3 Datenmodellierung 23
3.4 Datenbanksprachen 23
3.5 Objektorientierte Datenbanksysteme 24

4. Allgemeine Anforderungen an ein integriertes Datenbank- und Information Retrieval System **25**
4.1 Eigenschaften eines integrierten Systems 25
4.2 Datenunabhängigkeit im integrierten System 26
4.3 Der gewählte Ansatz 27

5. ProM - ein Objektmodell für integrierte Datenbank- und Information Retrieval Systeme **28**
5.1 Vorgehensweise 28
5.2 Das ODMG-Objektmodell als Grundlage für das Objektmodell ProM 29
5.2.1 Der ODMG-Standard 30
5.2.2 Das ODMG-Objektmodell 30
5.2.3 Die Relevanz des ODMG-Standards 32
5.3 Erweiterung um Information Retrieval Komponenten 33
5.4 Typen 34
5.4.1 Definition von Typen 35
5.4.2 Vererbungsrelationen 37
5.4.3 Typeigenschaften 40
5.5 Objekte 40
5.5.1 Erzeugung und Verwaltung von Objekten 40
5.5.2 Objektidentifikatoren 41
5.5.3 Objektnamen 42
5.5.4 Lebensdauer von Objekten 42
5.5.5 Atomare Objekte 42
5.5.6 Kollektionen 42
5.5.7 Strukturierte Objekte 50
5.6 Literale 50
5.6.1 Atomare Literale 50
5.6.2 Kollektionsliterale 53
5.6.3 Strukturierte Literale 53
5.6.4 Leere Literale 55
5.7 Typisierung und Typhierachie 55
5.8 Zustand, Verhalten und Semantik von Objekten 55
5.8.1 Attribute 55
5.8.2 Relationen 57
5.8.3 Operationen 59
5.8.4 Erweiterung um vage Prädikate 61
5.8.5 Erweiterung um vage Relationen 63
5.9 Metadaten 68
5.10 Datenbankfunktionalität 69

6. ProDL - eine Datendefinitionssprache für integrierte Datenbank- und Information Retrieval Systeme **70**
6.1 Vorgehensweise 70
6.2 Die Einordnung von ProDL 71
6.3 Die Spezifikation von Typen in ProDL 72
6.4 Die Spezifikation von Instanzeigenschaften in ProDL 74
6.4.1 Attribute 74
6.4.2 Boolesche und vage Relationen 75

6.4.3 Operationen 78
6.4.4 Vage Prädikate 80

7. ProQL - eine Anfragesprache für integrierte Datenbank- und Information Retrieval Systeme 82
7.1 Vorgehensweise 82
7.2 Die wichtigsten Merkmale der Anfragesprache OQL 82
7.3 Die probabilistische Erweiterung von OQL zu ProQL 83
7.3.1 Syntaktische Ergänzungen 84
7.3.2 Semantische Ergänzungen 85
7.3.3 Die Semantik vager select-from-where-Ausdrücke in ProQL 88
7.3.4 Weitere Optionen für die Realisierung von ProQL 93
7.4 Die formale Definition der Anfragesprache ProQL 93
7.4.1 Anfragen 94
7.4.2 Die Definition benannter Anfragen 94
7.4.3 Elementare Ausdrücke 95
7.4.4 Konstruktionsausdrücke 97
7.4.5 Atomare Typausdrücke 99
7.4.6 Objektausdrücke 101
7.4.7 Kollektionsausdrücke 103
7.4.8 Eindeutige select-from-where-Ausdrücke 106
7.4.9 Vage select-from-where-Ausdrücke in Erweiterung der Sprache OQL . . 108
7.4.10 Ausdrücke für geordnete Kollektionen 110
7.4.11 Binäre Mengenausdrücke 113
7.4.12 Konversionsausdrücke 114
7.4.13 Funktionsaufrufe 117
7.4.14 Sichtbarkeitsregeln 117

8. Eine prototypische Implementation der Anfragesprache ProQL in TL2 119
8.1 Der Umfang der Implementation 119
8.2 Tycoon-2 als Basis der Implementation 120
8.3 Das Ergebnis der prototypischen Implementation 121

9. Zusammenfassung 122

10 Ausblick 124

A. Beispiel zur Verdeutlichung der Konzepte von ProM, ProDL und ProQL 125

B. Die Syntax von ProDL 131

C. Die Syntax der Anfragesprache ProQL 144
C.1 Die Grammatik 145
C.2 Die Reihenfolge bei der Operatorauswertung 151

Literaturverzeichnis **153**

Abbildungsverzeichnis

2.1 Information Retrieval im Vektorraummodell 5
2.2 Die Struktur der Datenobjekte im Hyperlinksystem 7
2.3 Architekturschema eines Information Retrieval Systems 9

3.1 Das ANSI/X3/SPARC-Architekturmodell 22

5.1 Die Entwicklung des Objektmodells ProM durch Erweiterung eines Datenbankmodells um Information Retrieval Komponenten 29
5.2 Die Instanziierung von Objekt- und Literaltypen 35
5.3 Eine Gegenüberstellung von Semantik und Verhalten 36
5.4 Die Verwendung verschiedener Komponenten zur Typspezifikation 36
5.5 Die Unterscheidung zwischen Spezifikation und Implementation 37
5.6 Die Vererbung über Subtyp- und EXTENDS-Relation 38
5.7 Die Komponenten der Subtyprelation 39
5.8 Die Komponenten der EXTENDS-Relation 39
5.9 Die grundlegenden Konzepte von Objekttypen und Objekten 40
5.10 Veränderbare und nicht veränderbare Kollektionen 44
5.11 Die Relevanzrangfolge als Kollektionsform 48
5.12 Die grundlegenden Konzepte von Literaltypen und Literalen 50
5.13 Strukturen und ihre Komponenten 54
5.14 Die vordefinierte Typhierarchie des Objektmodells ProM 56
5.15 Instanzeigenschaften 57
5.16 Die Spezifikation von Relationen 58
5.17 Die Umsetzung semantischer Eigenschaften durch vage Prädikate 61
5.18 Die boolesche Relation verwendet/verwendet_in zwischen Projekten und Publikationen 63
5.19 Die vage Relation beschrieben_von/betrifft zwischen Projekten und Publikationen 64
5.20 Die Bildung von vagen Relationen 67
5.21 Unlogische Ergebnisse bei der Mehrfachtraversierung vager Relationen für ein Hyperlinksystem 68
5.22 Die Erstellung und Nutzung von Metadaten 69

6.1 Die Erweiterung der Definitionssprache IDL zu ODL und ProDL 71
6.2 ProDL als gemeinsame Schnittstelle zwischen Programmiersprachen und Systemen 72

7.1 Das Verhältnis zwischen dem ODMG-Standard, dem Objektmodell ProM und der Anfragesprache ProQL 83

1. Einleitung und Motivation

Datenbank- und Information Retrieval Systeme erfüllen, auf den ersten Blick betrachtet, ähnliche Aufgaben. Beide ermöglichen die effiziente und gezielte Suche auf unter Umständen sehr großen Datenbeständen. Diese Suche vollzieht sich jedoch jeweils in einem anderen Kontext, so daß Datenbanksysteme oft dort ihre Stärken haben, wo Information Retrieval Systeme Schwächen aufweisen und umgekehrt. Durch die zunehmende Globalisierung von Informationssystemen und die sich laufend verbessernden Möglichkeiten zur Verarbeitung von Multimediadaten wachsen die Anwendungsfelder des Information Retrieval und der Datenbanken jedoch immer weiter zusammen. Daher muß langfristig die Verwendung eines einheitlichen Systems, das auf das Potential beider Bereiche zurückgreift, für die Verwaltung und Verarbeitung von Daten unterschiedlichen Formats und Inhalts in Betracht gezogen werden.
Für den Aufbau eines solchen integrierten Systems müssen zunächst die Eigenschaften von Datenbank- und Information Retrieval Systemen getrennt voneinander herausgearbeitet werden. Information Retrieval Systeme realisieren Anfragen anhand inhaltlicher Aspekte (vgl. Kapitel 2). Die den Daten anhaftende Semantik ist dabei sowohl durch den Kontext der Anfrage als auch durch den Entstehungskontext der Daten selbst nicht eindeutig erfaßbar. Information Retrieval Systeme begegnen dieser Vagheit, indem sie zum einen auf speziell darauf abgestimmte Modelle zurückgreifen und zum anderen von umfangreichen Metadaten bezüglich der Semantik der verwalteten Daten Gebrauch machen. Die Ausarbeitung aller weiteren Aspekte von Information Retrieval Systemen steht hinter dem Umgang mit der vagen Semantik zurück. Die Systemarchitektur läßt zumeist weder eine Verteilung noch einen nebenläufigen, verändernden Zugriff mehrerer Benutzer auf die verwalteten Daten zu. Des weiteren liegen Information Retrieval Daten häufig ohne jede Struktur flach in Dateien vor.
Datenbanksysteme hingegen gewährleisten die persistente Verwaltung von Massendaten (vgl. Kapitel 3). Die Einhaltung von Architekturkonzepten ermöglicht hier einen effizienten nebenläufigen Zugriff mehrerer Benutzer auf denselben Datenbestand. Zudem können die Struktur der Daten und ihre Beziehungen untereinander mit Hilfe von Datenmodellen repräsentiert und unter Rückgriff auf Typisierungskonzepte realisiert werden. Dennoch bleibt die Perspektive auf Datenbankdaten stets eine eindeutige. Die Aspekte der vagen Semantik, wie sie in Information Retrieval Systemen umgesetzt sind, werden in Datenbanksystemen ausgeblendet, obwohl sie auch innerhalb eines strukturierten Datenbestandes zweckmäßig eingesetzt werden könnten.

Bei der Entwicklung eines integrierten System sollten die Vorteile von Datenbank- und Information Retrieval Systemen zusammengeführt und ihre Nachteile ausgeglichen werden. Die Zielsetzung der vorliegenden Arbeit ist es daher, erste Ansätze zur Entwicklung eines Systems aufzuzeigen, das einen nebenläufigen, aktualisierenden Mehrbenutzerbetrieb sowohl auf Daten mit eindeutiger als auch auf Daten mit vager Semantik ermöglicht, die persistent innerhalb des Systems abgelegt sind. Eines der wichtigsten Merkmale dieses integrierten Datenbank- und Information Retrieval Systems muß dabei seine übersichtliche Architektur sein, die die Basis für die Strukturierung der umfangreichen im System anfallenden Aufgaben bildet. Die Verwendung eines bereits bestehenden Architekturkonzeptes ist hier einer der vorstellbaren Lösungsansätze. Die vorliegende Arbeit unterteilt deshalb das integrierte Datenbank- und Information Retrieval System, dem ANSI/X3/SPARC-Architekturmodell für Datenbanksysteme folgend, in eine interne, eine konzeptuelle und eine externe Ebene. Die Funktionalität, die diesen Ebenen in Datenbanksystemen zukommt, bleibt dabei erhalten und wird um die für eine Integration von Information Retrieval Konzepten notwendigen Aufgaben erweitert (vgl. Kapitel 4).

Anhand der drei Systemebenen kann das Vorgehen bei der Entwicklung eines integrierten Datenbank- und Information Retrieval Systems gegliedert werden. Innerhalb dieser Arbeit wird dabei ein Schwerpunkt auf die externe und die konzeptuelle Ebene des Systems gesetzt, wobei der ODMG-Standard als anerkannter Standard für objektorientierte Datenbanksysteme die Grundlage bildet.

Auf der konzeptuellen Ebene wird ein Modell zur Repräsentation sowohl von Datenbank- als auch von Information Retrieval Daten entwickelt (vgl. Kapitel 5). Hier werden insbesondere Modellierungskomponenten bereitgestellt, die die Repräsentation einer vagen Semantik für Datentypen mit beliebiger Struktur ermöglichen, da solche Komponenten in bereits existierenden Datenbankmodellen fehlen.

In Ergänzung dieses Modells wird eine Datendefinitionssprache konzipiert, in der die konkret aus der Modellierung hervorgehenden Datenschemata formuliert werden können (vgl. Kapitel 6).

Als Benutzungsschnittstelle auf der externen Ebene des integrierten Systems wird im Rahmen der vorliegenden Arbeit eine Anfragesprache entwickelt, die auf den Konzepten des obigen Modells aufsetzt. Hierdurch werden Anfragen auf Daten mit eindeutiger und mit vager Semantik gleichermaßen unterstützt (vgl. Kapitel 7).

Die erarbeiteten Konzepte werden schließlich praktisch erprobt, indem sie zur Implementation einer prototypischen Schnittstelle für die Anfragesprache des integrierten Systems herangezogen werden (vgl. Kapitel 8).

Die vorliegende Arbeit schließt mit einer kurzen Zusammenfassung (vgl. Kapitel 9) und einem Ausblick, der insbsondere weitere Schritte bei der Realisierung der internen Ebene eines integrierten Datenbank- und Information Retrieval Systems umfaßt (vgl. Kapitel 10).

2. Information Retrieval

Traditionell wird die Aufgabe von Information Retrieval Systemen als die Suche auf Textdaten beschrieben. Die Konzepte des Information Retrieval können jedoch auf Daten von beliebigem Format ausgedehnt werden.
In klarer Abgrenzung zu Datenbanksystemen vollzieht sich die Suche anhand von semantischen Aspekten. Durch diese inhaltsorientierte Betrachtungsweise ergeben sich Anforderungen, die für das Information Retrieval spezifisch sind. Diese bilden den Schwerpunkt der bisherigen Forschung auf diesem Gebiet [Fuh96a] (vgl. Abschnitt 2.1).
Andere Zusammenhänge gewinnen erst nach und nach das Interesse, besitzen jedoch durchaus Relevanz für Information Retrieval Prozesse an sich und für ihre Integration in einen breiteren Anwendungskontext [Cro95]. So bedürfen Punkte, wie beispielsweise die Einpassung von Information Retrieval Systemen in verteilte Umgebungen, die Datenunabhängigkeit in Information Retrieval Systemen, die Einräumung von verändernden Mehrbenutzerzugriffen auf Information Retrieval Objekte[1] oder auch die Verwendung komplexer Datentypen, weiterer Betrachtung (vgl. Abschnitte 2.2 und 2.3).

2.1 Die semantische Suche in Information Retrieval Systemen

Die Suche in Information Retrieval Systemen erfolgt unter semantischen Aspekten. Somit werden nicht bloße Worterkennungsalgorithmen ausgeführt, sondern es geht vielmehr darum, Objekte zu suchen, deren Bedeutung zu einer entsprechenden Anfrage paßt.
Im Zusammenhang mit Information Retrieval fallen häufig die Begriffe "vage Anfragen" oder "unsicheres Wissen". Hiermit soll ausgedrückt werden, daß die Antworten auf Anfragen an ein Information Retrieval System a priori nicht eindeutig definierbar sind. Für diese Unsicherheit sind drei Gründe aufzuführen, die sich hauptsächlich aus dem Anspruch der semantischen Suche ableiten [Fuh92b]:

- Die Unsicherheit ist im angefragten Wissen selbst begründet. Ein Beispiel hierfür ist das Wissen, das in wissenschaftlichen Fakten-Datenbanken repräsentiert ist. Dieses Wissen ist oftmals unvollständig oder begründet sich auf Meßwerten begrenzter Genauigkeit.

[1] Der Begriff "Objekt" wird in diesem Kapitel für Daten von beliebigem Format benutzt

- Die Repräsentation des Wissens bzw. der Bedeutung von Objekten in Information Retrieval Systemen ist unvollständig, da die semantischen Aspekte der zugrundeliegenden Daten zu vielfältig sind, als daß sie vollständig algorithmisch erkannt werden könnten.
- Die Regeln, die ein Information Retrieval System zur Auswertung von Anfragen nutzt, können die Zusammenhänge zwischen den einzelnen semantischen Aspekten der Daten nur unzulänglich darstellen.

Auf dem Gebiet des Information Retrieval wurden verschiedene theoretische Modelle entwickelt, die sowohl die Repräsentation von Information Retrieval Objekten und Anfragen als auch die Auswertung dieser Repräsentationen beschreiben. Diese tragen der Vagheit des Wissens bei der semantischen Suche unterschiedlich stark Rechnung. Grundlegende Konzepte, wie die Gewichtung von semantischen Eigenschaften, können die Anschaulichkeit dieser Modelle fördern. Im Hinblick auf inhaltlich vernetzte Systeme, wie beispielsweise das World-WideWeb, sollten nicht nur die semantischen Aspekte einzelner Objekte untersucht werden, sondern auch die inhaltlichen Verflechtungen zwischen verschiedenen Objekten. Weiterhin kann eine angemessene Gestaltung der Benutzungschnittstellen von Information Retrieval Systemen den Anfragenden die semantische Suche wesentlich erleichtern.

2.1.1 Modelle des Information Retrieval

Die theoretischen Modelle des Information Retrieval lassen sich grob in zwei Kategorien einteilen.
Information Retrieval Systeme, die einem Modell mit exakter Übereinstimmung (*exact-match models*) folgen, ermitteln für jedes Objekt einen Wahrheitswert, der angibt, ob das Dokument für eine Anfrage Relevanz besitzt oder nicht. Systeme dieser Art realisieren also eine Abbildung $f : O \times A \longrightarrow \{0, 1\}$, wobei A die Menge der möglichen Anfragen und O die Menge der zur Verfügung stehenden Objekte beschreibt. Der Ansatz der exakten Übereinstimmung orientiert sich somit an der booleschen Suche, wie sie in Datenbanksystemen durchgeführt wird. [TC92]
Durch die Unsicherheit bei der semantischen Suche sind jedoch beliebige Abstufungen in der Relevanz, die ein Objekt für eine Anfrage besitzt, denkbar. Diese Abstufungen werden beim Information Retrieval mit exakter Übereinstimmung nicht berücksichtigt. Information Retrieval Systeme, die dem exakten Ansatz folgen, ermitteln daher häufig Ergebnismengen, die entweder wichtige Objekte nicht enthalten oder unübersichtlich groß werden. Der Umgang mit der semantischen Unsicherheit wird schließlich auf den Benutzer verlagert, dem die Aufgabe zukommt, durch umständliche Anfragen und durch eingehende Begutachtung der Ergebnismengen dennoch ein angemessenes Suchergebnis zu erzielen.
Ein Ansatz, der der Problematik des unsicheren Wissens eher gerecht wird, ist das Information Retrieval nach einem Modell mit bestmöglicher Übereinstimmung (*best-match model*). Hier werden den Objekten der Datensammlung keine Wahrheitswerte zugewiesen, sondern Retrieval-Statuswerte, die einen Vergleich der Objekte anhand ihrer Relevanz für eine Anfrage zulassen. Das Ergebnis dieses Vergleiches ist eine Relevanzrangfolge (*relevance ranking*)

der ausgewerteten Objekte. Formal läßt sich das Vorgehen beim Information Retrieval mit bestmöglicher Übereinstimmung als Abbildung $f : D \times A \longrightarrow [0,1]$ beschreiben. [TC92]
Die bekanntesten Ausprägungen der Modelle mit bestmöglicher Übereinstimmung sind das Vektorraummodell und das probabilistische Modell.

- Im Vektorraummodell werden Eigenschaften betrachtet, die von Objekten der Datensammlung erfüllt bzw. von Anfragen gefordert werden. Dabei erfüllt ein Objekt eine Eigenschaft zu einem gewissen Grad, ebenso wie eine Anfrage eine Eigenschaft zu einem gewissen Grad fordert. Im Modell werden nun Vektoren für die Anfrage und alle Objekte erstellt, wobei jede Vektorkomponente genau dem jeweiligen Erfüllungs- bzw. Anforderungsgrad für eine Eigenschaft entspricht. Die Auswertung der Anfrage wird durch Ermittlung der Distanz zwischen den Objektvektoren und dem Anfragevektor vorgenommen (vgl Abb. 2.1). Objekte, deren Vektoren einen geringen Abstand zum Anfragevektor aufweisen, erhalten einen hohen Retrieval-Statuswert und werden in der Relevanzrangfolge entsprechend weit oben eingeordnet.

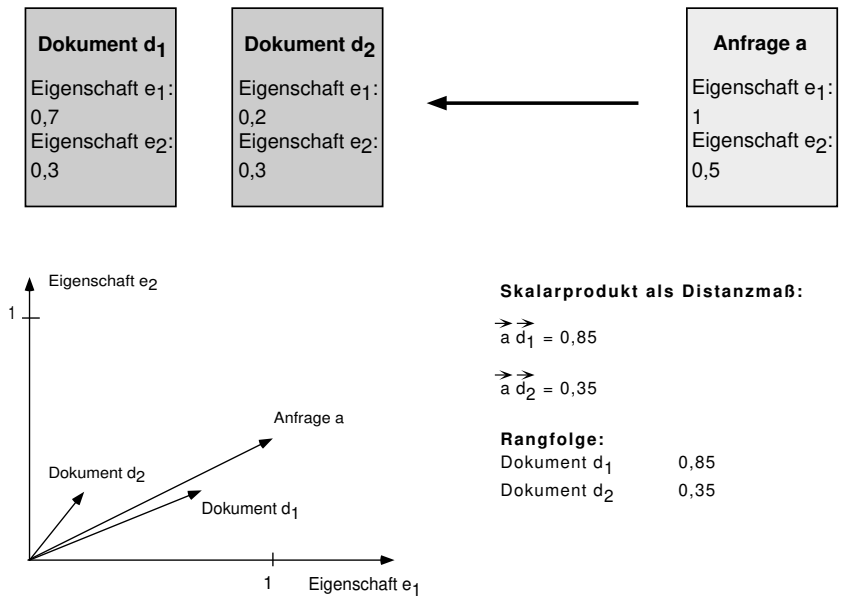


Abbildung 2.1: Information Retrieval im Vektorraummodell

- Im probabilistischen Modell erfolgt die Anfrageauswertung mit Hilfe von Wahrscheinlichkeiten $P(B \leftarrow o_m)$ für das Ereignis, daß ein Objekt o_m eine Anfragebedingung B erfüllt. Diese Wahrscheinlichkeiten werden genutzt, um die Relevanzrangfolge unter den Objekten der Datenbasis festzulegen.

Modelle mit bestmöglicher Übereinstimmung stellen eine wirklichkeitsnähere Abbildung der Gegebenheiten bei der semantischen Suche dar. Die Möglichkeit, eine Relevanzrangfolge zu

erstellen, entspricht der Vorstellung, daß Objekte mehr oder weniger relevant für eine Anfrage sein können. Eine Modellierung über Wahrscheinlichkeiten, wie sie im probabilistischen Modell vorgenommen wird, macht zusätzlich deutlich, daß verschiedene Anfragende sowie die Autoren der Objekte die jeweilige Semantik unterschiedlich beurteilen [Fuh92b].

2.1.2 Gewichtung von Information Retrieval Eigenschaften

Ein grundlegendes Konzept, um der Unsicherheit des Wissens im Information Retrieval entgegenzuwirken, ist die Gewichtung von Information Retrieval Eigenschaften. Diese Eigenschaften können von unterschiedlicher Natur sein. Bei der Volltextsuche handelt es sich zumeist um Wörter oder Wortterme, die in Texten enthalten sind. Doch auch das Auftreten bestimmter Muster in Bilddokumenten oder Formatierungsangaben in einer Dokumentenbeschreibungssprache sind Beispiele für Eigenschaften von Information Retrieval Objekten und sollen an dieser Stelle die Vielfalt der Möglichkeiten widerspiegeln. Die Gewichtung der Eigenschaften kann jeweils bei der Aufbereitung der Objekte oder bei der Auswertung von Anfragen erfolgen [Fuh92a].
Die Indizierung von Information Retrieval Objekten kann über die bloße Feststellung des Auftretens einer Eigenschaft in einem Objekt hinausgehen [vR79]. So kann beispielsweise die Vorkommenshäufigkeit eines Terms in einem Text relativ zu seiner Vorkommenshäufigkeit in der gesamten Datensammlung herangezogen werden. Die Textstelle, an der ein Term auftritt, kann zusätzlich Aufschluß darüber geben, wie sehr der Term den Textinhalt beeinflußt. Eine Gewichtung, die derartigen Überlegungen gerecht wird, ermöglicht es, zwischen Objekten, die von einer Eigenschaft nur am Rande betroffen sind, und solchen, in denen die Eigenschaft stark hervortritt, zu unterscheiden.
Ein ähnliches Ziel verfolgt die Gewichtung von Eigenschaften auf der Anfrageseite. Hier werden Gewichte für die einzelnen Eigenschaften festgelegt, die von der Anfrage gefordert werden. Diese Gewichte beschreiben, zu welchem Grad die Relevanz eines Ergebnisobjektes von der jeweiligen Anfrageeigenschaft abhängig ist. Die Gewichtung von Anfrageeigenschaften kann sowohl automatisch bei der Repräsentation der Anfrage durch das Information Retrieval System erfolgen als auch direkt durch den Benutzer vorgegeben werden, falls die Anfragesprache des Systems dieses erlaubt.

2.1.3 Information Retrieval in Hyperlinksystemen

Hyperlinksysteme, wie z.B. das WorldWideWeb, enthalten eine Vielzahl an Informationen, auf denen semantische Suchverfahren sinnvoll sind. In der Strukturierung der Datenobjekte unterscheiden sie sich jedoch deutlich von traditionellen Information Retrieval Systemen. Während Datenobjekte dort als atomare Einheiten betrachtet werden, die einander nicht beeinflussen, müssen für Hyperlinksysteme sowohl die Strukturen innerhalb von einzelnen Dokumenten[2] als auch die Verknüpfungen zwischen verschiedenen Dokumenten betrachtet werden [FR97].

[2]Datenobjekte in Hyperlinksystemen werden im Folgenden auch als Dokumente bezeichnet.

Dokumente in Hyperlinksystemen bestehen zumeist aus einzelnen, auch multimedialen Komponenten, die ihrerseits ebenfalls wieder als Dokumente betrachtet werden können. Daneben sind die Dokumente untereinander zum Teil durch Verweise verbunden (vgl. Abb. 2.2). So kann eine typische Seite im WorldWideWeb verschiedene Textabschnitte, Abbildungen, Video- oder Audiodaten und sogar Programme enthalten. Zudem existieren Verknüpfungen mit verschiedenen anderen Seiten, Komponenten anderer Seiten oder Komponenten derselben Seite.

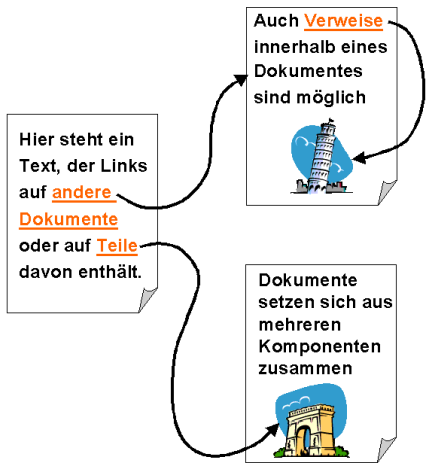


Abbildung 2.2: Die Struktur der Datenobjekte im Hyperlinksystem

Für das Information Retrieval ergeben sich hieraus neue Aspekte. Da Benutzer, die semantische Suchanfragen auf Dokumenten eines Hyperlinksystems stellen, mit der Struktur dieser Dokumente nicht vertraut sind, muß das entsprechende Information Retrieval System von dieser Struktur abstrahieren. Dieses kann durch die Konstruktion von Kontexten erreicht werden.
Theoretisch ist ein Kontext ein Teil eines oder mehrerer Datenobjekte, der auf sinnvolle Art und Weise betrachtet werden kann. Eine einfache praktische Repräsentation von Kontexten ist die Konjunktion aller Eigenschaften, die auf die Komponenten eines oder mehrerer Datenobjekte zutreffen. Bei der Anfrageauswertung wird aus dem jeweiligen Kontext dann genau das Minimum der Komponenten ermittelt, das notwendig ist, um den Informationsbedarf der Anfrage abzudecken. Hierdurch werden die nicht relevanten Teile eines Datenobjektes ausgeblendet. So erhält der Benutzer z.B. anstelle eines ganzen Artikels nur die Absätze, die mit seiner Anfrage korrespondieren. [FR97]
Da Kontexte aus Komponenten verschiedener Datenobjekte konstruiert werden, können in ihnen semantische Widersprüche enthalten sein. In diesen Fällen müssen die inkonsistenten Komponenten identifiziert und isoliert betrachtet werden, damit keine fehlerhaften Annahmen über den gesamten Kontext entstehen. Eine mehrwertige Logik, die den Wert "inkonsistent" enthält, bildet den Rahmen für den Umgang mit sich widersprechenden Kom-

ponenten. [FR97]
Durch die Verknüpfung verschiedener Dokumente erweitert sich deren Semantik. So ist beispielsweise zu beachten, daß ein Dokument, das auf ein anderes, relevantes Dokument verweist, hierdurch selbst ebenfalls eine indirekte Relevanz erzielt. Zusammenhänge dieser Art können durch Ergänzung einfacher Regeln (vgl. z.B. [FR97]) bei der Analyse dargestellt werden.
Sind alle Dokumente eines Hyperlinksystems indiziert und ihre Semantik somit für die Anfrageauswertung zugänglich, so erhöht die zusätzliche Beachtung von Verweisen die Retrievaleffektivität allerdings nicht. Der Inhalt eines jeden Dokumentes, das über Verweise erreicht werden kann, ist ohnehin schon im Retrievalindex erfaßt. Da Hyperlinksysteme aber multimedial ausgerichtet sind, existiert dort eine Vielzahl an Dokumenten, wie z.B. Abbildungen oder Videosequenzen, die mit derzeitigen Analysemethoden nur unzulänglich oder gar nicht für das Retrieval erschlossen werden können. Verweisen indizierte Dokumente auf nicht indizierbare Multimediadokumente, so kann eine Analyse dieser Verweise eine direkte semantische Untersuchung der Multimediadokumente zumindest teilweise ersetzen. [DvR93]
Die Gesamtheit der indizierten Dokumente, die auf ein nicht indizierbares Dokument verweisen, kann zu einer Gruppierung (*cluster*) zusammengefaßt werden. Innerhalb dieser Gruppierung wird dann für jeden Term des Indexes eine mittlere Gewichtung ermittelt. Der Vektor der gewichteten Eigenschaften wird als Deskriptor für das nicht indizierbare Dokument eingesetzt und unabhängig von den verweisenden Dokumenten zur Anfrageauswertung herangezogen. [DvR93]
Die Ermittlung von Deskriptoren für Multimediadokumente ist jedoch nur dann angebracht, wenn im Hyperlinksystem ein angemessenes Verhältnis von indizierten zu nicht indizierbaren Dokumenten besteht: Existiert für jedes Multimediadokument jeweils nur ein Verweis aus einem indizierten Dokument, so wird die Repräsentation durch Deskriptoren zu einer Variante typischer Suchverfahren auf multimedialen Daten, die mit textuellen Beschreibungen dieser Daten arbeiten. Wird hingegen aus allen indizierten Dokumenten auf ein nicht indizierbares Dokument verwiesen, so wird dessen Deskriptor zu einem Deskriptor der gesamten Datenbasis. [DvR93]

2.1.4 Benutzungsschnittstellen für Information Retrieval Systeme

Da die Möglichkeiten, die Semantik zu erfassen, die ein Objekt oder eine Anfrage für einen Benutzer besitzen, beschränkt sind, ist es notwendig, eine der semantischen Suche angemessene externe Schnittstelle zu Verfügung zu stellen [Fuh92b].
Zum einen sollte das System die ihm vorliegenden Informationen an den Benutzer weitergeben. Wird beispielsweise eine Relevanzrangfolge ermittelt, so muß diese optisch dargebracht werden, damit ein Anfragender die für ihn relevanten Objekte ohne Probleme finden kann. Beim Sichten der Objekte kann die optische Hervorhebung (*highlighting*) der angefragten Eigenschaften, die das Objekt beinhaltet, hilfreich sein.
Die Möglichkeit zur Interaktion mit dem Information Retrieval System beeinflußt die Qualität des Suchergebnisses und ist daher in vielen Systemen gegeben. Information Retrieval Systeme unterscheiden sich hier von Datenbanksystemen, die auch aufeinanderfolgende An-

fragen i.d.R. als unabhänig voneinander betrachten [Fuh92b]. Der Benutzer verliert seinen Einfluß auf die Datenbanksuche, sobald er seine Anfrage an das System absetzt. Information Retrieval Systeme greifen auf das Wissen des Benutzers selbst zurück, indem dieser den schrittweisen Ablauf eines Anfrageprozesses interaktiv mitgestalten kann. Bei der Relevanzrückkopplung (*relevance feedback*) beispielsweise erhält der Benutzer die Möglichkeit, die Objekte, die ein Anfrageergebnis enthält, zu bewerten. In einem nächsten Schritt paßt dann das Information Retrieval System seine ursprüngliche Repräsentation der Anfrage an die durch die Bewertung des Benutzers gewonnenen zusätzlichen Informationen an und wertet diese revidierte Anfragerepräsentation erneut aus. Hierdurch kann die Ergebnismenge schrittweise immer mehr dem Informationsbedarf des Benutzers angenähert werden. Durch differenzierte Beurteilungskriterien bei der Relevanzrückkopplung können somit effektive[3] Anfrageergebnisse erreicht werden [Fuh92b].

2.2 Die Architektur von Information Retrieval Systemen

Die wichtigste Datenstruktur vieler Information Retrieval Systeme ist der Retrieval Index (vgl. Abbildung 2.3). Dieser wird während einer umfangreichen Analyse erstellt, wobei häufig linguistische Zusatzdaten, wie zum Beispiel Stoppwortlisten oder Thesauri, zum Einsatz kommen [Ste96]. Die Verwaltung des Indexes vollzieht sich über das Dateisystem [VAB96]. Der Retrieval Index liegt meist in Form einer invertierten Datei (*inverted file*) vor. Eine invertierte Datei enthält für jeden Term der Datensammlung einen Eintrag, in dem vermerkt ist, in welchen Datenobjekten dieser Term vorkommt. Zusätzlich können hier auch Informationen wie Gewichtungen oder Angaben darüber, wo genau der Term im Datenbestand erscheint, abgelegt sein [BCCM94].

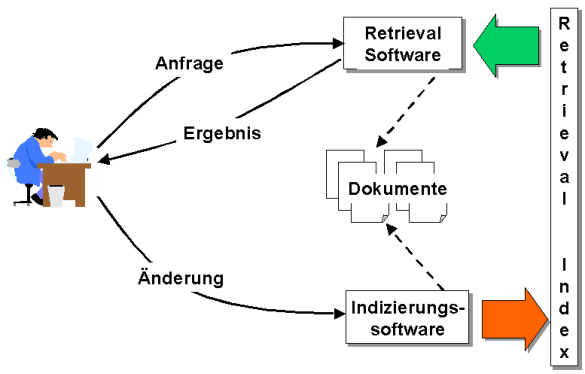


Abbildung 2.3: Architekturschema eines Information Retrieval Systems

[3] Im Information Retrieval wird zwischen der Effektivität und der Effizienz der Auswertung unterschieden. Die Effektivität bezieht sich auf die inhaltliche Qualität der Ergebnismenge. Die Effizienz wird durch gebräuchliche Parameter für die Leistung von Systemen beschrieben.

Die Anfrageauswertung erfolgt, wiederum unter Benutzung von Daten, wie beispielsweise Stoppwortlisten, über den Index. Die eigentlichen Datenobjekte werden zur Auswertung nicht mehr herangezogen. Hierdurch wird erreicht, daß die sprachliche Analyse der Datenobjekte vor und nicht während der Anfrageauswertung erfolgt. Die Analyseergebnisse liegen im Index in geeigneter Form vor und können von der Auswertungsroutine effizient genutzt werden [Fuh96a]. Der Umfang des Indexes, seine Struktur und die enthaltenen Metadaten sind somit entscheidend für die Geschwindigkeit der Auswertung und für die Qualität des Anfrageergebnisses. Auf die eigentlichen Datenobjekte wird erst nach der Anfrageauswertung zugegriffen, indem der Benutzer sie aus der Ergebnisrangfolge auswählt, weil sie seinem Informationsbedarf zu entsprechen scheinen.
Zumeist gewährt das System Benutzern nur lesenden Zugriff auf die Datenobjekte [Son96]. Eine Aktualisierung der Datensammlung ist nur den Administratoren des Information Retrieval Systems gestattet und wird zumeist außerhalb des Programmbetriebes durchgeführt. Um Änderungen für den Auswertungsvorgang sichtbar zu machen, ist die Anpassung des Indexes an den aktuellen Inhalt der Datenbasis notwendig. Häufig wir der Index auch nach geringfügigen Änderungen komplett neu erstellt (vgl. Abschnitt 2.2.3).
Das beschriebene Architekturschema soll ein performantes und effektives Information Retrieval unterstützen [Fuh96a]. Als Anwendungskontext wird hierbei die Suche durch einzelne Benutzer angenommen, die einen lokal begrenzten, lesenden Zugriff auf bestimmte Daten benötigen. Mit zunehmender Vernetzung und der Einführung von Informationssystemen zur Unterstützung von Geschäftsprozessen erlangen Information Retrieval Algorithmen jedoch eine zusätzliche Bedeutung. Sie werden zum Bestandteil von verteilten Systemen [CLC95], die einer Vielzahl von Benutzern sowohl die semantische Suche auf Daten als auch den nebenläufigen, verändernden Zugriff auf diese Daten ermöglichen sollen [KS96]. Die Datensammlung soll dabei möglichst einfach ergänzt und von verschiedenen Anwendungen, auch außerhalb des Information Retrieval, genutzt werden können [Cro95]. Hierdurch erwachsen neue Anforderungen an Information Retrieval Systeme, die von deren üblichem Architekturschema nur unzureichend berücksichtigt werden.

2.2.1 Information Retrieval in verteilten Systemen

Die voranschreitende globale Vernetzung und die damit verbundenen großen Datenmengen, wie sie beispielsweise im WorldWideWeb zur Verfügung stehen, machen Information Retrieval für verteilte Anwendungen interessant. Verschiedene Forschungsprojekte haben sich in diesem Bereich bereits entwickelt [CM96], [CLC95], [FY93]. Die angestrebten Lösungen sollen vielen Benutzern gleichzeitig einen einfachen und schnellen Zugriff auf verteilte Information Retrieval Datensammlungen ermöglichen. Hierdurch entstehen zusätzliche Anforderungen an die Architektur von Information Retrieval Systemen. So müssen Zugriffsinformationen für Datenbestände verwaltet werden, die sich in ihrem technischen Aufbau und ihren Inhalten stark voneinander unterscheiden können. Dieses sollte in einer Form geschehen, die eine automatische Auswahl einer Menge von Datensammlungen für eine Anfrage durch das Information Retrieval System ermöglicht. Die Ergebnisse der verschiedenen Datenbestände wiederum, die alle einer lokalen Relevanzrangfolge unterliegen, müssen in

eine einheitliche globale Rangfolge gebracht werden. Zusätzlich gilt es, die Rechenzeit des Systems so aufzuteilen, daß sowohl eine effiziente Anfrageauswertung als auch erträgliche Zugriffszeiten auf entfernte Datensammlungen erreicht werden.

Die Zahl der verfügbaren Datensammlungen, die über Weitverkehrsnetze zugänglich sind, ist groß. Der Zugriff auf sie bleibt jedoch schwierig, da es sich um Daten unterschiedlicher Struktur handelt, die oft durch heterogene Informationssysteme verwaltet werden. Häufig ermöglicht nur eine genaue Kenntnis der jeweiligen Systemschnittstelle und der Datenstrukturen den Zugang. Nicht immer verwenden die beteiligten Information Retrieval Systeme homogene Modelle, Stoppwortlisten oder Algorithmen zur Stammformreduktion. Hierdurch wird sowohl eine einheitliche rechnerinterne Repräsentation der Anfragen, die für alle verwendeten Datensammlungen einsetzbar ist, erschwert als auch die Ermittlung eines einheitlichen Anfrageergebnisses [CLC95].

Auch wenn die Datenobjekte des Netzes durch einheitliche Information Retrieval Systeme verwaltet werden, ergeben sich bei Anfragen an mehrere Datensammlungen semantische Schwierigkeiten. Bei der Masse der zur Verfügung stehenden Information kann kaum noch ermittelt werden, welcher Datenbestand Objekte zu welcher Themenkategorie beinhaltet. Über Verzeichnisdienste, wie den des Systems WAIS [SBGK94], wird versucht, möglichst viele Metainformationen zu den einzelnen Datensammlungen zusammenzutragen. Doch das allein ist nicht ausreichend, da auch diese Metainformationen bereits einen Umfang besitzen, der sie schwer überschaubar macht. Die Auswahl der für eine Anfrage adäquaten Datensammlungen sollte daher automatisch durch das verteilte Information Retrieval System erfolgen. Ein Ansatz in dieser Richtung ist, zunächst mit Hilfe von Information Retrieval Algorithmen relevante Datensammlungen auszuwählen, bevor in einem zweiten Schritt die darin enthaltenen Daten für das Retrieval herangezogen werden [CLC95] [FY93].

Die Zusammenführung der Ergebnisse für eine Anfrage auf verschiedenen Datensammlungen (*merging*) stellt ebenfalls hohe Ansprüche an ein Information Retrieval System. Werden die einzelnen Ergebnisse als bloße Relevanzrangfolgen angegeben, so können diese zu einer einzigen globalen Rangfolge verzahnt werden (*interleaving*). Dieser Schritt beeinträchtigt jedoch die Qualität des Gesamtergebnisses, da er außer acht läßt, daß die relevanten Datenobjekte innerhalb der einzelnen Datensammlungen unterschiedlich hohe Anteile besetzen. Bei der Verzahnung kann es dadurch beispielsweise dazu kommen, daß ein Datenobjekt nach einem anderen eingestuft wird, obwohl es relevanter ist als dieses, nur weil sehr viele Dokumente in der Datensammlung des ersten Datenobjektes ebenfalls relevant sind und es sich daher in der lokalen Relevanzrangfolge relativ weit hinten befindet. Die in einfachen Relevanzrangfolgen vorhandene Information über die Ergebnisse läßt präzisere Algorithmen zur Zusammenführung der Ergebnisse jedoch nicht zu. [CLC95]

Andere Retrievalprozesse liefern als Ergebnis nicht nur Rangfolgen, sondern auch Bewertungen (*scores*) für jedes Datenobjekt zurück, die angeben, wie gut das Objekt der Anfrage entspricht. Auch solche Bewertungen können, ebenso wie die einfachen Relevanzrangfolgen, innerhalb verschiedener Datensammlungen inkompatibel zueinander sein, wenn sich die unterschiedliche Verteilung relevanter Dokumente in den Datensammlungen bei ihrer Berechnung niederschlägt. Durch Gewichtung oder statistische Normalisierung können Be-

wertungen jedoch vereinheitlicht werden. Hierdurch wird es möglich, die Ergebnisobjekte aus verschiedenen Datensammlungen absteigend nach ihren Bewertungen in eine einheitliche Rangfolge zu bringen. [CLC95]
Die semantische Suche im Information Retrieval stellt zumeist hohe Ansprüche an die Performanz eines Systems, da dabei eine Vielzahl von semantischen Zusammenhängen innerhalb einer großen Datenmenge betrachtet werden müssen, um effektive Suchergebnisse zu erzielen. In verteilten Information Retrieval Systemen wird die Systemleistung zusätzlich noch stark durch den Zugriff auf das Netzwerk beansprucht. An der Universität von Massachusetts wurde im Zusammenhang mit der Forschung für das Information Retrieval System INQUERY ein Simulationsmodel geschaffen, mit dessen Hilfe leistungsfähige Systemarchitekturen für verteiltes Information Retrieval in verschiedenen Netzwerkkonfigurationen entwickelt werden können [CM96].

2.2.2 Datenunabhägigkeit in Information Retrieval Systemen

Umfangreiche Datensammlungen sollten für möglichst viele Anwendungen in einheitlicher Form zugänglich sein. Hierdurch läßt sich eine redundante und unter Umständen inkonsistente Speicherung der Datenobjekte durch die einzelnen Anwendungsprogramme vermeiden.
Je mehr Systemkomponenten einen Datenbestand nutzen, desto wichtiger wird es, daß die Struktur der Datenobjekte ohne großen Aufwand um beliebige Komponenten erweitert werden kann. Als Beispiel sei ein Datenbestand betrachtet, in dem Geschäftsbriefe eines Unternehmens abgelegt sind. Bekommt ein Anwendungsprogrammierer die Aufgabe, diese Briefe um ein Kommentarfeld für Sachbearbeiter zu ergänzen, so sollte dieses ohne weiteres möglich sein.
Weiterhin sollten sich Umstrukturierungen bei der Datenablage nicht auf den Zugang zu den Daten auswirken. Anwendungsprogramme sollten die Daten in einheitlicher Weise nutzen können, unabhängig davon, wo und wie diese abgelegt sind.
Der gemeinsame Zugriff auf Daten durch heterogene Systeme und die Erweiterbarkeit sowohl auf der logischen als auch auf der physikalischen Ebene, wie sie oben beschrieben wurden, lassen sich durch das Konzept der Datenunabhängigkeit erreichen. Bei der Organisation der Daten wird dann zwischen verschiedenen Abstraktionsniveaus unterschieden. Zumeist liegt dabei eine konzeptuelle Ebene zwischen der externen Anwendungsebene und der internen Speicher- und Optimierungsebene (vgl. Abschnitt 3.2). Hierdurch wird erreicht, daß alle Anwendungen voneinander unabhängig sind und sich außerdem Änderungen auf der internen Ebene auf diese beschränken [Fuh96b].
In Information Retrieval Systemen führt die Verwendung des Retrieval Indexes zu einem Verlust der Datenunabhängigkeit. Die Anfrageanwendung auf der externen Systemebene ist direkt von der Gestaltung des Indexes auf der internen Systemebene abhängig, da nur Suchterme, die im Index enthalten sind, angefragt werden können. Für dasselbe Wort existieren oft verschiedene Möglichkeiten, um es in den Index einzutragen. Der Eintrag muß jedoch für alle Datenobjekte in einheitlicher Form erfolgen, da nur so konsistente Anfrageergebnisse erzielt werden können [Son96]. Die Entscheidung für eine bestimmte Indexform schränkt das

Spektrum der Suchmöglichkeiten ein. Bei einigen Konzepten, wie beispielsweise der Suche nach Nominalphrasen oder der Stammformreduktion, entstehen hierdurch Nachteile für die Effektivität des Retrievalprozesses [Fuh96b].
Datenunabhängigkeit, wie sie im Bereich der Datenbanken bereits existiert, wird angestrebt, um einen unbeeinflußten Ablauf der Anfrageanwendungen zu sichern und um die Erweiterbarkeit von Information Retrieval Systemen zu gewährleisten.

2.2.3 Aktualisierung von Information Retrieval Objekten

Der ursprüngliche Zweck von Information Retrieval Anwendungen ist die semantische Suche auf Daten. Der verändernde Zugriff von Benutzern auf Objekte zur Systemlaufzeit ist dagegen nicht vorgesehen. In einem erweiterten Anwendungskontext reicht die bloße Suche auf Objekten jedoch nicht aus. Ziel ist, nicht nur semantisch und logisch in großen und unter Umständen auf verschiedenen Rechnern verteilten Datensammlungen nach Objekten suchen zu können, sondern auch Änderungen an diesen Objekten vorzunehmen und sie wieder abzulegen.
Ein verändernder Zugriff auf Datenobjekte stellt erweiterte Anforderungen an ein Informationssystem. Bislang erfolgt die Aufbereitung der Daten für das Information Retrieval außerhalb des Programmbetriebs. Dabei wird auch bei Erweiterungen der Datensammlung der Index zumeist nicht nur verändert, sondern komplett neu erstellt, da sein Umfang und seine flache Dateistruktur Veränderungen komplizieren [BCC94b]. Wird beliebigen Benutzern ein Schreibrecht auf den Daten eingeräumt, so müssen Verfahren gefunden werden, die dem Auswertungsalgorithmus die Semantik der Daten schon zur Laufzeit zugänglich machen. Wird der Index für die Auswertung verwendet, so muß er Änderungen in der Datensammlung direkt übernehmen können. Die vollständige Erneuerung des Indexes zur Laufzeit würde die Performanz eines Informationssystems jedoch negativ beeinflussen. Daher gilt es, ihn in einer Form zu verwalten, die seine einfache Erweiterung ermöglicht. Hierzu können beispielsweise Objektspeichersysteme [BCC94b] oder Datenbankmanagementsysteme [Son96] herangezogen werden. Neben der Anfrageauswertung ausschließlich über den Index sind weitere Verfahren vorstellbar, wie z.B. eine teilweise Analyse der Daten zur Programmlaufzeit [Fuh96a].
Aktualisierungsoperationen erfordern nicht nur veränderte Strategien zur Anfrageauswertung, sondern können auch Inkonsistenzen in der Datensammlung hervorrufen. Der Begriff der Datenintegrität läßt sich zwar für Information Retrieval Systeme nicht so eindeutig motivieren wie für Datenbanksysteme, da hier oftmals Daten von widersprüchlichem Inhalt innerhalb einer Datensammlung nebeneinander existieren [Fuh96b], bei eingehender Betrachtung unterliegen jedoch auch diese Daten gewissen Regeln. So ist es beispielsweise denkbar, daß in einer Artikelsammlung ein Objekt nur dann abgespeichert werden kann, wenn dort noch kein Objekt mit demselben Autor und demselben Titel existiert. Eine Regel dieser Art entspräche der Schlüsselintegrität, wie sie von Datenbanksystemen her bekannt ist. Ein Informationssystem, das Aktualisierungsoperationen zuläßt, sollte eine Schnittstelle zur Formulierung derartiger Integritätsbedingungen besitzen sowie die Einhaltung der Bedingungen selbständig überwachen, um zu gewährleisten, daß die Datensammlung weder

durch Einfügen noch durch Änderung von Objekten in einen inkonsistenten Zustand gerät. Da in bisherigen Information Retrieval Systemen in der Regel keine Änderungen während des Programmbetriebs durchgeführt werden können, ist eine solche Funktionalität dort nicht vorhanden.
Eine weitere Gefährdung für die Konsistenz der Daten besteht in der Möglichkeit, daß ein Systemausfall stattfindet, nachdem eine Änderungsoperation auf einem Objekt begonnen hat, aber bevor sie beendet ist. In diesem Fall sollte die Datenbasis auf ihren letzten konsistenten Zustand zurückgesetzt werden können. Derartige Methoden zur Fehlererholung (*Recovery*) sind jedoch in Information Retrieval Systemen nicht verfügbar. Es liegt somit in der Verantwortung des jeweiligen Benutzers oder Administrators entsprechende Sicherungskopien vorzunehmen, die die Wiederherstellung der Datenbasis in ihrem letzten konsistenten Zustand ermöglichen [Loo95].

2.2.4 Information Retrieval im nebenläufigen Mehrbenutzerbetrieb

Die Nutzung von Information Retrieval Algorithmen innerhalb von Informationssystemen, die nicht nur die Suche auf Daten, sondern auch deren Aktualisierung unterstützen (vgl. Abschnitt 2.2.3), geht einher mit der Forderung nach einem nebenläufigen Zugriff mehrerer Benutzer auf ein solches System. Die gleichzeitige Nutzung von Information Retrieval Systemen durch mehrere Anwender bzw. Anwendungsprogramme, die jeweils die angefragten Daten auch verändern können, wird bisher jedoch nicht unterstützt. Bei der Realisierung solcher Mehrbenutzersysteme muß sowohl ein Transaktionskonzept, wie es im Bereich der Datenbanken verbreitet ist, als auch die Zugriffskontrolle über die Datenobjekte gewährleistet werden.
Da Benutzer von Information Retrieval Systemen bislang rein lesenden Zugriff auf die Datenobjekte haben, ist ein Transaktionskonzept in diesen Systemen nicht verwirklicht worden. Will man aber Veränderungen der Daten zur Programmlaufzeit zulassen, so muß auch hier der nebenläufige Zugriff geregelt werden, ohne daß Anwender sich gegenseitig stören oder die Performanz des Information Retrieval Systems zu stark beeinträchtigt wird.
Benutzer von Information Retrieval Systemen haben einen einheitlichen Zugriff auf alle Datenobjekte des Systems. Für die zukünftige Verwendung von Information Retrieval Algorithmen, wie z.B. in Büroinformationssystemen, erscheint es sinnvoll, bestimmten Benutzern nur für bestimmte Teilmengen des Datenbestandes Lese- oder Schreibrechte einzuräumen. Hierbei muß in Erwägung gezogen werden, daß für einen Benutzer nicht sichtbare Datenobjekte auch nicht in die Berechnung der jeweiligen Relevanzrangfolgen (vgl. Abschnitt 2.1.1) für seine Anfragen einfließen sollten. [KS96]

2.3 Typisierung im Information Retrieval

Die Typisierung von Datenobjekten wird im Information Retrieval nicht betrachtet. Stattdessen wird in der Regel für alle Objekte angenommen, daß sie vom elementaren Datentyp "Text" sind [Fuh96b].

Durch diese einschränkende Annahme gehen jedoch schon bei der Speicherung der Daten Informationen verloren, die für das Information Retrieval verwendet werden könnten. Kaum ein Retrieval Objekt ist ohne Struktur. Texte bestehen aus Kapiteln, Abschnitten, Zitaten, Literaturlisten u.s.w. Sie werden durch Angaben zum Verfasser, zum Erscheinungsdatum o.ä. ergänzt. In gleicher Weise können multimediale Objekte in Komponenten unterteilt werden.
Die dem Information Retrieval eigene Vagheit macht es notwendig, auf einfache Art eine Vielzahl an Informationen über Datenobjekte zu sammeln. Oftmals wird daher versucht, die zwar vorhandene, aber nicht durch explizite Datentypen repräsentierte strukturelle Information mit Hilfe sprachlicher Konzepte für die Anfrageauswertung nutzbar zu machen. Algorithmen zur Begriffserkennung [BCC94a] oder zur Gewinnung bestimmter Information aus Volltexten (*Information Extraction*) sind Beispiele für diese Herangehensweise. Die Typisierung der Daten könnte diese Aufgaben wesentlich vereinfachen, da sie einen direkten Zugang zu den verschiedenen Eigenschaften eines Datenobjektes bieten kann.
Auch für die Darstellung der Anfrageergebnisse kann die strukturelle Information der Typisierung genutzt werden, indem die Elemente der Ergebnismenge verfeinert werden. Dem Benutzer werden dann nicht mehr ganze Datenobjekte als Ergebnisse präsentiert, sondern nur noch genau die Komponenten, die seinen Informationsbedraf decken. Er wird somit von der Aufgabe entlastet, selber den Teil eines Objektes ausfindig machen zu müssen, der für ihn relevant ist.

3. Datenbanksysteme

Auch in Datenbanksystemen[1] werden, ebenso wie in Information Retrieval Systemen, Anfragen ausgewertet. Jedoch liegt der Schwerpunkt hier nicht auf der Repräsentation einer vagen Semantik der verwalteten Daten, sondern darauf, für viele Anwender einen effizienten Zugang zu großen Datenmengen zu schaffen.

Die ersten Datenbanksysteme enstanden während der 70er Jahre aus dem Wunsch heraus, Rechensysteme nicht nur für die Ausführung von Algorithmen zu nutzen, sondern breit angelegte Information in Form von elektronischen Daten zu verwalten [LS87].

Die Daten werden dabei auch über das Ende von Anwendungsprogrammen hinaus dauerhaft gespeichert. Sie sind persistent [SST97]. Anwendungsprogramme und Benutzer können über entsprechende Schnittstellen direkt auf die gespeicherten Daten zugreifen. Dieser Zugriff kann nebenläufig erfolgen. Um hierbei zu vermeiden, daß sich die verschiedenen Anwender und Programme gegenseitig stören, und um zu gewährleisten, daß sich die persistenten Daten stets in einem konsistenten Zustand befinden, wird ein Datenbankmanagementsystem eingesetzt, das die Verwaltung des Datenbestandes übernimmt. Datenbankmanagementsysteme sind in der Lage, große Mengen von Daten, wie z.B. Kunden- und Kontenangaben einer Bank, effizienter zu verarbeiten als die meisten anderen Softwaresysteme [HS95].

In dem so vorgegebenen Verwendungskontext entstehen Grundanforderungen an die Funktionalität eines Datenbankmanagementsystems (vgl. Abschnitt 3.1). Die Erfüllung dieser Anforderungen wird durch entsprechende Architekturmodelle unterstützt (vgl. Abschnitt 3.2). Diverse Sprachen auf allen Ebenen einer Datenbankarchitektur vereinfachen die Datenverwaltung und die Anfragen nach benötigter Information (vgl. Abschnitt 3.4).

Daneben gewinnt das Paradigma der Objektorientierung auch im Datenbankbereich an Einfluß. Durch die Kombination objektorientierter Konzepte mit bestehenden Datenbankkonzepten entwickeln sich neue Fragestellungen (vgl. Abschnitt 3.5).

[1] Der Begriff "Datenbanksystem" bezeichnet die Koppelung eines Datenbankmanagementsystems mit der von ihm verwalteten Datenbank. Im Verlauf der Arbeit wird der Begriff hin und wieder als Synonym für den Begriff "Datenbankmanagementsystem" verwendet, wenn dadurch im jeweiligen Kontext keine Mißverständnisse entstehen können.

3.1 Datenbankfunktionalität

Datenbankmanagementsysteme verwalten persistente Massendaten, die sie verschiedenen Benutzern und Anwendungsprogrammen nebenläufig zugänglich machen, ohne dabei die Konsistenz des Datenbestandes zu gefährden. Hierfür ist eine konsequente Zugriffskontrolle sowie eine hohe Fehlertoleranz erforderlich.
Der Zugriff auf die Datenbank wird durch die Bereitstellung verschiedener, den Bedürfnissen der jeweiligen Anwender bzw. Anwendungen angepaßter Schnittstellen auf der externen Ebene des Datenbankmanagementsystems vereinfacht.
Die verwalteten Daten unterliegen semantischen Regeln, die durch Zugriffe auf die Datenbank nicht verletzt werden dürfen. Die Einhaltung dieser Integritätsbedingungen wird durch das Datenbankmanagementsystem überwacht.
Für die Erfüllung dieser an sie gestellten Anforderungen halten Datenbankmanagementsysteme eine Vielzahl an Verfahren bereit.

3.1.1 Persistenz

In Datenbanksystemen können persistente und transiente Datenobjekte unterschieden werden. Persistente Datenobjekte sind im stabilen Hintergrundspeicher resident und können von verschiedenen Anwendungsprogrammen genutzt werden. Sie sind über das Ende dieser Programme hinaus vorhanden und sollen System- und auch Mediafehler (vgl. Abschnitt 3.1.2.4) überleben. Transiente Datenobjekte hingegen sind hauptspeicherresident und stehen nur einer Anwendung lokal zur Verfügung. Spätestens nach Beendigung des Anwendungsprogramms gehen transiente Datenobjekte verloren. [SST97]
Die Speicherung der Datenobjekte in Dateien durch die einzelnen Anwendungsprogramme reicht nicht aus, um die Persistenz eines Systems zu gewährleisten. Im Bereich der Datenbanken wird die Persistenz daher durch das Datenbankmanagementsystem gesteuert. Es regelt die Erzeugung, die Aktualisierung und die Löschung von persistenten Datenobjekten sowie den Zugriff darauf [SST97]. Des weiteren verwaltet es redundante Datenkopien, um die Persistenz des Datenbestandes auch bei Systemfehlern gewährleisten zu können (vgl. hierzu auch Abschnitt 3.1.2.4).
Das Konzept der Persistenz führt in verschiedenen Datenbankmodellen zu unterschiedlichen Anforderungen. In relationalen Datenbanksystemen liegt eine klare Trennung von persistenten Datenbank- und transienten Programmobjekten vor. Dabei liegen die Datenbankobjekte in einem Speicherbereich, der für alle Anwendungen zugänglich ist, während die Programmobjekte im lokalen Speicherbereich abgelegt sind. Sollen persistente Objekte von einem Programm zugegriffen oder Programmvariablen persistent abgespeichert werden, so müssen diese von einem Speicherbereich in den anderen gebracht werden.

3.1.2 Mehrbenutzerbetrieb in Datenbanksystemen

Der Zugriff vieler Benutzer auf einen Gesamtdatenbestand stellt eine Reihe von Anforderungen an ein Datenbankmanagementsystem. Zum einen ist es notwendig, den Zugriff sowohl

auf persönliche Informationen als auch auf sensitive Unternehmensdaten zu kontrollieren [LS87].
Zum anderen sollte ein Datenbankmanagementsystem den effizienten Umgang mit den verwalteten Daten unterstützen. Dieses kann nur durch eine parallele Nutzung des Datenbestandes sowie durch eine hohe Fehlertoleranz des Systems erreicht werden. Das in Datenbanksystemen weit verbreitete Transaktionskonzept ist darauf ausgerichtet, auch bei nebenläufigem Zugriff die Konsistenz des Datenbestandes zu gewährleisten. [VGH93]

3.1.2.1 Zugriffskontrolle

Die Datenintegration führt dazu, daß eine Vielzahl von Bereichen einer Organisation eine einzige Datenbank nutzt. Dem unternehmerischen Interesse und den gesetzlichen Vorschriften zum Datenschutz folgend, müssen hierbei unzulässige Zugriffe oder Manipulationen auf diesem Datenbestand unterbunden werden. Die Zugriffskontrollkomponente eines Datenbankmanagementsystems erfüllt diese Aufgabe.
Bei der Zugriffskontrolle ergeben sich eine Reihe von Anforderungen an ein Datenbankmanagementsystem. Der Zugriffsschutz sollte in verschiedenen Granularitäten unterstützt werden. Man möchte sowohl die Zugriffsrechte für einzelne Attribute als auch für große Teile des Datenbestandes regeln. Der Zugriff auf die Daten soll nur über die definierten Schnittstellen erfolgen. Möglichkeiten, die Zugriffskontrolle unter Vermeidung dieser Schnittstellen zu umgehen oder Daten, die einer Zugriffskontrolle unterliegen, auf algorithmischen Umwegen zu ermitteln, darf es nicht geben. Neben diesen qualitativen Anforderungen an eine Zugriffskontrolle muß jedoch auch berücksichtigt werden, daß die Systemleistung dabei nur in erträglichem Maße beansprucht werden darf. [LS87]
Der Prozeß der Zugriffskontrolle läßt sich in mehrere Schritte einteilen [Ger93]. Bei der Autorisierung werden Zugriffsrechte vergeben. Bevor ein Benutzer auf den Datenbestand zugreifen kann, erfolgt seine Identifikation, bei der er sich dem System durch Angabe seiner Benutzerkennung bekannt macht, sowie die Authentifikation, bei der das System durch geeignete Verfahren prüft, ob der tatsächliche Benutzer mit dem durch die Kennung bezeichneten identisch ist [LS87]. Ist sichergestellt, daß ein berechtigter Benutzer auf das System zugreifen möchte, so wird während der Autorisation[2] überprüft, ob der Benutzer die Rechte besitzt, die für die geplante Arbeit mit der Datenbank notwendig sind.
Stellt die Zugriffskontrollkomponente Versuche unberechtigter Zugriffe fest, so muß sie entsprechende Maßnahmen einleiten, um den Datenbestand zu schützen [Ger93].

3.1.2.2 Transaktionen

Transaktionen sind Anwendungen auf einer Datenbank, die vom Datenbankmanagementsystem als logische Einheiten aufgefaßt werden. Das Transaktionskonzept fordert für die Verarbeitung einer Transaktion, daß sie so geschehen muß, als stünde das Datenbanksystem

[2]Die Begriffe "Autorisierung" und "Autorisation" werden hier streng unterschieden. Bei der Autorisierung handelt es sich um die Rechtevergabe, bei der Autorisation um die Rechteüberprüfung.

ausschließlich dieser Transaktion zur Verfügung. Werden weitere Transaktionen zur gleichen Zeit ausgeführt, so bleiben sie unbemerkt.
Auf der Grundlage dieser Überlegungen entstand das ACID-Prinzip[3], das die folgenden vier Eigenschaften von Transaktionen festschreibt [VGH93]:

- **Atomarität** (*atomicity*)

  Aus Sicht des Benutzers wird eine Transaktion entweder vollständig oder überhaupt nicht ausgeführt. Falls eine Transaktion also während der Ausführung abgebrochen wird, darf sich dieses nicht im Zustand der Datenbank niederschlagen.

- **Konsistenzerhaltung** (*consistency*)

  Transaktionen halten die für die Datenbank definierten Integritätsbedingungen ein. Wird eine Transaktion in einem konsistenten Datenbankzustand gestartet, so ist die Konsistenz auch nach der Ausführung gegeben. Transaktionen, deren Ausführung die Integrität verletzt, werden abgebrochen.

- **Isolation** (*isolation*)

  Eine Transaktion wird isoliert von allen anderen Transaktionen ausgeführt. Änderungen, die von anderen Transaktionen nebenläufig vorgenommen werden, sieht sie während ihrer Ausführung nicht.

- **Persistenz** (*durability*)

  Nachdem dem Benutzer die erfolgreiche Beendigung einer Transaktion mitgeteilt wurde, kann dieser davon ausgehen, daß alle Effekte dieser Transaktion dauerhaft im Datenbestand gespeichert sind.

Bei der Transaktionsverwaltung wird eine Komponente zur Nebenläufigkeitskontrolle (*concurrency control*) verwendet, die Konsistenzerhaltung und Isolation sicherstellt, sowie eine Komponente zur Fehlerbehandlung, die Atomarität und Persistenz unterstützt.

3.1.2.3 Kontrolle der Nebenläufigkeit

Die Nebenläufigkeit von Transaktionen bedingt deren zeitlich verzahnte Ausführung. Wird die Verzahnung unkontrolliert vorgenommen, so kann dieses zu Inkonsistenzen in der Datenbank führen, obwohl die Transaktionen einzeln betrachtet korrekt arbeiten [LS87].
Für die theoretische Betrachtung von Synchronisationsproblemen in Datenbanksystemen wird häufig von der Semantik der einzelnen Transaktionsschritte abstrahiert. Analysiert werden lediglich die Lese- (*reads*) und Schreibzugriffe (*writes*) auf den Datenbestand. Das entsprechende Modell wird als Read-Write-Modell bezeichnet. [VGH93]
Liegt ein Ablaufplan (*schedule*) aus verzahnten Lese- und Schreiboperationen verschiedener Transaktionen vor, so kann das Kriterium der Serialisierbarkeit herangezogen werden, um zu

[3] ACID ist ein Acronym, das sich aus den englischen Begriffen für die Eigenschaften, die eine Transaktion erfüllen soll, ableitet.

überprüfen, ob der Ablaufplan ein konsistentes Ergebnis liefert. Ein Ablaufplan heißt genau dann serialisierbar, wenn sein Ergebnis dem Ergebnis einer beliebigen seriellen Ausführung der beteiligten Transaktionen entspricht. Da seriell ausgeführte Transaktionen die Konsistenz des Datenbestandes wahren, sind auch serialisierbare Ablaufpläne konsistenzerhaltend. [SST97]
Bei der nebenläufigen Ausführung mehrerer Transaktionen kann die Serialisierbarkeit der Operationen durch verschiedene Methoden sichergestellt werden. Das am weitesten verbreitete Verfahren ist das Setzen von Sperren auf Elemente des Datenbestandes [LS87]. Man unterscheidet dabei zwischen Lese- und Schreibsperren. Bevor eine Transaktion eine Lese- oder Schreiboperation ausführt, muß sie die entsprechende Sperre auf das betreffende Datenelement erwerben. Nach Ausführung der Operation muß die Sperre freigegeben werden. Verschiedene Protokolle, wie beispielsweise das Zwei-Phasen-Sperr-Protokoll, die das Setzen und die Freigabe von Sperren regeln, gewährleisten die Serialisierbarkeit von Ablaufplänen (Fundamentalsatz des Sperrens) [LS87], [SST97].
Besonders in verteilten Datenbanksystemen werden neben Sperrprotokollen häufig Zeitstempelverfahren zur Synchronisation eingesetzt [HS95].

3.1.2.4 Fehlerbehandlung

Die Aufgabe der Komponente zur Fehlerbehandlung in einem Datenbankmanagementsystem ist es, typische Fehler zu erkennen und, nach Möglichkeit ohne menschliche Intervention, zu beheben. Hierbei müssen gegebenenfalls Transaktionen abgebrochen und der konsistente Zustand des Datenbestandes wiederhergestellt werden. Man spricht von einer Fehlererholung (*recovery*) des Datenbanksystems.
Bei der Fehlerbehandlung werden drei verschiedene Fehlerklassen unterschieden [SST97]. Fehler dieser Klassen betreffen jeweils unterschiedliche Bereiche des Datenbanksystems und ziehen für ihre Fehlerklasse spezifische Maßnahmen nach sich [LS87].

1. **Transaktionsfehler** führen zum Abbruch der betroffenen Transaktionen. Auslöser sind beispielsweise Programmfehler, ein expliziter Abbruch durch den Benutzer oder ein Abbruch durch das Datenbankmanagementsystem.

2. **Systemfehler** haben den Verlust der im Hauptspeicher befindlichen Daten zur Folge. Ursache hierfür können z.B. Hardwarefehler oder Fehler im Datenbankmanagementsystem sein.

3. **Mediafehler** führen zum völligen oder teilweisen Verlust der Daten im stabilen Speicher. Die Ursachen für Mediafehler sind vielfältig und reichen vom Stromausfall bis zur Naturkatastrophe.

Die Komponente zur Fehlerbehandlung muß nach dem Auftreten dieser Fehler den möglichst letzten konsistenten Zustand des Datenbestandes wiederherstellen. Hierzu verwendet sie Protokolldaten, die das Datenbankmanagementsystem über durchgeführte Transaktionen anlegt, sowie im Falle von Mediafehlern archivierte Kopien des Datenbestandes. Auf der

Grundlage dieser Informationen werden Transaktionen rückgängig gemacht oder erneut ausgeführt.

3.1.3 Datenintegrität

Damit ein Datenbanksystem überhaupt nutzbar ist, muß zu jeder Zeit sichergestellt werden, daß die gespeicherten Daten allen geforderten Integritätsbedingungen genügen und sich der Datenbestand somit in einem konsistenten Zustand befindet. Neben den Integritätbedingungen die sich automatisch aus Vorstellungen hinsichtlich eines einwandfreien Funktionierens eines Datenbanksystems ergeben und die beispielsweise durch die Nebenläufigkeitskontrolle oder ähnliche Systemkomponenten überprüft werden, können auch semantische Integritätbedingungen bezüglich des durch die Daten repräsentierten Anwendungsbereichs formuliert werden [KE97].
Es gibt statische und dynamische Integritätsbedingungen. Statische Integritätsbedingungen beziehen sich auf den jeweiligen Zustand des Datenbestandes. Dynamische Integritätsbedingungen hingegen legen fest, wie dieser Zustand verändert werden darf [Vos94].
Bei der expliziten Definition von Integritätsbedingungen muß das Datenbankmanagementsystem nicht nur deren syntaktische und semantische Korrektheit überprüfen, sondern auch feststellen, ob die formulierte Bedingung nicht schon zur Definitionszeit verletzt ist, ob sie nicht aus bereits vorhandenen Bedingungen abgeleitet werden kann und ob sie in keinem Widerspruch zu anderen Bedingungen steht. Des weiteren muß sichergestellt werden, daß die durch Konsistenzverletzungen ausgelösten Prozesse stets endlich sind.
Bei der Überwachung der Einhaltung von Integritätsbedingungen kann entweder Verletzungsvermeidung betrieben werden, so daß konsistenzverletzende Operationen vom Datenbankmanagementsystem gar nicht erst zugelassen werden, oder aber Verletzungserkennung, so daß Konsistenzverletzungen nach ihrem Auftreten durch das System wieder behoben werden.
Konkret können zur Integritätsüberwachung Monitore eingesetzt werden, die sämtliche Transaktionen überwachen und gegebenenfalls abbrechen. Auch Trigger, in denen genau definiert ist, wann eine Operation die Konsistenz des Datenbestandes verletzt und welche Maßnahmen in diesem Fall durchzuführen sind, werden häufig verwendet. [Vos94]

3.1.4 Externe Schnittstellen

Der Zugriff auf eine Datenbank erfolgt durch verschiedene Personengruppen. Zum einen implementieren Programmierer Anwendungen, die bestimmte Algorithmen auf dem Datenbestand durchführen. Zum anderen benutzen Endanwender das System, um auf direktem Weg an Informationen aus der Datenbank zu gelangen. Sowohl der technische Kenntnisstand als auch die Position innerhalb der Organisation, die das Datenbanksystem einsetzt, können sich von Anwender zu Anwender stark unterscheiden.
Datenbankmanagementsysteme versuchen diesen Unterschieden gerecht zu werden, indem sie mit Anfragesprachen, wie z.B. SQL oder OQL (vgl. Abschnitt 7.2), oder auch mit graphischen Oberflächen, die eine intuitive Navigation durch den Datenbestand erlauben, Be-

nutzungschnittstellen anbieten, die auf das Wissen des jeweiligen Anwenders zugeschnitten sind.
Des weiteren ist es möglich für unterschiedliche Anwender jeweils eigene Sichten (*views*) auf den Datenbestand anzulegen, so daß jeder Anwender bei der Arbeit mit dem Datenbanksystem jeweils nur mit dem für ihn relevanten Ausschnitt der Daten konfrontiert wird. Hierdurch wird der Anwender zum einen entlastet. Zum anderen kann ihm so der Zugang zu nicht für ihn bestimmten Informationen entzogen werden.

3.2 Die 3-Ebenen-Architektur

In einem Datenbanksystem fallen vielfältige Aufgaben an (vgl. Abschnitt 3.1). Verschiedene Architekturmodelle für Datenbanken versuchen diese Aufgaben in eine Struktur zu bringen, um sie effizient bewältigen zu können.
Eines dieser Modelle ist das ANSI/X3/SPARC-Architekturmodell [LS87]. Es unterteilt ein Datenbanksystem in drei Ebenen (vgl. Abb. 3.1). Auf der *internen Ebene* vollzieht sich die Speicherung der persistenten Daten. Verfahren für einen effizienten Zugriff, wie beispielsweise das Anlegen von Datenbankindizes, werden hier realisiert. Auf der *konzeptuellen Ebene* werden der logische Aufbau der Daten und ihre Zusammenhänge modelliert. Hierdurch entsteht ein Datenbankschema, das eine logische Gesamtsicht auf die Daten bietet. Da zumeist weder für Endanwender noch für Anwendungsprogramme der gesamte Datenbestand von Interesse ist, wird die Gesamtsicht der konzeptuellen Ebene innerhalb der *externen Ebene* auf verschiedene für Anwender oder Programme relevante Einzelsichten eingeschränkt.

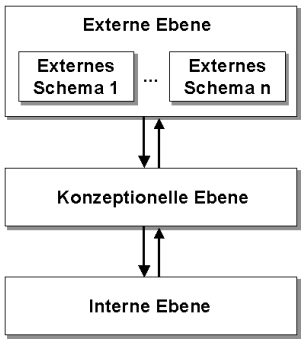


Abbildung 3.1: Das ANSI/X3/SPARC-Architekturmodell

Der Aufbau eines Datenbanksystems in diesen 3 Ebenen realisiert das Prinzip der Datenunabhängigkeit in zweierlei Hinsicht. Zum einen können Veränderungen auf der internen Ebene, die beispielsweise zur Erhöhung der Effizienz vorgenommen werden, auf diese Ebene beschränkt bleiben. Die darüberliegenden Ebenen profitieren von den Veränderungen, müssen jedoch keinerlei eigene Anpassungen vornehmen (*physische Datenunabhängigkeit*). Zum anderen sind die einzelnen Sichten der externen Ebene nicht aneinander gebunden, so

daß die Neudefinition oder Änderung einer Sicht die anderen Sichten nicht berührt (*logische Datenunabhängigkeit*).

3.3 Datenmodellierung

Auf der konzeptuellen Ebene der 3-Ebenen-Architektur wird eine logische Gesamtsicht auf den Datenbestand entworfen. Zu diesem Zweck sind verschiedene Modelle entwickelt worden, wie beispielsweise das relationale Datenbankmodell oder das ODMG-Objektmodell (vgl. Abschnitt 5.2.2) für objektorientierte Datenbanken. Die Komponenten dieser Modelle ermöglichen eine abstrakte Darstellung der Daten, ihrer Eigenschaften und ihrer Zusammenhänge untereinander.
Beim Entwurf der logischen Gesamtsicht wird der Ausschnitt der realen Anwendungsdomäne, der für die Verarbeitung durch das Datenbanksystem als relevant betrachtet wird, durch Datenstrukturbeschreibungen repräsentiert. Alle Annahmen, die hierbei getroffen werden, werden später vom Datenbanksystem als gesichert vorausgesetzt. Falls in der Realität abseits von den modellierten Datenstrukturen noch weitere Aspekte vorhanden sind, so werden diese ausgeblendet. Werden also beispielsweise Personen auf einen Datentyp mit den Attributen "Name", "Alter" und "Adresse" abgebildet, so besteht zum einen kein Zweifel daran, daß alle Instanzen des Typs für diese Attribute einen bestimmten Wert aufweisen, zum anderen sind darüber hinausgehende Eigenschaften einer realen Person, wie beispielsweise ihre Interessensgebiete, für das Datenbanksystem irrelevant. Die Datenmodellierung erzeugt somit eine eigene eingeschränkte Weltsicht, in der eindeutige Bedingungen gelten.
Datenbanksysteme repräsentieren die Realität somit anders als Information Retrieval Systeme. Während im Information Retrieval versucht wird, umfangreiche Informationen über das Anwendungsgebiet zu sammeln, um dadurch möglichst tiefe Erkenntnisse über die Semantik der vorliegenden Daten zu gewinnen, wird in Datenbanksystemen ein fest umrissener Teil des Anwendungsgebiets ausgewählt, für den dann eine effiziente Datenverarbeitung betrieben werden kann.
Andererseits geht mit der bei der Datenbankmodellierung durchgeführten Abstraktion ein Verständnis für die Zusammenhänge innerhalb des Datenbestandes einher, das im Information Retrieval, wo Daten untypisiert verwendet werden, nicht vorhanden ist (vgl. Abschnitt 2.3).

3.4 Datenbanksprachen

Für den Umgang mit dem Datenbankschema der konzeptuellen Ebene eines Datenbanksystems werden zwei verschiedene Sprachformen benötigt. So wird schon bei der Erstellung des Schemas zur Beschreibung der Datenstrukturen eine Datendefinitionssprache (*Data Definition Language, DDL*) verwendet (vgl. Kapitel 6).
Auf das fertige Schema kann dann mit einer Datenmanipulationssprache (*Data Manipulation Language, DML*) zugegriffen werden. Die DML läßt sich dabei noch weiter aufteilen in eine Anfragesprache (vgl. Kapitel 7) und in den Teil der Sprache, der Änderungen an den

persistenten Daten und damit die eigentliche Datenmanipulation vornimmt. Die DML kann von Endanwendern im interaktiven Dialog eingesetzt werden. Es ist aber auch möglich, ihre Anweisungen in ein Anwendungsprogramm einzubetten. DML sind häufig deklarativ. In ihren Konstrukten wird zwar angegeben, mit welchen Daten die Verarbeitung vorgenommen werden soll, nicht jedoch, in welchen Einzelschritten dieses zu geschehen hat. [KE97]

3.5 Objektorientierte Datenbanksysteme

Zwischen relationalen, hierarchischen oder Netzwerkdatenbanksystemen und den darauf aufsetzenden höheren Programmiersprachen besteht häufig ein Bruch. Der Blick beider Seiten auf die Daten und die darauf auszuführenden Algorithmen ist nicht derselbe. So können in den Datenbanksystemen beispielsweise nur Daten mit einer relativ einfachen Struktur abgelegt werden. Die Umsetzung der in Datenbanksystemen üblichen deklarativen Anfragen in Programmiersprachenkonstrukte ist schwierig. Insgesamt kann aus Programmiersprachen heraus demnach nur unter Aufwand auf durch Datenbanksysteme verwaltete persistente Daten zugegriffen werden.
Objektorientierte Datenbanksysteme bilden hier jedoch eine Ausnahme, da objektorientierte Datenbanksysteme und objektorientierte Programmiersprachen auf demselben Modellierungsprinzip für Daten und Algorithmen aufsetzen. In objektorientierten Datenbanksystemen können beliebig komplizierte Datenobjekte persistent abgelegt werden. Somit ist eine persistente Speicherung ohne vorherige Umwandlung der Objekte aus objektorientierten Programmen heraus möglich. Zudem sind sowohl dem Datenbanksystem als auch der Programmiersprachenumgebung die auf den Objekten definierten Operationen bekannt, so daß diese in Datenbankanfragen (vgl. Abschnitt 7.2) und in Anwendungsprogrammen verwendet werden können.

4. Allgemeine Anforderungen an ein integriertes Datenbank- und Information Retrieval System

Aus der Integration von Datenbank- und Information Retrieval Systemen sollte ein System hervorgehen, daß die Stärken des Information Retrieval und des Datenbankbereichs in sich vereinigt und hierdurch die auf beiden Seiten vorhandenen Schwächen ausgleicht (vgl. Kapitel 2 und 3) sowie neue Anwendungsfelder öffnet.
Da hier zwei Systemformen kombiniert werden, die bereits für sich genommen aufwendige Funktionen realisieren, sollte das entstehende integrierte System eine strukturierte Architektur aufweisen. Die vorliegende Arbeit greift hier auf die 3-Ebenen-Architektur von Datenbanksystemen (vgl. Abschnitt 3.2) zurück.

4.1 Eigenschaften eines integrierten Systems

Ein integriertes Datenbank- und Information Retrieval System sollte die Stärken beider Seiten zusammenführen. So sollte die Speicherung und die Verarbeitung von großen Datenmengen ebenso effizient durchgeführt werden wie in Datenbanksystemen. Insbesondere sollten hier Daten jeglicher Struktur abgelegt, zugegriffen und aktualisiert werden können. Information Retrieval Daten unterscheiden sich dabei in Umfang und Format von typischen Datenbankdaten. Beliebig lange Texte sollten ebenso verarbeitet werden können wie Multimediadaten. Hinzu kommt die Verwaltung umfangreicher Metadaten, wie beispielsweise Termindizes oder Thesauri, auf die Information Retrieval Algorithmen bei der Auswertung vager Anfragen zugreifen.
Alle vorhandenen Daten, unabhängig davon ob sie eine eindeutige oder vage Semantik besitzen, sollten persistent im integrierten System abgelegt und von diesem verwaltet werden können. Dabei sollte insbesondere auch die Aktualisierung von Daten mit vager Semantik möglich sein (vgl. Abschnitt 2.2.3). Diese Forderung gewinnt an Bedeutung, wenn gleichzeitig der nebenläufige Zugriff mehrerer Benutzer auf alle Daten angestrebt wird. Zum einen ist hier das gängige Transaktionskonzept von Datenbanksystemen (vgl. Abschnitt 3.1.2.2) zu überdenken, da Anwender mit vagen Daten, wie beispielsweise langen Texten, anders arbeiten als mit strukurierten Daten in einer Datenbank. Zum anderen können für Daten mit

vager Semantik unter Umständen Integritätsbedingungen (vgl. Abschnitt 3.1.3) zu Gunsten einer effizienteren Verarbeitung gelockert werden [KS96]. Die existierenden Lösungen für strukturierte Daten sollten hierdurch jedoch nicht in Frage gestellt werden.

Die bestehenden Konzepte der Datenbankmodellierung sollten im integrierten System so erweitert werden, daß sich auch eine vage Semantik für Daten von beliebiger Struktur repräsentieren läßt. Hierdurch wird zum einen die Strukturierung von Information Retrieval Daten ermöglicht, die bisher zumeist nur in flachen Dateien vorliegen (vgl. Abschnitt 2.3). Zum anderen kann auch für Datenbankdaten, die bisher auf eine eindeutige Semantik beschränkt sind (vgl. Abschnitt 3.3), eine Vagheit ausgedrückt werden, wie z.B. bei vagen Zeit- oder Ortsangaben oder bei phonetischer Gleichheit von Zeichenketten. Insgesamt können für das integrierte System beliebig strukturierte Daten modelliert werden, die jeweils sowohl eindeutige als auch vage semantische Anteile besitzen können.

4.2 Datenunabhängigkeit im integrierten System

Das integrierte System soll alle Aufgaben erfüllen, die auch ein Datenbanksystem ausführen kann. Hinzu kommen noch Anforderungen, die sich durch die Integration der Information Retrieval Funktionalität ergeben. Vor diesem Hintergrund wird die 3-Ebenen-Architektur der Datenbanksysteme für das integrierte System beibehalten.

Die drei Ebenen erfüllen dabei sowohl diejenigen Funktionen, die ihnen bereits in Datenbanksystemen zugedacht sind, als auch solche, die sich durch die Einführung der vagen Semantik ergeben:

- Die interne Ebene übernimmt neben der Speicherung von eindeutigen und vagen Daten sowie der Zugriffsoptimierung zusätzlich die Verwaltung der Metadaten für vage Anfragen. Hierbei stellt die angestrebte Aktualiserungsmöglichkeit für Daten mit vager Semantik die größte Herausforderung dar, weil bei jeder Änderung vager Daten die dazugehörigen Metadaten ebenfalls auf den neuesten Stand gebracht werden müssen. Diese sind jedoch wesentlich umfangreicher als die in Datenbanksystemen gepflegten Metadaten.

- Auf der konzeptuellen Ebene werden Daten modelliert, deren Semantik eindeutige und vage Anteile besitzt. Eine beliebige Strukturierung der Daten ist, wie auch in objektorientierten Datenbanksystemen, darstellbar.

- Auf der externen Ebene bleiben die individuellen Sichten auf den Datenbestand erhalten. Jede individuelle Sicht kann alle vagen und eindeutigen Aspekte der Gesamtsicht entweder aufnehmen oder ausblenden. Als ein allgemein verwendbares Werkzeug zum Zugriff auf die Daten des integrierten Systems wird eine Anfragesprache zur Verfügung gestellt, die eindeutige, vage und gemischte Anfragen zuläßt.

4.3 Der gewählte Ansatz

Aufgrund der für das integrierte System geforderten Datenunabhängigkeit ist es möglich, die einzelnen Ebenen isoliert zu behandeln. Die vorliegende Arbeit beschränkt sich daher zunächst auf die Betrachtung der konzeptuellen und der externen Ebene.
Hierbei wird auf eine bereits existierende Lösung für Datenbanksysteme zurückgegriffen, die um Information Retrieval Konzepte hinsichtlich der Verwaltung von Daten mit vager Semantik erweitert wird. Die vorliegende Arbeit verwendet hier den Standard der *Object Database Management Group* [Cat97] (vgl. Abschnitt 5.2.1). Dieser Standard definiert unter anderem ein Objektmodell mit einer dazugehörigen Datendefinitionssprache, die als Basis für das auf der konzeptuellen Ebene des integrierten Systems zu entwickelnde Modell und dessen Beschreibung dienen (vgl. Kapitel 5 und 6). Die im Standard beschriebene Anfragesprache OQL wird auf der externen Ebene des integrierten Systems zu einer Anfragesprache auf vagen und eindeutigen Daten erweitert (vgl. Kapitel 7).
Neben der Möglichkeit, die im ODMG-Standard enthaltenen Komponenten, die bereits einen Großteil der auch im integrierten System benötigten Funktionalität beinhalten, zu verwenden, spielt zudem die Tatsache, daß es sich beim ODMG-Standard um einen Standard für objektorientierte Datenbanksysteme (vgl. Abschnitt 3.5) handelt, bei seiner Auswahl als Grundlage für ein integriertes System eine Rolle. Zum einen unterstützt das objektorientierte Datenmodell die Repräsentation von Daten mit beliebiger Struktur, wie sie für das integrierte System angestrebt ist (vgl. Abschnitt 4.1). Zum anderen kommt die Erweiterbarkeit objektorientierter Informationssysteme der Integration von Datenbank- und Information Retrieval Konzepten direkt zugute.

5. ProM - ein Objektmodell für integrierte Datenbank- und Information Retrieval Systeme

Datenbank- und Information Retrieval Daten müssen auf der konzeptuellen Ebene eines integrierten Systems ohne Unterschiede modelliert werden können. Hierzu wird ein Satz von allgemeinen Modellierungskomponenten benötigt, der sowohl typische Konzepte der Datenbankmodellierung verwirklicht als auch den Anforderungen der im Information Retrieval inhärenten vagen Semantik gerecht wird. Dieser Komponentensatz ist im Objektmodell ProM enthalten, das im folgenden vorgestellt wird. Bei der Entwicklung dieses Modells wird das Objektmodell des ODMG-Standards um verschiedene Information Retrieval Komponenten zur Modellierung von Wahrscheinlichkeitswerten, vagen Objekteigenschaften und Relevanzrangfolgen erweitert.

5.1 Vorgehensweise

Sowohl im Bereich der Datenbanken als auch im Information Retrieval existieren bereits verschiedene Modelle, die die jeweiligen Kernfragestellungen ausführlich behandeln (vgl. beispielsweise [RBP+91], [Cat97], [Fuh92c] oder [TC91]). Bei Datenbankmodellen ist hier eher die Beschreibung strukturierter Daten maßgebend, während bei Information Retrieval Modellen die Repräsentation semantischer Aspekte im Vordergrund steht. Da für das Modell ProM beide Perspektiven von Bedeutung sind, wird in der vorliegenden Arbeit das Wissen aus Datenbank- und Information Retrieval Modellen zusammengeführt.
Modelle für objektorientierte Datenbanken bieten die Möglichkeit, Daten verschiedener Komplexität einheitlich zu modellieren. Des weiteren verfügen sie oft über ein umfangreiches Typsystem, in das Daten mit vager Semantik einfacher integriert werden können als beispielsweise in relationale Modelle [Abi97]. Daher wird das ODMG-Objektmodell als objektorientiertes Datenbankmodell der Entwicklung des Modells ProM (vgl. Abb. 5.1) zugrundegelegt.
Das ODMG-Objektmodell enthält dabei bereits alle Modellierungskomponenten, die erforderlich sind, um Daten mit eindeutiger Semantik sowie deren Beziehungen untereinander zu repräsentieren. Daher werden diesem Modell unter Rückgriff auf die semantischen Modelle

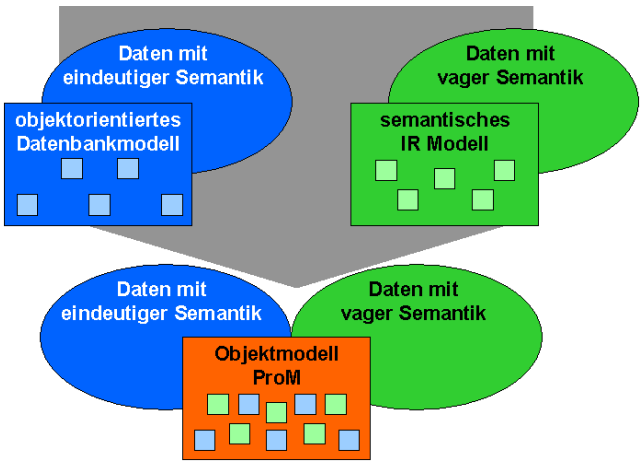


Abbildung 5.1: Die Entwicklung des Objektmodells ProM durch Erweiterung eines Datenbankmodells um Information Retrieval Komponenten

des Information Retrieval lediglich einige gleichberechtigte Komponenten hinzugefügt, die es ermöglichen, die vage Semantik von Datenobjekten und die Unschärfe in den Resultaten von Information Retrieval Anfragen zu modellieren. An dieser Stelle sei betont, daß es sich um eine Ergänzung, nicht aber um eine Veränderung der ursprünglich im ODMG-Objektmodell enthaltenen Datenbankmodellierungskomponenten handelt. Hierdurch bleibt das Modell ProM für Anfrage- und Manipulationssprachen sowie für Anwendungen offen, die auf der Basis des ODMG-Objektmodells konzipiert sind.

ProM kann demnach ebenso wie das ODMG-Objektmodell für die Modellierung herkömmlicher Datenbankanwendungen verwendet werden. Zusätzlich ermöglicht es aber auch die Darstellung vager Semantik im Sinne des Information Retrieval. Alle Komponenten von ProM sind ohne Einschränkungen für beliebige Daten anwendbar. Hierdurch eröffnet sich die Möglichkeit, Information Retrieval Techniken für Datenbankobjekte und umgekehrt Datenbanktechniken für Information Retrieval Objekte nutzbar zu machen.

5.2 Das ODMG-Objektmodell als Grundlage für das Objektmodell ProM

Der ODMG-Standard ist im Rahmen von Standardisierungsbestrebungen im Bereich der objektorientierten Datenbanktechnologie entstanden und wird von verschiedenen Produzenten objektorientierter Datenbanksysteme getragen. Einer seiner Kernbestandteile ist das ODMG-Objektmodell, das in der vorliegenden Arbeit dem Entwurf des Objektmodells ProM für integrierte Datenbank- und Information Retrieval Systeme zugrundeliegt.

5.2.1 Der ODMG-Standard

Die ODMG (*Object Database Management Group*) wurde 1991 von verschiedenen Herstellern objektorientierter Datenbanksysteme mit dem Ziel gegründet, einen einheitlichen Standard für objektorientierte Datenbanksysteme zu entwickeln. Ein solcher Standard soll die Portabilität von Anwendungen zwischen verschiedenen objektorientierten Datenbankmanagementsystemen erhöhen, indem er die Möglichkeit einer einheitlichen Definition von Datenbankschemata schafft und gemeinsame Sprachen für Datenbankzugriffe zur Verfügung stellt. Die Schwerpunkte liegen dabei zum einen auf einem allgemeinen Objektmodell und zum anderen auf der direkten Verwendung gängiger objektorientierter Programmiersprachen für die Datenbankprogrammierung.
Der ODMG-Standard liegt aktuell in der Version 2.0 vor [Cat97] und besteht aus

- einem *Objektmodell*, in dem dargestellt wird, welche Informationen auf welche Art und Weise in einem objektorientierten Datenbanksystem gespeichert und verarbeitet werden können
- zwei *Spezifikationssprachen, ODL (Object Definition Language)* zur konkreten Definition von Datenbankschemata und *OIF (Object Interchange Format)*, um den aktuellen Zustand einer Datenbank in Dateien speichern und wieder herauslesen zu können,
- einer deklarativen *Anfragesprache OQL (Object Query Language)* für lesende und verändernde Zugriffe auf Objekte einer Datenbank sowie
- *Sprachanbindungen*, die die direkte Verwendung der Programmiersprachen C++, Smalltalk und Java auf objektorientierten Datenbanken ermöglichen.

Die vorliegende Arbeit erweitert das Objektmodell, die Spezifikationssprache ODL (vgl. Kapitel 6) und die Anfragesprache OQL (vgl. Kapitel 7) in Hinblick auf die vagen Aspekte des Information Retrieval.
Die ODMG ist mittlerweile der OMG (*Object Mangement Group*) angegliedert worden, deren Vorhaben es ist, den Einsatz objektorientierter Konzepte in verteilten Systemen voranzubringen. Hieraus ergibt sich eine Zusammenarbeit der beiden Organisationen, die sich auch im ODMG-Standard niederschlägt. So basieren das Objektmodell, die Objektspezifikationssprachen und die Anbindung der Programmiersprachen teilweise auf ähnlichen, innerhalb der OMG erarbeiteten Komponenten.
Zudem besteht eine Kooperation zwischen der ODMG und dem ANSI (*American National Standards Institute*) hinsichtlich einer gegenseitigen Annäherung zwischen den Anfragesprachen OQL und SQL3. Dieses ist in Teilen von OQL und auch in einzelnen Komponenten des ODMG-Objektmodells erkennbar.

5.2.2 Das ODMG-Objektmodell

Das ODMG-Objektmodell stellt verschiedene Modellierungskomponenten zur Verfügung, die zum einen der Modellierung einzelner Datenobjekte und zum anderen dem Aufbau von

Strukturen zwischen verschiedenen Datenobjekten dienen. Die folgende Liste gibt eine grobe Übersicht über diese Komponenten und ihre Funktion:

- **Typ (*type*):** Ein Typ spezifiziert und implementiert einen Satz von Eigenschaften. Typen, die instanziierbar sind, heißen konkret. Nicht instanziierbare Typen heißen abstrakt.
- **Klasse (*class*):** Eine Klasse spezifiziert konkrete Typen, indem sie das Verhalten und den Zustand der Typinstanzen definiert.
- **Schnittstelle (*interface*):** Eine Schnittstelle spezifiziert abstrakte Typen, indem sie ein abstraktes Verhalten für diese Typen definiert.
- **Subtyprelation (*subtype relationship*):** Die Subtyprelation legt eine Vererbungshierarchie für abstraktes Verhalten zwischen abstrakten Typen auf der einen und abstrakten oder konkreten Typen auf der anderen Seite fest.
- **EXTENDS-Relation (*EXTENDS relationship*):** Die EXTENDS-Relation legt eine Vererbungshierarchie für alle Eigenschaften von konkreten Typen fest.
- **Objekt (*object*):** Ein Objekt ist eine durch einen Identifikator eindeutig bezeichnete Typinstanz. Es gibt atomare Objekte, Kollektionsobjekte und strukturierte Objekte.
- **Literal (*literal*):** Ein Literal ist eine Typinstanz, die nicht eindeutig identifizierbar, sondern nur durch ihren Wert zugreifbar ist. Es gibt atomare Literale, Kollektionsliterale, strukturierte Literale und Literale mit Leerwert (*null value*).
- **Attribut (*attribute*):** Ein Attribut enthält einen Wert, der zur Beschreibung des Zustandes einer Typinstanz beiträgt.
- **Relation (*relationship*):** Eine Relation beschreibt den Zustand einer Instanz, indem sie diese mit bestimmten Instanzen eines anderen Typs in Beziehung setzt.
- **Operation (*operation*):** Eine Operation spezifiziert einen Teil des Verhaltens, das für alle Instanzen eines Typs aufgerufen werden kann.
- **Metadaten (*metadata*):** Metadaten enthalten Zusatzinformationen, die nicht die eigentlichen Anwendungsdaten, sondern das Datenbankschema an sich beschreiben.
- **logische Datenbank (*logical database*):** Eine logische Datenbank ist ein Datenbankbereich, der nach außen hin eine Einheit darstellt. Sie kann aus mehreren physikalischen Datenbanken bestehen oder verteilt sein.
- **Transaktion (*transaction*):** Jeder Datenbankzugriff muß innerhalb einer Transaktion stattfinden, damit Datenintegrität, ein reibungsloser Mehrbenutzerbetrieb und Erholung im Fehlerfall gewährleistet werden können. Transaktionen arbeiten nach dem ACID-Prinzip (vgl. Abschnitt 3.1.2.2).

- **Sperre (*lock*):** Das Setzen von Sperren gewährleistet die Serialisierbarkeit von Datenbankzugriffen im Mehrbenutzerbetrieb und damit die Konsistenz der Datenbank.

Aus den aufgeführten Komponenten lassen sich einige Aspekte ableiten, die das Modell als Grundlage für das Objektmodell ProM geeignet erscheinen lassen. Durch das vorhandene Typsystem läßt sich präzise ausdrücken, wie die enthaltenen Komponenten zusammenhängen. Hierdurch ist es bei der Erweiterung um neue Modellierungskomponenten möglich, genau festzulegen, wo sie in das bestehende System eingefügt werden müssen und wie sie mit diesem zusammenwirken. So kann beispielsweise ein für Information Retrieval Anfragen adäquater Ergebnistyp definiert werden, für den das Typsystem sicherstellt, daß er bei entsprechenden Operationen zum Einsatz kommt.

Für vage Anfragen mit Hilfe von Information Retrieval Verfahren gibt es über die Volltextsuche hinaus noch eine Vielzahl anderer denkbarer Anwendungsfelder. Daher ist es wichtig, daß ein integriertes Datenbank- und Information Retrieval System um beliebige Datenstrukturen und Algorithmen erweitert werden kann. Für Systeme, die auf dem ODMG-Objektmodell aufsetzen, ist diese Anforderung dadurch erfüllt, daß über die Definition von Subtypen die bestehenden Vererbungshierarchien uneingeschränkt ergänzt werden können.

Sowohl für Objekte als auch für Literale definiert das ODMG-Objektmodell verschiedene Kollektionstypen, die zum einen geeignet sind, um interne Datenstrukturen des Information Retrieval, wie z.B. Retrieval Indizes, effizient zu verwalten, und die zum anderen als Muster für weitere, den Gegebenheiten des Information Retrieval angepaßte Kollektionen verwendet werden können.

Demnach sprechen verschiedene im ODMG-Objektmodell verwirklichte Modellierungskonzepte dafür, dieses Objektmodell als Grundlage für die Entwicklung des Objektmodells ProM für integrierte Datenbank- und Information Retrieval Systeme zu nutzen.

5.2.3 Die Relevanz des ODMG-Standards

In der ODMG ist ein Großteil der führenden Hersteller objektorientierter Datenbanksysteme, wie beispielsweise Gemstone oder POET, vertreten. Diese haben sich dazu verpflichtet, die Festlegungen des Standards in ihren Produkten umzusetzen. Daher beginnen der ODMG-Standard und insbesondere das enthaltene Objektmodell, sich im Bereich der objektorientierten Datenbanktechnologie als de-facto Standard durchzusetzen, der auch in der einschlägigen Literatur immer häufiger Beachtung findet (vgl. z.B. [Vos94] oder [KE97]).

Diese Position im Bereich der Datenbanken kann als ein weiterer Aspekt betrachtet werden, der den ODMG-Standard als Grundlage für ein integriertes System geeignet erscheinen läßt. Durch die Verbreitung des Standards können sowohl ein allgemeines Verständnis für die Grundlagen der vorliegenden Arbeit und der in ihr vorgenommenen Erweiterungen als auch die Existenz der entsprechenden technologischen Grundlagen vorausgesetzt werden.

5.3 Erweiterung um Information Retrieval Komponenten

Das Objektmodell ProM entsteht durch die Erweiterung des ODMG-Objektmodells um Komponenten, die zur Modellierung von Information Retrieval Daten und Verfahren erforderlich sind. Um die inhärente vage Semantik im Information Retrieval berücksichtigen zu können, wird hierbei auf probabilistische Konzepte (vgl. Abschnitt 2.1.1) zurückgegriffen. Da das Objektmodell ProM als ein Gesamtmodell gedacht ist, dessen Komponenten gleichberechtigt nebeneinander stehen und orthogonal kombinierbar sind, erfolgt eine genaue Beschreibung aller Komponenten, also sowohl der aus dem ODMG-Objektmodell übernommenen als auch der ergänzten probabilistischen Komponenten, gemeinsam innerhalb der nächsten Abschnitte. Um dennoch deutlich zu machen, wie sich das Modell ProM vom ODMG-Objektmodell unterscheidet, werden die Information Retrieval Erweiterungen in der detaillierten Beschreibung als solche gekennzeichnet[1]. Des weiteren soll dieser Abschnitt zuvor noch eine Übersicht über die vorgenommenen Ergänzungen bieten:

→ **Wahrscheinlichkeitswerte:** Wahrscheinlichkeitswerte werden für die Modellierung sowohl von vagen Eigenschaften für Datenobjekte als auch von probabilistischen Anfrageresultaten benötigt. Daher wird in Ergänzung der atomaren Literaltypen des ODMG-Objektmodells ein atomarer Literaltyp prob definiert, der Wahrscheinlichkeitswerte repräsentiert und entsprechende Operationen darauf zur Verfügung stellt.

→ **vage Eigenschaften:** Neben dem Zustand und dem Verhalten ist für Instanzen in Information Retrieval Anwendungen zusätzlich ihre vage Semantik entscheidend. Daher werden den im ODMG-Objektmodell als Eigenschaften für Instanzen definierten Attributen, Relationen und Operationen zwei weitere, vage Eigenschaften zur Seite gestellt. Hierbei handelt es sich zum einen um vage Prädikate, die von der jeweiligen Instanz nicht sicher, sondern nur mit einer bestimmten Wahrscheinlichkeit erfüllt werden. Der Aufbau dieser vagen Prädikate gleicht dem von Operationen. Sie berechnen für Objekte, unter Umständen in Abhängigkeit von Parametern, den entsprechenden Wahrscheinlichkeitswert. Daneben können in ProM vage Relationen definiert werden. Vage Relationen sind Beziehungen, durch die eine Typinstanz mit einer bestimmten Wahrscheinlichkeit mit Instanzen anderer Typen verbunden ist.

→ **Relevanzrangfolgen:** Die Ergebnisse probabilistischer Information Retrieval Anfragen liegen nicht als ungeordnete Mengen vor, sondern die gefundenen Instanzen werden nach ihrer wahrscheinlichen Relevanz für die gestellte Anfrage in eine Rangfolge eingefügt (vgl. Abschnitt 2.1.1). Die Resultate von Datenbankanfragen können im ODMG-Objektmodell auf einfache Weise durch Mengen, Multimengen oder Listen

[1] Diese Kennzeichnung vollzieht sich auf drei Ebenen. Zum einen wird innerhalb des Fließtextes ausdrücklich darauf hingewiesen, daß es sich bei den beschriebenen Eigenschaften um Erweiterungen des ODMG-Objektmodells handelt. Zum anderen werden in den OMT-Diagrammen, die die Modelleigenschaften in ihren Kontext einordnen, Erweiterungen durch die Verwendung roter Farbe hervorgehoben. Schließlich werden Erweiterungen innerhalb von Aufzählungen nicht durch den sonst in der Arbeit verwendeten Punkt, sondern durch das Symbol "→" eingeleitet.

modelliert werden, die als Kollektionsformen im Objektmodell enthalten sind. Um Information Retrieval Ergebnisse in gleicher Form darstellen zu können, wird die Relevanzrangfolge als eigene Kollektionsform in das Objektmodell ProM aufgenommen.

Die vorgenommenen Ergänzungen sind ebenso ein fester, grundlegender Bestandteil von ProM wie die aus dem ODMG-Objektmodell übernommenen Komponenten. Dieser Ansatz kann zwar die technische Integration der Information Retrieval Komponenten in bereits existierende Datenbanksysteme erschweren, da er Veränderungen der internen Systemstruktur voraussetzt. Das Objektmodell ProM soll jedoch die Informationen und Algorithmen der beiden integrierten Gebiete als geschlossenes Gebilde auf abstrakter Ebene reflektieren. Bei der technischen Realisierung integrierter Systeme kann die Eigenständigkeit der Information Retrieval Komponenten zugunsten einer einfacheren Integration eingeschränkt werden. So ist es z.B. möglich, den neu eingeführten atomaren Literaltyp prob als Klasse in eine bestehende Bibliothek des zugrundeliegenden Datenbanksystems einzufügen.

5.4 Typen

Typen legen Grenzen für Werte fest, die von Daten angenommen werden dürfen [Car96], und bestimmen Handlungen, die an oder mit Hilfe von Daten durchgeführt werden können. Hierdurch ergeben sich Schnittstellen[2], die Aussagen über die Kompatibilität von Daten untereinander zulassen [Neu95].
Das Objektmodell ProM ist streng typisiert. Alle in ihm auftretenden Daten besitzen mindestens einen Typ (vgl. Abb. 5.2)[3]. Konkrete Daten, die einem Typ zugeordnet sind, heißen Typinstanzen.
In ProM wird zwischen Objekt- und Literaltypen unterschieden. Objektinstanzen sind über einen Objektidentifikator eindeutig identifizierbar. Literalinstanzen hingegen besitzen keinen Identifikator, sondern sind lediglich über ihren Wert zugreifbar.
Für die Spezifikation von Typen stehen in ProM mit Literal-, Schnittstellen- und Klassendefinitionen drei unterschiedliche Modellkomponenten zur Verfügung, durch die verschiedene Eigenschaften für Typinstanzen beschrieben werden können.
Aus der Verwendung dieser Modellkomponenten leiten sich die Subtyp und die EXTENDS-Relation ab, die jeweils Typen miteinander verbinden. Mit Hilfe dieser beiden Relationen können bei der Erstellung eines Datenbankschemas Typhierarchien aufgebaut werden, entlang derer die Eigenschaften von übergeordneten Typen auf untergeordnete Typen übertragen werden. Dieser Vorgang wird als Vererbung (*inheritance*) bezeichnet.
Neben den Instanzeigenschaften, die für jede Instanz eines Typs unterschiedliche Ausprägungen besitzen, gibt es in ProM mit Extension und Schlüssel zusätzlich Eigenschaften, die einheitlich für den gesamten Typ definiert werden können.

[2]Der Begriff "Schnittstelle" wird weiter unten für das Objektmodell ProM neu definiert. An dieser Stelle wird er in seiner allgemeinen Bedeutung verwendet.
[3]Zur Darstellung der Modellierungskonzepte von ProM wird im folgenden die *Object Modelling Technique OMT* [RBP+91] verwendet.

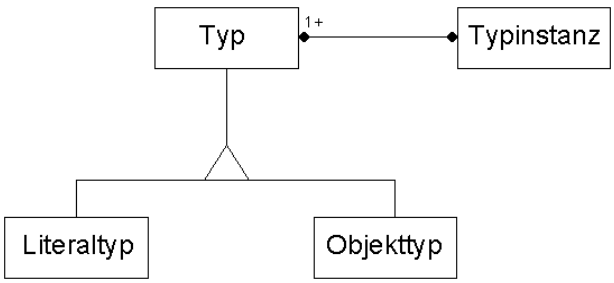


Abbildung 5.2: Die Instanziierung von Objekt- und Literaltypen

5.4.1 Definition von Typen

Die Definition eines Typs setzt sich in ProM aus einem Spezifikations- und einem Implementationsteil zusammen. Die Spezifikation beschreibt diejenigen Aspekte eines Typs, die für seine Benutzer sichtbar sind. Hierbei handelt es sich im einzelnen um

- Operationen, die für die Instanzen des Typs aufgerufen werden können,
- Ausnahmen, die während der Ausführung von Operationen auftreten können,
- Attribute und Relationen, deren Werte den Zustand einer Instanz beschreiben,

→ sowie um vage Prädikate und vage Relationen, die die Semantik einer Instanz zum Ausdruck bringen.

Bei probabilistischen Information Retrieval Verfahren stehen semantische Aspekte im Vordergrund, über deren Werte keine eindeutigen Aussagen getroffen werden können (vgl. Abschnitt 2.1). Die im letzten Punkt der obigen Auflistung aufgeführten vagen Prädikate und vagen Relationen modellieren diese unscharfe Semantik und werden dem ursprünglichen ODMG-Objektmodell hinzugfügt, um die Integration von Information Retrieval Funktionalität zu unterstützen.
Der Aufruf eines vagen Prädikates ergibt stets einen Wahrscheinlichkeitswert, dessen Berechnung unter Umständen in Abhängigkeit von Formalparametern (vgl. Abschnitt 5.8.4). Daneben werden vage Prädikate auch bei der Berechnung vager Relationen eingesetzt, deren Resultate immer Relevanzrangfolgen sind. Pragmatisch betrachtet kann die Semantik von Objekttypen daher als eine spezielle Form von Verhalten verstanden werden, die auf einen bestimmten Typ von Resultaten beschränkt ist (vgl. Abb. 5.3). Aufgrund dieser Einordnung erscheint es sinnvoll, die Semantik eines Typs auf die gleiche Weise zu spezifizieren wie sein Verhalten. Die Festlegungen des ODMG-Objektmodells für die Spezifikation von Verhalten werden daher im folgenden auf die Spezifikation von Semantik in ProM übertragen.

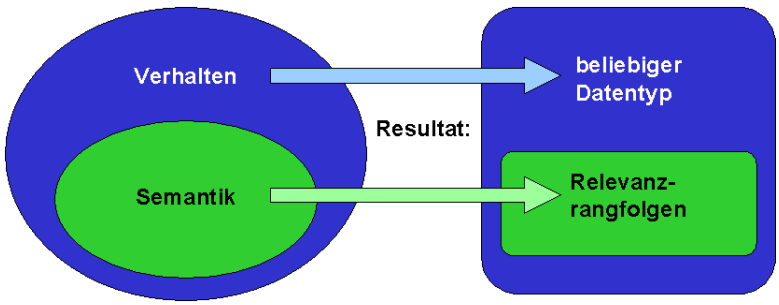


Abbildung 5.3: Eine Gegenüberstellung von Semantik und Verhalten

Zur Spezifikation eines Typs können verschiedene Komponenten von ProM herangezogen werden. Über Schnittstellendefinitionen (*interface definitions*) werden das abstrakte[4] Verhalten sowie die abstrakte Semantik von Objekttypen festgelegt (vgl. Abb. 5.4). Die resultierenden Schnittstellen (*interfaces*) sind nicht direkt instanziierbar. Sie können jedoch als Komponenten in der Subtyprelation eines Datenbankschemas auftreten und hierdurch abstraktes Verhalten und abstrakte Semantik an die entsprechenden Subtypen vererben (vgl. Abschnitt 5.4.2).

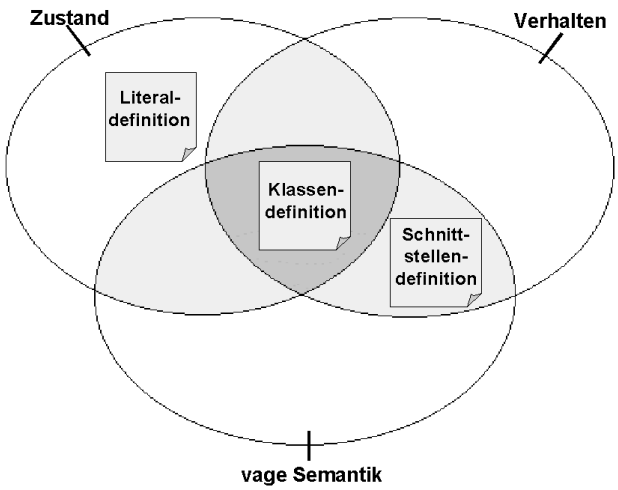


Abbildung 5.4: Die Verwendung verschiedener Komponenten zur Typspezifikation

[4]Der Begriff "abstrakt" wird in der vorliegenden Arbeit verwendet, um die Tatsache zu beschreiben, daß ein Aspekt keine dazugehörige Implementation besitzt. Abstraktheit im Sinne der Abgrenzung von Spezifikation und Implementation ist hier ausdrücklich nicht gemeint.

In Klassendefinitionen (*class definitions*) werden das Verhalten, die Semantik und auch der Zustand von Objekttypen spezifiziert. Hierdurch entstehen Klassen (*classes*), die direkt instanziierbar sind. Verhalten, Semantik und Zustand von Klassen werden über die EXTENDS-Relation vererbt (vgl. Abschnitt 5.4.2). Zu betonen ist, daß der Begriff "Klasse" hier ein Konzept bezeichnet und daher nicht mit den Implementationsklassen objektorientierter Programmiersprachen verwechselt werden sollte.
Literaldefinitionen (*literal definitions*) stellen ProMs dritte und letzte Modellkomponente zur Typspezifikation dar. Sie werden ausschließlich zur Festlegung des Zustandes von Literaltypen verwendet. Das Verhalten von Literaltypen läßt sich in ProM hingegen nicht spezifizieren. Die Operationen auf Literalen entsprechen vielmehr denen, die durch die verwendete Programmiersprache vorgegeben sind. Für Literaltypen bestehen keinerlei Vererbungsrelationen, sondern sie stellen einen grundlegenden Satz von nebengeordneten Typen dar, auf denen andere Datentypen aufgesetzt werden können.

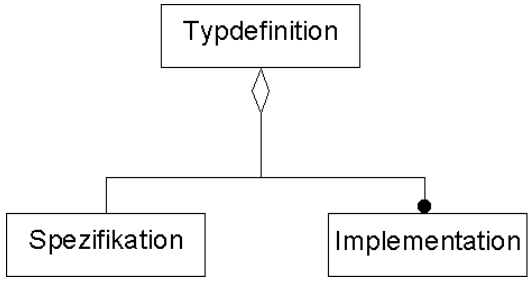


Abbildung 5.5: Die Unterscheidung zwischen Spezifikation und Implementation

Die Spezifikation von Typen wird in ProM durch deren Implementation ergänzt (vgl. Abb. 5.5). Dabei kann es zu einer Sepzifikation mehrere Implementationen geben. Eine Implementation enthält alle Aspekte eines Typs, die für den Benutzer nicht sichtbar sind. Konstrukte objektorientierter Programmiersprachen werden hier verwendet, um das in der Spezifikation festgelegte abstrakte Verhalten, die abstrakte Semantik und den abstrakten Zustand in Datenstrukturen und Methoden umzusetzen. Diese wiederum fließen direkt in die Anwendungsprogrammierung ein. Durch die Trennung von Spezifikation und Implementation eines Typs folgt ProM dem Prinzip der Datenkapselung (vgl. z.B. [Seb96]).

5.4.2 Vererbungsrelationen

Auf Objekttypen des Modells ProM lassen sich zwei verschiedene Arten von Relationen definieren, die die Basis für die Vererbung von Typeigenschaften bilden (vgl. Abb. 5.6).
Zum einen lassen sich abstraktes Verhalten und abstrakte Semantik über den Aufbau von Subtyprelationen vererben. Diese können als gerichtete Graphen zwischen Objekttypen aufgefaßt werden, in denen jede Kante einen Supertyp mit einem Subtyp verbindet. Der Subtyp erbt hierbei das gesamte Verhalten und die gesamte Semantik des Supertyps und transitiv

auch Verhalten und Semantik aller Supertypen des Supertyps. Dadurch kann dort, wo die Verwendung eines Objektes vom Supertyp gefordert ist, immer auch ein Objekt des Subtyps eingesetzt werden. Jede Instanz des Subtyps ist somit gleichzeitig auch eine logische Instanz des Supertyps.

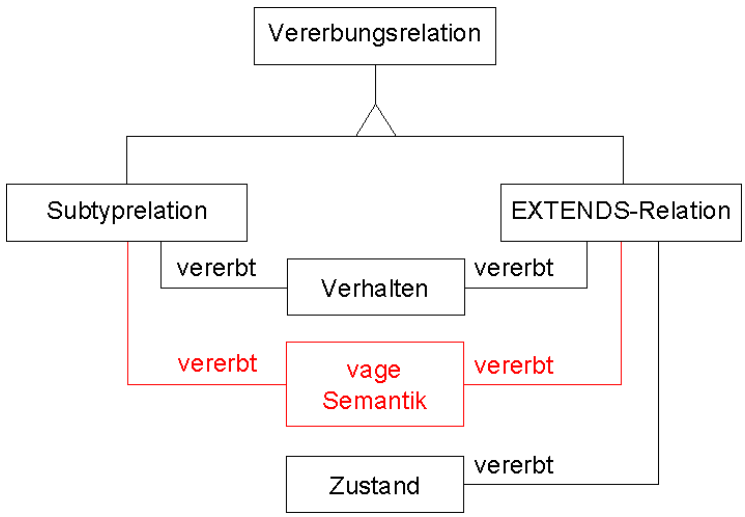


Abbildung 5.6: Die Vererbung über Subtyp- und EXTENDS-Relation

Neben den geerbten Eigenschaften kann ein Subtyp zusätzlich eigenes Verhalten und eigene Semantik definieren, die dann nur für diesen Typ und wiederum für alle seine Subtypen gültig sind. Subtypen sind in dieser Hinsicht also spezieller als ihre Supertypen, weshalb die Subtyprelation auch als Spezialisierung bzw. in der Gegenrichtung als Generalisierung bezeichnet wird.

Ererbte Eigenschaften müssen nicht unverändert übernommen werden. Vielmehr ist es möglich, die durch den Supertyp definierten Eigenschaften so umzugestalten, daß sie den speziellen Anforderungen des Subtyps gerecht werden. Durch das polymorphe Ausführungsparadigma objektorientierter Programmiersprachen, die auf den mit ProM definierten Datenbankschemata aufsetzen sollen, können zur Laufzeit diejenigen Operationen oder vagen Prädikate ausgewählt werden, die zum Typ der aufrufenden Instanz gehören (*late binding*).

Das Modell ProM unterstützt die Mehrfachvererbung von abstraktem Verhalten und abstrakter Semantik. Ein Typ kann somit Subtyp verschiedener Supertypen sein. Dadurch wäre grundsätzlich die Möglichkeit gegeben, daß ein Typ namensgleiche Operationen oder vage Prädikate mit verschiedenen Parametern von verschiedenen Supertypen erben könnte. Diese Form der Namensüberladung (*name overloading*) wird jedoch für ProM explizit ausgeschlossen.

Da eine Mehrfachvererbung für konkret implementierte Typeigenschaften durch Mehrdeutigkeiten zu Ineffizienzen führen kann [Cat96], bleibt die Subtyprelation in ProM auf die

Vererbung von abstraktem Verhalten und abstrakter Semantik beschränkt. Dieses wird dadurch erreicht, daß nur Schnittstellen als Supertypen in der Subtyprelation erscheinen dürfen (vgl. Abb. 5.7). Schnittstellen besitzen keine konkreten Instanzen, und Mehrdeutigkeiten bezüglich etwaiger Implementationen sind somit ausgeschlossen. Es ist ausreichend, diese Einschränkung für die Supertypseite der Subtyprelation einzuführen. Als Subtypen der Subtyprelation sind daher sowohl Schnittstellen als auch Klassen zugelassen.

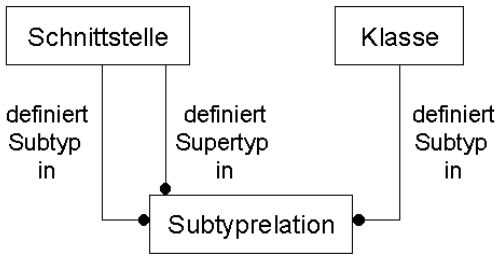


Abbildung 5.7: Die Komponenten der Subtyprelation

Um neben abstraktem Verhalten und abstrakter Semantik auch deren konkrete Implementationen sowie den Zustand von Objekttypen vererben zu können, wird in ProM zusätzlich zur Subtyprelation die EXTENDS-Relation eingeführt (vgl. Abb 5.8). Da der Zustand von Objekttypen zumeist in instanziierbaren Spezifikationen, mit anderen Worten also in Klassen, festgelegt wird, ist die EXTENDS-Relation in ProM als Beziehung zwischen jeweils zwei Klassen definiert. Die untergeordnete Klasse erbt von der übergeordneten Klasse ohne Ausnahme alle Typaspekte. Ebenso wie die Subtyprelation ist auch die EXTENDS-Relation transitiv. Um Mehrdeutigkeiten bei der Vererbung von konkreten Typaspekten auszuschließen, wird eine Mehrfachvererbung über die EXTENDS-Relation jedoch nicht zugelassen.

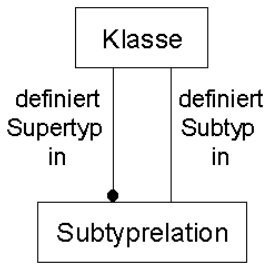


Abbildung 5.8: Die Komponenten der EXTENDS-Relation

5.4.3 Typeigenschaften

Neben dem Zustand, dem Verhalten und der Semantik eines Typs lassen sich in ProM weitere Eigenschaften festlegen, die für einen Typ unabhängig von seinen Instanzen stets gültig sind.
So wird die Menge aller Instanzen eines Typs innerhalb einer Datenbank als dessen Extension (*extent*) bezeichnet. Ist ein Typ A Subtyp eines Typs B, so ist die Extension von A eine Teilmenge der Extension von B, da alle Instanzen von A auch logische Instanzen von B sind. Extensionen können in ProM explizit benannt werden.
Können Instanzen eines Typs eindeutig durch ihre Werte für eine oder einige Instanzeigenschaften identifiziert werden, so wird die Vereinigungsmenge dieser Eigenschaften als Schlüssel (*key*) des Typs bezeichnet. Besteht ein Schlüssel aus genau einer Instanzeigenschaft, so heißt er einfacher Schlüssel (*simple key*) ansonsten zusammengesetzter Schlüssel (*compound key*). Der Bereich, für den die Eindeutigkeit eines Schlüssels gegeben ist, ist genau die Extension des Typs. Schlüssel sind daher nur für Typen mit Extension zulässig.

5.5 Objekte

Instanzen von Objekttypen werden als Objekte bezeichnet (vgl. Abb. 5.9). Jedes Objekt besitzt einen eindeutigen Objektidentifikator. Um die Nutzung der Objekte durch Programmierer und Endanwender zu vereinfachen, wird zusätzlich die Möglichkeit eingeräumt, den Objekten Namen zuzuweisen und diese für den Objektzugriff zu verwenden. Weiterhin wird ein Lebensdauerkonzept für Objekte definiert, welches je nachdem, wie der für Objekte alloziierte Speicherplatz verwaltet wird, zwischen transienten und persistenten Objekten unterscheidet. Zudem können Objekte in ProM anhand ihres Aufbaus in atomare Objekte, Kollektionen und strukturierte Objekte eingeteilt werden.

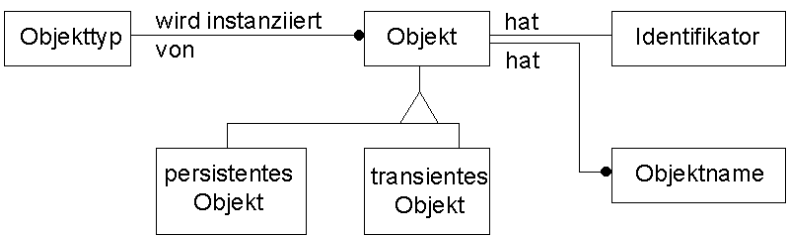


Abbildung 5.9: Die grundlegenden Konzepte von Objekttypen und Objekten

5.5.1 Erzeugung und Verwaltung von Objekten

Zur Erzeugung von Objekten stellt ProM Produktionsschnittstellen (*factory interfaces*) zur Verfügung. Diese enthalten Erzeugungsoperationen, die über Produktionsobjekte (*factory*

objects) aufgerufen werden können. Produktionsobjekte werden von der aufsetzenden objektorientierten Programmiersprache bereitgehalten.
Die folgende Produktionsschnittstelle[5] definiert die Operation new, die ein neues Objekt vom Typ Object erzeugt:

```
interface ObjectFactory {
     Object new();
};
```

Alle Objekte in ProM erben, ohne daß es bei ihrer Spezifikation explizit angegeben werden muß, das durch die folgende Schnittstelle definierte Verhalten:

```
interface Object {
    enum      Lock_Type(read, write, upgrade);
    exception LockNotGranted();
    void      lock(in Lock_Type mode) raises(LockNotGranted);
    boolean   try_lock(in Lock_Type mode);
    boolean   same_as(in Object anObject);
    Object    copy();
    void      delete();
};
```

Somit ist ein grundlegendes Verhalten für alle Objekte des Modells gewährleistet. Die Operation same_as vergleicht Objekte anhand ihrer Identität. Mit copy kann eine äquivalente Kopie des aufrufenden Objektes erzeugt werden, die dann einen eigenen Objektidentifikator (vgl. Abschnitt 5.5.2) erhält. Mit delete wird das aufrufende Objekt aus der Datenbank und dem flüchtigen Speicher entfernt.
Daneben definiert die Schnittstelle eine Reihe von Operationen zur Verwaltung von Sperren auf Objekten (vgl. Abschnitt 3.1.2.3). Es sei hier allerdings betont, daß in ProM prinzipiell davon ausgegangen wird, daß Sperren implizit durch das System verwaltet werden. Die spezifizierten Operationen stellen somit nur eine zusätzliche Möglichkeit zur Verwaltung von Sperren dar, die jedoch nicht das eigentliche Sperrkonzept des Modells repräsentiert.

5.5.2 Objektidentifikatoren

Jedes Objekt besitzt einen eindeutigen Objektidentifikator, über den auf dieses Objekt zugegriffen werden kann. Der Wert dieses Identifikators bleibt während der gesamten Lebensdauer des Objektes gleich. Ändert sich ein Attributwert oder eine Relation des Objektes, so wird sein Identifikator hiervon nicht berührt. Objekte können somit neue Werte annehmen, ohne ihre Identität zu verlieren.

[5] Die hier verwendete Notation entstammt der Definitionssprache ProDL (vgl. Kapitel 6). Im folgenden werden an einigen Stellen in Abweichung zu ProDL auch Kommentare verwendet, die dann durch "(* ... *)" eingebettet sind.

Die Generierung der Objektidentifikatoren ist Aufgabe des Datenbankmanagementsystems. Dabei gibt es eine Reihe von Implementationsmöglichkeiten, deren Beschreibung jedoch nicht Teil des Objektmodells ist.

5.5.3 Objektnamen

Um Programmierern und Endbenutzern den Zugriff zu erleichtern, bietet ProM die Möglichkeit, einem Objekt neben seinem Identifikator beliebig viele Namen zuzuweisen. Das Datenbankmanagementsystem nimmt eine entsprechende Abbildung zwischen den Namen eines Objektes und seinem Identifikator vor.
Der Namensraum, in dem der Name eines Objektes sichtbar ist, ist die gesamte Datenbank. Hierarchische Namensräume oder Namensräume, die sich über mehrere Datenbanken erstrecken, sind in ProM nicht vorgesehen.

5.5.4 Lebensdauer von Objekten

Die Lebensdauer eines Objektes legt fest, in welcher Weise der stabile und der flüchtige Speicherplatz für dieses Objekt verwaltet werden müssen. Jedem Objekt wird bei seiner Erzeugung eine Lebensdauer zugewiesen. Dabei wird zwischen transienten Objekten, die höchstens bis zum Ende der Laufzeit des entsprechenden Anwendungsprogramms existent sind, und persistenten Objekten, die durch das Datenbanksystem dauerhaft verwaltet werden, unterschieden.

5.5.5 Atomare Objekte

Atomare Objekttypen können durch Anwendungsprogrammierer bzw. Benutzer definiert werden, indem über Schnittstellen- oder Klassendefinitionen der Zustand, das Verhalten und die vage Semantik für Instanzen des Typs festgelegt wird. ProM selbst beinhaltet keine vordefinierten atomaren Objekttypen.

5.5.6 Kollektionen

Kollektionen fassen mehrere Objekte oder Literale gleichen Typs zusammen. Dabei kann die Zusammenfassung der Elemente anhand unterschiedlicher semantischer Regeln vorgenommen werden. Hierdurch ergeben sich sechs verschiedene Kollektionsformen:

- Menge (Set)
- Multimenge (Bag)
- Liste (List)
- Feld (Array)

- Verzeichnis (Dictionary)

→ Relevanzrangfolge (RankedList)

Relevanzrangfolgen werden dabei, abweichend vom ODMG-Objektmodell, als neues Konzept eingeführt. Sie sind als nach Wahrscheinlichkeitswerten geordnete Mengen zu verstehen und dienen der Repräsentation von Ergebnismengen probabilistischer Information Retrieval Verfahren.
Die Operationen auf Kollektionsobjekten werden durch die folgende Schnittstelle [6] spezifiziert:

```
interface Collection : Object {
    exception             InvalidCollectionType();
    unsigned long         cardinality();
    boolean               is_empty();
    boolean               is_ordered();
    boolean               allows_duplicates();
    boolean               allows_updates();
    boolean               contains_element(in any element);
    Iterator              create_iterator(in Boolean stable);
    BidirectionalIterator create_bidirectional_iterator(in Boolean stable)
                              raises(InvalidCollectionType);
};
```

Diese Operationen sind auf allen Kollektionsobjekten spezifiziert, unabhängig davon, um welche Kollektionsform es sich dabei handelt. So kann mit cardinality für jedes Kollektionsobjekt ermittelt werden, wieviele Elemente es enthält. Daneben gibt es einige boolesche Prädikate, die Auskunft über verschiedene Charakteristika von Kollektionsobjekten geben, beispielsweise darüber, ob ein Kollektionsobjekt geordnet ist oder ob ein bestimmtes Element darin vorkommt.
Zusätzlich ist es möglich, Iteratoren für Kollektionsobjekte anzulegen. Diese erlauben es, die enthaltenen Elemente eines nach dem anderen zu traversieren. Dabei können unidirektionale Iteratoren für Objekte jeglicher Kollektionsform erzeugt werden, während bidirektionale Iteratoren nur für geordnete Kollektionsformen, sprich für Listen, Felder und Relevanzrangfolgen, vorgesehen sind.
Der Typ Collection ist Subtyp des Typs Object und erbt somit dessen Operationen. Diese müssen jedoch teilweise an die Gegebenheiten des Typs Collection angepaßt werden. So wird für die Operation copy festgelegt, daß nur das Kollektionsobjekt an sich kopiert wird, während die enthaltenen Elemente identisch zu denen des Ausgangsobjektes sind. Es wird demnach eine flache Kopie (*shallow copy*) angelegt. Die Operation delete löscht zunächst das

[6] Der Platzhalter any wird hier analog zum ODMG-Standard verwendet, obwohl dieses zu Mißverständnissen führen kann. Daher sei an dieser Stelle ausdrücklich darauf hingewiesen, daß any nicht für einen beliebigen Typ, sondern stets für einen zum Ausdruck passenden Typ eingesetzt ist.

Kollektionsobjekt selbst. Waren hierin Literale enthalten, so werden diese automatisch mitgelöscht. Enthält ein Kollektionsobjekt Objekte, so bleiben diese auch nach seiner Löschung erhalten.
Neben der Semantik von Kollektionsobjekten stellt ihre Veränderlichkeit in ProM ein weiteres Unterscheidungskriterium dar. Das Modell weicht hierbei vom ODMG-Objektmodell ab, da die Einführung der Relevanzrangfolge als Kollektionsform dieses erforderlich macht. Die durch das ODMG-Objektmodell festgelegte Schnittstelle Collection definiert Operationen, um einzelne Elemente in eine Kollektion einzufügen oder daraus zu entfernen. Diese Operationen können für Mengen, Multimengen, Listen, Felder und Verzeichnisse aufgerufen werden. Für Relevanzrangfolgen dagegen ist ihre Verwendung nicht sinnvoll. Zum einen entstehen Relevanzrangfolgen vor dem Hintergrund einer bestimmten vagen Anfrage, die beim nachträglichen Einfügen eines Elementes rekonstruiert werden müßte. Zum anderen basieren die enthaltenen Relevanzwahrscheinlichkeiten auf der gesamten der Anfrage zugrundeliegenden Menge. Das Einfügen oder Löschen eines einzelnen Elementes stellt somit alle Wahrscheinlichkeitswerte in einer Relevanzrangfolge und damit auch die Ordnung darauf in Frage.

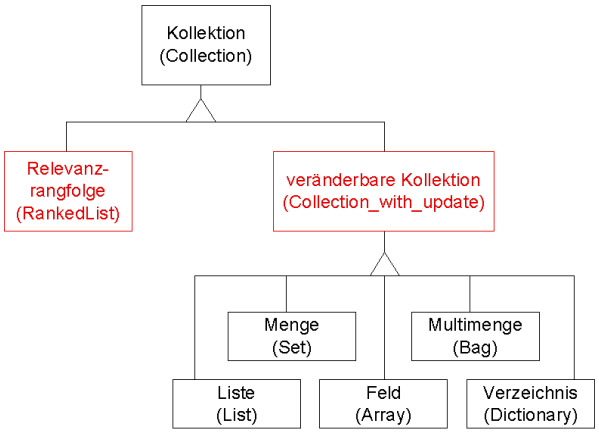


Abbildung 5.10: Veränderbare und nicht veränderbare Kollektionen

Relevanzrangfolgen besitzen abgesehen von der Möglichkeit zur Aktualisierung jedoch alle anderen Eigenschaften, die im ODMG-Objektmodell für Kollektionen definiert werden. Um diese Eigenschaften erben zu können und auch um die Bedeutung von Relevanzrangfolgen korrekt in das Objektmodell ProM einzubringen, wird hier der Typ Collection zunächst zu zwei verschiedenen Subtypen spezialisiert (vgl. Abb 5.10). Auf der einen Seite steht der Typ RankedList für Relevanzrangfolgen, dessen Schnittstelle weiter unten beschrieben wird. Auf deren Seite wird ein Typ Collection_with_updates definiert, der die vom ProM-Typ Collection geerbten Operationen genau um die Operationen zum Einfügen und Entfernen einzelner Elemente ergänzt:

```
interface Collection_with_updates : Collection {
    exception             ElementNotFound(any element);
    void                  insert_element(in any element);
    void                  remove_element(in any element)
                              raises(ElementNotFound);
};
```

Die Typen Set, Bag, List, Array und Dictionary sind Subtypen von Collection_with_updates, so daß ihnen der gleiche Satz an Operationen zur Verfügung steht wie auch den Kollektionen im ODMG-Objektmodell.
Das ODMG-Objektmodell wird an dieser Stelle nicht nur erweitert, sondern die vorgegebene Klassenhierarchie wird verändert. Dieses kann bei der praktischen Integration von Information Retrieval Funktionalität in bereits bestehende objektorientierte Datenbanksysteme zu Problemen führen. Bei der Entwicklung des Objektmodells ProM steht jedoch der Wunsch im Vordergrund, die durch Datenbank- und Information Retrieval Anwendungen gegebene Realität möglichst genau abbilden zu können. Daher werden Relevanzrangfolgen zunächst auf die beschriebene Art und Weise in das Modell ProM eingefügt.
Die Lösung etwaiger Schwierigkeiten bei der Übertragung auf bestehende, dem ODMG-Standard folgende Systeme wird auf die Implementationsebene verlagert. Ein Ansatz wäre hier beispielsweise, den Typ RankedList ebenso wie alle anderen Kollektionstypen direkt vom Typ Collection, wie er im ODMG-Objektmodell definiert ist, erben zu lassen und die Operationen zum Einfügen und Entfernen von Elementen für Relevanzrangfolgen als leere Operationen zu redefinieren.

5.5.6.1 Mengenobjekte

Mengenobjekte sind ungeordnete, veränderbare Kollektionen, in denen jedes Element nur einmal vorkommen darf.
Die Operationen für Mengenobjekte werden durch die Schnittstelle Set festgelegt. Sie entsprechen den bekannten Mengenoperationen. Außerdem werden boolesche Prädikate spezifiziert, die feststellen, ob zwischen dem aufrufenden und einem weiteren Mengenobjekt Teilmengenbeziehungen beliebiger Form bestehen.
Der Typ Set ist ein Subtyp des Typs Collection_with_updates. Die geerbte Operation insert_element ist an den Mengentyp anzupassen. Es werden nur Elemente in eine Menge eingefügt, die dort noch nicht vorhanden sind. Bei dem Versuch, ein schon enthaltenes Element einzufügen, bleibt die ursprüngliche Menge unverändert.

5.5.6.2 Multimengenobjekte

Multimengenobjekte sind ungeordnete, veränderbare Kollektionen, in denen jedes Element beliebig oft vorkommen darf.
Die Operationen für Multimengenobjekte werden durch die Schnittstelle Bag festgelegt. Eine Operation, die für ein bestimmtes Element ermittelt, wie oft es in der aufrufenden Multimenge vorkommt, wird spezifiziert. In den Mengenoperationen Vereinigung, Durchschnitt

und Differenz wird diese Vorkommenshäufigkeit der Semantik der jeweiligen Operation entsprechend erhöht oder verringert.
Auf die gleiche Weise werden auch die vom Supertyp Collection_with_updates geerbten Operationen für den Typ Bag verfeinert. Beim Einfügen eines Elementes in eine Multimenge durch insert_element wird dessen Vorkommenshäufigkeit um eins inkrementiert; beim Entfernen eines Elementes durch remove_element wird sie um eins dekrementiert.

5.5.6.3 Listenobjekte

Listenobjekte sind veränderbare Kollektionen, in denen jedes Element beliebig oft vorkommen kann. Zusätzlich sind Listen bezüglich eines Index geordnet. In ProM beginnt dieser Index mit 0. Die meisten Operationen, die über die Schnittstelle List spezifiziert werden, sind entsprechend auf eine bestimmte Position innerhalb der Liste bzw. auf den Listenanfang oder das Listenende bezogen. Die Auswahl, das Einfügen und das Entfernen von Elementen sind daher sowohl am Anfang und am Ende der Liste als auch an einer beliebig über den Index bestimmbaren Position innerhalb der Liste möglich.
Zusätzlich spezifiziert die Schnittstelle zwei Operationen zum Aneinanderhängen von Listenobjekten. Im ersten Fall entsteht dabei ein neues Listenobjekt. Im zweiten Fall wird das vordere Listenobjekt durch das Anhängen des hinteren verändert.
Die vom Supertyp Collection_with_updates geerbten Operationen insert_element und remove_element werden an den Typ List angepaßt. Mit insert_element wird das spezifizierte Element am Ende der Liste eingefügt. Durch remove_element wird es an der Position aus der Liste entfernt, an der es das erste Mal auftritt.

5.5.6.4 Feldobjekte

Ein Feldobjekt ist eine veränderbare Kollektion von Elementen, die beliebig oft vorkommen können und anhand ihrer Position innerhalb des Feldes zugreifbar sind.
Über die Schnittstelle Array werden Operationen spezifiziert, mit denen ein Feldelement an einer bestimmten Position ausgewählt, ersetzt oder entfernt werden kann. Beim Entfernen eines Elementes aus einem Feld ändert sich, im Gegensatz zur Liste, die Länge des Feldes nicht. An die Position des entfernten Elementes tritt ein Leerwert (*null value*).
Die Länge eines Feldes ist die maximale Anzahl der Elemente, die es aufnehmen kann. Sie kann mit Hilfe einer durch die Schnittstelle spezifizierten Operation geändert werden.
Die vom Supertyp Collection_with_updates geerbten Operationen insert_element und remove_element werden an den Typ Array angepaßt. Mit insert_element wird die Länge des aufrufenden Feldobjektes um eins inkrementiert und das spezifizierte Element in die freie Zelle am Ende des Feldes eingefügt. Mit remove_element wird das spezifizierte Element an der Position im Feldobjekt, an der es das erste Mal auftritt, durch einen Leerwert ersetzt.

5.5.6.5 Verzeichnisobjekte

Verzeichnisobjekte sind ungeordnete, veränderbare Folgen von Paaren aus Schlüsseln und Werten, wobei jeder Schlüssel nur einmal im gesamten Verzeichnis enthalten sein darf. Die

Schlüssel-Wert-Paare sind durch Instanzen der folgenden Struktur repräsentiert:

```
struct Association {any key; any value; };
```

Die Paare werden im Ganzen als Elemente des Verzeichnisobjektes aufgefaßt, so daß jeder Zugriff auf ein Verzeichniselement immer ein Schlüssel-Wert-Paar zum Ergebnis hat.

Die Schnittstelle Dictionary spezifiziert Operationen, mit denen Verzeichniselemente anhand ihres Schlüssels ausgewählt oder gelöscht werden können. Neue Schlüssel-Wert-Paare können in ein Verzeichnis eingefügt werden. Ist der Schlüssel bereits vorhanden, so wird das existierende Paar durch das neue Paar ersetzt. Eine Operation, die ermittelt, ob im aufrufenden Verzeichnis ein bestimmter Schlüssel bereits existiert, wird ebenfalls durch die Schnittstelle definiert.

Die vom Supertyp Collection_with_updates geerbten Operationen insert_element und remove_element sowie die vom indirekten Supertyp Collection geerbte Operation contains_element werden an den Typ Dictionary angepaßt. Bei ihrer Nutzung ist darauf zu achten, daß nur Schlüssel-Wert-Paare als Parameter übergeben werden können. Die Operation insert_element verhält sich wie die oben bereits spezifierte Einfügeoperation. Die Operation remove_element entfernt ein Schlüssel-Wert-Paar aus einem Verzeichnis, wenn dieses Paar darin genau in der angegebenen Kombination enthalten ist. Sollte ein Paar existieren, dessen Schlüssel zwar mit dem des spezifizierten Paares übereinstimmt, nicht aber sein Wert, so bleibt dieses Paar von der Operation remove_element unberührt. Die Operation contains_element liefert genau dann den booleschen Wert "wahr", wenn das spezifizierte Schlüssel-Wert-Paar in der angegebenen Kombination im aufrufenden Verzeichnis vorkommt.

5.5.6.6 Relevanzrangfolgeobjekte

Anfragen im Information Retrieval werden anders als im Datenbankbereich nicht unter Benutzung boolescher Logik durchgeführt (vgl. Abschnitt 2.1). Vielmehr wird die Wahrscheinlichkeit ermittelt, mit der ein betrachtetes Objekt die vagen Anfragebedingungen erfüllt. Nach diesen Wahrscheinlichkeitswerten kann dann die Ergebnismenge geordnet werden. Um Information Retrieval Resultate entsprechend repräsentieren zu können, wird in ProM die Relevanzrangfolge als Kollektionsform ergänzt.

Relevanzrangfolgen sind unveränderbare Kollektionen, in denen kein Element mehrfach vorkommen darf. Innerhalb einer Relevanzrangfolge ist jedem Element ein Wahrscheinlichkeitswert vom Typ prob zugeordnet (vgl. Abb. 5.11). Die Wahrscheinlichkeitswerte selbst sind jedoch im Gegensatz zu Schlüsseln in Verzeichnissen kein Teil der Elemente einer Relevanzrangfolge. Stattdessen werden die Elemente durch die Wahrscheinlichkeiten so geordnet, daß das Element mit dem höchsten Wahrscheinlichkeitswert an erster Position der Relevanzrangfolge steht, während das Element mit der kleinsten Wahrscheinlichkeit die letzte Position einnimmt. Relevanzrangfolgen beginnen ebenso wie beispielsweise Listen mit der

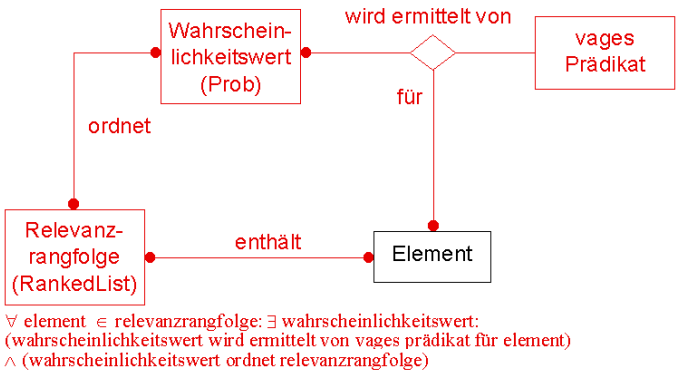


Abbildung 5.11: Die Relevanzrangfolge als Kollektionsform

Position 0. Ist der Wahrscheinlichkeitswert zweier Elemente identisch, so teilen sich diese beiden Elemente eine einzige Position innerhalb der Relevanzrangfolge. Das folgende Beispiel verdeutlicht den Aufbau von Relevanzrangfolgen:

Position	Element	Wahrscheinlichkeit
0	"Datenbankhandbuch"	0,89
1	"Datenbanken"	0,82
	"Objektbanken"	0,82
2	"Betriebssysteme"	0,51

Durch die folgende Schnittstelle RankedList sind verschiedene Operationen für Relevanzrangfolgen spezifiziert:

```
interface RankedList : Collection {
    RankedList set_lowest_prob(in prob lowest);
       (* begrenzt die Liste auf Elemente, deren
       Wahrscheinlichkeit fuer das vage Praedikat
       groesser oder gleich dem Wert "lowest" ist *)
    RankedList set_number(in unsigned long number);
       (* begrenzt die Liste auf "number" Elemente *)
    prob get_element_prob(in any element);
       (* Ergebnis: Die Relevanzwahrscheinlichkeit fuer ein
        bestimmtes Element *)
    unsigned long get_element_pos(in any element);
       (* Ergebnis: Die Position eines Elementes in der
```

```
            Rangfolge. Elemente mit gleicher Relevanz-
            wahrscheinlichkeit haben die gleiche Position *)
        set<any> get_prob_elements(in prob probability);
            (* Ergebnis: Die Menge der Elemente, die in der Liste
            ein Relevanzwahrscheinlichkeit vom Wert "probability"
            haben *)
        set<any> get_pos_elements(in unsigned long pos);
            (* Ergebnis: Die Menge der Elemente, die in der Liste
            die Position "pos" besetzen *)
        set<any> get_first_elements();
            (* Ergebnis: Die Menge der Elemente, die in der Liste
            die Position 0 besetzen *)
        set<any> get_last_elements();
            (* Ergebnis: Die Menge der Elemente, die in der Liste
            die letzte Position besetzen *)
};
```

Anders als die übrigen Kollektionsformen sind Relevanzrangfolgen nicht veränderbar. Daher ist der Typ RankedList zwar ein Subtyp des Typs Collection, aber kein Subtyp des Typs Collection_with_updates.
Die Größe der Ergebnismenge vager Anfragen ist von vornherein nicht eindeutig bestimmbar. Theoretisch betrachtet können sich alle Elemente, gegen die eine vage Anfrage gestellt wurde, auch in der Ergebnismenge wiederfinden. Hierbei wäre dann jedoch einigen Elementen ein sehr geringer Wahrscheinlichkeitswert zugeordnet, und es ist fragwürdig, ob solche Elemente dann überhaupt in einer Ergebnismenge enthalten sein sollten.
Aus diesen Überlegungen heraus werden für den Typ RankedList zwei Operationen spezifiziert, durch die Relevanzrangfolgen auf eine sinnvolle Größe beschränkt werden können. Mit set_lowest_prob kann angegeben werden, welche Wahrscheinlichkeit einem Element mindestens zugeordnet sein muß, damit dieses in der aufrufenden Relevanzrangfolge enthalten ist. Mit set_number kann die Anzahl der Elemente, die in der aufrufenden Relevanzrangfolge vorkommen dürfen, direkt begrenzt werden. Falls der Aufruf von set_number eine Relevanzrangfolge so abschneidet, daß einige Elemente einer mehrfach besetzten Position enthalten wären und einige nicht, wird die Anzahl der erlaubten Elemente so erhöht, daß alle Elemente der fraglichen Position in der Relevanzrangfolge bleiben. Wird der Umfang einer Relevanzrangfolge nicht begrenzt, so sind in ihr alle Instanzen ihres Elementtyps enthalten. Für jedes Element einer Relevanzrangfolge kann die ihm zugeordnete Wahrscheinlichkeit sowie seine Position in der Rangfolge ermittelt werden. Dieses geschieht mit Hilfe der Operationen get_element_prob und get_element_pos. Andersherum können über die Operationen get_prob_elements und get_pos_elements diejenigen Elemente in der Relevanzrangfolge berechnet werden, die mit einer bestimmten Wahrscheinlichkeit bewertet wurden bzw. die eine bestimmte Position in der Rangfolge belegen. Dabei ist zu beachten, daß eine einzelne Wahrscheinlichkeit für mehrere Elemente zutreffen kann und dadurch auch eine einzelne Position von mehreren Elementen belegt sein kann.

5.5.7 Strukturierte Objekte

Eine Reihe von Objekttypen ist in ProM bereits vordefiniert. Mit Hilfe des Typs Interval können Zeitintervalle dargestellt werden. Datumsangaben und Uhrzeiten können mit Date und Time verwaltet werden. Eine Kapselung beider Typen im Typ Timestamp ermöglicht das Setzen von Zeitmarken.
Für die aufgeführten Typen sind Schnittstellen und Produktionsschnittstellen definiert, die der ANSI SQL Spezifikation entsprechen. Eine detaillierte Ausführung dieser Schnittstellen würde den Rahmen der vorliegenden Arbeit jedoch überschreiten, so daß an dieser Stelle auf die entsprechende Literatur [Cat97] verwiesen sei.

5.6 Literale

Instanzen von Literaltypen werden als Literale bezeichnet (vgl. Abb 5.12). Literale besitzen keine Objektidentifikatoren, über die sie referenziert werden können. Ihre Existenz ist abhängig von ihrer Einbettung in ein Objekt oder ihrer Zuordnung zu einer Programmvariablen.

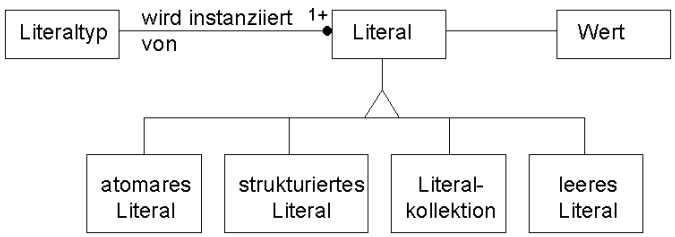


Abbildung 5.12: Die grundlegenden Konzepte von Literaltypen und Literalen

In ProM wird zwischen atomaren Literalen, Kollektionsliteralen, strukturierten Literalen und leeren Literalen (*null literals*) unterschieden.

5.6.1 Atomare Literale

Instanzen von atomaren Literaltypen werden nicht durch explizite Programmanweisungen erzeugt, sondern sind implizit im Wertebereich des Typs vorhanden.
Grundsätzliches Ziel von ProM ist es, daß bei der Anbindung an eine Programmiersprache alle durch das Modell definierten atomaren Literale unterstützt werden sollen. Dieses geschieht, indem entweder sprachspezifische Analoga der in ProM spezifizierten atomaren Literaltypen verwendet werden oder, falls in der Programmiersprache für einen der aufgeführten Literaltypen kein Gegenstück existiert, indem der Klassenbibliothek der Sprache eine entsprechende Implementation hinzugefügt wird. Sind in der aufsetzenden Programmiersprache noch weitere atomare Literaltypen definiert, so können auch sie im Rahmen von ProM verwendet werden.

ProM unterstützt folgende atomare Literale, wobei es sich beim Typ prob für Wahrscheinlichkeitswerte um eine Erweiterung des ODMG-Objektmodells handelt:

- ganze Zahlen mit Vorzeichen (long und short)
- positive ganze Zahlen (unsigned long und unsigned short)
- reelle Zahlen (float und double)
- Achtbitzeichen (octet)
- Zeichen (char) und Zeichenketten (string)
- Aufzählungen (enum)

→ Wahrscheinlichkeitswerte (prob) und Wahrheitswerte (boolean)

Für Aufzählungen steht der Typgenerator enum zur Verfügung, mit dem neue Literaltypen definiert werden können. Deren Instanzen können nur die bei der Deklaration aufgelisteten Werte annehmen. Als Beispiel mag hier ein Attribut dokument dienen, das durch folgende Deklaration definiert wird:

```
attribute enum dokument {brief, zeitschrift, buch};
```

Bei einem Rückgriff auf Information Retrieval Funktionalität in einem integrierten System werden Datenobjekte nicht mehr unter Betrachtung boolescher Eigenschaften ausgewertet. Vielmehr werden Wahrscheinlichkeitswerte ermittelt, die eine Aussage darüber treffen, wie wahrscheinlich ein Datenobjekt eine vage Eigenschaft erfüllt. Zur Repräsentation solcher probabilistischen Konzepte definiert ProM an dieser Stelle in Ergänzung des ODMG-Objektmodells den atomaren Literaltyp prob für Wahrscheinlichkeitswerte. Dieser wird sowohl als Ergebnistyp vager Prädikate (vgl. Abschnitt 5.8.4) als auch als Ordnungskriterium für Relevanzrangfolgen (vgl. Abschnitt 5.5.6.6) eingesetzt. Weiterhin kann er zur Weiterverarbeitung von Wahrscheinlichkeitswerten verwendet werden.
Eine Instanz des Typs prob ist ein Wahrscheinlichkeitswert im Intervall $[0, 1]$. Sie ist zumeist das Ergebnis eines vagen Prädikates und beschreibt, wie wahrscheinlich es ist, daß das durch das Prädikat beschriebene Ereignis eintritt. Der Wahrscheinlichkeitswert 1 bezeichnet das sichere Ereignis und ist die Entsprechung des Wahrheitswertes true. Der Wert 0 repräsentiert das unmögliche Ereignis und damit den Wahrheitswert false.
Für Literale des Typs prob werden verschiedene Operationen definiert. Den mathematischen Grundlagen für die Wahrscheinlichkeitsrechnung folgend werden zunächst die kombinatorischen Operatoren and, or und not eingeführt [CCH92]:

```
prob and(prob p1, ..., pn)
(* Wahrscheinlichkeit, dass alle Ereignisse pi eintreten =
    self * p1 * ... * pn    *)
```

```
prob or(prob p1, ..., pn)
(* Wahrscheinlichkeit, dass mindestens eines der Ereignisse pi
   eintritt = 1 - (1- self) * (1 - p1) * ... * (1 - pn)     *)

prob not()
(* Gegenwahrscheinlichkeit = 1 - self *)
```

Diese Operatoren folgen somit einer ähnlichen Semantik wie die gleichnamigen kombinatorischen Operatoren für Wahrheitswerte, nur daß hier keine boolesche sondern eine probabilistische Logik als Basis für die entsprechenden Berechnungen verwendet wird. Bei der Kombination von Wahrscheinlichkeitswerten wird die stochastische Unabhängigkeit der zugrundeliegenden Ereignisse angenommen, da dieses im allgemeinen den Voraussetzungen im Information Retrieval entspricht [Fuh96a].
Da alle weiteren kombinatorischen Operationen, wie beispielsweise Implikation oder Äquivalenz, mit Hilfe der obigen Operatoren dargestellt werden können [HSZ96], wird hier auf die Definition der entsprechenden Methoden verzichtet.
Neben den Operatoren der Kombinatorik sind zusätzliche Operatoren auf Wahrscheinlichkeitswerten definiert, die sich für Verfahren des Information Retrieval (vgl. z.B. [CCH92]) benötigt werden und eine Reihe von Wahrscheinlichkeitswerten zu einem einzigen Wert aggregieren.

```
prob avg(prob p1, ..., pn)
(* Durchschnittliche Wahrscheinlichkeit der Ereignisse pi =
   (self + p1 + ... + pn) / (n + 1)  *)

prob wavg(prob p1, ..., pn, Unsigned long w0, w1, ..., wn)
(* gewichteter Durchschnitt der Wahrscheinlichkeiten der Ereignisse pi =
   ( w0 * self + w1 * p1 + ... + wn * pn) / (w0 + w1 + ... + wn)  *)

prob max(prob p1, ..., pn)
(* die hoechste der Wahrscheinlichkeiten pi
   inklusive der aufrufenden Instanz *)

prob min(prob p1, ..., pi)
(* die geringste der Wahrscheinlichkeiten pi
   inklusive der aufrufenden Instanz *)
```

Weitere Operatoren entsprechen unären und binären Prädikaten auf Wahrscheinlichkeitswerten. Neben den üblichen Vergleichsoperatoren $=$, $\#$, $>$, $\geq$, $<$ und $\leq$ ist die Überprüfung auf das sichere und unmögliche Ereignis gegeben:

```
Boolean certain()
(* Ist ein Ereignis sicher: self ?= 1  *)
```

```
Boolean impossible()
(* Ist ein Ereignis unmoeglich: self ?= 0  *)
```

Für die Integration von Wahrscheinlichkeitswerten in das Modell ProM wird somit ein neuer atomarer Literaltyp eingeführt. Da hierdurch die durch das ODMG-Objektmodell vorgegebene Hierarchie der Basisklassen verändert wird, können sich Schwierigkeiten bei der technischen Anbindung von integrierten Datenbank- und Information Retrieval Systemen an bereits existierende Datenbanksysteme ergeben, die dem ODMG-Standard folgen. Dennoch wird der beschriebene Weg gewählt, da das Objektmodell ProM eine möglichst intuitive Modellierung sowohl von herkömmlichen Datenbankschemata als auch von Daten des Information Retrieval ermöglichen soll. Es sei jedoch darauf hingewiesen, daß für die Implementation eines integrierten Systems eine den praktischen Anforderungen entsprechende Vorgehensweise gefunden werden muß.

5.6.2 Kollektionsliterale

Neben Kollektionsobjekten können in ProM mit Hilfe der folgenden Typgeneratoren auch Kollektionsliterale definiert werden:

- set<t>
- bag<t>
- list<t>
- array<t>
- dictionary<t>

→ rankedList<t>

Kollektionsliterale unterscheiden sich von Kollektionsobjekten lediglich dadurch, daß sie keine eigene Identität besitzen. Die Bedeutung der einzelnen Kollektionsformen dagegen ist für Kollektionsobjekte und Kollektionsliterale gleich. Die Elemente von Kollektionsliteralen können Instanzen eines Objekttyps oder Instanzen eines Literaltyps sein.

5.6.3 Strukturierte Literale

Strukturierte Literale werden einfacher auch als Strukturen (*structures*) bezeichnet. Sie bestehen aus einer festen Anzahl von Komponenten, die entweder mit Literalen oder Objekten besetzt sein können (vgl. Abb. 5.13). Die einzelnen Komponenten können dabei von beliebigem Typ sein. Auf jede Komponente eines strukturierten Literals kann mit Hilfe der

Punktnotation und des Bezeichners der Komponente zugegriffen werden. So läßt sich beispielsweise die Straße innerhalb einer Adresse a durch den Ausdruck a.strasse ermitteln (vgl. Beispiel unten)[7] .

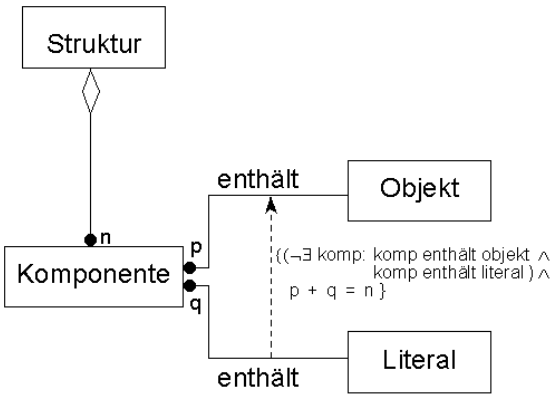


Abbildung 5.13: Strukturen und ihre Komponenten

Die strukturierten Literaltypen interval, date, time und timestamp sind durch ProM bereits vordefiniert. Ihre Semantik ist analog zu der der entsprechenden strukturierten Objekttypen festgelegt.

Daneben können Anwendungsprogrammierer mit Hilfe des struct-Konstrukts weitere strukturierte Literaltypen definieren. Die Definition eines Literaltyps Adresse könnte beispielsweise wie folgt aussehen:

```
struct Adresse {
    string         strasse;
    unsigned short number;
    string         plz;
    string         stadt
};
```

Strukturen können frei mit allen im Modell ProM auftretenden Typen kombiniert werden. So können beispielsweise Kollektionen Instanzen eines strukturierten Literaltyps als Elemente enthalten, oder eine Komponente einer Struktur kann wiederum durch eine Instanz eines anderen strukturierten Literaltyps besetzt sein.

[7] In Anhang A wird ein exemplarisches Datenbankschema beschrieben, das diesem und den meisten folgenden Beispielen zugrundeliegt. Hierdurch wird ein einheitlicher Kontext geschaffen, der das Verständnis für die Beispiele fördern soll.

5.6.4 Leere Literale

Zu jedem Literaltyp gibt es in ProM einen entsprechenden Literaltyp, der alle Instanzen des Ausgangstyps umfaßt und zusätzlich die Instanz "nil" als Leerwert beinhaltet. Die Semantik des Leerwertes ist ebenso definiert wie in SQL-92 [CO92].

5.7 Typisierung und Typhierachie

ProM verfolgt eine strenge Typisierung. Jede Instanz besitzt daher mindestens einen Typ. Alle Attribute und alle Traversierungen von booleschen und vagen Relationen sind typisiert ebenso wie die Argumente von Operationen und vagen Prädikaten. Zwei Instanzen sind genau dann vom gleichen Typ, wenn sie so deklariert wurden (*Namensäquivalenz*). Wurden zwei Objekte oder Literale als Instanzen von zwei unterschiedlichen Typen deklariert, so sind sie selbst dann nicht vom gleichen Typ, wenn sie die gleichen Attribute, Relationen, Operationen und vagen Prädikate aufweisen. Zwei atomare Literale sind vom gleichen Typ, wenn sie zu derselben Menge von Literalen gehören. In Abhängigkeit von der aufsetzenden Programmiersprache kann es implizite Konversionen zwischen atomaren Literaltypen geben. Das Objektmodell ProM selbst beinhaltet derartige Konversionen jedoch nicht.
Die Kompatibilität zweier Typen orientiert sich an der durch Subtyprelationen aufgebauten Typhierarchie eines Systems. Für die oben vorgestellten grundlegenden Objekt- und Literaltypen beinhaltet ProM bereits eine solche Hierarchie (vgl. Abb. 5.14). Falls T ein Typ und ST ein Subtyp von T ist, so kann jeder Variablen vom Typ T auch eine Instanz vom Typ ST zugewiesen werden. Eine Variable vom Subtyp ST dagegen kann keine Instanz vom Supertyp T aufnehmen.

5.8 Zustand, Verhalten und Semantik von Objekten

Eine Instanz kann aus verschiedenen Perspektiven betrachtet werden (vgl. Abb. 5.15). Sie verfügt über Zustandseigenschaften, deren Werte zugreifbar sind. Hierbei handelt es sich entweder um Attribute oder um Relationen. Attribute sind Werte, die sich allein auf eine Instanz beziehen, während Relationen die Instanz mit einer oder mehreren Instanzen eines anderen oder desselben Typs verbinden.
Auch das Verhalten von Instanzen ist nach außen hin sichtbar und kann ausgelöst werden, indem Operationen, die durch den Typ der Instanz spezifiziert sind, aufgerufen werden.
Eine dritte Betrachtungsweise für Instanzen, die ein fester Bestandteil des Objektmodells ProM ist, ergänzt das ODMG-Objektmodell. Über vage Prädikate und vage Relationen läßt sich die vage Semantik von Instanzen im Hinblick auf Information Retrieval Anfragen (vgl. Abschnitt 2.1) modellieren.

5.8.1 Attribute

In der Spezifikation eines Typs werden Attribute deklariert, denen in den einzelnen Instanzen des Typs Werte zugeordnet werden. Das folgende Beispiel zeigt verschiedene Möglichkeiten,

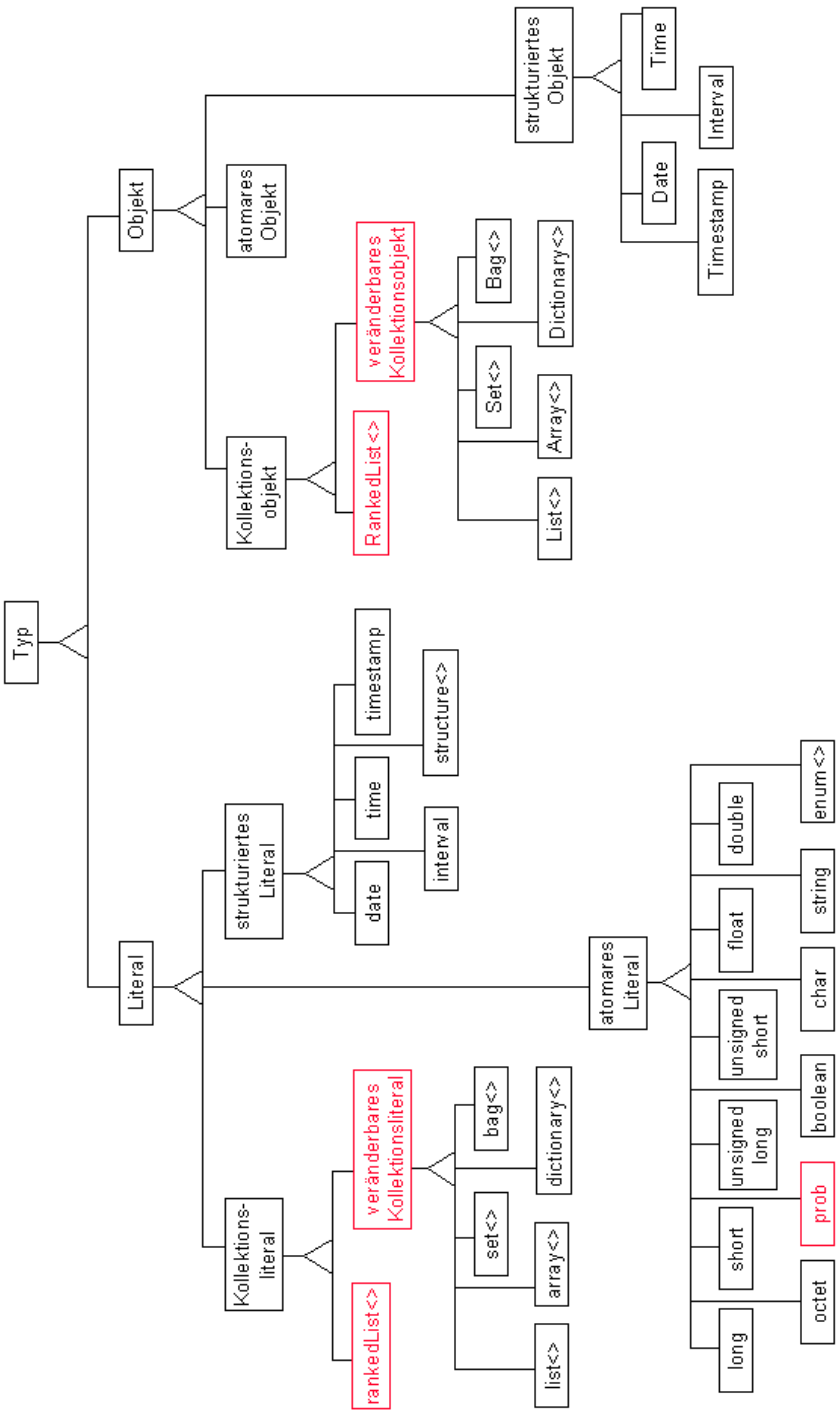


Abbildung 5.14: Die vordefinierte Typhierarchie des Objektmodells ProM

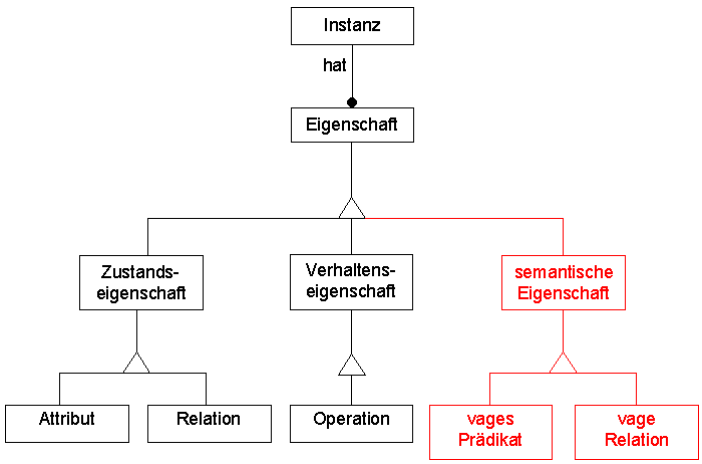


Abbildung 5.15: Instanzeigenschaften

wie Attributwerte spezifiziert werden können:

```
class Mitarbeiter: Person
{
attribute Abteilung abteilung;
attribute set<string> telefonnummern;
attribute unsigned short alter();
};
```

Ein Attribut kann Literalwerte annehmen, wie das Attribut telefonnummern. Es kann aber auch, wie das Attribut abteilung, über den Objektidentifikator ein Objekt referenzieren. Die referentielle Integrität, wie sie für Relationen besteht (vgl. Abschnitt 5.8.2), gibt es hier jedoch nicht. Verweist ein Attribut auf ein Objekt, das später gelöscht wird, so führt dieser Verweis anschließend ins Nichts, wenn er nicht explizit aktualisiert wird.
Attribute können jedoch nicht nur statisch durch Datenstrukturen, sondern auch dynamisch durch Operationen realisiert werden (*abgeleitete Attribute*). Das Attribut alter beispielsweise könnte anhand des Geburtsdatums der jeweiligen Person und anhand des aktuellen Tagesdatums errechnet werden. Der Übergang zwischen der Modellierung des Zustandes einer Instanz auf der einen Seite und ihres Verhaltens auf der anderen Seite ist hier fließend.
Attribute sind keine Modellkomponenten erster Ordnung und besitzen keine eigene Identität. Sie können weder verschachtelt eigene Attribute besitzen noch an Relationen teilhaben.

5.8.2 Relationen

Relationen beschreiben einen weiteren Teil des Zustandes von Instanzen. Im Gegensatz zu Attributen sind sie jedoch nicht auf einen Typ beschränkt, sondern sie repräsentieren die logische Verbindung zwischen zwei Typen.

Als Beispiel diene die Relation verfasst/verfasst_von zwischen Autoren und Publikationen. Der Zustand eines Objektes vom Typ Autor wird unter anderem dadurch bestimmt, welche Publikationen dieser Autor verfaßt hat. Der Zustand eines Objektes vom Typ Publikation ist abhängig davon, welche Autoren an dieser Publikation gearbeitet haben. Da die Zustände auf beiden Seiten offenbar nicht unabhängig voneinander zu betrachten sind, werden sie durch eine Relation modelliert.

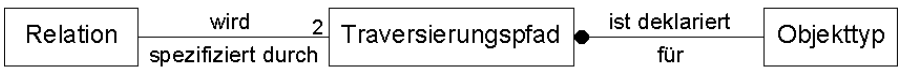


Abbildung 5.16: Die Spezifikation von Relationen

Relationen verbinden Objekte über deren Objektidentifikatoren miteinander (vgl. Abb. 5.16). Es werden lediglich binäre Relationen unterstützt. Relationen, die mehr als zwei Objekttypen kombinieren, sind nicht in ProM enthalten. Wie Attribute sind auch Relationen keine Sprachkomponenten erster Ordnung. Sie sind somit ohne Identität, verfügen über keine eigenen Attribute und können nicht geschachtelt werden.
Die Spezifikation von Relationen erfolgt durch die Angabe von Traversierungspfaden (*traversal paths*) für beide beteiligten Objekttypen. Die Relation verfasst beispielsweise ist in der Klasse des Typs Autor durch folgenden Traversierungspfad angegeben:

```
class Autor {
   ...
   relationship set<Publikation> verfasst
      inverse Publikation::verfasst_von;
   ...
};
```

Diesem Traversierungspfad entspricht ein Gegenstück in der Klasse des Typs Publikation:

```
class Publikation {
   ...
   relationship set<Autor> verfasst_von
      inverse Autor::verfasst;
      ...
};
```

Dadurch daß die Definition einer Relation über die Spezifikationen der beiden beteiligten Typen verteilt ist, kann sie auf beiden Seiten direkt nachvollzogen werden. Das einem Traversierungspfad im Zieltyp zugeordnete Gegenstück wird in der inverse-Klausel verzeichnet. Im obigen Beispiel ist in der inverse-Klausel des Traversierungspfades verfasst auf der Autorenseite Publikation::verfasst_von aufgeführt, da die Traversierungspfade verfasst und verfasst_von gemeinsam die Relation bilden.

Neben der inverse-Klausel und seinem Namen enthält ein Traversierungspfad einen Zieltyp. Im obigen Beispiel sind die Zieltypen beider Traversierungspfade jeweils Mengen. Hierdurch wird modelliert, daß es sich bei der Relation verfasst/verfasst_von um eine n:m-Relation handelt. Relationen der Kardinalität 1:n oder 1:1 sind ebenfalls darstellbar, indem in einem oder beiden Traversierungspfaden ein atomarer Objekttyp als Zieltyp angegeben wird. In der folgenden Spezifikation beispielsweise wird auf diese Art die 1:n-Relation verlegt/erschienen_bei zwischen Verlagen und Büchern definiert:

```
class Verlag {
   ...
   relationship set<Buch> verlegt
      inverse Buch::erschienen_bei;
      ...
};


class Buch {
   ...
   relationship Verlag erschienen_bei
      inverse Verlag::verlegt;
      ...
};
```

Ist ein Objekt über eine Relation der Kardinalität 1:n oder n:m mit mehreren Zielobjekten verbunden, so kann im entsprechenden Traversierungspfad eine Ordnung auf diesen Zielobjekten festgelegt werden, indem für den Zieltyp anstelle einer Menge eine Liste eingesetzt wird. Tabelle 5.1 faßt zusammen, wie die Kardinalitäten von Relationen in den Traversierungspfaden des Objektmodells ProM mit Hilfe von Zieltypen dargestellt werden.
Die referentielle Integrität von Relationen wird durch das System kontrolliert. Wird ein Objekt gelöscht, so werden alle Relationenpaare, an denen das gelöschte Objekt beteiligt war, aus den entsprechenden Relationen entfernt. Relationen unterscheiden sich hierin beispielsweise von Objektattributen (vgl. Abschnitt 5.8.1), für die eine referentielle Integrität in dieser Form nicht gegeben ist.
ProM stellt verschiedene Operationen auf Relationen zur Verfügung. Relationen können erzeugt und zerstört werden. Die mit einem Objekt verbundenen Objekte der Gegenseite können anhand des Traversierungspfades ermittelt werden. Auf Relationen der Kardinalität 1:n oder n:m können außerdem diejenigen Operationen, die für die Kollektionen in den Zieltypen spezifiziert sind, ausgeführt werden.

5.8.3 Operationen

Operationen bestimmen das Verhalten von Instanzen. Sie werden, wie im folgenden Beispiel, über Signaturen spezifiziert.

Kardinalität	Traversierungspfad für Typ A	Traversierungspfad für Typ B
1:1	relationship B a_name inverse B::b_name	relationship A b_name inverse A::a_name
1:n ungeordnet	relationship set<B> a_name inverse B::b_name	relationship A b_name inverse A::a_name
1:n geordnet	relationship list<B> a_name inverse B::b_name	relationship A b_name inverse A::a_name
n:m ungeordnet	relationship set<B> a_name inverse B::b_name	relationship set<A> b_name inverse A::a_name
n:m einseitig geordnet	relationship list<B> a_name inverse B::b_name	relationship set<A> b_name inverse A::a_name
n:m beidseitig geordnet	relationship list<B> a_name inverse B::b_name	relationship list<A> b_name inverse A::a_name

Tabelle 5.1: Darstellung von Kardinalitäten in Traversierungspfaden

```
class Buch {
   ...
   void erstelle_verweis(inout Datei bibtex)
      raises(kapitel_nicht_vorhanden);
   ...
};
```

Die Signatur enthält Angaben über den Namen der Operation, über den Typ ihres Rückgabe- und ihrer Formalparameter sowie über eventuelle Ausnahmefälle, die von der Operation ausgelöst werden können. Eine Operation muß nicht zwingend einen Rückgabeparameter besitzen, wie das obige Beispiel zeigt. Weiterhin sind Seiteneffekte bezüglich der Formalparameter möglich. Die Signatur spezifiziert nicht die Semantik einer Operation. Sie trifft also keine Aussage darüber, wie die Operation ausgeführt wird und was sie bewirkt. [8]
Jede Operation ist genau für die Instanzen eines Typs und seiner Subtypen definiert. Der Namensraum, in dem der Name der Operation eindeutig sein muß, ist somit die Spezifikation dieses Typs. Operationen in unterschiedlichen Typen können durchaus denselben Namen führen (*overloading*). Wird ein solcher mehrfach definierter Operationenname aufgerufen, so muß dynamisch entschieden werden, welche Methode zu diesem Namen gehört und ausgeführt wird (*dispatching*).
Der Subtyp B eines Typs A kann in A spezifizierte Operationen neu definieren, so daß die Definition in A durch die Definition in B überschrieben wird (*overriding*).
Für die Ausführung von Operationen wird zunächst eine sequentielle Verarbeitung angenommen. Jedoch ist auch die Ausnutzung der Performanzvorteile von Mehrprozessorsystemen

[8] Die vorliegende Arbeit übernimmt an dieser Stelle die Empfehlung der ODMG, die Semantik von Operationen durch Kommentare und/oder die Angabe von Prä- und Postkonditionen sowie von Invarianten offenzulegen. Die in ProM neu eingeführten Operationen sind daher ebenfalls kommentiert worden.

zulässig.
Zu jeder Operation kann ein Satz von Ausnahmefällen spezifiziert werden, die diese Operation auslösen kann. Die Ausnahmefälle selbst sind wiederum als Objekte modelliert. Unter dem zugehörigen Wurzeltyp Exception kann eine Subtyphierarchie von Ausnahmetypen aufgebaut werden.
Für jeden Ausnahmetyp A kann ein Konzept zur Ausnahmebehandlung (*exception handler*) definiert werden. Mit diesem Konzept sollten alle Ausnahmeinstanzen von A sowie alle Instanzen der Subtypen von A behandelt werden können. Wird für einen Subtyp eine speziellere Fehlerbehandlung notwendig, so kann das geerbte Konzept entsprechend angepaßt werden.

5.8.4 Erweiterung um vage Prädikate

Ein vages Prädikat überprüft die Instanz i, für die es definiert ist, auf die Erfüllung einer semantischen Eigenschaft (vgl.Abb. 5.17). Da bei semantischen Eigenschaften nicht eindeutig entschieden werden kann, ob diese von der Instanz erfüllt werden (vgl. Abschnitt 2.1), wird anstelle eines Wahrheitswertes ein Wahrscheinlichkeitswert $P(i)$ ermittelt. Dieser drückt dann aus, wie wahrscheinlich es ist, daß die semantische Eigenschaft auf die aufrufende Instanz zutrifft.
Die dynamische Auswertung, die hier zur Belegung einer Objekteigenschaft führt, deutet auf die semantische Ähnlichkeit zwischen vagen Prädikaten und abgeleiteten Attributen hin (vgl. Abschnitt 5.8.1). Doch auch mit Operationen sind vage Prädikate aufgrund ihres Aufbaus vergleichbar.
Bei der Ermittlung der Wahrscheinlichkeit, die ein vages Prädikat zurückliefert, können andere Instanzen beliebigen Typs herangezogen werden, um Werte für etwaige Vergleiche oder zusätzliche Informationen für die Auswertung zu gewinnen. Diese Instanzen können dem vagen Prädikat als Formalparameter übergeben oder auch im Rumpf des Prädikats aufgerufen werden.

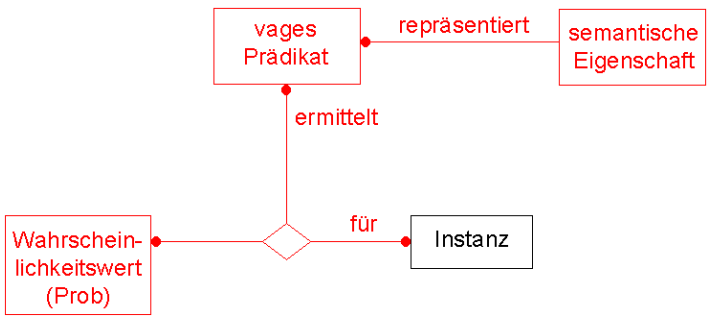


Abbildung 5.17: Die Umsetzung semantischer Eigenschaften durch vage Prädikate

Betrachtet man beispielsweise Objekte vom Typ Publikation, so kann man versuchen, zu

ermitteln, ob eine Publikation durch eine andere vertieft wird. Ein Weg, die Entscheidung hierüber herbeizuführen, wäre es, beide Publikationen zu lesen. Doch auch dann ist es möglich, daß zwei verschiedene Leser zu unterschiedlichen Schlüssen hinsichtlich der Zusammenhänge zwischen den Publikationen kommen. Die Algorithmen eines Informationssystems greifen nur auf eine Teilmenge der Regeln zurück, die ein menschlicher Leser bei der Überprüfung anwendet. Daher ist ein Informationssystem noch weniger in der Lage, mit Sicherheit festzustellen, ob die eine Publikation die andere vertieft. Werden die vorhandenen Regeln jedoch korrekt miteinander verbunden, so kann eine Wahrscheinlichkeit für die Vertiefung der Publikation ermittelt werden. Hierzu wird in der Spezifikation des Typs Publikation ein vages Prädikat vertieft definiert, das als Formalparameter die zweite Publikation erhält:

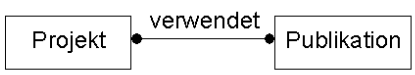
```
class Publikation {
   ...
   vague vertieft(Publikation publikation)
      (raises publikation_nicht_vorhanden);
   ...
};
```

Bei der Ausführung des vagen Prädikates kommen typische Information Retrieval Konzepte [Ste96] zum Einsatz. Beispielsweise könnte ermittelt werden, wie häufig und in welcher lokalen Konzentration die Themenbezeichnung oder sinnverwandte Wörter in dem Buch vorkommen.
Ähnlich wie bei Operationen wird auch bei der Definition vager Prädikate der Typ etwaiger Formalparameter festglegt. Im obigen Beispiel wurde für den Parameter publikation der Typ Publikation gewählt. Ein Typ für den Rückgabeparameter muß hingegen innerhalb des vagen Prädikates nicht spezifiziert werden. Da ausschließlich Wahrscheinlichkeitswerte vom Typ prob zurückgegeben werden, steht der Typ des Rückgabeparameters von vornherein implizit fest.
Vage Prädikate dienen lediglich der Überprüfung semantischer Eigenschaften. Sie sollen jedoch keine Veränderung der Datenbasis vornehmen. Daher werden in ProM Seiteneffekte auf den Formalparametern vager Prädikate von vornherein ausgeschlossen, und die Angabe über solche Seiteneffekte in der Parameterliste entfällt.
Da vage Prädikate auch als Berechnungsvorschriften für Wahrscheinlichkeitswerte interpretiert werden können, muß bei ihrer Ausführung mit Ausnahmefällen gerechnet werden. Diese Ausnahmefälle werden ebenfalls bei der Spezifikation eines vagen Prädikates berücksichtigt. Im obigen Beispiel wird die Ausnahme publikation_nicht_vorhanden ausgelöst, wenn das System den Parameter publikation nicht findet. Ausnahmefälle für vage Prädikate sind in die Subtyphierarchie unter der Wurzel Exception einzuordnen, so daß ihre Behandlung demselben Muster folgt wie bei Operationen.
Für vage Prädikate gilt derselbe Namensraum wie für Operationen. Der Name eines vagen Prädikates muß somit innerhalb eines Typs eindeutig sein. Es ist möglich in verschiedenen Typen den gleichen Namen für vage Prädikate zu verwenden, was beim Aufruf zu einer dynamischen Auswahl des entsprechenden Prädikates führt.

Auch für Daten, die nicht der Volltextsuche als typischer Information Retrieval Domäne entstammen, sind vage Prädikate denkbar [Fuh96a]. So kann man sich für Namen vage Vergleiche vorstellen, die den ähnlichen Klang oder die ähnliche Schreibweise zweier Namen bewerten. Die auf Zahlen definierten Vergleiche können um vage Varianten ergänzt werden. Unter Einbeziehung der zugrundeliegenden gesamten Zahlenmenge beispielsweise läßt sich die Wahrscheinlichkeit für die ungefähre Gleichheit zweier Zahlen berechnen.

5.8.5 Erweiterung um vage Relationen

So wie in den Traversierungspfaden einer Relation durch die Angabe der Zieltypen die Kardinalität dieser Relation modelliert wird, ist es nach Einführung der Relevanzrangfolge als neue Kollektionsform (vgl. Abschnitt 5.5.6.6) außerdem möglich, auf ähnliche Weise vage Relationen zu definieren.
Ist auf den Typen zweier Objekte eine eindeutige Relation[9] (vgl. Abschnitt 5.8.2) spezifiziert, so kann für diese beiden Objekte eine eindeutige Aussage darüber getroffen werden, ob sie an dieser booleschen Relation teilnehmen oder nicht. Doch ebenso wie die Eigenschaften einzelner Objekte kann auch der Zusammenhang zwischen verschiedenen Objekten einer vagen Semantik unterliegen. Dann läßt sich in den meisten Fällen nicht mit Sicherheit, sondern nur mit einer bestimmten Wahrscheinlichkeit annehmen, daß zwei Objekte miteinander in Beziehung stehen. Die Darstellung dieser vagen Semantik erfolgt in ProM durch vage Relationen.



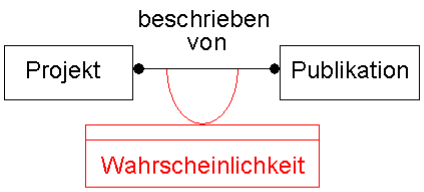

```
class Projekt
{
...
relationship set<Publikation> verwendet
        inverse Publikation::verwendet_in;
...
};

class Publikation
{
...
relationship set<Projekt> verwendet_in
        inverse Projekt::verwendet;
...
};
```

Abbildung 5.18: Die boolesche Relation verwendet/verwendet_in zwischen Projekten und Publikationen

Ein Beispiel soll den Unterschied zwischen booleschen und vagen Relationen verdeutlichen. Betrachtet man die Typen Projekt und Publikation, so sind verschiedene Relationen zwischen

[9] Solche eindeutigen Relationen werden im folgenden oft auch als boolesche Relationen bezeichnet, um sie von vagen Relationen abzugrenzen.

deren Objekten vorstellbar. Beispielsweise werden Publikationen in Projekten als Literatur verwendet. Dabei läßt sich eindeutig feststellen, ob ein Projekt eine Publikation verwendet, indem beispielsweise im System eine boolesche Variable auf **true** gesetzt wird, sobald ein Mitarbeiter des Projektes auf die Publikation zugreift. Dieser eindeutige Zusammenhang zwischen Projekten und Publikationen wird durch die Definition einer booleschen n:m-Relation **verwendet/verwendet_in** repräsentiert (vgl. Abb. 5.18).
Als weitere Verbindung zwischen Projekten und Publikationen soll nun modelliert werden, daß der Inhalt eines Projektes durch eine Publikation näher beschrieben wird. Über einen solchen Zusammenhang zwischen Projekt und Publikation liegt jedoch in den meisten Fällen nur eine vage Annahme vor, für deren Richtigkeit bestenfalls eine Wahrscheinlichkeit ermittelt werden kann. Zur Modellierung dieses vagen Zusammenhangs wird die vage Relation **beschrieben_von/betrifft** definiert (vgl. Abb. 5.19).




```
class Projekt
{
...
relationship rankedList<Publikation>
                beschrieben_von
        inverse Publikation::betrifft
        {rank by Publikation::ist_quelle_fuer};
...
};

class Publikation
{
...
vague ist_quelle_fuer(Projekt p);
relationship rankedList<Projekt> betrifft
        inverse Publikation::beschrieben_von;
...
};
```

Abbildung 5.19: Die vage Relation **beschrieben_von/betrifft** zwischen Projekten und Publikationen

Die Beispielrelation **beschrieben_von/betrifft** wird im weiteren zur Betrachtung des Aufbaus und der formalen Semantik einer vagen Relation verwendet. Zur Berechnung vager Relationen werden vage Prädikate herangezogen. So ist für Objekte des Typs **Publikation** das vage Prädikat **ist_quelle_fuer** spezifiziert. Dieses ermittelt die Wahrscheinlichkeit, daß eine Publikation inhaltlich als Quelle für ein Projekt, das dem vagen Prädikat als Formalparameter übergeben wird, in Frage kommt:

$$P(publikation \mid projekt) := publikation.ist_quelle_fuer(projekt) \tag{5.1}$$

Der Aufruf des vagen Prädikates erfolgt nun für alle Objekte der Extension des Typs **Publi-**

kation mit jeweils demselben Projektparameter. Anschließend wird eine Menge von Paaren gebildet, in denen jeder Publikation die Wahrscheinlichkeit zugeordnet wird, daß diese eine Quelle für das betrachtete Projekt ist:

$$\{(publikation_i, P(publikation_i \mid projekt)) \mid \\ publikation_i \in Extension(Publikation)\} \tag{5.2}$$

Diese Paarmenge läßt sich anhand der Wahrscheinlichkeiten ordnen. Als Resultat ergibt sich eine Relevanzrangfolge mit Elementen vom Typ Publikation. Diese Relevanzrangfolge entspricht genau dem Wert des vagen Traversierungspfades beschrieben_von für das als Parameter verwendete Projektobjekt.

$$\begin{aligned} &projekt.beschrieben_von := \\ &(publikation_{i_1}, P(publikation_{i_1} \mid projekt)), \\ &publikationi_2, P(publikation_{i_2} \mid projekt)), \ldots, \\ &(publikation_{i_n}, P(publikation_{i_n} \mid projekt)) \\ &\text{mit } \forall \;\; 1 \leq j,k \leq n, \quad j,k \in I\!N, \quad j \neq k, \quad n = |Extension(Publikation)| : \\ &j < k \;\; \Rightarrow \;\; P(publikation_{i_j} \mid projekt) \geq P(publikation_{i_k} \mid projekt) \end{aligned} \tag{5.3}$$

Die Verwendung des vagen Prädikates ist_quelle_fuer wird in der Spezifikation der vagen Relation auf der Seite angegeben, auf der es zur Berechnung des Traversierungspfades direkt herangezogen wird (vgl. Abb. 5.19). Im Falle des Beispiels erscheint es somit in der Definition von beschrieben_von auf der Projektseite. Zur Angabe des vagen Prädikates wird die rank-by-Klausel verwendet.
Es bleibt nun noch die entgegengesetzte Traversierung der vagen Relation zu berechnen, also die Relevanzrangfolge, die den Wert von betrifft für eine bestimmte Publikation darstellt. Eine Vorgehensweise könnte es sein, diesen Wert ebenso zu bestimmen wie den Wert von beschrieben_von, indem ein weiteres vages Prädikat verwendet würde, das in diesem Fall jedoch für Projektobjekte spezifiziert wäre und ein Publikationsobjekt als Parameter übergeben bekäme. Dieser Ansatz erweist sich jedoch als semantisch fehlerhaft, da er die Richtigkeit der folgenden Gleichung 5.4 nicht gewährleisten kann:

$$P(publikation \mid projekt) = P(projekt \mid publikation) \tag{5.4}$$

Da diese Forderung jedoch der Realität von vagen Zusammenhängen zwischen verschiedenen Objekten entspricht, wird ein anderer Weg für die Ermittlung der entgegengesetzten Traversierung gewählt. Dabei wird die Relevanzrangfolge betrifft auf der Publikationsseite unter Rückgriff auf die Relevanzrangfolge beschrieben_von auf der Projektseite ermittelt. Sei betrifft für eine bestimmte Publikation zu ermitteln, so werden für alle Projekte Paare gebildet, die den Projekten die Wahrscheinlichkeit zuweisen, die die Publikation in der Relevanzrangfolge beschrieben_von des jeweiligen Projektes besitzt:

$$\{(projekt_i, P(projekt_i \mid publikation)) \mid projekt_i \in Extension(Projekte)\}$$
$$\text{mit } P(projekt \mid publikation) :=$$
$$projekt.beschrieben_von.get_element_prob(publikation) \tag{5.5}$$

Analog zu Zuweisung 5.3 kann die Menge dieser Paare durch Ordnung über die Wahrscheinlichkeiten zu einer Relevanzrangfolge umgewandelt werden, die dann dem Wert des vagen Traversierungspfades betrifft für das Publikationsobjekt entspricht.

$$publikation.betrifft :=$$
$$(projekt_{i_1}, P(projekt_{i_1} \mid publikation)),$$
$$(projekt_{i_2}, P(projekt_{i_2} \mid publikation)), \ldots,$$
$$(projekt_{i_n}, P(projekt_{i_n} \mid publikation))$$
$$\text{mit } \forall \;\; 1 \leq j,k \leq n, \quad j,k \in I\!N, \quad j \neq k, \quad n = |Extension(Projekt)| :$$
$$j < k \;\Rightarrow\; P(projekt_{i_j} \mid publikation) \geq P(projekt_{i_k} \mid publikation) \tag{5.6}$$

Indirekt wird somit auch bei der Ermittlung des entgegengesetzten Traversierungspfades das vage Prädikat ist_quelle_fuer verwendet. Bei der Spezifikation muß es in dem indirekt ermittelten Traversierungspfad der vagen Relation nicht mehr explizit angegeben werden, da es sich automatisch aus der Spezifikation des direkt ermittelten Traversierungspfades ergibt (vgl. Abb. 5.19).
Die aufgeführten Einzelschritte der Bildung von vagen Relationen lassen sich vereinfachend zusammenfassen. Eine vage Relation stellt semantische Zusammenhänge zwischen den Objekten zweier Typen A und B dar, indem sie für alle Objekte des Typs A mit Hilfe eines auf Typ B definierten vagen Prädikates jeweils eine Relevanzrangfolge anlegt, die die Objekte des Typs B ordnet (vgl. Abb. 5.20). Die Wahrscheinlichkeiten dieser Relevanzrangfolgen werden dann genutzt, um auch für alle Objekte des Typs B entsprechende Relevanzrangfolgen aufzustellen.
Die referentielle Integrität vager Relationen wird ebenso durch das System sichergestellt wie die von booleschen Relationen. Wird eine Objektinstanz, die an einer vagen Relation beteiligt ist, gelöscht, so werden alle Paare, die die zu löschende Instanz enthalten, aus dieser Relation entfernt. Des weiteren verändern sich die Relevanzrangfolgen auf beiden Seiten der Relation, da vage Prädikate häufig Angaben über die gesamte betrachtete Objektmenge verwenden. Diese hat sich jedoch durch die Löschung der Objektinstanz verändert, so daß die ursprünglichen Wahrscheinlichkeitswerte in den Relevanzrangfolgen nicht mehr zutreffen.
Vage Relationen werden nicht von vornherein für alle beteiligten Objekte berechnet, sondern erst ermittelt, wenn direkt auf die vage Relation eines Objektes zugegriffen wird. Hierdurch sinkt der Aufwand für die Verwaltung vager Relationen erheblich. Zudem können vage Relationen auf diese Weise automatisch konsistent gehalten werden.

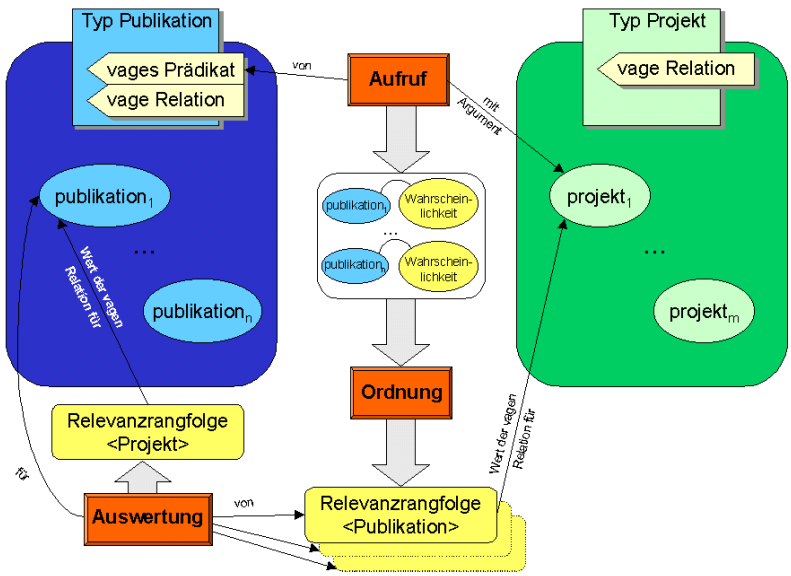
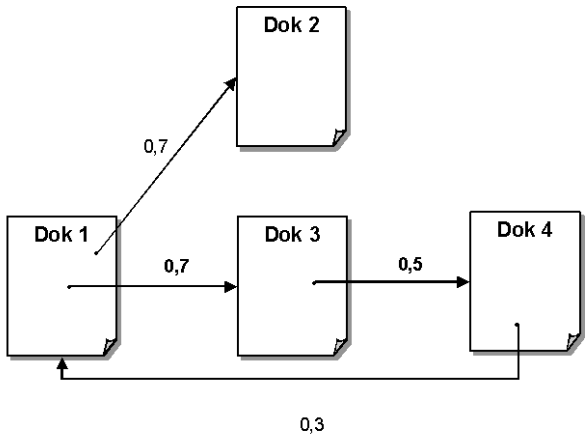


Abbildung 5.20: Die Bildung von vagen Relationen

Die Einführung vager Relationen in das Objektmodell ProM erhöht dessen Mächtigkeit. Während vage Prädikate semantische Aspekte lediglich für das einzelne Objekt modellieren, für das sie spezifiziert sind, vermögen vage Relationen, semantische Abhängigkeiten zwischen Objekten auszudrücken. Die referentielle Integrität sowie die Möglichkeit, vage Relationen in beide Richtungen gleichberechtigt zu traversieren, hat eine Betonung des semantischen Zusammenhangs zwischen den Objekten zur Folge.

Durch diese Hervorhebung semantischer Beziehungen zwischen Objekttypen sind vage Relationen auch für die Modellierung von Hyperlinks (vgl. Abschnitt 2.1.3) geeignet. Beispielsweise kann man sich eine reflexive vage Relation für in einem Hyperlinksystem verbundene Dokumentenknoten vorstellen, die für zwei Knoten angibt, ob der Inhalt des einen durch den Inhalt des anderen ergänzt wird. Durch wiederholtes Traversieren einer solchen vagen Relation und die logische Kombination der jeweiligen Zwischenergebnisse kann für Dokumentenpfade beliebiger Länge berechnet werden, wie wahrscheinlich das Anfangs- und das Enddokument inhaltlich zusammenhängen.

Bei der Implementation derartiger Mehrfachtraversierungen von vagen Relationen ist darauf zu achten, daß keine abwegigen Ergebnisse ermittelt werden können. Als Beispiel hierzu kann man wiederum Dokumentenknoten eines Hyperlinksystems betrachten, die sich jetzt jedoch teilweise auch zyklisch referenzieren (vgl. Abb. 5.21). Nimmt man ausreichend viele Traversierungen vor, die schrittweise immer geringer gewichtet werden, und kombiniert die Zwischenergebnisse logisch durch Konjunktion, so kann man beobachten, daß für einen mehrfach durchlaufenen Dokumentenknoten die Wahrscheinlichkeit, daß er den Inhalt eines bestimmten anderen Dokumentenknotens ergänzt, mit jedem Durchlauf geringer wird.



	P(Dok 3 ergänzt Dok 1)
Traversierung 1	0,7
Traversierung 4	0,7 x 0,5 x 0,3 x 0,7 = 0,00735
...	...

Abbildung 5.21: Unlogische Ergebnisse bei der Mehrfachtraversierung vager Relationen für ein Hyperlinksystem

ProM verhindert solche abwegigen Ergebnisse teilweise, indem es in einer Relevanzrangfolge keine zwei identischen Elemente zuläßt. Für das obige Beispiel ist das allein jedoch nicht ausreichend. Hier muß der Anwendungsprogrammierer darauf achten, daß die Wahrscheinlichkeit für einen durchlaufenen Dokumentenknoten nur beim ersten Durchlauf in der Relevanzrangfolge festgehalten wird. Wo im einzelnen Schwierigkeiten auftreten können, ist stark von den zu modellierenden Daten abhängig. Ein Ausschluß solcher Probleme obliegt daher nicht dem Objektmodell sondern der Implementation.

5.9 Metadaten

Mit Hilfe der bisher beschriebenen Komponenten des Objektmodells ProM können die Anwendungsdaten, auf denen ein integriertes Datenbank- und Information Retrieval System arbeitet, in einem entsprechenden Schema modelliert werden. Für eine korrekte Ausführung von Anwendungsprogrammen benötigt das System über das Anwendungsdatenmodell hinaus jedoch noch weitere Informationen. Diese Informationen beschreiben den allgemeinen Umgang mit den Modellierungskomponenten von ProM und werden Metadaten genannt (vgl. Abb. 5.22).
Metadaten werden genau wie Anwendungsdaten in einer Datenbeschreibungssprache wie ProDL oder ODL spezifiziert. Hierdurch wird für alle wichtigen Modellierungskomponenten

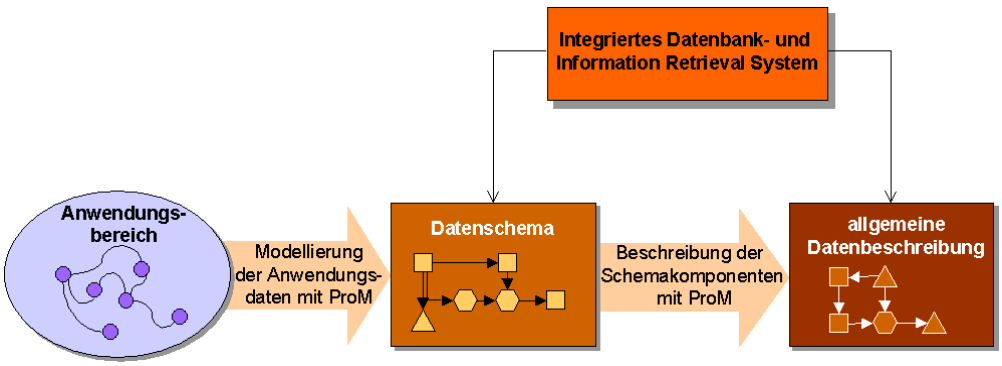


Abbildung 5.22: Die Erstellung und Nutzung von Metadaten

festgelegt, welche Operationen auf ihren Instanzen möglich sind. Die Spezifikationen werden dann in einer allgemeinen Datenbankbeschreibung (*repository*) so gespeichert, daß alle Werkzeug- und Anwendungsprogramme zur Laufzeit auf diese Information genauso zugreifen können wie auch auf die Anwendungsdaten selbst.
Das ODMG-Objektmodell enthält Metadaten in Form verschiedener ODL-Schnittstellen, die die wichtigsten Komponenten dieses Modells spezifizieren. Unter anderem werden hier Typen, Klassen und Schnittstellen sowie Attribute und Relationen beschrieben. Diese Metadaten werden zunächst für das Objektmodell ProM übernommen. Ein solches Vorgehen kann aber offensichtlich nicht ausreichen, um alle Komponenten von ProM durch Metadaten zu beschreiben, da ProM das ODMG-Objektmodell um einige für das Information Retrieval spezifische Komponenten erweitert. Die Erstellung der fehlenden Metadaten für die Erweiterungen in ProM überschreitet jedoch den Rahmen der vorliegenden Arbeit und kann daher als Gegenstand weiterer Forschung betrachtet werden.

5.10 Datenbankfunktionalität

Das ODMG-Objektmodell ist ein Modell für objektorientierte Datenbanksysteme. Daher bietet es Komponenten zur Modellierung von Datenbankfunktionlität an. Da auch die gesamten Daten des integrierten Datenbank- und Information Retrieval Systems in der Datenbank verwaltet werden sollen, werden diese Modellierungskomponenten in das Objektmodell ProM übernommen. Sie betreffen die Definition logischer Datenbanken, die Transaktionsverarbeitung nach dem ACID-Prinzip (vgl. Abschnitt 3.1.2.2) sowie die Verwaltung von Sperren zur Nebenläufigkeitskontrolle (vgl. Abschnitt 3.1.2.3).

6. ProDL - eine Datendefinitionssprache für integrierte Datenbank- und Information Retrieval Systeme

Im Rahmen des ODMG-Standards (vgl. Abschnitt 5.2.1) wurde die Datendefinitionssprache ODL (*Object Definition Language*) entwickelt. Diese ist eine Erweiterung der Sprache IDL (*Interface Definition Language IDL*) der *Common Object Request Broker Architecture CORBA*. Die in diesem Kapitel vorgestellte Definitionssprache ProDL (*Probabilistic Object Definition Language*) ergänzt ODL in Anlehnung an das vorgestellte Objektmodell ProM. Dabei werden Sprachkonstrukte zur Spezifikation von Daten mit vager Semantik eingeführt, so daß auf durch ProDL spezifizierten Daten Information Retrieval Anfragen möglich sind. Die durch den ODMG-Standard für Datendefinitionssprachen festgelegten Prinzipien werden hierbei übernommen (vgl. Abschnitt 6.2).
ProDL ist vornehmlich für die Beschreibung nach außen hin sichtbarer Eigenschaften von Typen und Instanzen vorgesehen. Die entsprechenden Sprachkonstrukte werden in den Abschnitten 6.3 und 6.4 anhand von Beispielen erläutert. Einen vollständigen Überblick über ProDL bietet die Syntaxbeschreibung in Anhang B.
ProDL-Beschreibungen bilden die Grundlage für vage und eindeutige Anfragen an ein integriertes Datenbank- und Information Retrieval System, die in der Sprache ProQL (vgl. Kapitel 7) formuliert werden.

6.1 Vorgehensweise

ProDL erweitert die im Rahmen des ODMG-Standards eingeführte Datendefinitionssprache ODL (*Object Definition Language*) (vgl. Abb. 6.1). ODL wiederum ergänzt die von der OMG entwickelte Sprache IDL. Dabei fügt die ODMG der Sprache IDL jedoch nur diejenigen Konstrukte hinzu, die notwendig sind, um die Konzepte objektorientierter Datenbanken zu realisieren. Möglichst viele Sprachkonstrukte von IDL sollen unverändert erhalten bleiben. Bei der Definition von ProDL wird derselbe Weg verfolgt. ProDL enthält daher den Großteil der Sprachkonstrukte aus ODL und damit auch aus IDL. Hierdurch erhöht sich

die Kompatibilität der drei Sprachen untereinander und damit auch die Portabilität von Anwendungen zwischen integrierten Datenbank- und Information Retrieval Systemen, objektorientierten Datenbankmanagementsystemen und CORBA.

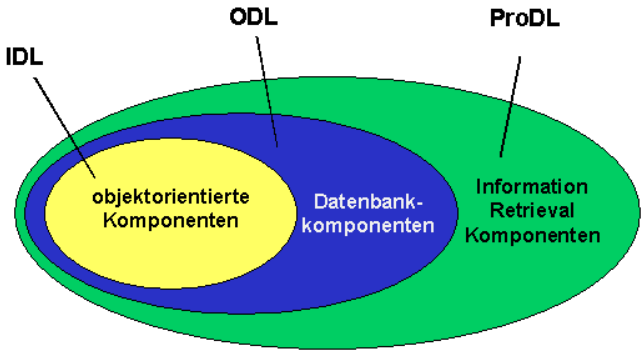


Abbildung 6.1: Die Erweiterung der Definitionssprache IDL zu ODL und ProDL

ODL und IDL enthalten verschiedene Konstrukte, die im Objektmodell ProM (vgl. Abschnitt 5) keine Entsprechung haben, diesem jedoch auch nicht entgegenwirken. Als Beispiele seien die Bildung von Modulen oder die Definition von Kontexten für Operationen genannt. Die Möglichkeit, diese Konstrukte zu verwenden, bleibt in ProDL bestehen. Auf eine eingehende Beschreibung wird jedoch unter Hinweis auf die entsprechenden Standards (siehe [OMG95], [Cat97]) an dieser Stelle verzichtet.

6.2 Die Einordnung von ProDL

Als Datendefinitionssprache dient ProDL der Spezifikation von Datentypen mit Hilfe von Schnittstellen und Klassen (vgl. auch Abschnitt 5.4.1). Alle nach außen hin sichtbaren Aspekte eines Typs können mit Hilfe von Sprachkonstrukten aus ProDL festgelegt werden. Dabei werden jedoch weder die Algorithmen noch die Datenstrukturen determiniert, durch die die Implementation dieser Aspekte erfolgt. ProDL kann somit als Spezifikationssprache, nicht aber als Manipulationssprache bezeichnet werden.
Um in ProDL definierte Daten zu realisieren und zu manipulieren, wird auf Programmiersprachen zurückgegriffen. Über zu entwickelnde Sprachanbindungen werden die in ProDL erstellten Datendefinitionen in Datenstrukturen der entsprechenden Programmiersprache umgewandelt, die dann den Typspezifikationen entsprechend weiterverarbeitet werden können. Dadurch daß eine solche Umwandlung angestrebt wird, kann ProDL als Bindeglied zwischen diesen Programmiersprachen auf der einen und Datenbanksystemen und Information Retrieval Systemen auf der anderen Seite angesehen werden (vgl. Abb. 6.2). Eine erhöhte Portabilität von in ProDL spezifizierten Datenschemata ist das Resultat.
ProDL unterstützt alle Konzepte des Objektmodells ProM. Daher können sowohl Daten

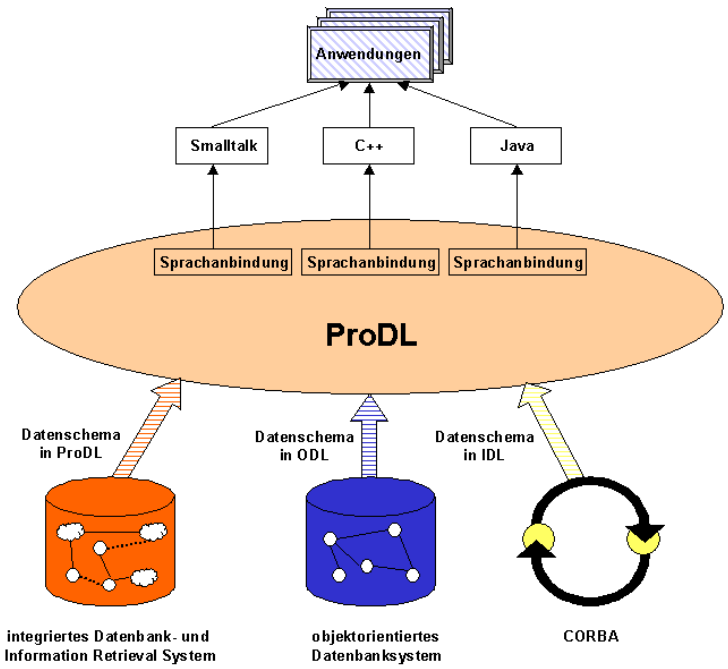


Abbildung 6.2: ProDL als gemeinsame Schnittstelle zwischen Programmiersprachen und Systemen

mit eindeutiger als auch Daten mit vager Semantik spezifiziert werden, so daß auf einem mit ProDL spezifizierten Datenbestand neben booleschen Anfragen, wie sie von herkömmlichen Datenbankmanagementsystemen bekannt sind (vgl. Abschnitt ?), auch vage Anfragen nach probabilistischen Information Retrieval Modellen (vgl. Abschnitt 2.1.1) gestellt werden können.

6.3 Die Spezifikation von Typen in ProDL

Ein Typ wird in ProDL innerhalb einer Schnittstelle oder einer Klasse spezifiziert (vgl. Abschnitt 5.4.1). Diese besitzen die folgende Form[1]:

```
<interface> ::= <interface_dcl>
        | <forward_dcl>
<interface_dcl> ::= <interface_header>
        { [<interface_body>] }
```

[1]Die Syntax der wichtigsten Konstrukte der Sprache ProDL in Backus-Naur-Form wird in den folgenden Abschnitten direkt für die Beschreibung dieser Konstrukte herangezogen. Die vollständige Syntax in BNF befindet sich in Anhang B.

<forward_dcl> ::= **interface** <identifier>
<interface_header> ::= **interface** <identifier>
[<inheritance_spec>]
<inheritance_spec> ::=
:<scoped_name>[, <inheritance_spec>]

<class> ::= <class_header> { <interface_body> }
<class_header> ::= **class** <identifier>
[**extends** <scoped_name>]
[<inheritance_spec>]
[<type_property_list>]
<type_property_list> ::=
([<extent_spec>][<key_spec>])
<extent_spec> ::= **extent** <string>
<key_spec> ::= **key[s]** <key_list>

Im Kopf einer Schnittstelle folgt dem Schlüsselwort interface zunächst der Name des zu spezifizierenden abstrakten Typs. Hieran schließt sich die Festlegung etwaiger Supertypen an, denen der entsprechende Typ in Subtyprelationen (vgl. Abschnitt 5.4.2) untergeordnet ist. Ein Typ kann dabei beliebig viele Supertypen besitzen. Im Rumpf der Schnittstelle werden sämtliche Eigenschaften, die spezifiziert werden sollen, aufgelistet. Diese Liste kann ebenso wie die Angabe der Supertypen weggelassen werden.
Im Kopf einer Klasse kann nach dem Schlüsselwort class und dem Typnamen die extends-Klausel erscheinen. In ihr wird ein einzelner Typ angegeben, dem der spezifizierte Typ in der EXTENDS-Relation (vgl. Abschnitt 5.4.2) untergeordnet ist. Die Angabe mehrerer Typen ist hier nicht möglich. Stattdessen kann eine Klasse ebenso wie eine Schnittstelle mit beliebig vielen weiteren abstrakten Typen in einer Subtyprelation stehen. Diese Supertypen der Klasse werden im Anschluß an die extends-Klausel aufgelistet. Zusätzlich kann der Kopf einer Klasse den Namen der Typextension sowie einen oder mehrere Schlüssel festlegen (vgl. Abschnitt 5.4.3). Der Rumpf einer Klasse spezifiziert alle Eigenschaften für die Instanzen des entsprechenden Typs.
Als Beispiel wird der Typ Publikation spezifiziert. Der Kopf der entsprechenden Klasse hat die folgende Form:

```
class Publikation
   (extent publikationen
    key titel)
   {
   /* Spezifikationsrumpf mit Instanzeigenschaften */
   };
```

Um den Aufbau von Subtyp- und EXTENDS-Relationen innerhalb einer Klasse zu skizzieren, sei zusätzlich der Typ Zugreifbare_Publikation spezifiziert:

```
class Zugreifbare_Publikation extends Publikation : Bibliotheksobjekt
   (extent zugreifbare_publikationen)
   {
   /* Spezifikationsrumpf mit Instanzeigenschaften */
   };
```

Eigene_Publikation enthält Veröffentlichungen die innerhalb des eigenen Fachbereichs vorliegen und eingesehen werden können. Die Instanzen des Typs erben zunächst über die EXTENDS-Relation alle Eigenschaften des Typs Publikation. Des weiteren sind für im Fachbereich vorhandene Schriften zusätzlich abstraktes Verhalten und abstrakte Semantik spezifiziert worden, die der Typ Zugreifbare_Publikation über die Subtyprelation vom Supertyp Bibliotheksobjekt erbt.

6.4 Die Spezifikation von Instanzeigenschaften in ProDL

Im Rumpf sowohl von Klassen als auch von Schnittstellen können Eigenschaften für Typinstanzen definiert werden. Zusätzlich werden hier Ausnahmen, Variable und Konstanten, die für den Typ von Belang sind, festgelegt:

```
<interface_body> ::= <export>
      |<export><interface_body>
<export> ::= <type_dcl>;
      |<const_dcl>;
      |<except_dcl>;
      |<attr_dcl>;
      |<rel_dcl>;
      |<op_dcl>;
      |<vague_pred_dcl>;
```

Instanzeigenschaften sind hierbei Attribute, boolesche und vage Relationen, Operationen sowie vage Prädikate (vgl. Kapitel 5.8).

6.4.1 Attribute

Attribute für Instanzen eines Typs (vgl. Abschnitt 5.8.1) werden im Spezifikationsrumpf wie folgt definiert:

```
<attr_dcl> ::= [readonly] attribute
      <domain_type><attribute_name>
      [<fixed_array_size>]
<domain_type> ::= <simple_type_spec>
      |<struct_type>
      |<enum_type>
```

Die Definition eines Attributs wird durch das Schlüsselwort attribute eingeleitet. Diesem kann das Schlüsselwort readonly vorangestellt werden, falls von außen nur lesend auf dieses Attribut zugegriffen werden soll. Weiterhin wird der Typ des Attributs festgelegt, wobei sowohl Objekttypen als auch Literaltypen beliebiger Form zulässig sind. Die Definition des Attributs wird durch die Angabe eines Attributnamens abgeschlossen.
Für den Beispieltyp Publikation seien hier verschiedene Attribute definiert:

```
class Publikation
   (extent publikationen
    key titel)
   {

   attribute string titel;
   attribute date erscheinungsdatum;
   attribute Text zusammenfassung;
   attribute Text text;
   attribute set<struct Kommentarfeld {Text kommentar;
      Mitarbeiter von_wem}> kommentare;

   /* weitere Instanzeigenschaften */
   };
```

Hier werden sowohl Attribute definiert, die Literale enthalten, wie beispielsweise titel, als auch solche, die Referenzen auf Objekte enthalten, wie beispielsweise zusammenfassung, wo auf ein Textobjekt verwiesen wird.

6.4.2 Boolesche und vage Relationen

Für die Definition vager Relationen (vgl. Abschnitt 5.8.5) wird die ODL-Syntax für boolesche Relationen (vgl. Abschnitt 5.8.2) wie folgt erweitert:

```
<rel_dcl> ::= relationship
        <target_of_path>
        <identifier>
        inverse <inverse_traversal_path>
        [ { rank by <ranking_aspect> } ]
<target_of_path> ::= <identifier>
        |<rel_collection_type>  < <identifier> >
<inverse_traversal_path> ::=
        <identifier>::<identifier>
<rel_collection_type> ::= set | list
        | bag | rankedList
```

Da Relationen zwei Objekttypen miteinander verbinden, wird ihre Spezifikation ebenfalls in zwei Teilen vorgenommen. Dazu wird für jeden der beteiligten Typen je ein Traversierungspfad angegeben. Beide Traversierungspfade gemeinsam bilden die eigentliche Relation.

Die Angabe eines Traversierungspfades erfolgt im Anschluß an das Schlüsselwort relationship. Hier wird zunächst der Zieltyp des Traversierungspfades festgelegt. Wird dabei in genau einem der beiden beteiligten Traversierungspfade eine Kollektion angegeben, so liegt eine Relation der Kardinalität 1:n vor. Sind die Zieltypen beider Traversierungspfade Kollektionen, so handelt es sich um eine Relation der Kardinalität n:m. Vage Relationen setzen sich stets aus zwei Traversierungspfaden mit dem Zieltyp rankedList zusammen. An den Zieltyp schließen sich der Name des Traversierungspfades sowie die inverse-Klausel an. Letztere benennt den Traversierungspfad, mit dem zusammen der definierte Traversierungspfad die eigentliche Relation bildet.

Für vage Relationen wird in einem der Traversierungspfade mit Hilfe der rank-by-Klausel festgelegt, welches vage Prädikat zur Erstellung der Zielrelevanzrangfolgen auf beiden Seiten der Relation genutzt werden soll.

Für den Beispieltyp Publikation wird im Folgenden zunächst der Traversierungspfad verfasst_von innerhalb einer booleschen n:m-Relation definiert[2]. Über diesen Traversierungspfad können die Autoren einer Veröffentlichung ermittelt werden:

```
class Publikation
   (extent publikationen
    key titel)
   {

   attribute string titel;
   attribute date erscheinungsdatum;
   attribute Text zusammenfassung;
   attribute Text text;
   attribute set<struct Kommentarfeld {Text kommentar;
         Mitarbeiter von_wem}> kommentare;

   relationship set<Autor> verfasst_von inverse Autor::verfasst;

   /* weitere Instanzeigenschaften */
   };
```

Dementsprechend ist auf dem Typ Autor der inverse Traversierungspfad verfasst definiert. Er ordnet einem Autoren alle seine Publikationen zu:

[2] In Anhand A werden noch weitere Eigenschaften für den Typ Publikation definiert, um diesen in einen vollständigen Sinnzusammenhang mit dem Gesamtbeispiel zu bringen

```
class Autor
   ...
   {
   ...

   relationship set<Publikation> verfasst
        inverse Publikation::verfasst_von;

   ...
   };
```

Um die Definition von vagen Relationen in ProDL zu illustrieren, wird für den Typ Publikation zusätzlich der Traversierungspfad erweitert definiert, der Teil einer solchen vagen Relation ist. Er soll für Publikationen festhalten, wie wahrscheinlich es ist, daß sie den Inhalt anderer Publikationen aufnehmen und erweitern. Da sich aus dieser Semantik eine Reflexivität der vagen Relation ableitet, wird der inverse Traversierungspfad wird_erweitert_von ebenfalls auf dem Typ Publikation definiert:

```
class Publikation
   (extent publikationen
    key titel)
   {

   attribute string titel;
   attribute date erscheinungsdatum;
   attribute Text zusammenfassung;
   attribute Text text;
   attribute set<struct Kommentarfeld {Text kommentar;
        Mitarbeiter von_wem}> kommentare;
   relationship set<Autor> verfasst_von inverse Autor::verfasst;

   relationship rankedList<Publikation> erweitert
        inverse Publikation::erweitert_von;
   relationship rankedList<Publikation> erweitert_von
        inverse Publikation::erweitert
        {rank by Publikation::vertieft};

/* weitere Instanzeigenschaften, insbesondere das vage Praedikat
   "vertieft" */
};
```

Auf dem Typ Publikation sei das vage Prädikat vertieft definiert. Wie das geschieht, ist in Abschnitt 6.4.4 nachzulesen. vertieft dient hier als direkte Grundlage für die Erstellung der

Zielrelevanzrangfolge des Traversierungspfades erweitert_von und muß daher dort angegeben werden (vgl. Abschnitt 5.8.5). Daneben wird dasselbe Prädikat indirekt auch für die Berechnung der Zielrelevanzrangfolge des Traversierungpfades erweitert verwendet. Dort ist die Angabe des vagen Prädikates nicht mehr erforderlich, da die entsprechende Information aus dem Traversierungspfad erweitert_von abgeleitet werden kann. Grundsätzlich darf ein vages Prädikat, das zur Berechnung einer vagen Relation benutzt wird, nur in demjenigen Traversierungspfad angegeben werden, zu dessen Berechnung es direkt beiträgt, nicht aber in dem dazugehörigen inversen Traversierungspfad, da speziell bei reflexiven vagen Relationen nur hierdurch eine korrekte Auswertungsreihenfolge gewährleistet werden kann.
Insgesamt ist die Einführung der rank-by-Klausel sowie die Zulassung von Relevanzrangfolgen als Zieltypen für die Erweiterung der ODL-Syntax in Hinblick auf die Definition von vagen Relationen ausreichend.

6.4.3 Operationen

Um das Verhalten von Instanzen zu festzulegen, können im Spezifikationsrumpf Operationen unter Benutzung der folgenden Syntax definiert werden:

```
<op_dcl> ::= [<op_attribute>]<op_type_spec>
        <identifier><parameter_dcls>
        [<raises_expr>][<context_expr>]
<op_attribute> ::= oneway
<op_type_spec> ::= <simple_type_spec>
        | void
<parameter_dcls> ::= ( [<param_dcl_list>] )
<param_dcl_list> ::= <param_dcl>
        |<param_dcl>, <param_dcl_list>
<param_dcl> ::= <param_attribute><simple_type_spec>
        <declarator>
<param_attribute> ::= in
        | out
        | inout
<raises_expr> ::= raises   ( <scoped_name_list> )
<scoped_name_list> ::= <scoped_name>
        |<scoped_name>, <scoped_name_list>
<context_expr> ::= context   ( <string_literal_list> )
<string_literal_list> ::= <string_literal>
        |<string_literal>, <string_literal_list>
```

Die Definition von Operationen erfolgt ohne die ausdrückliche Angabe eines Schlüsselwortes, indem der Typ des Rückgabeparameters, der Name der Operation, sowie Namen und Typen etwaiger Formalparameter festgelegt werden. Formalparameter lassen sich weiter einteilen in Wertparameter, die der Operation lediglich als Eingabe dienen, jedoch nicht verändert

werden, in Variablenparameter, die nur verändert, aber nicht gelesen werden, und schließlich in Variablen-Wert-Parameter, die sowohl gelesen als auch verändert werden (vgl. hierzu z.B. auch [Wat90]). Die Festlegung, um welche Art von Formalparameter es sich jeweils handelt, findet in der Parameterliste durch Voranstellung der Schlüsselwörter in, out oder inout statt. Der Operationsspezifikation kann das Schlüsselwort oneway vorangestellt werden, das die Aufrufsemantik der Operation in der Form beeinflußt, daß die aufrufende Instanz nach dem Aufruf keine weiteren Informationen über die Ausführung der Operation erhält [OMG95]. Als weiterer optionaler Zusatz kann der Spezifikation unter dem Schlüsselwort raises eine Liste von Ausnahmen folgen, die durch die Operation ausgelöst werden können.
Die context-Klausel schließlich erlaubt es, der Operation Systemgegebenheiten für das aufrufende Objekt mitzuteilen, damit diese gegebenenfalls bei der Ausführung der Operation berücksichtigt werden können.
Die Beispielspezifikation des Typs Publikation wird im Folgenden um eine Operation erstelle_verweis erweitert, die unter Angabe einer Datei im Bibtexformat einen Eintrag für die aufrufende Publikation erstellt:

```
class Publikation
   (extent publikationen
    key titel)
   {

   attribute string titel;
   attribute date erscheinungsdatum;
   attribute Text zusammenfassung;
   attribute Text text;
   attribute set<struct Kommentarfeld {Text kommentar;
        Mitarbeiter von_wem}> kommentare;
   relationship set<Autor> verfasst_von inverse Autor::verfasst;
   relationship rankedList<Publikation> erweitert
        inverse Publikation::erweitert_von;
   relationship rankedList<Publikation> erweitert_von
        inverse Publikation::erweitert
        {rank by Publikation::vertieft};


   void erstelle_verweis (inout Datei bibtex)
        raises (keine_bibtex_datei);

/* weitere Instanzeigenschaften */
};
```

Die Verwendung des Schlüsslewortes void besagt, daß die Operation keinen Rückgabeparameter besitzt. Der Parameter bibtex ist ein Variablen-Wert-Parameter, was durch die

Voranstellung des Schlüsselwortes inout ausgedrückt wird. Der Aufruf der Operation löst eine Ausnahme aus, falls die angegebene Datei nicht im Bibtexformat vorliegt.

6.4.4 Vage Prädikate

Ein wichtiges Konzept, um das das ODMG-Objektmodell durch das Objektmodell ProM ergänzt wird, sind vage Prädikate (vgl. Abschnitt 5.8.4). Der Sprache ProDL werden an dieser Stelle die entsprechenden Sprachkonstrukte zur Spezifikation vager Prädikate hinzugefügt:

<vague_pred_dcl> ::= **vague** <identifier><in_param_dcl_list>
[<raises_expr>]
<in_param_dcl_list> ::= <in_param_dcl>
| <in_param_dcl>, <in_param_dcl_list>
<in_param_dcl> ::= <simple_type_spec><declarator>

Das Schlüsselwort vague leitet die Definition eines vagen Prädikates ein. Ihm folgt der Name des vagen Prädikates sowie eine Liste mit Parameter. Die Parameter eines vagen Prädikates sind stets Wertparameter (in). Damit entfällt die Angabe über die Art der Parameter in der Parameterliste von vagen Prädikaten. Die Schlüsselwörter in, out und inout, die für die Spezifikation von Operationen verwendet werden, sin hier also nicht notwendig.
Die Angabe eines Rückgabeparameters, die für die Spezifikation von Operationen vorgeschrieben ist, entfällt bei der Spezifiaktion vager Prädikate ebenfalls, da diese stets Wahrscheinlichkeitswerte, also Werte vom Typ prob, zurückgeben.
Trotz dieser Einschränkungen handelt es sich bei vagen Prädikaten semantisch um eine spezielle Form von Operationen. Sie sind also Algorithmen zur Bestimmung von Wahrscheinlichkeitswerten. In dieser Funktion können sie Ausnahmen auslösen. Die Ausnahmen, die ein vages Prädikat auslösen kann, werden in der raises-Klausel spezifiziert.
Für den Beispieltyp Publikation sei nun das vage Prädikat vertieft spezifiziert, auf das bereits für die Definition der vagen Relation erweitert/erweitert_von (vgl. Abschnitt 6.4.2) zurückgegriffen wird:

```
class Publikation
   (extent publikationen
    key titel)
   {

   attribute string titel;
   attribute date erscheinungsdatum;
   attribute Text zusammenfassung;
   attribute Text text;
   attribute set<struct Kommentarfeld {Text kommentar;
        Mitarbeiter von_wem}> kommentare;
```

```
relationship set<Autor> verfasst_von inverse Autor::verfasst;
relationship rankedList<Publikation> erweitert
     inverse Publikation::erweitert_von;
relationship rankedList<Publikation> erweitert_von
     inverse Publikation::erweitert
     {rank by Publikation::vertieft};
void erstelle_verweis (inout Datei bibtex)
     raises (keine_bibtex_datei);

vague vertieft (Publikation publikation)
     raises (publikation_nicht_vorhanden);

};
```

Das vage Prädikat vertieft erhält als Wertparameter ein weiteres Objekt vom Typ Publikation. Falls das übergebene Publikationsobjekt im Datenbestand nicht vorhanden ist, wird eine Ausnahme ausgelöst.

7. ProQL - eine Anfragesprache für integrierte Datenbank- und Information Retrieval Systeme

Das Objektmodell ProM bietet die notwendigen Komponenten zur Modellierung von Daten sowohl mit eindeutiger als auch mit vager Semantik und füllt damit die konzeptuelle Ebene eines integrierten Datenbank- und Information Retrieval Systems aus. Auf der externen Ebene sollen die modellierten Daten nun dem Anwender zugänglich gemacht werden. Hierfür wird die Anfragesprache ProQL definiert. ProQL baut auf der Anfragesprache OQL des ODMG-Standards auf. Durch syntaktische und semantische Erweiterung von OQL können in ProQL vage und eindeutige Anfragen formuliert werden.

7.1 Vorgehensweise

Das Objektmodell ProM basiert auf dem ODMG-Objektmodell. Für Anfragen auf Daten, die unter Benutzung des ODMG-Objektmodells beschrieben worden sind, definiert der ODMG-Standard die Anfragesprache OQL. In der Anfragesprache ProQL sollen Anfragen auf Daten, die anhand von ProM modelliert worden sind, formuliert werden. Bei der Entwicklung von ProQL können zunächst alle Sprachkonstrukte aus OQL übernommen werden, da sie dem Modell ProM ebenso entsprechen wie dem ODMG-Objektmodell (vgl. Abb. 7.1). Zusätzlich muß ProQL um syntaktische und semantische Konzepte ergänzt werden, die die Nutzung der im Modell ProM eingeführten vagen Prädikate und vagen Relationen sowie der dazugehörigen Datentypen rankedList und prob ermöglichen.
Im Folgenden werden zunächst einige Merkmale von OQL beschrieben. Daraufhin wird erläutert, durch welche Erweiterungen von OQL die probabilistische Anfragesprache ProQL entsteht. Als wichtigste Erweiterung werden vage select-from-where-Ausdrücke im Detail beschrieben. Das Kapitel endet mit einer formalen Beschreibung der Sprache ProQL.

7.2 Die wichtigsten Merkmale der Anfragesprache OQL

Die Anfragesprache OQL ist Teil des ODMG-Standards. Da sie auf dem ODMG-Objektmodell aufbaut, eignet sie sich als Grundlage für die vage Anfragesprache ProQL, die auf dem Ob-

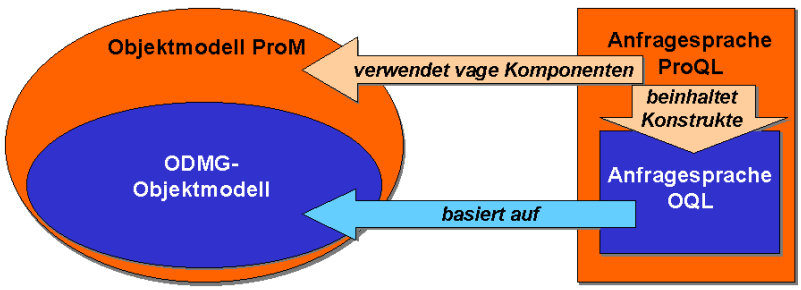


Abbildung 7.1: Das Verhältnis zwischen dem ODMG-Standard, dem Objektmodell ProM und der Anfragesprache ProQL

jektmodell ProM basiert. Syntaktisch weist OQL starke Ähnlichkeiten zu SQL 92 [CO92] auf. So sind beispielsweise deklarative select-from-where-Anfragen einer der Eckpfeiler von OQL. Neben den von SQL 92 bekannten Aspekten verfügt OQL jedoch über einige weitere nennenswerte Eigenschaften.
Die objektorientierten Konzepte des ODMG-Objektmodells setzen sich innerhalb von OQL fort. So lassen sich in OQL komplexe Objekte mit eigener Objektidentität erzeugen. Über Pfadausdrücke kann in beliebiger Tiefe durch die Objekte navigiert und auf ihre Eigenschaften zugegriffen werden. Hierbei ist insbesondere der Aufruf von Operationen zugelassen, wodurch in OQL auch auf Funktionen zurückgegriffen werden kann, die über die Möglichkeiten der Anfragesprache an sich hinausführen.
Auch die Aktualisierung von Objekten vollzieht sich in OQL durch Operationsaufrufe, so daß explizite Sprachkonstrukte zur Aktualisierung von Daten ihre Bedeutung verlieren und daher nicht in der Sprache enthalten sind.
Während SQL 92 ein starkes Gewicht auf Mengen (Relationen) als Kollektionsform legt, sind in OQL die weiteren durch das ODMG-Objektmodell definierten Kollektionsformen in gleichem Umfang nutzbar. Dabei realisiert OQL das Prinzip des parametrischen Polymorphismus, so daß jede Kollektionsform Elemente eines beliebigen Typs zusammenfassen kann.
Alle Ausdrücke in OQL unterliegen dem Typsystem der Sprache. Sie können daher nur mit Operanden verwendet werden, die zu der jeweiligen Operationssignatur kompatibel sind. Das ermittelte Ergebnis besitzt ebenfalls einen wohldefinierten Typ. Das Typsystem stellt die einzige Grenze für die Komposition von OQL-Ausdrücken dar. Die Ausdrücke der Sprache können somit beliebig kombiniert werden, falls die dazugehörigen Typen zueinander passen.

7.3 Die probabilistische Erweiterung von OQL zu ProQL

Die probabilistische Anfragesprache ProQL entsteht zum einen durch Erweiterung der Grammatik der Anfragesprache OQL um nur wenige syntaktische Konstrukte. Zum anderen wird durch Rückgriff auf die im Objektmodell ProM eingeführten Typen für Wahrscheinlichkeits-

werte und Relevanzrangfolgen sowie auf die dort als neue Objekteigenschaften vorgestellten vagen Prädikate und vagen Relationen die Funktionalität der Sprache OQL erweitert, ohne daß sich dieses syntaktisch auf die Sprache ProQL auswirkt. Diese zweite Vorgehensweise wird insbesondere bei der Auswertung von select-from-where-Ausdrücken verfolgt.

7.3.1 Syntaktische Ergänzungen

Der Großteil der Funktionalität, die notwendig ist, um aus OQL eine Anfragesprache zu machen, die auch vage Anfragen im Sinne des Information Retrieval akzeptiert, erschließt sich aus den Erweiterungen, die das ODMG-Objektmodell durch das Objektmodell ProM erfährt (vgl. Kapitel 5).
Andererseits besitzen die neu eingeführten Modellkomponenten jedoch zusätzliche Eigenschaften, für die innerhalb der Anfragesprache ProQL direkte Verwendungs- und Modifizierungsmöglichkeiten geschaffen werden sollen. Dieses geschieht durch explizite Ergänzung der OQL-Syntax um die entsprechenden Sprachkonstrukte.
Die Modellierungsmächtigkeit von Relevanzrangfolgen begründet sich hauptsächlich darauf, daß dort jedem Element eine Wahrscheinlichkeit zugewiesen ist (vgl. Abschnitt 5.5.6.6). Innerhalb der Anfragesprache ProQL kann so eine Wahrscheinlichkeit verwendet werden, um auf die Elemente zuzugreifen, denen sie zugeordnet ist. Da ähnliche Konzepte in OQL bisher nicht bekannt sind, erweitert ProQL an dieser Stelle die Syntax von OQL um den Ausdruck r with p, der genau die Elemente der Relevanzrangfolge r ermittelt, die dort die Wahrscheinlichkeit p tragen.
Relevanzrangfolgen, die sich aus einer vagen Anfrage ergeben, beinhalten zunächst ausnahmslos alle angefragten Elemente mit Wahrscheinlichkeitswerten von 1.0 bis 0.0. Solche Ergebnismengen sind im allgemeinen jedoch zu mächtig, um für den Anfragenden von Nutzen zu sein. Daher werden in der ProQL-Syntax für select-from-where-Anfragen zwei Klauseln ergänzt, die es ermöglichen, die Ergebnismengen vager Anfragen auf unterschiedliche Weise einzuschränken (vgl. Abschnitte 7.3.3 und 7.4.9).
Durch die number-Klausel kann die Ergebnisrangfolge auf eine bestimmte Anzahl von Elementen eingeschränkt werden. Falls die Verwendung der number-Klausel das Ergebnis so begrenzt, daß einige Elemente einer mehrfach zugeordneten Wahrscheinlichkeit enthalten wären und einige nicht, wird die Anzahl der erlaubten Elemente so erhöht, daß alle Elemente dieser Wahrscheinlichkeit in der Ergebnisrangfolge erhalten bleiben (vgl. Abschnitt 5.5.6.6).
Die zweite Klausel zur Einschränkung der Ergebnisrangfolgen vager select-from-where-Anfragen ist die lowest-prob-Klausel, die es erlaubt eine Mindestwahrscheinlichkeit festzusetzen, die ein Element haben muß, um in die Ergebnisrangfolge einer vagen Anfrage aufgenommen zu werden.
Werden innerhalb einer Anfrage number und lowest prob verwendet, so ist die kleinere der beiden sich ergebenden Relevanzrangfolgen das endgültige Ergebnis.
Des weiteren bietet ProQL dem Anfragenden die Möglichkeit, Ergebnisse vager Anfragen in der gewohnten Weise in eindeutigen Anfragen weiterzuverarbeiten, indem er Relevanzrangfolgen, die aus vagen Anfragen resultieren, in ungeordnete Mengen umwandeln kann (vgl.

Abschnitt 7.4.12). Dieses geschieht entweder mit Hilfe des Ausdrucks rankedlisttoset(r), der alle Elemente der Relevanzrangfolge r in einer Menge zusammenfaßt, oder mit Hilfe des Ausdrucks rankedlisttoset(r,p), der alle Elemente, denen in der Relevanzrangfolge r mindestens die Wahrscheinlichkeit p zugeordnet ist, in einer Menge zusammenfaßt.

7.3.2 Semantische Ergänzungen

Das Objektmodell ProM erweitert das ODMG-Objektmodell auf zwei Ebenen. Zum einen definiert es mit Relevanzrangfolgen und Wahrscheinlichkeitswerten zwei neue Datentypen, deren Semantik und Beziehungen zueinander über das Typsystem der Sprache ProQL auf deren Ausdrücke Einfluß nehmen. Zum anderen werden in ProM mit vagen Prädikaten und vagen Relationen zwei neue Objekteigenschaften definiert, die ebenfalls in der Sprache ProQL berücksichtigt werden müssen.
Wahrscheinlichkeitswerte sind atomare Literale, die in ihrer Semantik die Semantik des Datentyps boolean erweitern (vgl Abschnitt 5.6.1). Ein Wert vom Typ boolean beschreibt, ob eine Bedingung wahr oder falsch ist. Ein Wahrscheinlichkeitswert vom Typ prob dagegen beschreibt die Wahrscheinlichkeit dafür, daß eine Bedingung gilt, und stellt somit eine Verallgemeinerung eines Wertes vom Typ boolean dar.
Aus der semantischen Ähnlichkeit von Wahrscheinlichkeits- und Wahrheitswerten läßt sich ableiten, daß nahezu überall, wo Wahrheitswerte verwendet werden, auch Wahrscheinlichkeitswerte einsetzbar sind. Insbesondere lassen sich Wahrscheinlichkeits- und Wahrheitswerte logisch kombinieren, indem die Wahrheitswerte zunächst implizit in die Wahrscheinlichkeiten 1.0 oder 0.0 umgewandelt werden, bevor sie mit weiteren Wahrscheinlichkeitswerten kombiniert werden.

Beispiel:

```
0.9 or 0.3 or false = 0.9 or 0.3 or 0.0 =
1.0 - (1.0 - 0.9) * (1.0 - 0.3) * (1.0 - 0.0) = 0.93
```

Im where-Teil von select-from-where-Ausdrücken werden Bedingungen formuliert, die in ProQL sowohl Wahrheits- als auch Wahrscheinlichkeitswerte zurückliefern können. Ergibt die Berechnung der where-Klausel einer select-from-where-Anfrage einen Wahrscheinlichkeitswert, so wird diese Anfrage als vage betrachtet (vgl. Abschnitt 7.3.3).
In Relevanzrangfolgen wird jedem Element eine Wahrscheinlichkeit zugeordnet. Auch diese Wahrscheinlichkeitswerte beschreiben Bedingungen. Sie drücken für die Elemente, denen sie zugewiesen sind aus, wie wahrscheinlich es ist, daß das jeweilige Element überhaupt zu der Kollektion gehört, die durch die Relevanzrangfolge repräsentiert ist (vgl. Abschnitt 5.5.6). Diese vage Zugehörigkeit schlägt sich in der Mehrzahl der für Relevanzrangfolgen definierten Operationen nieder. Zunächst ergibt der Ausdruck e in r nicht wie bei anderen Kollektionsformen einen Wahrheitswert, sondern einen Wahrscheinlichkeitswert (vgl. Abschnitt 7.4.7). Dieser Wahrscheinlichkeitswert entspricht genau der Wahrscheinlichkeit die dem Element e in der Relevanzrangfolge zugeordnet ist, oder er nimmt den Wert 0.0 an, falls das Element nicht in der Relevanzrangfolge enthalten ist.

Sei beispielsweise die folgende Relevanzrangfolge r gegeben

Position	Element	Wahrscheinlichkeit
0	e_1	0,89
1	e_2	0,82
	e_3	0,82
2	e_4	0,51

so gilt:

e_3 in r = 0.82 und
e_5 in r = 0.0.

Auch bei Quantifizierungen bezüglich einer Relevanzrangfolge wirkt sich die vage Zugehörigkeit der Elemente aus, unahängig davon, ob die Quantifizierungsbedingung einen Wahrheits- oder Wahrscheinlichkeitswert zurückliefert (vgl. Abschnitt 7.4.7). Zunächst wird ermittelt, ob bzw. mit welcher Wahrscheinlichkeit die Elemente der Relevanzrangfolge jeweils die Quantifizierungsbedingung erfüllen. Da sich aber jedes Element überhaupt nur mit einer bestimmten Wahrscheinlichkeit in der Relevanzrangfolge befindet, werden anschließend die sich aus der Quantifizierungsbedingung ergebenden Wahrheits- bzw. Wahrscheinlichkeitswerte mit den Wahrscheinlichkeiten, die den Elementen in der Relevanzrangfolge zugeordnet sind, durch den Operator and verknüpft. Es wird somit für jedes Element die Wahrscheinlichkeit berechnet, daß es die Quantifizierungsbedingung erfüllt *und* sich in der Relevanzrangfolge befindet. Diese Wahrscheinlichkeiten werden für die universelle Quantifizierung nochmals mit and und für die existenzielle Quantifizierung mit or verknüpft. Hierdurch läßt sich die Wahrscheinlichkeit berechnen, daß sich alle bzw. mindestens ein Element der Relevanzrangfolge in derselben befinden und die Quantifizierungsbedingung erfüllen. Sei beispielsweise auf dem Elementtyp der oben angegebenen Relevanzrangfolge r das vage Prädikat vp mit den folgenden Werten für die Elemente von r definiert:

e_1.vp = 0.45
e_2.vp = 0.17
e_3.vp = 0.83
e_4.vp = 0.29

So gilt ergibt sich für die universelle Quantifizierung

for all e in r: e.vp =
(e_1.vp and 0.89) and (e_2.vp and 0.82) and (e_3.vp and 0.82) and (e_4.vp and 0.51) $\approx$
0.00561

und für die Existenzquantifizierung

exists e in r: e.vp =
(e_1.vp and 0.89) or (e_2.vp and 0.82) or (e_3.vp and 0.82) or (e_4.vp and 0.51) ≈
0.85958

Da Relevanzrangfolgen Kollektionen sind, können sie innerhalb des from-Teils einer select-from-where-Anfrage verwendet werden (vgl. Abschnitte 7.3.3 und 7.4.9). Auch dann wird die Anfrage als vage betrachtet. Falls im from-Teil dieser Anfrage noch weitere Kollektionen deklariert sind, so werden diese mit der Relevanzrangfolge zu einer einzigen übergreifenden Kollektion verschmolzen. Da die Elemente der beteiligten Relevanzrangfolge nur mit einer bestimmten Wahrscheinlichkeit zu dieser Relevanzrangfolge und damit auch zu der übergeordneten Kollektion gehören, muß auch diese übergeordnete Kollektion eine Relevanzrangfolge sein.
Die Wahrscheinlichkeiten in Relevanzrangfolgen legen eine Ordnung auf den Elementen fest. Da jedoch verschiedenen Elementen dieselbe Wahrscheinlichkeit zugeordnet sein kann, ist es möglich, daß eine Position innerhalb der Ordnung durch mehrere Elemente belegt sein kann. Hierin unterscheiden sich Relevanzrangfolgen von allen anderen geordneten Kollektionen des Objektmodells ProM. Diese Mehrfachbelegung wirkt sich insbesondere auf den Zugriff auf Elemente anhand ihrer Position aus. So ergeben die Ausdrücke r[i], first(r) bzw. last(r) stets die Menge, die alle Elemente der Relevanzrangfolge r an der Position i, bzw. an der ersten oder an der letzten Position enthält (vgl. Abschnitte 7.4.10 und 7.4.10). Für die oben angegebene Relevanzrangfolge r ergibt sich dadurch beispielsweise

r[1] = { e_1, e_2 }

Eine weitere semantische Eigenschaft von Relevanzrangfolgen ist ihre Unveränderbarkeit. Relevanzrangfolgen werden im Objektmodell als unveränderbar definiert, da sie sich zumeist aus vagen Anfragen auf der Grundlage des gesamten aktuellen Suchraums ergeben und somit nur eine Momentaufnahme der vagen Semantik des Datenbestandes darstellen. Hieraus folgt für die Anfragesprache ProQL, daß Ausdrücke zur expliziten Konstruktion oder Modifikation nicht zur Verfügung gestellt werden.
Neben den Typen für Wahrscheinlichkeitswerte und Relevanzrangfolgen führt das Modell ProM vage Prädikate und vage Relationen als Eigenschaften von Objekten ein. Auf diese Eigenschaften muß inerhalb der Anfragesprache ProQL ebenfalls zurückegegriffen werden können. Da erhebliche semantische Ähnlichkeiten zwischen vagen Relationen und eindeutigen Relationen sowie zwischen vagen Prädikaten und Operationen bestehen, können die in der Sprache OQL bereits vorhandenen Zugriffskonstrukte für eindeutige Relationen und Operationen ebenfalls für vage Relationen und vage Prädikate verwendet werden. Konkret bedeutet dieses, daß sowohl auf vage Relationen als auch auf vage Prädikate mit Hilfe der Punkt- oder auch mit Hilfe der Pfeilnotation zugegriffen werden kann. Beim Aufruf eines vagen Prädikats ist unter Umständen zusätzlich die Angabe einer Parameterliste erforder-

lich. Betrachtet man beispielsweise das Objekt o, auf dem das vage Prädikat vp mit einem Parameter vom Typ P definiert ist, so kann vp für o alternativ durch einen der beiden Ausdrücke

o.vp(p) oder
o -> vp(p)

aufgerufen werden, falls p ein Wert vom Typ P ist.

7.3.3 Die Semantik vager select-from-where-Ausdrücke in ProQL

Die Einführung von Relevanzrangfolgen, Wahrscheinlichkeiten, vagen Prädikaten und vagen Relationen wirkt sich insbesondere auf die Möglichkeiten in den select-from-where-Anfragen der Anfragesprache ProQL aus (vgl. Abschnitt 7.4.9), da dort an beliebiger Stelle auf die neuen Komponenten zurückgegriffen werden kann. Daher soll in diesem Abschnitt detailliert darauf eingegangen werden, was eine vage select-from-where-Anfrage ausmacht und wie vage select-from-where-Anfragen ausgewertet werden.
Eine select-from-where-Anfrage wird dann als vage bezeichnet,

- → wenn entweder in ihrem from-Teil mindestens eine Relevanzrangfolge deklariert ist,
- → wenn ihre where-Klausel einen Wahrscheinlichkeitswert zurückgibt oder
- → wenn sie die Schlüsselworte number oder lowest prob (siehe unten) enthält.

Das Ergebnis einer vagen Anfrage ist in jedem Fall eine Relevanzrangfolge, die zunächst ein Gesamtergebnis mit allen überhaupt möglichen Elementen darstellt. Hierin sind jedoch auch Elemente enthalten, denen nur eine geringe Wahrscheinlichkeit zugeordnet ist. Um solche Elemente aus dem Ergebnis zu entfernen, können die number- und die lowest-prob-Klausel verwendet werden (vgl. Abschnitt 7.3.1). Theoretisch ist die Verwendung der beiden Klauseln auch in Anfragen möglich, die keinen vagen where-Teil haben und in deren from-Teil keine Relevanzrangfolge deklariert wird. So formulierte Anfragen ergeben ebenfalls Relevanzrangfolgen und werden daher der Vollständigkeit halber auch als vage bezeichnet, obwohl ihnen eindeutige Anfragen, die mit den Sprachkonstrukten aus OQL formuliert werden können, vorzuziehen sind.
Der Verlauf der Auswertung einer vagen Anfrage wird nun schrittweise anhand des folgenden Ausdrucks nachvollzogen:

```
select p.name
       from projekte p
            p.beschrieben_von q
       where q.handelt_von("Datenbanken")
```

Diese Anfrage ergibt eine Relevanzrangfolge mit den Namen von Projekten, deren Projektbibliothek Publikationen enthält, die das Thema "Datenbanken" behandeln.

Als Datenbasis seien die folgenden Projekte und Publikationen angenommen[1]:

projekte	
name	...
"Digitale Bibliotheken"	...
"Objektorientierte Datenbanken"	...
"Internet-AG"	...

publikationen	
titel	...
"Datenbankhandbuch"	...
"Objektorientierte Konzepte"	...
"Textretrieval"	...
"HTML-Einführung"	...
"Zugriffskontrolle"	...

Auf Publikationen ist das vage Prädikat handelt_von spezifiziert, dessen Auswertung bei Verwendung des Aktualparameters "Datenbanken" für die einzelnen Publikationsobjekte die folgenden Wahrscheinlichkeitswerte ergibt:

("Datenbankhandbuch", ...).handelt_von("Datenbanken") = 0.9

("Objektorientierte Konzepte", ...).handelt_von("Datenbanken") = 0.4

("Textretrieval", ...).handelt_von("Datenbanken") = 0.2

("HTML-Einführung", ...).handelt_von("Datenbanken") = 0.2

("Zugriffskontrolle", ...).handelt_von("Datenbanken") = 0.7

Des weiteren besteht zwischen Projekten und Publikationen die vage Relation beschrieben_von, die auf der Seite der Projekte wie folgt belegt ist:

[1]Zur übersichtlichen Darstellung werden hier Tabellen verwendet. Diese Tabellen repräsentieren Kollektionen mit Objektelementen und sind damit semantisch von Tabellen zu unterscheiden, wie sie in relationalen Datenbanksystemen gebräuchlich sind.

projekte				
name	...	**beschrieben_von**		
		Pos.	**publikation**	Prob.
"Digitale Bibliotheken"	...	0	"Objektorientierte Konzepte"	0.8
		1	"Textretrieval"	0.7
		2	"Datenbankhandbuch"	0.6
		3	"HTML-Einführung"	0.3
		4	"Zugriffskontrolle"	0.3
"Objektorientierte Datenbanken"	...	0	"Objektorientierte Konzepte"	0.9
		1	"Datenbankhandbuch"	0.9
		2	"Zugriffskontrolle"	0.6
		3	"Textretrieval"	0.1
		4	"HTML-Einführung"	0.1
"Internet-AG"	...	0	"HTML-Einführung"	0.9
		1	"Objektorientierte Konzepte"	0.7
		2	"Textretrieval"	0.7
		3	"Datenbankhandbuch"	0.6
		4	"Zugriffskontrolle"	0.5

Anhand dieser Informationen läßt sich die select-from-where-Anfrage auswerten. Dazu wird das durch die where-Klausel gegebene vage Prädikat

q.handelt_von("Datenbanken")

auf die Relevanzrangfolgen des from-Teils

p.beschrieben_von q

angewendet. Semantisch betrachtet soll hier ermittelt werden, wie wahrscheinlich das vage Prädikat auf die einzelnen Elemente der Relevanzrangfolgen zutrifft. Dabei muß beachtet werden, daß jedes Element nur mit einer bestimmten Wahrscheinlichkeit überhaupt in der jeweiligen Relevanzrangfolge existiert. Deshalb muß die bei der Anwendung des vagen Prädikats für ein Element entstehende Wahrscheinlichkeit jeweils mit der Wahrscheinlichkeit des Elements in der Relevanzrangfolge durch eine boolesche and-Verknüpfung verrechnet werden. Für die innerhalb der Beispielobjekte vom Typ Projekt als Belegung der vagen Relation beschrieben_von bestehenden Relevanzrangfolgen ergeben sich dadurch die folgenden Veränderungen:

projekte *(nach Anwendung der where-Klausel)*				
name	...	**beschrieben_von**		
		Pos.	**publikation**	Prob.
"Digitale Bibliotheken"	...	0	"Datenbankhandbuch"	0.6 and 0.9 = 0.54
		1	"Objektorientierte Konzepte"	0.8 and 0.4 = 0.32
		2	"Zugriffskontrolle"	0.3 and 0.7 = 0.21
		3	"Textretrieval"	0.7 and 0.2 = 0.14
		4	"HTML-Einführung"	0.3 and 0.2 = 0.06
"Objektorientierte Datenbanken"	...	0	"Datenbankhandbuch"	0.9 and 0.9 = 0.81
		1	"Objektorientierte Konzepte"	0.9 and 0.4 = 0.36
		2	"Zugriffskontrolle"	0.6 and 0.7 = 0.42
		3	"Textretrieval"	0.1 and 0.2 = 0.02
		4	"HTML-Einführung"	0.1 and 0.2 = 0.02
"Internet-AG"	...	0	"Datenbankhandbuch"	0.6 and 0.9 = 0.54
		1	"Zugriffskontrolle"	0.5 and 0.7 = 0.35
		2	"Objektorientierte Konzepte"	0.7 and 0.4 = 0.28
		3	"HTML-Einführung"	0.9 and 0.2 = 0.18
		4	"Textretrieval"	0.7 and 0.2 = 0.14

Auf dieser Grundlage kann nun die Anweisung

p.name

in der select-Klausel durchgeführt werden. Da sich diese Anweisung auf Objekte vom Typ Projekt bezieht, die im from-Teil gemeinsam mit in ihnen enthaltenen Relevanzrangfolgen aufgerufen werden, wird nun über diese Projektobjekte selbst ebenfalls eine Relevanzrangfolge aufgebaut. Semantisch wird hier ausgedrückt, wie wahrscheinlich es ist, daß ein Projekt durch eine Publikation beschrieben wird, die das Thema "Datenbanken" behandelt. Zu dieser Gesamtwahrscheinlichkeit könnte jede der Publikationen beitragen, durch die das Projekt wahrscheinlich beschrieben wird. Verknüpft man nun für ein Projekt alle Wahrscheinlichkeiten derjenigen Publikationen durch or, durch die nach der Anwendung der where-Klausel die vage Relation beschrieben_von belegt ist, so erhält man die gesuchte Gesamtwahrscheinlichkeit für das Projekt. Für die konkreten Objekte leitet sich dabei die folgende Relevanzrangfolge ab:

projekte *(in Relevanzrangfolge)*						
Pos.	**name**	...	**beschrieben_von**			**Prob.**
			Pos.	**publikation**	Prob.	
0	"Objektorientierte Datenbanken"	...	0	"Datenbankhandbuch"	0.81	0.81 or
			1	"Objektorientierte Konzepte"	0.36	0.36 or
			2	"Zugriffskontrolle"	0.42	0.42 or
			3	"Textretrieval"	0.02	0.02 or
			4	"HTML-Einführung"	0.02	0.02 ≈ 0.9323
1	"Internet-AG"	...	0	"Datenbankhandbuch"	0.54	0.54 or
			1	"Zugriffskontrolle"	0.35	0.35 or
			2	"Objektorientierte Konzepte"	0.28	0.28 or
			3	"HTML-Einführung"	0.18	0.18 or
			4	"Textretrieval"	0.14	0.14 ≈ 0.8481
2	"Digitale Bibliotheken"	...	0	"Datenbankhandbuch"	0.54	0.54 or
			1	"Objektorientierte Konzepte"	0.32	0.32 or
			2	"Zugriffskontrolle"	0.21	0.21 or
			3	"Textretrieval"	0.14	0.14 or
			4	"HTML-Einführung"	0.06	0.06 ≈ 0.8002

Die in der select-Klausel formulierte Projektion wird nun auf diese Relevanzrangfolge angewendet, so daß sich das folgende Resultat für die select-from-where-Anfrage ergibt:

select-from-where		
Pos.	**name**	**Prob.**
0	"Objektorientierte Datenbanken"	0.9323
1	"Internet-AG"	0.8481
2	"Digitale Bibliotheken"	0.8002

Die Formulierung und die Auswertung der Beispielanfrage ist am objektorientierten Charakter der Sprache ProQL ausgerichtet. Dieses ist insbesondere in der from-Klausel sichtbar. Hier erscheinen die Relevanzrangfolgen in Abhängigkeit von ebenfalls dort deklarierten Objekten. Durch diese Möglichkeit sind Verbundoperationen in ProQL wie auch in anderen objektorientierten Anfragesprachen meist überflüssig.

Der Weg, der bei der Auswertung der Anfrage beschritten wurde, ist daher direkt von der Formulierung der from-Klausel abhängig. Wie sich die verschiedenen Möglichkeiten, innerhalb der from-Klausel Relevanzrangfolgen aufzubauen, auf die Auswertungssemantik vager select-from-where-Anfragen auswirkt, soll hier nicht im Detail untersucht werden, sondern ist Gegenstand zukünftiger Forschung.

Über die folgenden semantischen Zusammenhänge lassen sich jedoch vor dem Hintergrund der Beispielanfrage klare Aussagen treffen:

→ Ist das Resultat des from-Teils einer vagen select-from-where-Anfrage eine Relevanzrangfolge, so erfolgt die Anwendung des im where-Teil enthaltenen Prädikats, ungeachtet dessen, ob es sich dabei um ein vages oder ein boolesches Prädikat handelt, durch die probabilistische and-Verknüpfung des sich aus dem Prädikat für ein Element ergebenden Werts mit der Wahrscheinlichkeit dieses Elements in der Relevanzrangfolge.

→ Das Resultat einer vagen select-from-where-Anfrage ist stets eine Relevanzrangfolge, auch wenn im from-Teil teilweise oder ausschließlich eindeutige Kollektionen erscheinen.

Zu erwähnen bleibt, daß eine vage select-from-where-Anfrage zusätzlich noch die oben bereits beschriebenen Klauseln number und lowest prob sowie die order-by-Klausel enthalten kann. Für die Auswertung der order-by-Klausel gilt dabei, daß alle dort angebenen Ordnungskriterien in der Priorität hinter der durch die Ergebnisrelevanzrangfolge implizit vorgebenen Ordnung zurückstehen. Daher werden die Ordnungskriterien der order-by-Klausel nur innerhalb der einzelnen Positionen der Relevanzrangfolge angewendet.

7.3.4 Weitere Optionen für die Realisierung von ProQL

Die in ProQL eingebrachten Erweiterungen der Sprache OQL stellen nur eine Teilmenge aller möglichen Erweiterungen dar, die sich aus der Definition von vager Semantik im Objektmodell ProM ergeben. So wäre beispielsweise eine vage Gleichheitsoperation zwischen Wahrscheinlichkeitswerten denkbar.
Eine Übernahme aller möglichen vagen Konstrukte kann jedoch nicht das unbedingte Ziel bei der Entwicklung der Anfragesprache ProQL sein. Vielmehr ist darauf zu achten, daß die Sprache für Anwender, die zumeist entweder mit dem Umgang mit Datenbanksystemen oder mit dem Umgang mit Information Retrieval Systemen vertraut sind, trotz der Integration beider Welten gut verständlich und intuitiv einsetzbar belibt. Zugunsten dieser Anforderung verzichtet die vorliegende Arbeit auf die Einführung weiterer vager Konzepte in die Sprache ProQL.

7.4 Die formale Definition der Anfragesprache ProQL

Die Menge der gültigen Ausdrücke der Sprache ProQL entspricht der Menge der in dieser Sprache formulierbaren Anfragen[2]. Jeder Ausdruck ist eine Anfrage und umgekehrt, so daß die beiden Begriffe synonym verwendet werden können.
ProQL ist eine typisierte Sprache. Eine gültige Anfrage setzt sich aus Operatoren und getypten Operanden zusammen, die rekursiv kombiniert werden können. Auch dem Resultat einer Anfrage kann eindeutig ein Typ zugeordnet werden. Dieser ergibt sich aus der Struktur der Anfrage, den Typdeklarationen des Datenschemas (vgl. Kapitel 6) sowie aus dem Typ etwaiger Literalwerte oder benannter Objekte, die in der Anfrage enthalten sind.

[2]Dieses ist auch der Syntax der Sprache zu entnehmen, die sich im Anhang C befindet

Unter bestimmten Voraussetzungen können Instanzen eines Typs auch für Instanzen eines anderen Typs eingesetzt werden, ohne daß dabei das Typsystem verletzt wird. Diese Typkompatibilität ergibt sich durch die Vererbungsbeziehungen zwischen den verschiedenen Typen und wird in ProQL durch die folgenden Regeln rekursiv definiert:

(1) t ist kompatibel zu t

(2) Falls t kompatibel ist zu t', so gilt
set(t) ist kompatibel zu set(t')
bag(t) ist kompatibel zu bag(t')
list(t) ist kompatibel zu list(t')
array(t) ist kompatibel zu array(t')
rankedList(t) ist kompatibel zu rankedList(t')

(3) Falls es ein t gibt, so daß t Supertyp von t_1 und t_2 ist, dann sind t_1 und t_2 kompatibel.

Des weiteren kann für eine Menge von kompatiblen Typen eindeutig festgestellt werden, welcher ihrer gemeinsamen Supertypen der speziellste ist. Auf diesen Typ wird an mehreren Stellen dieser Sprachdefinition zurückgegriffen. Formal kann er wie folgt definiert werden: Sind $t_1, t_2, \ldots, t_n$ zueinander kompatibel, dann existiert genau ein t mit:

(1) $t >^3 t_i$ für alle $1 \leq i \leq n$

(2) Für alle t', für die gilt $t' \neq t$ und $t' > t_i$ für alle $1 \leq i \leq n$,
gilt $t' > t$

Die vorliegende Arbeit verwendet für t die Notation $\mathsf{lub}(t_1, t_2, \ldots, t_n)$[4].

7.4.1 Anfragen

Eine Anfrage ist ein Ausdruck, in dem keine Variablen vorkommen, die im Kontext dieses Ausdrucks gebunden sind.

7.4.2 Die Definition benannter Anfragen

In dem Ausdruck define $\mathsf{id}(x_1, x_2, \ldots, x_n)$ as $\mathsf{e}(x_1, x_2, \ldots, x_n)$ seien id ein Bezeichner, e ein Ausdruck und $x_1, x_2, \ldots, x_n$ freie Variable innerhalb von e. Dieser Ausdruck definiert eine benannte Anfrage, die einer Funktion mit n Parametern entspricht.
Für den Bezeichner id darf innerhalb des existierenden Datenschemas noch keine Bindung an ein benanntes Objekt, eine Methode oder eine Klasse bestehen. Des weiteren darf id

[3] Im Zusammenhang mit Typen werden im Folgenden die Symbole ">" und "<" zur Darstellung der Super- bzw. Subtyprelation benutzt. Erscheinen diese Symbole in anderem Zusammenhang, so tragen sie die in der Mathematik gebräuchliche Bedeutung.

[4] lub ist die Abkürzung für den englischen Begriff *lowest upper bound*.

nicht der Name einer vordefinierten Funktion sein. Sind diese Bedingungen nicht erfüllt, so führt der Versuch, eine Anfragedefinition an id zu binden, zu einem Fehler.
Bei der Auswertung einer Anfrage, in der der Bezeichner id benutzt wird, wird dieser durch den Ausdruck e ersetzt. Hierdurch kann die Definition benannter Anfragen beispielsweise als Mechanismus zur Realisierung von Datenbanksichten (vgl. Abschnitt 3.1.4) verwendet werden.
Anfragedefinitionen sind persistent. Sie bleiben bestehen, bis sie entweder durch eine neue Anfragedefinition für denselben Bezeichner überschrieben werden oder explizit durch den Ausdruck delete definition id gelöscht werden.
Anfragedefinitionen können nicht überladen werden. Daher ist es nicht möglich, zusätzlich zu einer Definition für den Bezeichner id mit n Parametern eine weitere Definition für id mit $p \neq n$ Parametern vorzunehmen, ohne daß die erste Definition von id dabei überschrieben wird.
Falls die Definition einer benannten Anfrage keine Parameter festlegt, können die runden Klammern weggelassen werden.

Beispiele:

```
define alter(x) as
   select p.alter
   from personen p
   where p.name = x

define einsteins() as
   select p
   from personen p
   where p.name = "Einstein"
```

7.4.3 Elementare Ausdrücke

Atomare Literale, benannte Objekte und Anfragen sowie die Iterationsvariable innerhalb einer select-from-where-Anfrage sind elementare Ausdrücke der Sprache ProQL.

Atomare Literale in Erweiterung der Sprache OQL

Falls l ein atomares Literal ist, so ist l ein Ausdruck, dessen Wert das Literal selbst ist. Literale unterliegen der gewohnten Syntax:

- Leeres Literal: nil
- Boolesches Literal: false, true
- Ganzzahliges Literal: Ziffernfolge mit optionalem negativem Vorzeichen.
  Beispiel: 27 oder -347

- Gleitkommaliteral: positive oder negative Mantisse/ positiver oder negativer Exponent. Der Exponent ist optional.

  Beispiel: 3.14 oder 314.16e-2

→ Probabilistisches Literal: wie Gleitkommaliteral, jedoch nur Werte zwischen 0 und 1.

  Beispiele: 1. oder 23.456e-2

- Symbolliteral: Symbol zwischen einfachen Anführungszeichen.

  Beispiel: 'a'

- Zeichenkettenliteral: Zeichenkette zwischen doppelten Anführungszeichen.

  Beispiel: "eine Zeichenkette"

Benannte Objekte

Falls e ein Objektname ist, so ist e ein Ausdruck, der die Instanz, die an den Objektnamen gebunden ist, zurückgibt. Der Typ von e entspricht dem Typ des benannten Objektes, so wie er im Datenschema deklariert wurde.

`Beispiel:`

```
Datenbankspezialist
```

Dieser Beispielausdruck hat ein bestimmtes Objekt aus der Extension der Klasse Person zum Ergebnis, unter der Voraussetzung, daß der Bezeichner Datenbankspezialist an dieses Objekt gebunden ist.

Iterationsvariable

Ist x eine Variable, die in einer select-from-where-Klausel definiert ist, so ist x ein Ausdruck. Der Wert dieses Ausdrucks entspricht dem aktuellen Element der Iteration, die innerhalb der select-from-where-Klausel über der zu durchsuchenden Kollektion durchgeführt wird. Falls x im from-Teil einer select-from-where-Klausel in der Form

```
e as x
```

oder

```
e x
```

oder

```
x in e
```

deklariert wird, wobei e vom Typ collection(t) ist, so ist x vom Typ t.

Benannte Anfragen

Falls define id(x_1, x_2, ..., x_n) as e(x_1, x_2, ..., x_n) die Definition einer benannten Anfrage ist, wobei es sich bei e um einen Ausdruck des Typs t mit den freien Variablen x_1, x_2, ..., x_n handelt, so ist id(x_1, x_2, ..., x_n) ebenfalls ein Ausdruck vom Typ t.

Beispiel:

```
einsteins()
```

Diese Anfrage bezieht sich auf die weiter oben erfolgte Anfragedefinition und ergibt die Menge aller Personen, die den Namen Einstein tragen.

7.4.4 Konstruktionsausdrücke

Die Sprache ProQL umfaßt Ausdrücke zur Konstruktion von Objekten, Strukturen und veränderbaren Kollektionen.

Die Konstruktion von Objekten

Falls t ein Typbezeichner ist und p_1, p_2, ..., p_n Eigenschaften dieses Typs mit den dazugehörigen Typen t_1, t_2, ..., t_n sind, so ist t(p_1: e_1, p_2: e_2, ..., p_n: e_n) ein Ausdruck vom Typ t.
Das Resultat dieses Ausdrucks ist ein neues Objekt vom Typ t, dessen Eigenschaften p_1, p_2, ..., p_n durch die Ausdrücke e_1, e_2, ..., e_n initialisiert werden. Dabei entspricht der Typ eines jeden e_i dem Typ des dazugehörigen p_i bzw. einem Subtyp davon.
Falls t der Typname einer Kollektion und e ein Kollektionsliteral ist, so ist t(e) ein Kollektionsobjekt. Der Typ von e muß dabei t sein.

Beispiele:

```
Person(name: "Einstein", vorname: "Albert",
       adresse: Datenbankspezialist.adresse)
```

erzeugt ein Objekt vom Typ Person.

```
zahlenmenge(set(1,2,5,7))
```

erzeugt unter der Voraussetzung, daß zahlenmenge der Typname für eine Menge von ganzen Zahlen ist, ein entsprechendes Kollektionsobjekt.

Die Konstruktion von Strukturen

Falls p_1, p_2, ..., p_n Komponentennamen eines strukturierten Typs und e_1, e_2, ..., e_n Ausdrücke mit den entsprechenden Typen t_1, t_2, ..., t_n sind, so ist struct(p_1: e_1, p_2: e_2, ..., p_n: e_n) ein Ausdruck.
Dieser Ausdruck erzeugt eine Instanz vom Typ struct(p_1: t_1, p_2: t_2, ..., p_n: t_n), falls für alle $1 \leq i \leq n$ gilt, daß e_i vom Typ t_i ist.

Beispiel:

```
struct(name: "Albert", alter: 71)
```

ergibt eine Struktur mit den beiden Komponenten name und alter, die mit den Werten "Albert" und 71 belegt sind.

Die Konstruktion von Mengen

Falls e_1, e_2, ..., e_n Ausdrücke mit zueinander kompatiblen Typen t_1, t_2, ..., t_n sind, so ist set(e_1, e_2, ..., e_n) ein Ausdruck vom Typ set(t), wobei gilt t = lub(t_1, t_2, ... , t_n). Dieser Ausdruck erzeugt eine Menge mit den Elementen e_1, e_2, ..., e_n.

Beispiel:

```
set(18,20,2)
```

ergibt eine Menge mit den drei Elementen 18, 20 und 2.

Die Konstruktion von Listen

Falls e_1, e_2, ..., e_n Ausdrücke mit zueinander kompatiblen Typen t_1, t_2, ..., t_n sind, so sind list(e_1, e_2, ..., e_n) und auch einfach (e_1, e_2, ..., e_n) Ausdrücke vom Typ list(t), wobei gilt t = lub(t_1, t_2, ... , t_n). Diese Ausdrücke erzeugen jeweils Listen mit den Elementen e_1, e_2, ..., e_n.
Falls min und max zwei Ausdrücke sind, die entweder beide ganze Zahlen oder beide Symbole sind, so daß min < max gilt, so sind list(min .. max) und auch (min .. max) Ausdrücke. Diese Ausdrücke haben den Wert list(min, min+1, ..., max-1, max). Der Typ der Ausdrücke ist list(int)[5] oder list(char), je nachdem von welchem Typ min und max sind.

Die Konstruktion von Multimengen

Falls e_1, e_2, ..., e_n Ausdrücke mit zueinander kompatiblen Typen t_1, t_2, ..., t_n sind, so ist bag(e_1, e_2, ..., e_n) ein Ausdruck vom Typ bag(t), wobei gilt t = lub(t_1, t_2, ... , t_n). Dieser Ausdruck erzeugt eine Multimenge mit den Elementen e_1, e_2, ..., e_n.

[5] Der Typ int steht hier als Platzhalter für einen der in Abschnitt 5.6.1 aufgeführten Typen long, short, unsigned long oder unsigned short.

`Beispiel:`

```
bag(1,1,2,3,3)
```

ergibt eine Menge mit fünf Elementen.

Die Konstruktion von Feldern

Falls e_1, e_2, ..., e_n Ausdrücke mit zueinander kompatiblen Typen t_1, t_2, ..., t_n sind, so ist array(e_1, e_2, ..., e_n) ein Ausdruck vom Typ array(t), wobei gilt t = lub(t_1, t_2, ..., t_n). Dieser Ausdruck erzeugt ein Feld mit den Elementen e_1, e_2, ..., e_n.

`Beispiel:`

```
array(3,4,2,1,1)
```

ergibt ein Feld mit fünf Elementen.

7.4.5 Atomare Typausdrücke

Die atomaren Ausdrücke der Sprache ProQL lassen sich unterteilen in unäre und binäre Ausdrücke sowie in Ausdrücke für Zeichenketten.

Unäre Ausdrücke in Erweiterung der Sprache OQL

Falls e ein Ausdruck ist und <op> eine unäre Operation, die auf dem Typ von e definiert ist, so ist <op>e ein Ausdruck. Dieser ergibt das Resultat der Anwendung von <op> auf e.

Arithmetische unäre Operatoren:	+, −, abs
Boolescher unärer Operator:	not
→ Probabilistischer unärer Operator:	not

`Beispiele:`

```
not true
```

ergibt false.

```
not 0.75
```

ergibt 0.25.

Falls <op> einer der Operatoren +, − oder abs ist und falls e vom Typ int oder float ist, so ist <op>e vom selben Typ wie e.
Falls e vom Typ boolean ist, so ist not e ebenfalls vom Typ boolean.
Falls e vom Typ prob ist, so ist not e ebenfalls vom Typ prob.

Binäre Ausdrücke in Erweiterung der Sprache OQL

Falls e_1 und e_2 Ausdrücke gleichen Typs sind und <op> ein binärer Operator ist, der auf dem Typ von e_1 und e_2 definiert ist, so ist e_1<op>e_2 ein Ausdruck. Dieser ergibt das Resultat der Anwendung von <op> auf e_1 und e_2.

Arithmetische binäre Operatoren für ganze Zahlen:	+, −, ∗, /, mod
Arithmetische binäre Operatoren für Gleitkommazahlen:	+, −, ∗, /
Binäre Vergleichsoperatoren:	=, !=, <, <=, >, >=

Die binären Vergleichsoperatoren sind auf allen atomaren Typen definiert.

Boolesche binäre Operatoren:	and, or
→ Probabilistische binäre Operatoren:	and, or

`Beispiel:`

```
count(publikationen) - count(buecher)
```

ergibt die Differenz zwischen der Anzahl an Publikationen und der Anzahl an Büchern.
Falls <op> einer der Operatoren +, −, ∗ oder / ist und e_1 und e_2 vom Typ int oder float sind, so ist e_1<op>e_2 entweder vom Typ float, falls e_1 oder e_2 vom Typ float ist, oder andernfalls vom Typ int.
Falls <op> einer der Operatoren =, !=, <, ≤, > oder ≥ ist und e_1 und e_2 von kompatiblem Typ sind, wobei hier auch die Typen float und int als zueinander kompatibel betrachtet werden, so ist e_1<op>e_2 vom Typ boolean.
Falls <op> einer der Operatoren and oder or ist und e_1 und e_2 vom Typ boolean oder prob sind, so ist e_1<op>e_2 entweder vom Typ prob, falls e_1 oder e_2 vom Typ prob ist, oder andernfalls vom Typ boolean.

Ausdrücke für Zeichenketten

Falls s_1 und s_2 Ausdrücke vom Typ string sind, so sind $s_1 \parallel s_2$ und $s_1 + s_2$ Ausdrücke vom Typ string, die den Wert der Konkatenation von s_1 und s_2 ergeben.
Falls c ein Ausdruck vom Typ char ist und s ein Ausdruck vom Typ string, so ist c in s ein Ausdruck vom Typ boolean, der genau dann wahr ist, wenn c zu s gehört.
Falls s ein Ausdruck vom Typ string ist und i ein Ausdruck vom Typ int, so ist s[i] ein Ausdruck vom Typ char, dessen Wert das Symbol an der Stelle i+1 der Zeichenkette s ist.
Falls s ein Ausdruck vom Typ string ist und low und up Ausdrücke vom Typ int sind, so ist s[low:up] ein Ausdruck vom Typ string, dessen Wert diejenige Teilzeichenkette in s ist, die sich von der Stelle low+1 bis zur Stelle up+1 erstreckt.
Sei s ein Ausdruck vom Typ string. Sei pattern ein Zeichenkettenliteral, das die Platzhaltersymbole "?" oder "_" für ein beliebiges Symbol und die Platzhaltersymbole "∗" oder "%" für eine beliebige Teilzeichenkette, inklusive der leeren Zeichenkette, enthalten darf. Dann ist s like pattern ein Ausdruck vom Typ boolean, der genau dann wahr ist, wenn s dem durch pattern vorgegebenen Muster entspricht.

Beispiel:

```
'Zeichenketten sind toll' like 'Z_ic%k%en sind _oll'
```

ergibt den Wert true.

7.4.6 Objektausdrücke

Die Sprache ProQL stellt Objektausdrücke in Form von Vergleichen und Operationen für den Zugriff auf Objekteigenschaften zur Verfügung.

Der Objektvergleich

Falls e_1 und e_2 Ausdrücke sind, die Objekte von zueinander kompatiblen Objekttypen bezeichnen, so sind $e_1 = e_2$ und e_1 != e_2 Ausdrücke vom Typ boolean. Der Ausdruck $e_1 = e_2$ ist genau dann wahr, wenn e_1 und e_2 dasselbe Objekt bezeichnen. Der Ausdruck e_1 != e_2 ist äquivalent zu not($e_1 = e_2$).

Beispiel:

Unter der Voraussetzung, daß Datenbankspezialist ein Objekt benennt, das das einzige Objekt in der benannten Menge mitarbeiter ist, das als Attribut leiter desjenigen Objektes der Menge projekte erscheint, dessen Attribut name mit dem Wert "Datenbankprojekt" belegt ist, so ergibt der Ausdruck

```
Datenbankspezialist =
element(select m
        from mitarbeiter m
             projekte p
        where p.name = "Datenbankprojekt"
              and p.leiter = m)
```

den Wert true.

Der Literalvergleich

Falls e_1 und e_2 Ausdrücke sind, die Literale von zueinander kompatiblen Literaltypen bezeichnen, so sind $e_1 = e_2$ und e_1 != e_2 Ausdrücke vom Typ boolean. Der Ausdruck $e_1 = e_2$ ist genau dann wahr, wenn e_1 und e_2 mit demselben Wert belegt sind. Der Ausdruck e_1 != e_2 ist äquivalent zu not($e_1 = e_2$).
Die Gleichheit von Literalen berechnet sich nach den folgenden Regeln:

- Strukturierte Literale müssen dieselbe Struktur aufweisen, und alle ihre Komponenten müssen gleich sein.
- Mengen müssen dieselbe Elemente aufweisen.

- Multimengen müssen dieselben Elemente und dieselbe Anzahl von Elementen aufweisen.
- Listen oder Felder müssen dieselben Elemente in derselben Reihenfolge aufweisen.

Der Zugriff auf Attribute und die Traversierung von Relationen eines Objektes

Falls e ein Objekt eines Typs bezeichnet, der ein Attribut p oder eine Relation p vom Typ t definiert, so sind e.p und e ->p äquivalente Ausdrücke vom Typ t. Diese Ausdrücke greifen auf die Eigenschaft p des Objektes e zu.

Die Traversierung von vagen Relationen eines Objektes in Erweiterung der Sprache OQL

Falls e ein Objekt eines Typs bezeichnet, der eine vage Relation v vom Typ rankedList(t) definiert, so sind e.v und e ->v äquivalente Ausdrücke vom Typ rankedList(t). Diese Ausdrücke greifen auf diejenige Relevanzrangfolge zu, die durch die vage Relation v bezüglich des durch e bezeichneten Objektes berechnet wird.

Der Aufruf einer Operation für ein Objekt

Falls e ein Ausdruck von einem Typ ist, für den eine Operation f definiert ist, die keine Parameter hat und ein Resultat vom Typ t zurückgibt, so sind e.f und e ->f äquivalente Ausdrücke vom Typ t. Durch diese Ausdrücke wird für das Objekt e die Operation f aufgerufen. Der Wert der Ausdrücke ist derjenige, der von der Operation zurückgegeben wird, oder nil, falls die Operation keinen Wert zurückgibt.

Beispiel:

```
Datenbankspezialist->alter
```

ruft die Operation alter für das Objekt Datenbankspezialist auf.

Der Aufruf eines vagen Prädikats für ein Objekt in Erweiterung der Sprache OQL

Falls e ein Ausdruck von einem Typ ist, für den ein vages Prädikat v definiert ist, das keine Parameter hat, so sind e.v und e ->v äquivalente Ausdrücke vom Typ prob. Durch diese Ausdrücke wird für das Objekt e das vage Prädikat v aufgerufen. Der Wert der Ausdrücke ist derjenige, der durch das vage Prädikat v zurückgegeben wird.

Beispiel:

```
meine_Publikation->ist_aktuell
```

ruft das vage Prädikat ist_aktuell für das Objekt meine_Publikation auf.

Der Aufruf einer Operation mit Parametern für ein Objekt

Sei e ein Ausdruck von einem Typ, für den eine Operation f mit Parametern vom Typ t_1, t_2, ..., t_n definiert ist, die ein Resultat vom Typ t zurückgibt. Seien e_1, e_2, ..., e_n Ausdrücke vom Typ t_1, t_2, ..., t_n. Dann sind e.f(e_1, e_2, ..., e_n) und e->f(e_1, e_2, ..., e_n) äquivalente Ausdrücke vom Typ t. Durch diese Ausdrücke wird für das Objekt e die Operation f mit den Parametern e_1, e_2, ..., e_n aufgerufen. Der Wert der Ausdrücke ist derjenige, der von der Operation zurückgegeben wird, oder nil, falls die Operation keinen Wert zurückgibt.

Beispiel:

```
meine_publikation.erstelle_verweis("literatur.bib")
```

ruft die Operation erstelle_verweis mit dem Parameter "literatur.bib" für das Objekt meine_publikation auf.

Der Aufruf eines vagen Prädikats mit Parametern für ein Objekt in Erweiterung der Sprache OQL

Sei e ein Ausdruck von einem Typ, für den ein vages Prädikat v mit Parametern vom Typ t_1, t_2, ..., t_n definiert ist. Seien e_1, e_2, ..., e_n Ausdrücke vom Typ t_1, t_2, ..., t_n. Dann sind e.v(e_1, e_2, ..., e_n) und e ->v(e_1, e_2, ..., e_n) äquivalente Ausdrücke vom Typ prob. Durch diese Ausdrücke wird für das Objekt e das vage Prädikat v mit den Parametern e_1, e_2, ..., e_n aufgerufen. Der Wert der Ausdrücke ist derjenige, der durch das vage Prädikat zurückgegeben wird.

Beispiel:

```
meine_Publikation->handelt_von("Information Retrieval")
```

ruft das vage Prädikat handelt_von mit dem Parameter "Information Retrieval" für das Objekt meine_Publikation auf.

7.4.7 Kollektionsausdrücke

Für Kollektionen sind in der Sprache ProQL die Quantifizierung, die Überprüfung auf Zugehörigkeit sowie verschiedene Aggregatoperationen definiert.

Die Quantifizierung

Falls x ein Variablenname ist und e_1 und e_2 Ausdrücke sind, wobei e_1 eine Kollektion bezeichnet und e_2 ein Ausdruck vom Typ boolean ist, so sind for all x in e_1: e_2 und exists x in e_1: e_2 Ausdrücke vom Typ boolean. Der Ausdruck for all x in e_1: e_2 ist genau dann wahr, wenn alle Elemente der Kollektion e_1 den Ausdruck e_2 erfüllen. Der Ausdruck exists x in e_1: e_2 ist genau dann wahr, wenn mindestens ein Element der Kollektion e_1 den Ausdruck e_2 erfüllt.

Beispiele:

```
for all x in publikationen: x.erscheinungsdatum > 1994
```

ergibt den Wert true, falls alle Objekte in der Menge publikationen jeweils nach 1994 erschienen sind.

```
exists x in publikationen: x.erscheinungsdatum > 1994
```

ergibt den Wert true, falls mindestens ein Objekt in der Menge publikationen nach 1994 erschienen ist.
Falls e ein Kollektionsausdruck ist, so sind exists(e) und unique(e) Ausdrücke vom Typ boolean. Der Ausdruck exists(e) ist genau wahr, wenn sich mindestens ein Element in der Kollektion e befindet. Der Ausdruck unique(e) ist genau dann wahr, wenn sich genau ein Element in der Kollektion e befindet.

Die vage Quantifizierung in Erweiterung der Sprache OQL

Falls x ein Variablenname ist und e_1 und e_2 Ausdrücke sind, wobei e_1 eine Kollektion bezeichnet und e_2 ein Ausdruck vom Typ prob ist, so sind for all x in e_1: e_2 und exists x in e_1: e_2 Ausdrücke vom Typ prob. Der Ausdruck for all x in e_1: e_2 ermittelt für alle Elemente der Kollektion e_1 die Wahrscheinlichkeit, die der Ausdruck e_2 im Zusammenhang mit dem jeweiligen Element ergibt, und verknüpft diese Wahrscheinlichkeiten durch den probabilistischen and-Operator. Der Ausdruck exists x in e_1: e_2 ermittelt für alle Elemente der Kollektion e_1 die Wahrscheinlichkeit, die der Ausdruck e_2 im Zusammenhang mit dem jeweiligen Element ergibt, und verknüpft diese Wahrscheinlichkeiten durch den probabilistischen or-Operator.

Beispiele:

```
for all x in publikationen: x.handelt_von("Information Retrieval")
```

berechnet eine Wahrscheinlichkeit dafür, daß alle Objekte in der Menge publikationen das Thema "Information Retrieval" behandeln dadurch, daß die Konjunktion aller Einzelwahrscheinlichkeiten gebildet wird.

```
exists x in publikationen: x.handelt_von("Information Retrieval")
```

berechnet eine Wahrscheinlichkeit dafür, daß mindestens ein Objekt in der Menge publikationen das Thema "Information Retrieval" behandelt dadurch, daß die Disjunktion aller Einzelwahrscheinlichkeiten gebildet wird.

Die Überprüfung auf Zugehörigkeit

Seien e_1 und e_2 Ausdrücke, bezeichne e_2 eine Kollektion, die nicht vom Typ rankedList ist, und sei e_1 eine Instanz vom Elementtyp der Kollektion e_2 oder von einem Subtyp des Elementyps der Kollektion e_2. Dann ist e_1 in e_2 ein Ausdruck vom Typ boolean. Dieser Ausdruck ist genau dann wahr, wenn das Element e_1 zur Kollektion e_2 gehört.

Beispiel:

```
Datenbankspezialist in mitarbeiter
```

ergibt den Wert true, wenn sich das Objekt Datenbankspezialist in der Menge mitarbeiter befindet.

Die Überprüfung auf vage Zugehörigkeit in Erweiterung der Sprache OQL

Seien e_1 und e_2 Ausdrücke, bezeichne e_2 eine Kollektion vom Typ rankedList, und sei e_1 eine Instanz vom Elementtyp der Kollektion e_2 oder von einem Subtyp des Elementyps der Kollektion e_2. Dann ist e_1 in e_2 ein Ausdruck vom Typ prob. Der Wert dieses Ausdrucks ist diejenige Wahrscheinlichkeit, die dem Element e_1 in der Relevanzrangfolge e_2 zugeordnet ist. Ist das Element nicht in der Relevanzrangfolge vorhanden, so ist der Wert des Ausdrucks 0.0 .

Beispiel:

```
meine_publikation in
     select p from publikationen p
     where  p.handelt_von("Information Retrieval")
```

ergibt die Wahrscheinlichkeit, die dem Element meine_publikation in der durch die vage select-from-where-Klausel berechneten Relevanzrangfolge zugeordnet ist.

Aggregatoperatoren

Falls e ein Ausdruck ist, der eine Kollektion bezeichnet, und <op> ist einer der Operatoren min, max, count, sum oder avg, so ist <op>(e) ein Ausdruck.

Beispiel:

```
min(select alter from mitarbeiter)
```

ergibt das Alter des jüngsten Mitarbeiters.
Sei e vom Typ collection(t), so ist count(e) ein Ausdruck vom Typ int.
Sei e vom Typ collection(t), wobei t entweder dem Typ int, dem Typ float oder dem Typ prob entspricht. Sei <op> ein anderer Aggregatoperator als der Operator count. Dann ist <op>(e) ein Ausdruck vom Typ t.

7.4.8 Eindeutige select-from-where-Ausdrücke

Eindeutige select-from-where-Ausdrücke lassen im Gegensatz zu vagen select-from-where-Ausdrücken (siehe unten) keine Ausdrücke vom Typ rankedList innerhalb der from-Klausel zu. Der Typ des where-Teils ist stets ein boolesches Prädikat. Die Schlüsselworte number und lowest prob kommen in eindeutigen select-from-where-Ausdrücken nicht vor.
Die allgemeine Form eines eindeutigen select-from-where-Ausdrucks ist die folgende:

select [distinct] f(x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+p})
from x_1 in e_1(x_{n+1}, x_{n+2}, ..., x_{n+p})
    x_2 in e_2(x_1, x_{n+1}, x_{n+2}, ..., x_{n+p})
    x_3 in e_3(x_1, x_2, x_{n+1}, x_{n+2}, ..., x_{n+p})
    ...
    x_n in e_n(x_1, x_2, ..., x_{n-1}, x_{n+1}, x_{n+2}, ..., x_{n+p})
[where p(x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+p})]
[order by f_1(x_1, x_2, ..., x_{n+p}), f_2(x_1, x_2, ..., x_{n+p}), ..., f_q(x_1, x_2, ..., x_{n+p})]

Oder:

select [distinct] f(x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+p})
from e_1(x_{n+1}, x_{n+2}, ..., x_{n+p}) as x_1
    e_2(x_1, x_{n+1}, x_{n+2}, ..., x_{n+p}) as x_2
    e_3(x_1, x_2, x_{n+1}, x_{n+2}, ..., x_{n+p}) as x_3
    ...
    e_n(x_1, x_2, ..., x_{n-1}, x_{n+1}, x_{n+2}, ..., x_{n+p}) as x_n
[where p(x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+p})]
[order by f_1(x_1, x_2, ..., x_{n+p}), f_2(x_1, x_2, ..., x_{n+p}), ..., f_q(x_1, x_2, ..., x_{n+p})]

x_{n+1}, x_{n+2}, ..., x_{n+p} sind freie Variablen, die zur Auswertung des Ausdrucks gebunden werden müssen.
Die e_i müssen Kollektionen sein. Der Typ rankedList ist für die Kollektionen e_i jedoch nicht zugelassen. Der Ausdruck p muß vom Typ boolean sein. Die f_i müssen von einem Typ sein, auf dem eine Ordnung definiert ist.
Das Resultat des eindeutigen select-from-where-Ausdrucks ist eine Kollektion mit Elementen des Typs t, wobei t der Typ des Resultats von f ist.
Unter der Annahme, daß die Variablen x_{n+1}, x_{n+2}, ..., x_{n+p} an die Ausdrücke X_{n+1}, X_{n+2}, ..., X_{n+p} gebunden werden, wird eine eindeutige select-from-where-Anfrage wie folgt ausgewertet:

(1) Das Resultat der from-Klausel ist eine Multimenge mit Elementen des Typs struct(x_1: X_1, x_2: X_2, ..., x_n: X_n), wobei

der Wertebereich von X_1 der Kollektion bagof($e_1(X_{n+1}, X_{n+2}, ..., X_{n+p})$) entspricht,

der Wertebereich von X_2 der Kollektion bagof($e_2(X_1, X_{n+1}, X_{n+2}, ..., X_{n+p})$) entspricht,

der Wertebereich von X_3 der Kollektion bagof($e_3(X_1, X_2, X_{n+1}, X_{n+2}, ..., X_{n+p})$) entspricht,

...

der Wertebereich von X_n der Kollektion bagof($e_n(X_1, X_2, ..., X_{n-1}, X_{n+1}, X_{n+2}, ..., X_{n+p})$) entspricht,

wobei bagof(C) für eine Kollektion C wie folgt definiert ist:

- C, falls C eine Multimenge ist,
- die Multimenge, die alle Elemente von C enthält, falls C eine Liste oder eine Menge ist.

(2) Das Resultat der from-Klausel wird so gefiltert, daß es anschließend nur noch diejenigen Tupel $(X_1, X_2, ..., X_n)$ enthält, die das Prädikat $p(X_1, X_2, ..., X_{n-1}, X_n, X_{n+1}, X_{n+2}, ..., X_{n+p})$ erfüllen.

(3) Falls der eindeutige select-from-where-Ausdruck das Schlüsselwort order by enthält, wird die in (2) entstandene Kollektion unter Verwendung der Funktionen $f_1, f_2, ..., f_q$ lexikographisch sortiert und in eine Liste umgewandelt. Die lexikographische Ordnung anhand einer Menge von Funktionen wird wie folgt durchgeführt: Zunächst wird die Kollektion anhand der Funktion f_1 sortiert. Alle Elemente, die für f_1 denselben Wert tragen, werden dann anhand der Funktion f_2 sortiert u.s.w.

(4) Auf jedes Tupel der Kollektion wird nun die Funktion

$f(X_1, X_2, ..., X_{n-1}, X_n, X_{n+1}, X_{n+2}, ..., X_{n+p})$

angewendet. Falls f die Funktion "$*$" ist, bleibt die Kollektion aus (3) erhalten.

(5) Falls der eindeutige select-from-where-Ausdruck das Schlüsselwort distinct enthält, werden eventuelle Duplikate aus der Kollektion entfernt. Es entsteht eine Menge oder eine Liste ohne Duplikate.

Zusammenfassend kann über den Typ des Resultats einer eindeutigen select-from-where-Anfrage folgendes gesagt werden:

- Das Resultat ist immer eine Kollektion.
- Der Typ dieser Kollektion hängt nicht direkt von den Typen der Kollektionen im from-Teil des Ausdrucks ab.

- Der Typ der Kollektion ist lediglich von der Form der eindeutigen Anfrage abhängig: Die Verwendung von order by ergibt eine Liste; die Verwendung von distinct ohne order by ergibt eine Menge; das Ergebnis einer eindeutigen Anfrage ohne order by und ohne distinct ist eine Multimenge.

Beispiel:

```
select p.titel
from publikationen p
     p.verwendet_in q
where q.leiter = Datenbankspezialist
```

ermittel die Titel aller Publikationen, die in Projekten verwendet werden, die unter der Leitung von Datenbankspezialist stehen.

7.4.9 Vage select-from-where-Ausdrücke in Erweiterung der Sprache OQL

Die allgemeine Form eines vagen select-from-where-Ausdrucks ist die folgende:

select [distinct] f(x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+p})
from r(x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+p})
[where p(x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+p})]
[number numfun(x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+p})]
[lowest prob probfun(x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+p})]
[order by f_1(x_1, x_2, ..., x_{n+p}), f_2(x_1, x_2, ..., x_{n+p}), ..., f_q(x_1, x_2, ..., x_{n+p})]

x_{n+1}, x_{n+2}, ..., x_{n+p} sind freie Variablen, die zur Auswertung des Ausdrucks gebunden werden müssen.
Der Ausdruck r muß eine Kollektion sein. Der Ausdruck p muß vom Typ prob oder vom Typ boolean sein. Der Ausdruck numfun muß vom Typ int sein. Der Ausdruck probfun muß vom Typ prob oder vom Typ boolean sein. Die f_i müssen von einem Typ sein, auf dem eine Ordnung definiert ist.
Eine select-from-where-Anfrage ist somit genau dann vage, wenn

- sie eines der Schlüsselwörter number oder lowest prob enthält oder
- im from-Teil mindestens eine Relevanzrangfolge deklariert ist oder
- das Prädikat des where-Teils vage ist, d.h. einen Wert vom Typ prob zurückliefert.

Das Resultat des vagen select-from-where-Ausdrucks ist eine Relevanzrangfolge mit Elementen des Typs t, wobei t der Typ des Resultats von f ist.
Unter der Annahme, daß die Variablen x_{n+1}, x_{n+2}, ..., x_{n+p} an die Ausdrücke X_{n+1}, X_{n+2}, ..., X_{n+p} gebunden werden, wird eine vage select-from-where-Anfrage wie folgt ausgewertet:

(1) Das Resultat der from-Klausel ist die Relevanzrangfolge $\mathsf{r}(\mathsf{X}_{n+1}, \mathsf{X}_{n+2}, ..., \mathsf{X}_{n+p})$, die sich zumeist als kartesisches Produkt der einzelnen dort deklarierten Kollektionen darstellt, in das zusätzlich die innerhalb dieser Kollektionen erscheinenden Relevanzwahrscheinlichkeiten einbezogen werden.

(2) Auf jedes Element dieser Relevanzrangfolge wird das vage Prädikat p angewendet. Die sich ergebende Wahrscheinlichkeit $\mathsf{p}(\mathsf{X}_1, \mathsf{X}_2, ..., \mathsf{X}_{n+p})$ wird in die Relevanzrangfolge aufgenommen, indem sie mit der Wahrscheinlichkeit, die dem Element dort zugeordnet ist, durch den probabilistischen Operator and verknüpft wird und das Ergebnis dieser Operation dem Element als neue Wahrscheinlichkeit in der Relevanzrangfolge zugewiesen wird.

(3) Falls der select-from-where-Ausdruck das Schlüsselwort order by enthält, werden diejenigen Tupel, denen in der in (2) entstandenen Relevanzrangfolge dieselbe Wahrscheinlichkeit zugeordnet ist, unter Verwendung der Funktionen $\mathsf{f}_1, \mathsf{f}_2, ..., \mathsf{f}_q$ lexikographisch sortiert. Die lexikographische Ordnung anhand einer Menge von Funktionen wird wie folgt durchgeführt: Zunächst wird die zu sortierende Menge anhand der Funktion f_1 sortiert. Alle Elemente, die für f_1 denselben Wert tragen, werden dann anhand der Funktion f_2 sortiert u.s.w.

(4) Auf jedes Tupel der Relevanzrangfolge wird nun die Funktion

$\mathsf{f}(\mathsf{X}_1, \mathsf{X}_2, ..., \mathsf{X}_{n-1}, \mathsf{X}_n, \mathsf{X}_{n+1}, \mathsf{X}_{n+2}, ..., \mathsf{X}_{n+p})$

angewendet. Falls f die Funktion "$*$" ist, bleibt die Relevanzrangfolge[6] aus (3) erhalten.

(5) Falls der select-from-where-Ausdruck das Schlüsselwort number, nicht jedoch das Schlüsselwort lowest prob enthält, wird auf die in (5) entstandene Relevanzrangfolge die Operation set_number (vgl. Abschnitt 5.5.6.6) angewendet. Falls der select-from-where-Ausdruck das Schlüsselwort lowest prob, nicht jedoch das Schlüsselwort number enthält, wird auf die Relevanzrangfolge die Operation set_lowest_prob (vgl. Abschnitt 5.5.6.6) angewendet. Falls der select-from-where-Ausdruck sowohl das Schlüsselwort number als auch das Schlüsselwort lowest prob enthält, so wird diejenige der Operationen set_number und set_lowest_prob auf die Relevanzrangfolge angewendet, die die kleinere Relevanzrangfolge zum Ergebnis hat.

Zusammenfassend kann über den Typ des Resultats einer vagen select-from-where-Anfrage folgendes gesagt werden:

→ Unabhängig von den Typen der Kollektionen im from-Teil des Ausdrucks und unabhängig vom Typ des Prädikats im where-Teil ist das Resultat immer eine Relevanzrangfolge.

[6] Dadurch daß Relevanzrangfolgen von vornherein keine Duplikate enthalten, verliert das Schlüsselwort distinct in vagen Anfragen seine Bedeutung. Um eine einheitliche Nutzung der Sprache ProQL zu unterstützen, kann distinct dennoch in vagen Anfragen verwendet werden.

→ Falls der from-Teil keine Relevanzrangfolge enthält und das Prädikat im where-Teil einem Ausdruck vom Typ boolean entspricht, so kann eine Anfrage dennoch vage sein, falls sie die Schlüsselworte number oder lowest prob enthält. Auch das Resultat einer solchen vagen Anfrage ist vom Typ rankedList. Es enthält jedoch nur Elemente, denen entweder die Wahrscheinlichkeit 0.0 oder die Wahrscheinlichkeit 1.0 zugeordnet ist. Die Teilmenge der Elemente mit der zugeordneten Wahrscheinlichkeit 1.0 entspricht semantisch dem Ergebnis einer eindeutigen select-from-where-Anfrage, in der das Schlüsselwort distinct vorkommt und die dieselben Kollektionen, Funktionen und Bindungen verwendet wie die vage Anfrage.

Beispiel:

```
select p.name
from projekte p
     p.beschrieben_von q
where q.handelt_von("Datenbanken")
```

ergibt eine Relevanzrangfolge mit den Namen von Projekten, deren Projektbibliothek Publikationen enthält, die das Thema "Datenbanken" behandeln.

7.4.10 Ausdrücke für geordnete Kollektionen

Die Sprache ProQL ermöglicht den Zugriff auf ein oder mehrere Elemente einer geordneten Kollektion anhand ihrer Position. Auf Listen und Feldern ist die Konkatenation definiert. Des weiteren können Elemente in Verzeichnissen über ihren Schlüssel und Elemente in Relevanzrangfolgen über ihre Wahrscheinlichkeit ermittelt werden.

Der Zugriff auf das i-te Element einer Liste oder eines Feldes

Falls e_1 ein Ausdruck vom Typ list(t) oder array(t) und e_2 ein Ausdruck vom Typ int ist, so ist $e_1[e_2]$ ein Ausdruck vom Typ t. Dieser Ausdruck ermittelt das e_2+1. Element der geordneten Kollektion e_1. Dabei ist zu beachten, daß das erste Element einer Liste oder eines Feldes stets die Position 0 einnimmt.

Beispiel:

```
list(a,b,c,d)[1]
```

ergibt b.

Der Zugriff auf die Elemente einer Relevanzrangfolge an der Position i in Erweiterung der Sprache OQL

Falls e_1 ein Ausdruck vom Typ rankedList(t) und e_2 ein Ausdruck vom Typ int ist, so ist $e_1[e_2]$ ein Ausdruck vom Typ set(t). Dieser Ausdruck ermittelt alle Elemente, die die e_2+1.

Position der Relevanzrangfolge e_1 einnehmen. Dabei ist zu beachten, daß die Positionen einer Relevanzrangfolge durch mehrere Elemente besetzt sein können und daß eine Relevanzrangfolge stets an der Position 0 beginnt.

Beispiel:

Sei Publikationsrangfolge ein benanntes Objekt, das die folgende Relevanzrangfolge beinhaltet:

Pos.	Element	Prob.
0	meine_publikation	0.9
1	fremde_publikation	0.8
	fremde_publikation_2	0.8
2	meine_publikation_2	0.5

So ergibt der Ausdruck

```
Publikationsrangfolge[1]
```

die Menge mit den Elementen fremde_publikation und fremde_publikation_2.

Der Zugriff auf eine Unterkollektion einer Liste oder eines Feldes

Falls e_1 ein Ausdruck vom Typ list(t) oder array(t) ist und e_2 und e_3 Ausdrücke vom Typ int sind, so ist $e_1[e_2:e_3]$ ein Ausdruck vom Typ list(t) bzw. vom Typ array(t). Dieser Ausdruck ermittelt diejenige Unterkollektion der geordneten Kollektion e_1, die an der Position e_2 beginnt und an der Position e_3 endet.

Beispiel:

```
list(a,b,c,d)[1:3]
```

ergibt list(b,c,d).

Der Zugriff auf eine Unterrelevanzrangfolge einer Relevanzrangfolge in Erweiterung der Sprache OQL

Falls e_1 ein Ausdruck vom Typ rankedList(t) ist und e_2 und e_3 Ausdrücke vom Typ int sind, so ist $e_1[e_2:e_3]$ ein Ausdruck vom Typ rankedList(t). Dieser Ausdruck ermittelt diejenige Unterrelevanzrangfolge der Relevanzrangfolge e_1, die an der Position e_2 beginnt und an der Position e_3 endet. Den Elementen der Unterrelevanzrangfolge sind dabei dieselben Wahrscheinlichkeiten zugeordnet wie in der Relevanzrangfolge e_1.

Beispiel:

```
Publikationsrangfolge[1:2]
```

ergibt die folgende Relevanzrangfolge:

Pos.	Element	Prob.
0	fremde_publikation	0.8
	fremde_publikation_2	0.8
1	meine_publikation_2	0.5

Der Zugriff auf das erste oder das letzte Element einer Liste oder eines Feldes

Falls e ein Ausdruck vom Typ list(t) oder array(t) und <op> ein Operator aus der Menge { first, last } ist, so ist <op>(e) ein Ausdruck vom Typ t. Dieser Ausdruck ermittelt das erste bzw. das letzte Element der geordneten Kollektion e.

Beispiel:

```
last(list(a,b,c,d))
```

ergibt d.

Der Zugriff auf die Elemente an der ersten oder letzten Position einer Relevanzrangfolge in Erweiterung der Sprache OQL

Falls e ein Ausdruck vom Typ rankedList(t) und <op> ein Operator aus der Menge { first, last } ist, so ist <op>(e) ein Ausdruck vom Typ set(t). Dieser Ausdruck ermittelt diejenigen Elemente, die die erste bzw. die letzte Position der Relevanzrangfolge e einnehmen. Dabei ist zu beachten, daß die Positionen einer Relevanzrangfolge durch mehrere Elemente besetzt sein können.

Beispiel:

```
first(Publikationsrangfolge)
```

ergibt eine Menge mit dem Element meine_publikation.

Die Konkatenation zweier Listen oder zweier Felder

Falls e_1 und e_2 Ausdrücke der Typen list(t_1) und list(t_2) bzw. der Typen array(t_1) und array(t_2) sind, wobei t_1 und t_2 zueinander kompatibel sind, so $e_1 + e_2$ ein Ausdruck vom Typ list(lub(t_1, t_2)) bzw. vom Typ array(lub(t_1, t_2)). Dieser Ausdruck berechnet die Konkatenation von e_1 und e_2.

Beispiel:

```
list(a,b) + list(c,d)
```

ergibt list(a,b,c,d).

Der Zugriff auf ein Element eines Verzeichnisses über seinen Schlüssel

Falls e_1 ein Ausdruck vom Typ dictionary(k,v) und e_2 ein Ausdruck vom Typ k ist, so ist $e_1[e_2]$ ein Ausdruck vom Typ v. Dieser Ausdruck ermittelt den Wert, der im Verzeichnis e_1 unter dem Schlüssel e_2 abgelegt ist.

Beispiel:

```
mein_verzeichnis["mein_wert"]
```

ergibt den Wert, der im Verzeichnis mein_verzeichnis mit dem Schlüssel "mein_wert" verknüpft ist.

Der Zugriff auf Elemente einer Relevanzrangfolge über ihre Wahrscheinlichkeit in Erweiterung der Sprache OQL

Falls e_1 ein Ausdruck vom Typ rankedList(t) und e_2 ein Ausdruck vom Typ prob ist, so ist e_1 with e_2 ein Ausdruck vom Typ set(t). Dieser Ausdruck ermittelt alle Elemente, denen in der Relevanzrangfolge e_1 die Wahrscheinlichkeit e_2 zugeordnet ist.

Beispiel:

```
Publikationsrangfolge with 0.5
```

ergibt eine Menge mit dem Element meine_publikation_2.

7.4.11 Binäre Mengenausdrücke

In der Sprache ProQL sind die bekannten Mengenoperationen sowie eine Operation zur Überprüfung der Teilmengenbeziehung definiert.

Die Vereinigung, der Durchschnitt und die Differenz

Falls e_1 ein Ausdruck vom Typ set(t_1) oder vom Typ bag(t_1) und e_2 ein Ausdruck vom Typ set(t_2) oder vom Typ bag(t_2) ist, wobei t_1 und t_2 zueinander kompatible Typen sind, und falls <op> einer der Operatoren aus { union, except, intersect } ist, so ist e_1<op>e_2 ein Ausdruck vom Typ set(lub(t_1, t_2)), wenn die Ausdrücke e_1 und e_2 beide vom Typ set sind. Ist mindestens einer der Ausdrücke e_1 oder e_2 vom Typ bag, so ist e_1<op>e_2 ein Ausdruck vom Typ bag(lub(t_1, t_2)). Der Ausdruck berechnet die mengentheoretischen Operationen der Vereinigung, der Differenz und des Durchschnitts.
Falls die Kollektionstypen der Operanden voneinander verschieden sind, d.h. falls e_1 eine Menge und e_2 eine Multimenge ist oder umgekehrt, so wird die Menge zunächst in eine Multimenge konvertiert. Das Ergebnis des Ausdrucks ist wiederum eine Multimenge.

Beispiel:

```
bag(2,2,3,3,3) except bag(2,3,3,3)
```

ergibt bag(2).

Die Teilmengenbeziehung

Falls e_1 und e_2 Ausdrücke vom Typ set oder bag mit zueinander kompatiblen Elementtypen sind und <op> ein Operator aus { <, <=, >, >= } ist, so ist e_1<op>e_2 ein Ausdruck vom Typ boolean.
Falls die Kollektionstypen der Operanden voneinander verschieden sind, d.h. falls e_1 eine Menge und e_2 eine Multimenge ist oder umgekehrt, so wird die Menge zunächst in eine Multimenge konvertiert.

Der Ausdruck $e_1 < e_2$ ist wahr, falls e_1 in e_2 enthalten ist, aber e_1 und e_2 nicht gleich sind.

Der Ausdruck $e_1 <= e_2$ ist wahr, falls e_1 in e_2 enthalten ist.

Der Ausdruck $e_1 > e_2$ ist wahr, falls e_2 in e_1 enthalten ist, aber e_1 und e_2 nicht gleich sind.

Der Ausdruck $e_1 >= e_2$ ist wahr, falls e_2 in e_1 enthalten ist.

Beispiel:

```
set(1,2,3) < set(3,4,2,1)
```

ergibt true.

7.4.12 Konversionsausdrücke

Für Kollektionen definiert die Sprache ProQL verschiedene Umwandlungsoperationen. Zusätzlich lassen sich Ausdrücke explizit typisieren.

Die Umwandlung einer einelementigen Menge in ein Element

Falls e ein Ausdruck vom Typ collection(t) ist, so ist element(e) ein Ausdruck vom Typ t. Dieser Ausdruck ergibt das Element der einelementigen Menge e. Falls e nicht einelementig ist, löst die Verwendung Ausdrucks eine Ausnahme aus.

Beispiel:

```
element(bag(a))
```

ergibt a.

Die Umwandlung einer Liste in eine Menge

Falls e ein Ausdruck vom Typ list(t) ist, so ist listtoset(e) ein Ausdruck vom Typ set(t). Dieser Ausdruck wandelt die Liste e in eine Menge um, indem er diejenige Menge bildet, die alle Elemente aus der Liste und nur diese Elemente enthält. Falls in der Liste e Duplikate vorhanden sind, wird hiervon jeweils nur ein Exemplar in die Menge übernommen.

Beispiel:

```
listtoset(list(a,b,c,d))
```

ergibt set(a,b,c,d).

Die Umwandlung einer Relevanzrangfolge in eine Menge in Erweiterung der Sprache OQL

Falls e ein Ausdruck vom Typ rankedList(t) ist, so ist rankedlisttoset(e) ein Ausdruck vom Typ set(t). Dieser Ausdruck wandelt die Relevanzrangfolge e in die Menge um, die alle Elemente aus der Relevanzrangfolge und nur diese Elemente enthält. Die Wahrscheinlichkeiten, die den Elementen in der Relevanzrangfolge e zugeordnet sind, gehen in der Menge verloren.

Beispiel:

```
rankedlisttoset(Publikationsrangfolge)
```

ergibt set(meine_publikation, fremde_publikation, fremde_publikation_2, meine_publikation_2).

Das Entfernen von Duplikaten

Falls e ein Ausdruck vom Typ col(t) ist, wobei col für den Typ set oder bag steht, so ist distinct(e) ein Ausdruck vom Typ set(t). Der Ausdruck ergibt die Kollektion e ohne mehrfach auftretende Elemente. Falls e ein Ausdruck vom Typ col(t) ist, wobei col für den Typ list oder array steht, so ist distinct(e) ein Ausdruck vom Typ col(t). Der Ausdruck ergibt die Kollektion e ohne mehrfach auftretende Elemente, wobei immer das erste Auftreten eines Elements berücksichtigt wird, während alle weiteren Auftreten ignoriert werden.

Beispiel:

```
distinct(list(a,b,b,c,a,c,d))
```

ergibt list(a,b,c,d).

Das Auflösen von Unterkollektionen innerhalb von Kollektionen

Die Typen col, col_1 und col_2 seien Platzhalter für einen der Typen set, bag, list oder array. Sei e ein Kollektionsausdruck vom Typ col_1, so ist flatten(e) ein Ausdruck. Dieser Ausdruck wandelt eine Kollektion vom Typ col_1, deren Elemente wiederum Kollektionen sind, in eine einfache Kollektion vom Typ col(t) um. Dabei ist zu beachten, daß diese Umwandlung nur die erste Ebene der Verschachtelung berührt.
Angenommen e sei vom Typ col_1<col_2<t>>, so gilt für das Resultat von flatten(e):

- Falls col_2 dem Typ set oder dem Typ bag entspricht, so wird die Vereinigung aller Elemente von col_1 durchgeführt. Es ergibt sich ein Reusltat vom Typ set<t> bzw. vom Typ bag<t>.

- Falls und col_1 und col_2 dem Typ list oder dem Typ array entsprechen, so wird die Konkatenation aller Elemente von col_1 der Ordnung von col_1 entsprechend durchgeführt. Duplikate bleiben hierbei erhalten. Es ergibt sich ein Resultat vom Typ col_2<t>.

- Falls col_1 dem Typ set bzw. bag und col_2 dem Typ array oder list entspricht, so werden alle Elemente von col_1 in Mengen bzw. Multimengen umgewandelt, die anschließend vereinigt werden. Es ergibt sich ein Resultat vom Typ set<t> bzw. bag<t>.

Beispiel:

```
flatten(list(set(1,2,3), set(3,4,5), set(7)))
```

ergibt set(1,2,3,4,5,7).

Die explizite Typisierung von Ausdrücken

Sei e ein Ausdruck vom Typ t und t' ein Typname. Seien t und t' vergleichbar, d.h. es gilt entweder t >= t' oder t <= t'. Dann ist (t')e ein Ausdruck vom Typ t'. Dieser Ausdruck hat zwei verschiedene Auswirkungen:

(1) Zur Übersetzungszeit wird für die Typüberprüfung die Information bereitgestellt, daß e als eine Instanz des Typs t' zu verstehen ist.

(2) Zur Laufzeit wird zugesichert, daß e tatsächlich vom Typ t' ist. Nur in diesem Fall liefert der Ausdruck e ein Ergebnis. In allen anderen Fällen wird eine Ausnahme ausgelöst.

Die explizite Typisierung erlaubt es dem Benutzer, Anfragen zu formulieren, die ohne Verwendung expliziter Typisierung als fehlerhaft typisiert zurückgewiesen würden.

7.4.13 Funktionsaufrufe

Falls f eine Funktion vom Typ $(t_1, t_2, ..., t_n \rightarrow t)$ ist und $e_1, e_2, ..., e_n$ Ausdrücke vom Typ $t_1, t_2, ..., t_n$ sind, so ist $f(e_1, e_2, ..., e_n)$ ein Ausdruck vom Typ t, dessen Wert dem Wert entspricht, der von der Funktion f bei Verwendung der Parameter $e_1, e_2, ..., e_n$ zurückgegeben wird, oder dem Objekt nil entspricht, falls die Funktion keinen Wert zurückgibt. Falls f eine Funktion ohne Parameter mit Rückgabetyp t ist, so ist f() ein Ausdruck vom Typ t, dessen Wert dem Wert entspricht, der von der Funktion f zurückgegeben wird, oder dem Objekt nil entspricht, falls die Funktion keinen Wert zurückgibt.
ProQL definiert nicht, in welcher Programmiersprache der Rumpf einer solchen Funktion implementiert sein soll. Dadurch kann die Funktionalität von ProQL erweitert werden, ohne die Sprache zu verändern.

7.4.14 Sichtbarkeitsregeln

Der from-Teil einer select-from-where-Anfrage führt explizit oder implizit Variablen ein, die verwendet werden, um die dort ebenfalls benannten Kollektionen Element für Element zu durchlaufen. Ein Beispiel für eine explizit deklarierte Variable ist

```
select ... from Personen p ... ,
```

während

```
select ... from Personen ...
```

das implizite Vorhandensein einer Variablen voraussetzt. Solche expliziten und impliziten Variablen sind innerhalb des gesamten select-from-where-Ausdrucks und auch in etwaigen Unterausdrücken sichtbar.
Die group by-Klausel einer select-from-where-Anfrage deklariert den Namen partition zusammen mit den explizit anzugebenden Namen für verschiedene Partitionsattribute. Diese Namen sind in den dazugehörigen select- und having-Klauseln und ihren Unterausdrücken sichtbar.
Innerhalb eines Sichtbarkeitsbereichs werden die Variablennamen verwendet, um Pfadausdrücke zu bilden, durch die auf Operationen, Relationen und Attribute von durch die Variablen bezeichneten komplexen Objekten zugegriffen wird. Ist eine Variable nur implizit vorhanden, so kann der Name der entsprechenden Kollektion direkt verwendet werden, um Pfadausdrücke zu bilden. So ist der Ausdruck Personen.alter in Zusammenhang mit dem zweiten Beispiel oben ein korrekter Pfadausdruck.
Falls keine Mehrdeutigkeiten vorhanden sind, kann der Variablenname weggelassen und auf eine Eigenschaft eines komplexen Objekts direkt über ihren Namen zugegriffen werden. Mehrdeutigkeiten sind genau dann nicht vorhanden, wenn keine weiteren Objekte im Sichtbarkeitsbereich existieren, für die eine Eigenschaft gleichen Namens deklariert ist.
Zusammenfassend wird ein Name in einer Anfrage wie folgt aufgefaßt:

- ein Variablenname im aktuellen Sichtbarkeitsbereich oder

- eine benannte Anfrage, die durch die define-Klausel definiert wurde, oder
- ein benanntes Objekt oder
- ein Name eines Attributs, einer Relation oder einer Operation auf einer Variablen des aktuellen Sichtbarkeitsbereichs, falls es keine Mehrdeutigkeiten gibt, d.h. falls der Name nur auf eine einzige Variable des aktuellen Sichtbarkeitsbereichs angewendet werden kann.

8. Eine prototypische Implementation der Anfragesprache ProQL in TL2

Um zu überprüfen, ob die Konzepte des Objektmodells ProM (vgl. Kapitel 5) und der Anfragesprache ProQL in der Praxis einzuhalten sind, wird im praktischen Teil der vorliegenden Arbeit ein Prototyp entwickelt. Dieser realisiert eine Benutzungsschnittstelle, über die ausgewählte ProQL-Konstrukte zur eindeutigen und vagen Suche auf einem Testdatenbestand abgesetzt werden können.
Die hierfür verwendete Programmierumgebung ist das System Tycoon-2, das in einer Kooperation zwischen dem Arbeitsbereich Softwaresysteme an der Technischen Universität Hamburg-Harburg und dem Hamburger Unternehmen Higher-Order Informations- und Kommunikationssysteme GmbH entstanden ist.

8.1 Der Umfang der Implementation

Der implementierte Prototyp realisiert eine Schnittstelle zur Formulierung eindeutiger und vager select-from-where-Anfragen in der Anfragesprache ProQL, anhand derer gezeigt wird, daß die Konzepte des Modells ProM und der Sprache ProQL sinnvoll innerhalb praktischer Anwendungen einsetzbar sind.
Im Vordergrund steht hierbei die Implementation der Datentypen rankedList und prob, wie sie im Objektmodell ProM definiert sind, und des Algorithmus zur Auswertung einer select-from-where-Anfrage. Auf die Verwendung von Datenbanktechnologie wird hierbei zunächst verzichtet. Stattdessen wird der persistente Objektspeicher des Tycoon-2-Systems zur Ablage von Nutz- und Programmdaten verwendet. Um den Prototypen einfach zu halten, werden anstelle einer Anbindung ausgereifter Information Retrieval Algorithmen einfache vage Prädikate ausprogrammiert. Die hierfür benötigten Metadaten werden ebenfalls vom Tycoon-2-System verwaltet.
Als Datenbasis werden ungefähr 100 bibliographische Datensätze verwendet, die wissenschaftliche Veröffentlichungen beschreiben und auch die dazugehörigen Zusammenfassungen beinhalten. Diesen Daten wurden in der Extension des Datentyps Document abgespeichert, der folgendermaßen aufgebaut ist:

```
class Document
{
   attribute Text title;
   attribute set<Name> authors;
   attribute VagueDate publicationDate;
   attribute Text abstract;
   vague isAbout(string aString);
   vague vagueWrittenBy(string aString);
   vague appearedAbout(int aDay, aMonth, aYear);
};
```

Die für die Spezifikation der Attribute verwendeten Typen Text, Name und VagueDate sind ebenfalls im Rahmen des Prototypen implementiert worden und definieren ihrerseits einfache vage Prädikate, auf die die vagen Prädikate des Typs Document in ihrer Implementation zurückgreifen. Das vage Prädikat isAbout ermittelt die Wahrscheinlichkeit dafür, daß ein Dokument ein im Parameter aString angegebenes Thema behandelt. Das vage Prädikat vagueWrittenBy berechnet, wie wahrscheinlich es ist, daß ein Dokument von einem Autoren verfaßt wurde, dessen Name dem Parameter aString gleicht. Das Resultat des vagen Prädikats appearedAbout ist die Wahrscheinlichkeit, daß ein Dokument um einen bestimmten Zeitpunkt herum erschienen ist, der durch die drei Parameter angegeben wird.

8.2 Tycoon-2 als Basis der Implementation

Die Programmierumgebung Tycoon-2 bietet verschiedene Konzepte, die die Implementation des Prototypen unterstützen. So ist Tycoon-2 ein persistentes Objektsystem, das unter anderem eine objektorientierte Programmiersprache umfaßt[1]. Die objektorientierten Konzepte aus ProM und ProQL können daher direkt übernommen werden.
Der für ProQL geforderte parametrische Polymorphismus ist ebenfalls in Tycoon-2 realisiert. Er kann bei der Definition von Datentypen jederzeit eingesetzt werden. Für den Prototypen kam er insbesondere bei der Realisierung des Datentyps RankedList zum tragen, dessen Elemente von beliebigem Typ sein können.
Die durch das Tycoon-2-System gegebene orthogonale Persistenz ermöglicht es, für den Prototypen die Existenz eines Datenbanksystems zu simulieren. Alle Programmvariablen und auch die Nutzdaten werden dabei persistent im Objektspeicher des Systems abgelegt und können nach einem Neustart des Systems weiterhin zugegriffen werden.
Um die den vagen Prädikaten zugrundeliegenden Algorithmen dynamisch ändern zu können, wird von Möglichkeit Gebrauch gemacht, in der Sprache Tycoon-2 Funktionen höherer Ordnung zu implementieren. Die Parameter von Funktionen höherer Ordnung sind wiederum Funktionen. Im Falle des Prototypen werden Funktionen höherer Ordnung genutzt, die mit der Funktion parametrisiert sind, die den jeweiligen Auswertungsalgorithmus für ein vages Prädikat realisiert.

[1] Die zum Tycoon-2-System gehörende Programmiersprache wird ebenfalls Tycoon-2 genannt.

Die Gestaltung der Benutzungsoberfläche kann in Tycoon-2 durch die Verwendung des SGML-Dialektes *STML (Structured Tycoon Markup Language)* erfolgen. STML erlaubt es, Internetseiten zu entwerfen, innerhalb derer Tycoon-2-Funktionen direkt aufgerufen werden.

8.3 Das Ergebnis der prototypischen Implementation

Über die entwickelte Schnittstelle wurden verschiedene vage Anfragen an den Prototypen abgesetzt, in deren from-Teil jeweils die Extension des Typs Document (vgl. Abschnitt 8.1) eingesetzt wurde. Im where-Teil wurde eines der für Document spezifizierten vagen Prädikate verwendet. Schließlich erfolgte im select-Teil eine Projektion auf einzelne Attribute des Typs Document. Aus den so formulierten Anfragen resultierten stets die erwarteten Relevanzrangfolgen.
Hieraus läßt sich schließen, daß die Konzepte des Modells ProM und der Sprache ProQL in einer objektorientierten Umgebung prinzipiell direkt umsetzbar sind. Dabei sollte jedoch nicht außer acht gelassen werden, daß es sich bei dem beschriebenen Prototypen lediglich um eine eingeschränkte Implementation handelt. Es bleibt zu überprüfen, ob sich der Prototyp ebenfalls für Anfragen bewährt, die sich über mehrere Kollektionen erstrecken, vage und eindeutige Prädikate kombinieren oder beliebige Pfadausdrücke enthalten. Des weiteren muß untersucht werden, unter welchem Aufwand Teile der Funktionalität aus bestehenden Datenbank- oder Information Retrieval Systemen in den Prototypen integriert werden können und wie sich eine solche Integration auf die Qualität der Anfrageergebnisse auswirkt.

9. Zusammenfassung

In Datenbank- und Information Retrieval Systemen werden jeweils unterschiedliche Konzepte hinsichtlich der Verwaltung von Daten und der Formulierung von Anfragen auf diesen Daten verfolgt. Im Information Retrieval wird der Schwerpunkt darauf gesetzt, Anfragen auf Daten mit einer vagen Semantik auswerten zu können. Hierzu werden aufwendige Algorithmen entwickelt, die auf umfangreiche Metadaten zurückgreifen. Alle weiteren Aspekte, wie beispielsweise die Verteilung oder der Mehrbenutzerbetrieb, treten hierbei in den Hintergrund (vgl. Kapitel 2).
Im Gegensatz hierzu verwalten Datenbanksysteme einen fest umrissenen Ausschnitt des Anwendungsfeldes in Daten, die innerhalb des Systems eine eindeutige Semantik besitzen. Diese Daten sind persistent gespeichert und werden durch das Datenbankmanagementsystem so verwaltet, daß darauf ein effizienter, nebenläufiger Mehrbenutzerbetrieb in allen seinen Konsequenzen stattfinden kann (vgl. Kapitel 3).
Aus der Integration von Datenbank- und Information Retrieval Systemen sollte ein System hervorgehen, das die Möglichkeit bietet, Daten mit eindeutiger und mit vager Semantik einheitlich zu verarbeiten. Alle Stärken eines herkömmlichen Datenbanksystems sollten hierbei für das integrierte System weitgehend erhalten bleiben (vgl. Kapitel 4).
Die Integration läßt sich auf bereits vorhandenen Konzepten, Methoden und Technologien aufbauen. So kann die Architektur des integrierten Systems ebenso wie die eines Datenbanksystems in eine interne, eine externe und eine konzeptuelle Ebene unterteilt werden. Diese Ebenen erfüllen alle Aufgaben, die ihnen auch innerhalb des Datenbanksystems zukommen. Zusätzlich werden auf der internen Ebene die Information Retrieval Algorithmen implementiert und die dazugehörigen Metadaten verwaltet. Auf der konzeptuellen Ebene wird mit dem Objektmodell ProM und der dazugehörigen Definitionssprache ProDL ein Werkzeug zur Repräsentation von Daten mit allen ihren vagen und eindeutigen Eigenschaften sowie mit ihren Beziehungen untereinander bereitgestellt. Auf der externen Ebene des integrierten Systems ist mit der Anfragesprache ProQL ein potentiell orthogonaler Zugriff auf alle Daten gegeben, die mit Hilfe der Konstrukte der Datendefinitionssprache ProDL beschrieben werden können.
Neben der 3-Ebenen-Architektur für Datenbanken wird der ODMG-Standard als wichtige Grundlage für die Integration genutzt. Die in ihm verwirklichten objektorientierten Konzepte erleichtern seine Erweiterung hinsichtlich der Verwirklichung von Information Retrieval Anforderungen. So kann das Objektmodell ProM aus dem Objektmodell des Standards abgeleitet werden (vgl. Kapitel 5). Die Sprachen ProDL und ProQL entstehen durch den syn-

taktischen und semantischen Ausbau der im ODMG-Standard enthaltenen Sprachen ODL bzw. OQL (vgl. Kapitel 6 und 7).
Die Anfragesprache ProQL kann auf der Basis des Objektmodells ProM unter Verwendung einer objektorientierten Programmiersprache implementiert werden. Eine solche Implementation ist prototypisch innerhalb der Programmierumgebung Tycoon-2 vorgenommen worden.

10. Ausblick

Die vorliegende Arbeit setzt ihren Schwerpunkt auf die konzeptuelle und die externe Ebene von integrierten Datenbank- und Information Retrieval Systemen. Auf der internen Ebene hingegen steht die Realisierung konkreter Verfahren noch aus. Zum einen ist hier die Anbindung bereits bestehender Algorithmen und Systemkomponenten aus beiden Bereichen zu verfolgen. Zum anderen sind insbesondere Algorithmen zu entwerfen, die Information Retrieval Daten und die dazugehörigen Metadaten im Mehrbenutzerbetrieb effizient verwalten können. Dabei kann teilweise auf im Bereich Information Retrieval bereits entwickelte Verfahren zurückgegriffen werden (vgl. hierzu beispielsweise [vR79] oder [BCC94b]).
Durch die Integration von Datenbank- und Information Retrieval Systemen ergeben sich Aspekte, aus denen unter Umständen zusätzliche Informationen für die Auswertung vager Anfragen gewonnen werden können. So werden beispielsweise Daten mit vager Semantik strukturiert dargestellt und außerdem auch untereinander in Beziehung gesetzt. Es bleibt zu untersuchen, ob sich Erweiterungen dieser Art auch auf Information Retrieval Algorithmen auswirken und zu deren Verbesserung beitragen können.
Auf der externen Ebene herkömmlicher Datenbanksysteme werden individuelle Sichten auf die verwalteten Daten angelegt. Hier sollte untersucht werden, inwieweit ein solches Konzept auf integrierte Datenbank- und Information Retrieval Systeme übertragbar ist und wo es sinnvoll erweitert werden kann. So ist das Anlegen umfangreicher Benutzerprofile denkbar, die beispielsweise häufig gestellte Anfragen oder Aufzeichnungen vorangehender Sitzungen mit dem System enthalten und so zu einer effizienteren und effektivere Anfrageauswertung können.
Zudem sollte die allgemeine Fragestellung nach möglichen Anwendungsfeldern eines integrierten Datenbank- und Information Retrieval Systems nicht aus den Augen verloren werden, insbesondere im Hinblick auf die speziellen Anforderungen, mit denen ein integriertes System innerhalb eines solchen Anwednungsfeldes konfrontiert werden kann.

A. Beispiel zur Verdeutlichung der Konzepte von ProM, ProDL und ProQL

Zur Verdeutlichung der Konzepte des Objektmodells ProM, der Defintionssprache ProDL und der Anfragesprache ProQL werden im Laufe der vorliegenden Arbeit immer wieder Beispiele angeführt. Um etwaigen Unklarheiten vorzubeugen und die allgemeine Verständlichkeit zu erhöhen, wird hier ein exemplarisches Datenbankschema in ProDL-Notation vorgestellt, aus dem sich der Großteil der im Text angegebenen Beispiele rekrutiert. Dieses Kapitel stellt somit eine Referenz dar, auf die zurückgegriffen werden kann, falls ein Beispiel in seinem Gesamtkontext betrachtet werden soll.
Den Kern des Beispielschemas bildet der Objekttyp Publikation, der eine Menge von Veröffentlichungen repräsentiert, die als Quellen zur wissenschaftlichen Arbeit dienen:

```
class Publikation
   (extent publikationen
    key titel)
   {
    attribute string titel;
    attribute date erscheinungsdatum;
    attribute Text zusammenfassung;
    attribute Text text;
    attribute set<struct Kommentarfeld {Text kommentar;
       Mitarbeiter von_wem}> kommentare;
    relationship set<Autor> verfasst_von
       inverse Autor::verfasst;
    relationship set<Projekt> verwendet_in
       inverse Projekt::verwendet;
    relationship rankedList<Projekt> betrifft
       inverse Projekt::beschrieben_von;
    relationship rankedList<Publikation> erweitert
       inverse Publikation::erweitert_von;
    relationship rankedList<Publikation> erweitert_von
```

```
        inverse Publikation::erweitert
        {rank by Publikation::vertieft};
    vague ist_aktuell();
    vague handelt_von (string thema);
    vague vertieft (Publikation publikation)
        raises (publikation_nicht_vorhanden);
    vague ist_quelle_fuer (Projekt projekt)
        raises (projekt_nicht_vorhanden);
    void kommentiere (in Text kommentar);
    void erstelle_verweis (inout Datei bibtex)
        raises (keine_bibtex_datei);
};
```

Die Extension des Typs Publikation heißt publikationen. Publikationen besitzen einen Titel, anhand dessen sie auch eindeutig identifizierbar sind. Weitere Attribute sind das Erscheinungsdatum der Publikation, eine Zusammenfassung sowie der eigentliche Text. Außerdem gibt es zu einer Publikation eine Menge von Kommentaren, in die in Verbindung mit dem jeweiligen Urheber des Kommentars abgelegt werden. Für den Großteil dieser Attribute wird der Typ Text als gegeben angenommen, der eine gängige Repräsenation von Volltexten beliebiger Länge darstellt und hier nicht weiter spezifiziert werden soll.
Eine Publikation steht mit ihren Autoren in der eindeutigen Relation verfasst_von/verfasst. Des weiteren kann über die eindeutige Relation verwendet_in/verwendet ermittelt werden, in welchen Forschungsprojekten eine Publikation als Literatur verwendet wird. Zusätzlich besteht zwischen Publikationen und Projekten auch die vage Relation betrifft/beschrieben_von, die für verschiedene Projekte ausdrückt, wie wahrscheinlich der Inhalt einer Publikation diese Projekte betrifft. Die vage Relation erweitert/erweitert_von schließlich setzt Publikationen reflexiv zueinander in Beziehung, indem sie darstellt, wie wahrscheinlich diese sich gegenseitig erweitern.
Verschiedene vage Prädikate sind für den Typ Publikation definiert. So wird ist_aktuell verwendet, um zu ermitteln, mit welcher Wahrscheinlichkeit eine Publikation zum Zeitpunkt des Aufrufs noch von Bedeutung ist. Durch handelt_von kann berechnet werden, wie wahrscheinlich ein beliebiges Thema in einer Publikation behandelt wird. Mit vertieft wird die Wahrscheinlichkeit ausgedrückt, mit der eine Publikation eine andere vertieft. Dieses vage Prädikat ist Grundlage für die Auswertung der vagen Relation erweitert/erweitert_von. Als Basis der vagen Relation betrifft/beschrieben_von dient das vage Prädikat ist_quelle_fuer, das ermittelt, wie wahrscheinlich eine Publikation ein bestimmtes Forschungsprojekt betrifft.
Schließlich sind für Publikationen zwei Operationen definiert. Die Operation kommentiere ermöglicht es, einen neuen Kommentar, der der Operation als Parameter übergeben wird, in die Kommentarliste einer Publikation einzufügen. Die Operation erstelle_verweis generiert für eine Publikation einen entsprechenden Eintrag in einer Bibtexdatei, die als Variablen-Wert-Parameter übergeben wird. Falls die übergebene Datei nicht im Bibtexformat vorliegt, wird eine Ausnahme ausgelöst.
Publikationen lassen sich weiter aufteilen in Artikel und Bücher. Für diese beiden Publikati-

onsformen wird daher jeweils ein Typ definiert, der alle Eigenschaften des Typs Publikation erbt und daher über die EXTENDS-Relation mit diesem verbunden ist:

```
class Artikel extends Publikation
(extent artikel)
{
attribute string zeitschrift;
attribute unsigned short band;
attribute unsigned short nummer;
attribute unsigned short von_seite;
attribute unsigned short bis_seite;
void erstelle_verweis (inout Datei bibtex)
   raises (keine_bibtex_datei);
};

class Buch extends Publikation
(extent buecher)
{
attribute string isbn;
relationship Verlag erschienen_bei inverse Verlag::verlegt;
void erstelle_verweis (inout Datei bibtex)
   raises (keine_bibtex_datei);
};
```

Die Operation erstelle_verweis wird sowohl in Artikel als auch in Buch überschrieben, so daß bei einem Aufruf der Operation für ein Objekt des jeweiligen Typs eine speziell für diesen Typ implementierte Methode zur Ausführung gelangt.
Für Objekte des Typs Artikel werden einige zusätzliche Attribute spezifiziert, die Informationen darüber enthalten, in welcher Zeitschrift, in welcher Ausgabe und an welcher Stelle der entsprechende Artikel erschienen ist. Für Objekte des Typs Buch ist ein Attribut spezifiziert, das die ISBN des Buches beinhaltet. Des weiteren besteht für ein Buch eine binäre 1:n-Relation zu dem Verlag, bei dem das Buch herausgegeben wurde.
Die Objekttypen, die in den Spezifikationen der Typen Publikation, Artikel und Buch erscheinen, sollen nun ebenfalls definiert werden. Zum einen werden Publikationen im Zusammenhang mit verschiedenen Forschungsprojekten betrachtet, für die der Typ Projekt wie folgt festgelegt wird:

```
class Projekt
(extent projekte
 key name)
{
attribute string name;
attribute Mitarbeiter leiter;
```

```
attribute set<Mitarbeiter> beteiligte;
attribute set<string> schlagworte;
relationship set<Publikation> verwendet
   inverse Publikation::verwendet_in
relationship rankedList<Publikation> beschrieben_von
   inverse Publikation::betrifft
   {rank by Publikation::ist_quelle_fuer};
void erweitere_schlagwortliste (in string schlagwort,
   in Mitarbeiter von_wem)
   raises (keine_berechtigung);
};
```

Jedes Forschungsprojekt hat einen Namen, anhand dessen es eindeutig identifizierbar ist. Weitere Attribute legen einen Mitarbeiter fest, der das Projekt leitet, sowie eine Menge von Mitarbeitern, die ebenfalls an dem Projekt beteiligt sind. Außerdem ist einem Projekt eine Menge von Schlagwörtern zugeordnet, die in Zusammenhang mit dem betrachteten Forschungsgebiet stehen. Diese Schlagwortliste kann mit Hilfe der Operation erweitere_schlagwortliste ergänzt werden. Als Parameter müssen hierbei das zu ergänzende Schlagwort sowie der Mitarbeiter, der die Ergänzung vornimmt, angegeben werden. Falls der angegebene Mitarbeiter nicht berechtigt ist, die Schlagwortliste zu erweitern, wird eine Ausnahme ausgelöst. Als Gegenstück des Traversierungspfades verwendet_in in der Spezifikation des Typs Publikation wird in der Spezifikation des Typs Projekt der Traversierungspfad verwendet definiert, so daß insgesamt die n:m-Relation verwendet/verwendet_in entsteht. verwendet gibt für ein Projekt an, welche Publikationen dort als Literatur dienen. Auf ähnliche Weise wird der Traversierungspfad betrifft in der Spezifikation des Typs Publikation durch den vagen Traversierungspfad beschrieben_von zur vagen Relation betrifft/beschrieben_von ergänzt. beschrieben_von gibt für ein Projekt an, welche Publikationen mit welcher Wahrscheinlichkeit Inhalte des Projektes beschreiben.
Mit den Autoren einer Publikation und den an einem Forschungsprojekt beteiligten Mitarbeitern sind bei der bisherigen Definition des Datenbankschemas bereits zwei verschiedene Personengruppen aufgetreten. Um die dazugehörigen Objekte entsprechend definieren zu können, wird zunächst der Typ Person eingeführt:

```
class Person
(extent Personen)
{
attribute string name;
attribute string vorname;
attribute Adresse adresse;
};
```

Eine Person besitzt einen Namen, einen Vornamen und eine Adresse, die durch ein atomares Literal des folgenden Typs repräsentiert ist:

```
Struct Adresse {
   string strasse;
   unsigned short number;
   string plz;
   string stadt
};
```

Sowohl Autoren als auch Mitarbeiter sind Personen, so daß entsprechende Typen, die dem Typ Person durch die EXTENDS-Relation untergeordnet sind, definiert werden können:

```
class Autor extends Person
(extent autoren)
{
relationship set<Publikation> verfasst
   inverse Publikation::verfasstVon;
};

class Mitarbeiter extends Person
(extent mitarbeiter)
{
attribute Abteilung abteilung;
attribute set<string> telefonnummern;
attribute unsigned short alter();
};
```

Der Typ Autor definiert zusätzlich zu den vom Typ Person geerbten Attributen den Traversierungspfad verfasst, der zusammen mit dem Traversierungspfad verfasst_von des Typs Publikation die binäre n:m-Relation verfasst_von/verfasst ergibt. Diese beschreibt, welche Publikationen von welchen Autoren verfaßt wurden. Objekte des Typs Mitarbeiter besitzen neben den Attributen des Supertyps Person weitere Attribute. Der Objekttyp Abteilung für das Attribut, das die Abteilung festlegt, in der der jeweilige Mitarbeiter beschäftigt ist, wird hier nicht näher spezifiziert. Ein Mitarbeiter kann unter einer Menge von Telefonnummern erreichbar sein. Sein Alter wird im Attribut alter durch eine Operation definiert, die das Alter des Mitarbeiters bei jedem Zugriff aktuell berechnet (vgl. hierzu Abschnitt 5.8.1).
Bei der Spezifikation des Typs Buch wurde außerdem der Objekttyp Verlag verwendet, der hier der Vollständigkeit halber ebenfalls spezifiziert wird:

```
class Verlag
(extent verlage)
{
attribute string name;
attribute Adresse adresse;
relationship set<Buch> verlegt
```

```
    inverse erschienen_bei;
};
```

Neben den Attributen name und adresse ist hier der Traversierungspfad verlegt definiert, der zusammen mit seinem Gegenstück erschienen_ei in der Spezifikation des Typs Buch eine binäre 1:n-Relation bildet, die jedem Buch den Verlag zuordnet, bei dem es herausgegeben wurde.

B. Die Syntax von ProDL

Die gesamte Syntax der Definitionssprache ProDL wird im Folgenden in Backus-Naur-Form aufgeführt. Hierbei werden sowohl die Syntaxregeln der Definitionssprache IDL aus der OMG CORBA Spezifikation [OMG95] als auch die Syntaxregeln von ODL, die durch den ODMG-Standard [Cat97] vorgegeben sind, übernommen.
Die Numerierung der Regeln wurde aus der ODMG-Spezifikation übernommen. Dort wurden Veränderungen der OMG CORBA Spezifikation durch Sterne, z.B. (5∗), gekennzeichnet. Hinzugefügte Regeln wurden mit Kleinbuchstaben versehen, wie z.B. (5a).
Veränderungen einzelner Regeln, die im Rahmen der Entwicklung von ProDL stattgefunden haben, sind in der folgenden Syntaxbeschreibung durch Pluszeichen, z.B. (9∗+), kenntlich gemacht worden. Wurden ganze Regeln hinzugefügt, so sind diese innerhalb der Syntaxbeschreibung durch numerische Suffixe markiert, wie z.B. (24.1). Die Regeln ab Nummer (75) bis Ende sind ergänzt worden, ohne dieses besonders kenntlich zu machen.

(1) <specification> ::= <definition>

| <definition><specification>

(2) <definition> ::= <type_dcl>;

| <const_dcl>;

| <except_dcl>;

| <interface>;

| <module>;

| <class>;

(2a) <class> ::= <class_header> { <interface_body> }

(2b) <class_header> ::= **class** <identifier>

[**extends** <scoped_name>]

[<inheritance_spec>]

[<type_property_list>]

(2c) <type_property_list> ::=

([<extent_spec>][<key_spec>])

(2d) <extent_spec> ::= **extent** <string>

(2e) <key_spec> ::= **key[s]** <key_list>

(2f) <key_list> ::= <key> | <key> , <key_list>

(2g) <key> ::= <property_name> | (<property_list>)

(2h) <property_list> ::= <property_name>

| <property_name> , <property_list>

(2i) <property_name> ::= <identifier>

(3) <module> ::= **module** <identifier> { <specification> }

(4) <interface> ::= <interface_dcl>

| <forward_dcl>

(5) <interface_dcl> ::= <interface_header>

{ [<interface_body>] }

(6) <forward_dcl> ::= **interface** <identifier>

(7) <interface_header> ::= **interface** <identifier>

[<inheritance_spec>]

(8) <interface_body> ::= <export>

|<export><interface_body>

(9*+) <export> ::= <type_dcl>;

|<const_dcl>;

|<except_dcl>;

|<attr_dcl>;

|<rel_dcl>;

|<op_dcl>;

|<vague_pred_dcl>;

(10) <inheritance_spec> ::=

:<scoped_name>[, <inheritance_spec>]

(11) <scoped_name> ::= <identifier>

| ::<identifier>

| <scoped_name>::<identifier>

(12) <const_dcl> ::= **const** <const_type><identifier> **=**

<const_exp>

(13+) <const_type> ::= <integer_type>

|<char_type>

|<probability_type>

|<floating_pt_type>

|<string_type>

|<scoped_name>

(14) <const_exp> ::= <or_expr>

(15) <or_expr> ::= <xor_expr>

| <or_expr> | <xor_expr>

(16) <xor_expr> ::= <and_expr>

| <xor_expr> ∧ <and_expr>

(17) <and_expr> ::= <shift_expr>

| <and_expr> & <shift_expr>

(18) <shift_expr> ::= <add_expr>

| <shift_expr> ≫ <add_expr>

| <shift_expr> ≪ <add_expr>

(19) <add_expr> ::= <mult_expr>

| <add_expr> + <mult_expr>

| <add_expr> - <mult_expr>

(20) <mult_expr> ::= <unary_expr>

| <mult_expr> * <unary_expr>

| <mult_expr> / <unary_expr>

| <mult_expr> % <unary_expr>

(21) <unary_expr> ::= <unary_operator><primary_expr>

| <primary_expr>

(22) <unary_operator> ::= -

| +

| ~

(23) <primary_expr> ::= <scoped_name>

| <literal>

| (<const_exp>)

(24+) <literal> ::= <integer_literal>

| <string_literal>

| <charcter_literal>

| <floating_pt_literal>

| <prob_literal>

| <boolean_literal>

(24.1) <prob_literal> ::= **1.**

| **0.**{<digit>}

| <digit>.{<digit>}< scale_factor>

(24.2) <scale_factor> ::= **e+**<digit>{<digit>}

| **e–**<digit>{<digit>}

(25) <boolean_literal> ::= **TRUE**

| **FALSE**

(26) <positive_int_const> ::= <const_exp>

(27) <type_dcl> ::= **typedef** <type_declarator>

| <struct_type>

| <union_type>

| <enum_type>

(28) <type_declarator> ::= <type_spec><declarators>

(29) <type_spec> ::= <simple_type_spec>

| <constr_type_spec>

(30) <simple_type_spec> ::= <base_type_spec>

| <template_type_spec>

| <scoped_name>

(31*+) <base_type_spec> ::= <floating_pt_type>

| <integer_type>

| <char_type>

| <boolean_type>

| <prob_type>

| <octet_type>

| <any_type>

| <date_type>

| <time_type>

| <interval_type>

| <timestamp_type>

(31a) <date_type> ::= **date**

(31b) <time_type> ::= **time**

(31c) <interval_type> ::= **interval**

(31d) <timestamp_type> ::= **timestamp**

(32*) <template_type_spec> ::= <array_type>

| <string_type>

| <coll_type>

(32a) <coll_type> ::= <coll_spec> < <simple_type_spec> >

| **dictionary** < <simple_type_spec>,

<simple_type_spec> >

(32b+) <coll_spec> ::= **set** | **list** | **bag** | **rankedList**

(33) <constr_type_spec> ::= <struct_type>

| <union_type>

| <enum_type>

(34) <declarators> ::= <declarator>

| <declarator>, <declarators>

(35) <declarator> ::= <simple_declarator>

| <complex_declarator>

(36) <simple_declarator> ::= <identifier>

(37) <complex_declarator> ::= <array_declarator>

(38) <floating_pt_type> ::= **float**

| **double**

(39) <integer_type> ::= <signed_int>

| <unsigned_int>

(40) <signed_int> ::= <signed_long_int>

| <signed_short_int>

(41) <signed_long_int> ::= **long**

(42) <signed_short_int> ::= **short**

(43) <unsigned_int> ::= <unsigned_long_int>

| <unsigned_short_int>

(44) <unsigned_long_int> ::= **unsigned long**

(45) <unsigned_short_int> ::= **unsigned short**

(46) <char_type> ::= **char**

(47) <boolean_type> :: **boolean**

(47.1) <prob_type> ::= **prob**

(48) <octet_type> ::= **octet**

(49) <any_type> ::= **any**

(50) <struct_type> ::= **struct** <identifier> { <member_list> }

(51) <member_list> ::= <member>

| <member><member_list>

(52) <member> ::= <type_spec><declarators>

(53) <union_type> ::= **union** <identifier> **switch**

(<switch_type_spec>) { <switch_body> }

(54) <switch_type_spec> ::= <integer_type>

| <char_type>

| <boolean_type>

| <enum_type>

| <scoped_name>

(55) <switch_body> ::= <case>| <case><switch_body>

(56) <case> ::= <case_label_list><element_spec>;

(56a) <case_label_list> ::= <case_label>

| <case_label><case_label_list>

(57) <case_label> ::= **case** <const_exp> :

| **default :**

(58) <element_spec> ::= <type_spec><declarator>

(59) <enum_type> ::= **enum** <identifier> { <enumerator_list> }

(59a) <enumerator_list> ::= <enumerator>

|<enumerator>,<enumerator_list>

(60) <enumerator> ::= <identifier>

(61*) <array_type> ::= <array_spec> < <simple_type_spec>,

<positive_int_const> >

|<array_spec> < <simple_type_spec> >

(61a*) <array_spec> :: **array** | **sequence**

(62) <string_type> ::= **string** < <positive_int_const> >

| **string**

(63) <array_declarator> ::= <idenitifier><array_size_list>

(63a) <array_size_list> ::= <fixed_array_size>

|<fixed_array_size><array_size_list>

(64) <fixed_array_size> ::= [<positive_int_const>]

(65*) <attr_dcl> ::= [**readonly**] **attribute**

<domain_type><attribute_name>

[<fixed_array_size>]

(65a) <domain_type> ::= <simple_type_spec>

| <struct_type>

| <enum_type>

(65b+) <rel_dcl> ::= **relationship**

<target_of_path>

<identifier>

inverse <inverse_traversal_path>

[{ **rank by** <ranking_aspect> }]

(65c) <target_of_path> ::= <identifier>

| <rel_collection_type> < <identifier> >

(65d) <inverse_traversal_path> ::=

<identifier>::<identifier>

(65e+) <rel_collection_type> ::= **set** | **list**

| **bag** | **rankedList**

(65.1) <ranking_aspect> ::= <identifier>::<identifier>

(66) <except_dcl> ::= **exception** <identifier>

{ [<member_list>] }

(67) <op_dcl> ::= [<op_attribute>]<op_type_spec>

<identifier><parameter_dcls>

[<raises_expr>][<context_expr>]

(68) <op_attribute> ::= **oneway**

(69) <op_type_spec> ::= <simple_type_spec>

| **void**

(70) <parameter_dcls> ::= ([<param_dcl_list>])

(70a) <param_dcl_list> ::= <param_dcl>

|<param_dcl>, <param_dcl_list>

(71) <param_dcl> ::= <param_attribute><simple_type_spec>

<declarator>

(72) <param_attribute> ::= **in**

| **out**

| **inout**

(73) <raises_expr> ::= **raises** (<scoped_name_list>)

(73a) <scoped_name_list> ::= <scoped_name>

|<scoped_name>, <scoped_name_list>

(74) <context_expr> ::= **context** (<string_literal_list>)

(74a) <string_literal_list> ::= <string_literal>

|<string_literal>, <string_literal_list>

(75) <vague_pred_dcl> ::= **vague** <identifier><in_param_dcl_list>

[<raises_expr>]

(76) <in_param_dcl_list> ::= <in_param_dcl>

|<in_param_dcl>, <in_param_dcl_list>

(77) <in_param_dcl> ::= <simple_type_spec><declarator>

C. Die Syntax der Anfragesprache ProQL

In Anlehnung an den ODMG-Standard wird die Grammatik in einer Notation beschrieben, die der formalen BNF nur teilweise entspricht. Um dennoch eine eindeutige Interpretation der Ableitungen zu gewährleisten, werden zunächst die folgenden Regeln festgelegt:

- { Symbol } beschreibt eine Folge von 0 oder n Symbolen.
- *[*Symbol*]* beschreibt ein optionales Symbol. Die kursiven eckigen Klammern sind dabei nicht mit einfachen eckigen Klammern zu verwechseln, die als Teil der Sprache zur Trennung von Ausdrücken verwendet werden.
- **Schlüsselwort** ist ein Terminal der Grammatik.
- xxx_name ist die Syntax einer Bezeichners.
- xxx_literal ist selbsterklärend. Die Zeichenkette "Hallo" beispielsweise ist ein string_literal.
- bind_argument steht für einen Parameter, der durch Bindung an einen Ausdruck in einer Programmiersprache entsteht, falls eine Sprachanbindung von ProQL an diese Programmiersprache existiert.

Das Nonterminal query steht für einen gültigen Ausdruck der Sprache ProQL. Die vorliegende Grammatik stellt Ableitungen zur Verfügung, die einen rekursiven Aufbau solcher Ausdrücke erlauben. Daher erscheint das Nonterminal query häufig auf der linken Seite der Ableitungen.
Die Grammatik setzt eine korrekte Typisierung der ableitbaren Ausdrücke voraus, ohne sie in den Ableitungen zu gewährleisten. Wie die korrekte Typisierung von ProQL-Ausdrücken erfolgt, wird in Abschnitt 7.4 festgelegt.
Des weiteren werden die Ableitungen der Grammatik durch Prioritätsregeln für ProQL-Operatoren ergänzt. Diese werden im Anschluß an die Grammatik aufgeführt.
Ein großer Teil der Ableitungen wurde aus der Grammatik der Sprache OQL übernommen, die im ODMG-Standard niedergeschrieben ist. Die Stellen, an denen die OQL-Grammatik um neue Ableitungen ergänzt wurde, sind durch das Symbol + vor der neuen Ableitung

gekennzeichnet. Ableitungen der OQL-Grammatik, die im Rahmen der ProQL-Grammatik verändert wurden, sind durch das Symbol $*$ vor der eigentlichen Ableitung kenntlich gemacht.

C.1 Die Grammatik

Axiom

(vgl. Abschnitte 7.4.1 und 7.4.2)

query_program ::= {define_query;} query

define_query ::= **define** identifier **as** query

Grundlagen

(vgl. Abschnitt 7.4.3)

query ::= **nil**

query ::= **true**

query ::= **false**

query ::= integer_literal

query ::= float_literal

+query ::= prob_literal

query ::= character_literal

query ::= string_literal

query ::= entry_name

query ::= query_name

query ::= bind_argument

query ::= from_variable_name

query ::= (query)

Einfache Ausdrücke

(vgl. Abschnitt 7.4.5)

query ::= query + query[1]

query ::= query - query

query ::= query * query

query ::= query / query

query ::= - query

query ::= query **mod** query

query ::= **abs**(query)

query ::= query || query

Vergleich

(vgl. Abschnitt 7.4.5)

query ::= query comparison_operator query

query ::= query **like** string_literal

comparison_operator ::= =

comparison_operator ::= !=

[1] Der Operator + wird auch zur Konkatenation von Listen und Feldern verwendet

comparison_operator ::= >

comparison_operator ::= <

comparison_operator ::= >=

comparison_operator ::= <=

Boolesche und probabilistische Ausdrücke

(vgl. Abschnitt 7.4.5)

query ::= **not** query

query ::= query **and** query

query ::= query **or** query

Konstruktoren

(vgl. Abschnitt 7.4.4)

query ::= type_name (*[* query *]*)

query ::= type_name (identifier: query {, identifier: query})

query ::= **struct** (identifier: query {, identifier: query})

query ::= **set** (*[* query {, query} *]*)

query ::= **bag** (*[* query {, query} *]*)

query ::= **list** (*[* query {, query} *]*)

query ::= (query, query {, query})

query ::= *[* **list***]*(query .. query)

query ::= **array** (*[* query {, query} *]*)

Zugriff

(vgl. Abschnitte 7.4.6, 7.4.10 und 7.4.14)

query ::= query dot attribute_name

query ::= query dot relationship_name

query ::= query dot operation_name(query {, query})

dot ::= . | ->

query ::= * query

query ::= query [query]

+query ::= query **with** query

query ::= query [query:query]

query ::= **first** (query)

query ::= **array** (*[* query {, query} *]*)

Kollektionsausdrücke

(vgl. Abschnitt 7.4.7)

query ::= **for all** identifier **in** query: query

query ::= **exists** identifier **in** query: query

query ::= **exists**(query)

query ::= **unique**(query)

query ::= query **in** query

query ::= query comparison_operator quantifier query

quantifier ::= **some**

quantifier ::= **any**

quantifier ::= **all**

query ::= **count** (query)

query ::= **count** (∗)

query ::= **sum** (query)

query ::= **min** (query)

query ::= **max** (query)

query ::= **avg** (query)

select-from-where-Ausdrücke

(vgl. Abschnitte 7.4.8 und 7.4.9)

∗query ::= **select** *[* **distinct** *]* projection_attributes

from variable_declaration {, variable_declaration}

[**where** query *]*

[**number** query *]*

[**lowest prob** query *]*

[**group by** partition_attributes *]*

[**having** query *]*

[**order by** sort_criterion {, sort_criterion} *]*

projection_attributes ::= projection {, projection}

projection_attributes ::= ∗

projection ::= query

projection ::= identifier: query

projection ::= query **as** identifier

variable_declaration ::= query *[[***as***]* identifier *]*

variable_declaration ::= identifier **in** query

partition_attributes ::= projection {, projection}

sort_criterion ::= query *[*ordering*]*

ordering ::= **asc**

ordering ::= **desc**

Mengenausdrücke

(vgl. Abschnitt 7.4.11)

query ::= query **intersect** query

query ::= query **union** query

query ::= query **except** query

Konversionsausdrücke

(vgl. Abschnitt 7.4.12)

query ::= **listtoset**(query)

+query ::= **rankedlisttoset**(query)

+query ::= **rankedlisttoset**(query, query)

query ::= **element**(query)

query ::= **distinct**(query)

query ::= **flatten**(query)

query ::= (class_name) query

C.2 Die Reihenfolge bei der Operatorauswertung

Die folgenden Operatoren sind nach abnehmender Priorität geordnet. Operatoren in derselben Zeile besitzen dieselbe Priorität.

() [] . ->

not - (unr) + (unr)

in

* / **mod** **intersect**

\+ - **union** **except** ||

< > <= >= <**some** <**any** <**all** (auch für weitere Vergleichsoperatoren ...)

= != **like**

and **exists** **for all**

or

.. :

,

(identifier) Steht hier für die Typkonversion

order

having

group by

where

from

select

Literaturverzeichnis

Abi97 S. Abiteboul. Object Database Support for Digital Libraries. In *Proceedings of the First European Conference on Research and Advanced Technology for Digital Libraries*, pages 11–23. Springer-Verlag, 1997.

BCC94a J. Broglio, J. P. Callan, and W. B. Croft. INQUERY System Overview. In *Proceedings of the TIPSTER Text Programm (Phase 1)*, pages 47–67, San Francisco, 1994. Morgan Kaufmann.

BCC94b E.W. Brown, J.P. Callan, and W.B. Croft. Fast Incremental Indexing for Full-Text Information Retrieval. In *Proceedings of the 20th International Conference on Very Large Databases*, 1994.

BCCM94 E.W. Brown, J.P. Callan, W.B. Croft, and E. Moss. Supporting full-text information retrieval with a persistent object store. In *Proceedings of the 4th International Conference on Extending Database Technology*, pages 365–378, 1994.

Car96 L. Cardelli. Type Systems. In *CRC Handbook of Computer Science and Engineering*. CRC Press, September 1996.

Cat96 R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

Cat97 R.G.G. Cattell, editor. *Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, San Mateo, CA, 1997.

CCH92 J.P. Callan, W.B. Croft, and S.M. Harding. The INQUERY Retrieval System. In *Proceedings of the Third International Conference on Database and Expert Systems*, pages 78–83. Springer-Verlag, 1992.

CLC95 J.P. Callan, Z. Lu, and W.B. Croft. Searching Distributed Collections With Inference Networks. In *Proc. 18th ACM SIGIR Conf.* ACM, 1995.

CM96 B. Cahoon and K. S. McKinley. Performance Evaluation of a Distributed Architecture for Information Retrieval. In H.-P. Frei et al., editors, *Proc. 19th ACM SIGIR Conf.*, pages 110–118. ACM, 1996.

CO92 S. J. Cannan and G. A. M. Otten. *SQL - The Standard Handbook*. McGraw Hill, 1992.

Cro95 W.B. Croft. What Do People Want From Information Retrieval? *d-lib Magazine*, November 1995.

DvR93 M. Dunlop and C.J. van Rijsbergen. Hypermedia and free text retrieval. *Information Processing and Management*, 29(3):287–298, 1993.

FR97 N. Fuhr and T Rölleke. HySpirit - a Probabilistic Inference Engine for Hypermedia Retrieval in Large Databases. to appear, 1997.

Fuh92a N. Fuhr. Integration of Probabilistic Fact and Text Retrieval. In N. Belkin et al., editors, *Proc. 15th ACM SIGIR Conf.*, pages 211–222. ACM, 1992.

Fuh92b N. Fuhr. Konzepte zur Gestaltung zukünfiger Information-Retrieval-Systeme. In R. Kuhlen, editor, *Experimentelles und praktisches Information Retrieval*, pages 59–75. Universitätsverlag Konstanz, 1992.

Fuh92c N. Fuhr. Probabilisitc Models in Information Retrieval. *Computer Journal*, 35(3):243–255, 1992.

Fuh96a N. Fuhr. Models for Integrated Information Retrieval and Database Systems. In D. Lomet and E. Moss, editors, *Special Issue on Integrating Text Retrieval and Databases*, volume 19 of *Bulletin of the Technical Committee on Data Engineering*, pages 3–13. IEEE Computer Society, 1996.

Fuh96b N. Fuhr. Object-oriented and Database Concepts for the Design of Networked Information Retrieval Systems. In K. Barker and M. Özsu, editors, *Proceedings of the 5th International Conference on Information and Knowledge Management*, pages 164–172, 1996.

FY93 D.W. Flater and Y. Yesha. Towards Flexible Distributed Information Retrieval. In N.R. Adam and B.K. Bhargava, editors, *Advanced Database Systems*, pages 259–276. Springer-Verlag, 1993.

Ger93 W. Gerhardt. *Zugriffskontrolle bei Datenbanken*. Oldenbourg Verlag, 1993.

HS95 A. Heuer and G. Saake. *Datenbanken - Konzepte und Sprachen*. International Thomson Publishing, 1995.

HSZ96 W. Hackbusch, H. R. Schwarz, and E. Zeidler. *Teubner-Taschenbuch der Mathematik*. Teubner, 1996.

KE97 A. Kemper and A. Eickler. *Datenbanksysteme: eine Einführung*. Oldenbourg, second edition, 1997.

KS96 D. Knaus and P. Schäuble. The System Architecture and the Transaction Concept of the SPIDER Information retrieval System. In D. Lomet and E. Moss, editors, *Special Issue on Integrating Text Retrieval and Databases*, volume 19 of *Bulletin of the Technical Committee on Data Engineering*, pages 43–52. IEEE Computer Society, 1996.

Loo95 M.E.S. Loomis. *Object Databases - The Essentials.* Addison-Wesley, 1995.

LS87 P.C. Lockemann and J.W. Schmidt, editors. *Datenbank-Handbuch.* Springer-Verlag, 1987.

Neu95 H.A. Neumann. *Objektorientierte Entwicklung von Software-Systemen.* Addison-Wesley, first edition, 1995.

OMG95 OMG. *CORBA 2.0: Architecture and Specification.* Object Management Group, 1995.

RBP+91 J. Rumbaugh, M. Blaha, M. Premerlani, E. Frederick, and Lorensen W. *Object-Oriented Modelling and Design.* Prentice Hall, 1991.

SBGK94 M. Scheller, K-P. Boden, A. Geenen, and J. Kampermann. *Internet: Werkzeuge und Dienste.* Springer-Verlag, 1994.

Seb96 R.W. Sebesta. *Programming Languages.* Addison-Wesley, third edition, 1996.

Son96 G. Sonnenberger. Exploiting the Functionality of Object-Oriented Database Management Systems for Information Retrieval. In D. Lomet and E. Moss, editors, *Special Issue on Integrating Text Retrieval and Databases*, volume 19 of *Bulletin of the Technical Committee on Data Engineering*, pages 14–23. IEEE Computer Society, 1996.

SST97 G. Saake, I. Schmitt, and C. Türker. *Objektdatenbanken - Konzepte, Sprachen, Architekturen.* International Thomson Publishing, 1997.

Ste96 U. Steffens. Integration von Information Retrieval Funktionalität in eine offene, persistente Programmierumgebung. Technical Report FBI-HH-M-257/96, Universität Hamburg, Fachbereich Informatik, April 1996.

TC91 H.R. Turtle and W.B. Croft. Evaluation of an Inference Network-Based Retrieval Model. *ACM Transactions on Information Systems*, 9(3):187–222, 1991.

TC92 H.R. Turtle and W.B. Croft. A comparision of text retrieval models. *Computer Journal*, 35(3):279–290, 1992.

VAB96 M. Volz, K. Aberer, and K. Böhm. An OODBMS-IRS Coupling for Structured Documents. In D. Lomet and E. Moss, editors, *Special Issue on Integrating Text Retrieval and Databases*, volume 19 of *Bulletin of the Technical Committee on Data Engineering*, pages 34–42. IEEE Computer Society, 1996.

VGH93 G. Vossen and M. Groš-Hardt. *Grundlagen der Transaktionsverarbeitung.* Addison-Wesley, 1993.

Vos94 G. Vossen. *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme.* Addison-Wesley, second edition, 1994.

vR79 C.J. van Rijsbergen. *Information retrieval.* Butterworths, 1979. 2nd edition.

Wat90 D.A. Watt. *Programming Language Concepts and Paradigms.* Prentice Hall, 1990.

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, den 10. Dezember 1997

(Ulrike Steffens)