Semantische Anreicherung eines Datenmodells für komplexe Objekte

von
Sven Thelemann

Dissertation
vorgelegt am Fachbereich Mathematik/Informatik
der Universität Gesamthochschule Kassel
zur Erlangung des Doktorgrades der Naturwissenschaften
(Dr. rer. nat.)

Kassel
Mai 1996

Vom Fachbereich Mathematik/Informatik der Universität Gesamthochschule Kassel als Dissertation angenommen am 13. Juni 1996

Erstgutachter: Prof. Dr. L. Wegner (Universität Gesamthochschule Kassel)
Zweitgutachter: Prof. Dr. K. Küspert (Friedrich-Schiller-Universität Jena)

Tag der mündlichen Prüfung: 1. Juli 1996

Hiermit versichere ich, daß ich die vorliegende Dissertation selbständig und ohne unerlaubte Hilfe angefertigt und andere als die in der Dissertation angegebenen Hilfsmittel nicht benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen sind, habe ich als solche kenntlich gemacht. Kein Teil dieser Arbeit ist in einem anderen Promotions- oder Habilitationsverfahren verwendet worden.

Inhaltsverzeichnis

Inhaltsverzeichnis i

Kapitel 1 : Einleitung 1

1.1 ESCHER – ein interaktiver Datenbankeditor 4
1.2 Ein einfaches CAD-Beispiel 8
1.3 Defizite des bisherigen Datenmodells 10
1.4 Ziele und Aufbau der Arbeit 19

Kapitel 2 : Entwicklungslinien der Datenmodellierung 23

2.1 Semantische Datenmodelle 24
2.1.1 Abstraktionsmechanismen 25
2.1.2 Integritätsbedingungen 27
2.1.3 Beispiele für semantische Datenmodelle 28
2.2 Die Drei-Schichten-Architektur 34
2.2.1 Erfahrungen mit frühen DBMS 34
2.2.2 Trennung der Ebenen 36
2.3 Das Relationenmodell 38
2.3.1 Relationentheorie und Integritätsbedingungen 38
2.3.2 Relationale Anfragesprachen 39
2.3.3 Relationale Sichten und das View-Update-Problem 41
2.3.4 Nachteile des Relationenmodells 44
2.4 Erweiterungen des Relationenmodells 47
2.4.1 Verallgemeinerungen des Strukturteils 47
2.4.2 Erweiterungen um Typkonzepte 51
2.4.3 Weitere Konzepte postrelationaler DBMS 53
2.4.3.1 Komplexe Objekt-Sichten 53
2.4.3.2 ECA-Regeln 58
2.5 Objektorientierte Datenmodelle und DBMS 60
2.5.1 Was ist „Objektorientierung“? 60
2.5.2 Objektorientierte DBMS 64
2.5.3 Kritik an objektorientierten DBMS 65

Kapitel 3 : Der strukturelle Teil des Datenmodells 69

3.1 Typen und einfache Wertebereiche 71

3.1.1 Basistypen und atomare Werte 72
3.1.2 Aufzählungstypen 73
3.1.3 Benutzerdefinierte Typen und Objekte 75
3.1.4 Variante Typen 76
3.2 Strukturen und komplexe Wertebereiche 77
3.2.1 Komplexe Wertebereiche 77
3.2.1.1 Die komplexen Wertebereiche des eNF^2-Modells 78
3.2.1.2 Multimengen 80
3.2.1.3 Felder 81
3.2.2 Strukturen und Typausdrücke 83
3.3 Die Semantik varianter Typen 86
3.4 Getypte Datenobjekte 88
3.5 Repräsentation von Datenobjekten durch Bäume 88
3.5.1 Motivation 88
3.5.2 Konstruktion von Baumrepräsentationen 91
3.5.2.1 Baumrepräsentationen für Strukturen 91
3.5.2.2 Baumrepräsentationen für Datenobjekte 94
3.6 Die strukturelle Beschreibung von Objekttypen 100
3.7 Die Subtyp-Beziehung auf Objekttypen 103
3.7.1 Grundlagen 103
3.7.2 Dynamische Spezialisierung 107
3.8 Tabellen 108
3.9 Datenbankschema und Datenbankinstanz 111
3.10 Metamodellierung 115
3.11 Zusammenfassung und Diskussion 121

Kapitel 4 : Der operationale Teil des Datenmodells 125

4.1 Laufzeitzustand 127
4.2 Signaturen für Operationen 129
4.3 Basisoperationen 130
4.3.1 Elementare Operationen 132
4.3.2 Generische Update-Operationen 132
4.3.3 Weitere generische Operationen 135
4.3.4 Operationen für Objekttypen 136
4.4 Links und Iteratoren 138
4.5 Benutzerdefinierte Operationen 142
4.5.1 Kodierte Operationen 142

4.5.2 Methoden für Objekttypen 143
4.5.3 Weitere benutzerdefinierte Operationen 146
4.6 Die Sprache BASESCRIPT 147
4.6.1 Die Syntax des Anweisungsteils eines Basisscriptes 147
4.6.2 Semantik von einfachen Ausdrücken 148
4.6.3 Aufruf und Abarbeitung einer Operation 150
4.6.4 Die Semantik des Anweisungsteils eines Basisscriptes 151
4.6.5 Die Semantik von Anfragen in BASESCRIPT 153
4.7 Eine Variante der Sprache SCRIPT 156
4.7.1 Grundlegende Vorbemerkungen 156
4.7.2 Die Semantik von Anfragen in SCRIPT$^+$ 157
4.7.3 Beispiele zur Übersetzung von SCRIPT$^+$ nach BASESCRIPT 160
4.7.4 Weitere Aspekte der Sprache SCRIPT$^+$ 165
4.7.4.1 Variante Typen 165
4.7.4.2 Dynamische Spezialisierung 168
4.8 Erweiterbarkeit des Datenmodells. 171
4.9 Zusammenfassung und Diskussion 174

Kapitel 5 : Integritätsbedingungen und Konsistenzerhaltung 179

5.1 Transaktionen 180
5.2 Strukturpfadausdrücke. 183
5.3 Integritätsbedingungen 185
5.3.1 Schlüsselbedingungen 188
5.3.2 Inklusionsbedingungen 193
5.3.3 Disjunktheitsbedingungen 196
5.3.4 Lokale Wertebedingungen. 197
5.4 Überwachung der Integritätsbedingungen. 199
5.5 Konsistenzerhaltung durch Einkapselung 207
5.5.1 Einkapselung für Attribute von Objekttypen 207
5.5.2 Einkapselung für Operationen. 210
5.6 Konsistenz von Datenbankschemata 211
5.7 Zusammenfassung und Diskussion 214

Kapitel 6 : Beziehungen 219

6.1 Die Behandlung von Beziehungen in Datenmodellen. 220
6.1.1 Ein einführendes Beispiel 220
6.1.2 Beziehungsredundanz 224

6.1.3 Schema-Modifikation und Beziehungen 227
6.2 Beziehungen in ESCHER+ 228
6.3 Formale Definitionen. 233
6.3.1 Beziehungen und Beziehungssichten 233
6.3.2 Beziehungssichten auf der Instanzenebene 237
6.3.3 Zulässige Beziehungssichten. 238
6.3.4 Der Zugriff auf die Attribute einer Beziehungssicht. 242
6.4 Integritätsbedingungen für Beziehungen. 246
6.4.1 Schlüsselbedingungen 246
6.4.2 Kardinalitätsbedingungen 247
6.4.3 Wertebedingungen für Komponenten und Attribute. 248
6.4.4 Inklusionsbedingungen 249
6.5 Konsistenz bei Vorliegen von Beziehungsredundanz. 250
6.6 Kontrolle der Beziehungsredundanz 253
6.7 Zusammenfassung und Diskussion 255

Kapitel 7 : Zusammenfassung und Ausblick 259

7.1 Zusammenfassung. 259
7.2 Ausblick. 263

Anhang A : Mathematische Grundlagen 265

A.1 Elementare Notationen und Definitionen 265
A.2 Definitionen aus der Graphentheorie. 266

Anhang B : Syntaxregeln 269

B.1 Vereinbarungen zur Angabe von Grammatiken 269
B.2 Die DDL für ESCHER+. 270
B.3 Die Syntax von SCRIPT+ 273

Anhang C : Beispiele zur Ableitung von ECA-Regeln aus Beziehungsdefinitionen. . 275

C.1 Regeln für die Beziehung `Teilnahme` 275
C.2 Regeln für die Beziehung `Ausleihe` 277

Literatur 281

Kapitel 1

Einleitung

Im Bereich der Datenverarbeitung spielen heute bei nahezu jeder größeren Anwendungsentwicklung Datenbankmanagementsysteme (DBMS) eine entscheidende Rolle. Sie bilden sozusagen das Rückgrat heutiger DV-Anwendungen. Datenhaltungssysteme, die sich allein auf die herkömmliche Verwaltung von Dateien beschränken, spielen nur noch eine untergeordnete Rolle.

Datenbankmanagementsysteme gehen in ihrer Funktionalität weit über die Leistungsfähigkeit traditioneller Dateiverwaltungssysteme hinaus, was auch letztendlich für den Erfolg von DBMS in der Praxis verantwortlich ist. Die wesentlichen Eigenschaften lassen sich wie folgt zusammenfassen:

- Vorhandensein eines logischen *Datenmodells*, das von physischen Gegebenheiten abstrahiert
- Verwaltung von und effizienter Zugriff auf ggf. sehr große persistente Datenbestände
- *Transaktionsmanagement* zur Unterstützung der Konsistenz der Datenbank
- Synchronisation im *Mehrbenutzerbetrieb*
- *Zugriffskontrolle* aufgrund von Zugriffsberechtigungen (Datenschutz)
- *Recovery*-Mechanismen zur Gewährleistung hoher Datensicherheit bei Hard- oder Software-Fehlern
- *Benutzerfreundliche (Sprach-)Schnittstellen* zur Definition von Schemata, Manipulation von Daten und zur Formulierung von Anfragen in deklarativer Form

In dieser Arbeit verwenden wir die Begriffe DBMS und Datenbanksystem synonym, obwohl einige Autoren hier genauer unterscheiden, indem sie die Gleichung „Datenbanksystem = DBMS + *n* Datenbanken“ aufstellen (vgl. [Vos91]).

Aufgrund der konzeptionellen Einfachheit des *Relationenmodells* haben auf diesem Modell aufsetzende sog. relationale Datenbanksysteme im kommerziellen Bereich heute eine dominierende Position erlangt, wenn auch nach dem Aufkommen erster Prototypen Mitte der 70er Jahre eine längere Anlaufzeit zu überwinden war[1].

Das traditionelle Anwendungsgebiet für Datenbanken sind betriebswirtschaftliche, kaufmännische oder administrative Informationssysteme, wie z.B. Auftrags-, Lager- und Personalverwaltung. Für diese Anwendungsbereiche haben sich das Relationenmodell mit seiner schlichten, „flachen" Grundstruktur und relationale Datenbanksysteme mit hinreichender Leistungsfähigkeit bewährt. Anders sieht es im technischen oder ingenieurwissenschaftlichen Bereich aus. Auch hier wächst der Bedarf am Einsatz von Datenbanken, nachdem sie in diesem Sektor lange nur eine unwesentliche Rolle spielten. Anwendungen etwa aus den Bereichen CAD (Computer Aided Design), CAM (Computer Aided Manufacturing), CASE (Computer Aided Software Engineering), Büroinformationssysteme, VLSI-Design u.ä. werden daher als *Nicht-Standard*-Anwendungen bezeichnet. In diesen Bereichen gibt es ebenfalls sehr große und komplexe Datenbestände, die jedoch von einer Vielzahl unterschiedlicher Tools bearbeitet werden. Diese Tools arbeiten i.d.R. rein auf Datei-Basis mit zueinander inkompatiblen Datenformaten, so daß ein Datenaustausch zwischen diesen Tools sehr mühsam ist. Daher bietet sich der Einsatz von Datenbanksystemen in ihrer Rolle als *Integrationswerkzeuge* auch für die Nicht-Standard-Anwendungen an: Im Idealfall ist eine Datenbank eine gemeinsame Kommunikationsbasis für alle Teilbereiche und Teilschritte einer Anwendung, und unterschiedliche Teilanwendungen arbeiten auf einem gemeinsamen Datenbestand.

Es hat sich gezeigt, daß die Ausdrucksmittel des Relationenmodells nicht ausreichen, um Nicht-Standard-Anwendungen adäquat zu modellieren. Bleibt man dennoch beim Relationenmodell, so werden relationale DBMS oftmals den gestellten Performance-Anforderungen nicht gerecht [GS82]. Neben Unzulänglichkeiten im strukturellen Teil des Datenmodells tritt auch die Forderung nach Unterstützung weiterer Konzepte (z.B. „lange" Transaktionen, Versionierung, kooperatives Arbeiten, weitergehende Unterstützung von Integritätsbedingungen), die von den konventionellen relationalen DBMS nicht unterstützt werden [Loc+85, KW87, Mai89]. Gute Überblicke der Anforderungen von Nicht-Standard-Applikationen sind in [Cat91, KM94] zu finden.

Um diesen Mißstand zu überwinden, sind im Laufe der 80er Jahre zwei Wege eingeschlagen worden: In einem als *evolutionär* zu charakterisierenden Ansatz werden Verallgemeinerungen des Relationenmodells entwickelt bzw. neue Konzepte in eher „konservativer" Manier hinzugefügt. Eine andere Strömung vertritt den *revolutionären* Ansatz und sieht die Lösung in den sog. *objektorientierten Datenbanken* (OODB). Hier wird zunächst mit dem Relationenmodell gebrochen, neue Grundlage sind Konzepte der Objektorientierung. Daß dabei anfangs auch Bewährtes, wie z.B. die Deklarativität der Anfragesprache, geopfert wurde, störte zunächst nicht. Mittlerweile gibt es eine kaum noch zu übersehende Vielzahl von Vorschlägen in beiden Lagern. Es gibt bis heute auch keine Einigkeit über *das* objektorientierte Datenmodell, obwohl mittlerweile Standardisierungsvorschläge vorliegen [Cat+94].

Auf der Benutzerseite haben in allen Bereichen der Datenverarbeitung *graphische Benutzeroberflächen* sehr starke Verbreitung gefunden. Die Ansprüche der Endanwender sind diesbe-

1. In seiner Zeit als Softwareentwickler konnte der Autor die Erfahrung machen, daß noch Ende der 80er Jahre den relationalen Datenbanken in der Praxis mit einigem „Mißtrauen" begegnet wurde.

züglich mittlerweile auch sehr hoch. Als Interaktionstechnik hat sich die sog. *direkte Manipulation* [Shn87] durchgesetzt. Dabei steuert der Benutzer die Interaktion mit der Benutzerschnittstelle über die sog. WIMP-Operationen (WIMP = Window, Icon, Mouse, Pointer). Vorteile sind schnelle Erlernbarkeit einer solchen Oberfläche und eine ergonomischere Arbeitsweise. Wichtig ist, daß die Oberfläche sofort auf die Aktionen des Benutzers reagiert (unmittelbares Feedback). Im Zuge dieser Entwicklung und des damit einhergehenden Anwachsens der Möglichkeiten der visuellen Präsentation gewinnen Forschungsarbeiten weiter an Bedeutung, deren Ziel es ist, fortschrittliche graphische Benutzeroberflächen für DBMS zu entwikkeln [SAD+93]. Aufgrund der Anforderungen durch Nicht-Standard-Anwendungen geht es vor allem darum, komplex strukturierte Daten in einer leicht verständlichen Form visuell zu präsentieren und auf intuitive Weise zu manipulieren.

Neben der für den Anwendungsprogrammierer entscheidenden Programmiersprachen-Schnittstelle rücken also die Schnittstellen zu anderen Benutzergruppen eines Datenbanksystems weiter in den Vordergrund. Es werden folgende Benutzerklassen unterschieden (vgl. [Zeh89]):

- *parametrischer Benutzer* (parametric user):
  Sie sind Endanwender einer Applikation und i.d.R. Sachbearbeiter, die mit speziell für Routinetätigkeiten erstellten Anwendungsprogrammen arbeiten. Die Interaktion erfolgt dabei über das Ausfüllen von Bildschirmmasken bzw. -formularen. Von parametrischen Benutzern können nur die über die Anwendungsprogramme vorbereiteten Anfragen ausgeführt werden.
- *gelegentlicher Benutzer* (casual user):
  Diese Art von Benutzer wird oft auch als „anpruchsvoller Laie“ bezeichnet, wobei der Begriff „Laie“ sich darauf bezieht, daß es sich nicht um einen Datenbankspezialisten handelt. Gleichwohl ist der gelegentliche Benutzer Spezialist auf seinem Fachgebiet, hat genaue Kenntnisse über das externe und logische (bzw. konzeptuelle) Schema seiner „Miniwelt“. Er ist in der Lage und daran interessiert, Ad-hoc-Anfragen zu stellen, durch den Datenbestand zu stöbern (*browsen*), Ad-hoc-Updates zu machen sowie seine individuellen Benutzersichten zu erzeugen.
- *Spezialist* (professional user):
  Darunter sind Anwendungsprogrammierer, Datenbank- und Systemadministratoren zu verstehen. Ihnen werden weitere Sprachmittel und Tools zur Verfügung gestellt, mit denen sie die unterschiedlichen Schemata (extern, logisch, intern) definieren und auch Parameter für weitere Funktionalitäten eines DBMS (z.B. Steuerung der Zugriffsrechte) setzen können.

Neben rein formular- oder maskenorientierten Oberflächen für den Endanwender sind generische bzw. durch den Benutzer konfigurierbare graphische Schnittstellen wie *Browser* und *graphische Anfragesprachen* von großem Interesse, die sich auch an andere Benutzergruppen wenden und einen „spontaneren“ Umgang mit einer Datenbank möglich machen.

1.1 ESCHER – ein interaktiver Datenbankeditor

Diese Arbeit geht aus den Forschungsaktivitäten im Projekt ESCHER hervor, dessen Anfänge auf das Ende der 80er Jahre zurückgehen [Weg89, KTW90]. Kern des Projektes ESCHER ist ein gleichnamiger interaktiver Datenbankeditor, der auf der Grundlage geschachtelter Tabellen komplexe Daten visualisiert und auch die Manipulation dieser Daten erlaubt [The92, PTW94]. ESCHER greift also den in der Einleitung zuletzt angesprochenen Aspekt der Entwicklung fortgeschrittener graphischer Benutzeroberflächen für das Arbeiten mit Datenbanksystemen auf.

Dabei bedient sich ESCHER des eNF^2-Datenmodells, das als eine Erweiterung des sog. NF^2-Datenmodells angesehen werden kann[2]. Letzteres unterscheidet sich von dem bekannten Relationenmodell dadurch, daß im Relationenmodell nur „flache“[3] Relationen zum persistenten Datenbestand gehören können, während das NF^2-Modell immerhin relationenwertige Attribute zuläßt. Das eNF^2-Modell geht noch einen Schritt weiter und erlaubt noch größere strukturelle Vielfalt, indem man durch geschachtelte Anwendung der Konstruktoren *Set*, *List* und *Tuple* und auf der Basis von vordefinierten elementaren Typen beliebig *komplexe Strukturen* erzeugen kann. Das eNF^2-Datenmodell wurde im Rahmen des „AIM-P“-Projektes [DKA+86, PA86, PD89] der IBM entwickelt. Durch die erhöhte strukturelle Flexibilität erreicht man eine adäquatere und direktere Datenmodellierung für die bereits erwähnten Nicht-Standard-Anwendungen.

Mit dem derzeit implementierten Prototyp von ESCHER kann man

- *Schemata* erzeugen, die jeweils eine benamte komplexe (eNF^2-)Struktur darstellen
- eine oder mehrere *Tabellen* als Instanzen zu einem Schema erzeugen
- in Tabellen browsen und Datenmanipulationsoperationen ausführen („editieren“)

Wir wollen in dieser Arbeit den Begriff „Schema“ als Synonym für *eine* benamte eNF^2-Struktur vermeiden, da unter „Schema“ allgemein ein Datenbankschema für eine vollständige Anwendung verstanden wird, innerhalb dessen ein ESCHER-Schema i.d.R. nur einen kleinen Teil ausmacht. Lediglich in diesem den Prototypen vorstellenden Abschnitt verwenden wir „Schema“ als Synonym für eine einzelne eNF^2-Struktur.

Beispiel 1.1

Die Abb. 1.1 zeigt ein Arbeitsfenster des ESCHER-Prototyps, in dem ein Schema als „Tabellenkopf“ und darunter die Tabelle als Instanz zum Schema zu sehen ist. Gezeigt wird ein Ausschnitt aus einer vereinfachten Bibliotheksanwendung. Die Tabelle `Leser` ist eine Menge (durch das Präfix `{}` von `Leser` angedeutet) von Tupeln, wobei jedes Tupel die Daten eines Bibliotheksbenutzers enthält. In der Tupelkomponente `Entleihungen` ist die Menge der aktuellen Ausleihen des jeweiligen Lesers vermerkt, die wiederum Tupel bestehend aus der Exemplarnummer des ausgeliehenen Buchs und des Ausleihdatums sind. Ferner werden in derselben Tabelle auch die `Vormerkungen` für jeden Leser erfaßt. Diese bestehen aus dem Datum der Vormerkung und der Signatur des vorzumerkenden Buches. Es wird zwischen

2. NF^2 = *N*on *F*irst *N*ormal *F*orm, eNF^2 = extended NF^2.
3. Dabei bedeutet „flach“, daß alle Komponenten der Tupel in einer Relation einen elementaren Datentyp besitzen. Diese Eigenschaft bezeichnet man auch als Erste Normalform (First Normal Form).

Exemplarnummer und Signatur unterschieden, da zu einem Buch (d.h. einer Signatur) eventuell mehrere Exemplare entleihbar sind. ❒

Abbildung 1.1: Ein ESCHER-Arbeitsfenster mit der Tabelle `Leser`

In Abb. 1.1 ist auch leicht zu erkennen, daß das generische Tabellenlayout Tupelkomponenten horizontal und die Elemente von Mengen (oder auch Listen) vertikal anordnet. Die Schachtelung gibt die Daten aus der Perspektive der Leser wieder. Der Prototyp zeigt in einem Arbeitsfenster immer eine vollständige Basistabelle, d.h. ein vom Datenbankschema separiertes Sichtenkonzept gibt es bislang nicht.

Sowohl das Erzeugen von Schemata als auch das Browsen und Editieren von Tabellen geschieht interaktiv. Der Benutzer *navigiert* dazu mit sog. *Fingern* in einem Schema bzw. in einer Tabelle [Weg91b].

Aufgrund des hierarchischen Aufbaus aller eNF^2-Strukturen läßt sich jedes Schema und jede Tabelle als ein Baum auffassen, und wir sprechen dann auch von Schema- bzw. Tabellenbäumen. Ein Finger „zeigt“ immer auf einen Knoten in einem solchen Baum. Die Navigation erfolgt daher auf der Benutzeroberfläche in zwei verschiedenen Dimensionen:

- *In*/*Out*: Navigationsschritt von einem Knoten zum ersten Sohnknoten bzw. von einem Knoten zurück zu seinem Vater
- *Prev(ious)*/*Next*: Navigationsschritt innerhalb einer Ebene von einem Knoten zu seinem vorangehenden/nächsten Bruder (d.h. zur vorangehenden/nächsten Komponente eines Tupels bzw. zum vorangehenden/nächsten Element einer Menge oder Liste)

Ein Finger ist also ein verallgemeinerter Cursor, wie man ihn z.B. aus der SQL-Cursor-Schnittstelle in relationalen Datenbanken oder auch aus Texteditoren kennt. Während ein SQL-Cursor immer auf genau ein Tupel einer Relation „zeigt“, berücksichtigen die Finger in ESCHER die hierarchische Struktur des dargestellten Objektes: Alle Operationen beziehen sich auf die Position des Fingers. Die grundlegenden Änderungsoperationen sind das Modifizieren atomarer Werte sowie das Einfügen bzw. Löschen von Elementen in bzw. aus Mengen oder Listen. Im Prototyp sind die Navigations- und Änderungsoperationen an bestimmte Tasten bzw. Tastenkombinationen gebunden, wobei auf größtmögliche Analogie zur Editierfunktionalität bei gängigen Texteditoren geachtet wurde: Zeigt ein Finger z.B. gerade auf ein Element einer Menge, dann führt das Drücken der Delete-Taste zum Löschen des Elementes aus dieser Menge. Ein weiteres Analogon zu bekannten Tools ist die Verwendung eines Clipboard [The92] zur temporären Ablage komplexer Objekte im Zusammenhang mit *Cut*, *Copy* und *Paste*-Operationen.

Beispiel 1.2 (Fortsetzung von Beispiel 1.1)

In Abb. 1.1 „zeigt“ der aktive[4] Finger F2 gerade auf die Daten einer Entleihung der Leserin Vera Heck. Dies wird durch die farbliche Unterlegung des kompletten Objektes visualisiert, auf dessen Wurzel der aktive Finger gerade zeigt[5]. Eine *In*-Operation bewegt den aktiven Finger auf die `ExNr`-Komponente des Tupels. Die *Next*-Operation bewegt den aktiven Finger auf das nachfolgende Tupel in der Menge der Entleihungen. Drücken der *Delete*-Taste löscht das aktuelle Tupel aus der Menge der Entleihungen.
Ein weiterer (momentan nicht aktiver) Finger ist auf dem Nachnamen „Thiele“ positioniert. Nach seiner Aktivierung und einer *In*-Operation, durch die vom Browse- in den Edit-Modus übergegangen wird, kann der atomare Wert modifiziert werden. ❒

Das Browsen in sehr großen Datenmengen wird zusätzlich durch relativ rudimentäre Anfragemöglichkeiten unterstützt. In einem Tabellen-*Template* können Eintragungen gemacht werden, die den aktiven Finger veranlassen, daß er sukzessive auf diejenigen Objekte positioniert wird, für die es ein „pattern matching“ mit dem Tabellen-Template gibt. Eine mächtigere graphische Anfragesprache in der Tradition des QBE-Ansatzes wurde für ESCHER in [WTW+96] vorgeschlagen.

Nähere Ausführungen zu Implementierungsaspekten des existierenden Prototyps können [Weg90, Weg91a, Weg91b] entnommen werden.

Zum Abschluß dieses Abschnittes soll nun noch das bereits angerissene Anwendungsbeispiel „Bibliothek“ vervollständigt werden.

Beispiel 1.3 (Fortsetzung von Beispiel 1.1)

Auf den ersten Blick scheint die Struktur der Tabelle `Leser` aus Abb. 1.1 die erfreuliche Eigenschaft zu haben, verschiedene Verarbeitungsvorgänge in einer Bibliothek adäquat zu erfassen. Die Aufnahme oder das Löschen von Lesern kann innerhalb der Tabelle `Leser` ebenso erfolgen wie die Protokollierung von Entleihungen und Vormerkungen. Tatsächlich bedient die Tabelle `Leser` aber nur einen Teilaspekt der Globalanwendung „Bibliothek“. In

4. In einer Tabelle können gleichzeitig mehrere Finger existieren, aber genau einer ist der aktive, während die anderen als „Lesezeichen“ fungieren und bei Bedarf „reaktiviert“ werden können.
5. Beim Öffnen einer Tabelle ist der aktive Finger auf der Wurzel des Tabellenbaums positioniert, demzufolge ist anfangs auch die gesamte Tabelle eingefärbt.

Abb. 1.8 ist ein Entity-Relationship-Diagramm [Che76] zur strukturellen Beschreibung der gesamten Anwendung angegeben, welches ohne weiteres verständlich sein sollte. Die Beziehungen sind mit Kardinalitätsbedingungen in *min-max*-Notation versehen, d.h. ist `A` eine Komponente einer Beziehung `R` mit der Kardinalitätsbedingung (*n*, *m*), dann soll nach Projektion aller Beziehungsinstanzen auf die Komponente `A` die resultierende Menge eine Kardinalität zwischen *n* und *m* haben. Falls *m* = "∗", dann gibt es für die Kardinalität keine obere Schranke.

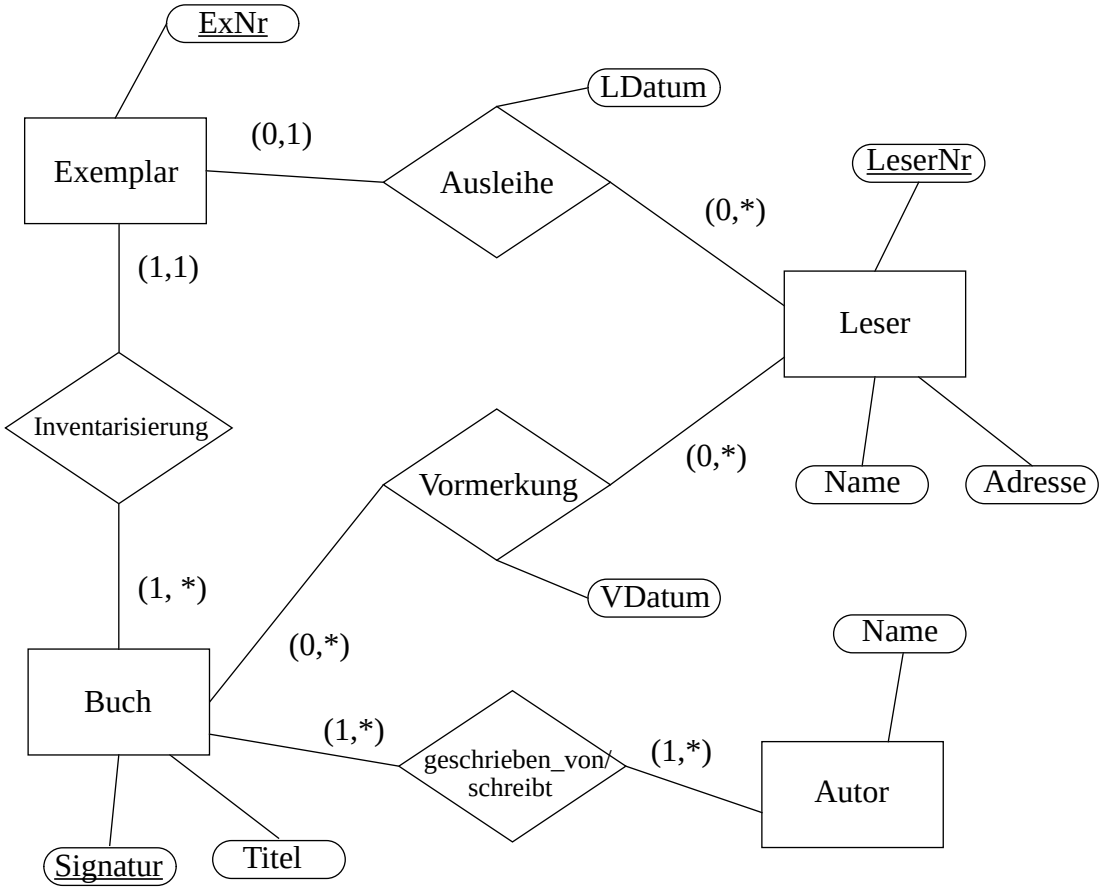


Abbildung 1.2: Ein ER-Schema für eine vereinfachte Bibliotheksanwendung

Um das gegebene ER-Schema vollständig über eNF^2-Tabellen zu erfassen, muß eine weitere Tabelle `Bestand` existieren, wie sie in Abb. 1.3 angegeben ist. Über diese Tabelle können nun auch zwei weitere Verarbeitungsvorgänge in einer Bibliothek abgehandelt werden. Bei der Bestellung neuer Bücher können die Grunddaten (Signatur, Titel, Autoren usw.) sofort erfaßt werden, und die Liste der Exemplare bleibt zunächst leer. Bei der Anlieferung der Exemplare wird die Liste dann mit den neu vergebenen Exemplarnummern gefüllt.
Man beachte, daß die Autoren eines Buches als Liste erfaßt werden (im Schema angezeigt durch <>), so daß eine für das Buch festgelegte Rangfolge der Autoren festgehalten werden kann. ❒

Abbildung 1.3: Die ESCHER-Tabelle `Bestand`

Das Bibliotheksbeispiel gehört mit seiner verwaltungstechnischen Natur eher zum Bereich der traditionellen Datenbankanwendungen. An dem Beispiel wurde gezeigt, daß geschachtelte Strukturen auch für solche Anwendungen Sinn machen. Für bestimmte Verarbeitungsvorgänge kann durch eine Schachtelungsstruktur eine geeignete „Sicht“ definiert werden, die an den jeweiligen Verarbeitungsvorgang besonders gut angepaßt ist.

1.2 Ein einfaches CAD-Beispiel

Es soll nun auf ein weiteres Beispiel eingegangen werden, das zum Bereich der Nicht-Standard-Anwendungen zählt. Der hierarchische Aufbau der im eNF^2-Datenmodell zu bildenden komplexen Strukturen und der in der Folge ebenfalls hierarchische Aufbau der Instanzen macht das Modell insbesondere für solche Anwendungen nutzbar, in denen streng hierarchisch aufgebaute *komplexe Objekte* vorkommen, bei denen Teilobjekte von ihrem umgebenden Objekt existenzabhängig sind. Dies ist z.B. bei CAD-Objekten der Fall, die aus anderen CAD-Objekten zusammengesetzt sind.

Beispiel 1.4

Es sollen 3-dimensionale geometrische Objekte angelegt und manipuliert werden. Zur Darstellung der Objekte wird die *Begrenzungsflächendarstellung* (*Boundary-Representation, BREP*) [KW87] verwendet, d.h. die Objekte werden durch eine Polyederdarstellung approximiert. Ein Polyeder wird durch eine mehrstufige Hierarchie „Polyeder → Flächen → Kanten → Ecken“ beschrieben. In Abb. 1.4 sind ein Quader und eine ihm aufgesetzte Pyramide dargestellt, und die entsprechenden Daten sind in der ESCHER-Tabelle `Polyeder` in Abb. 1.5 enthalten. In

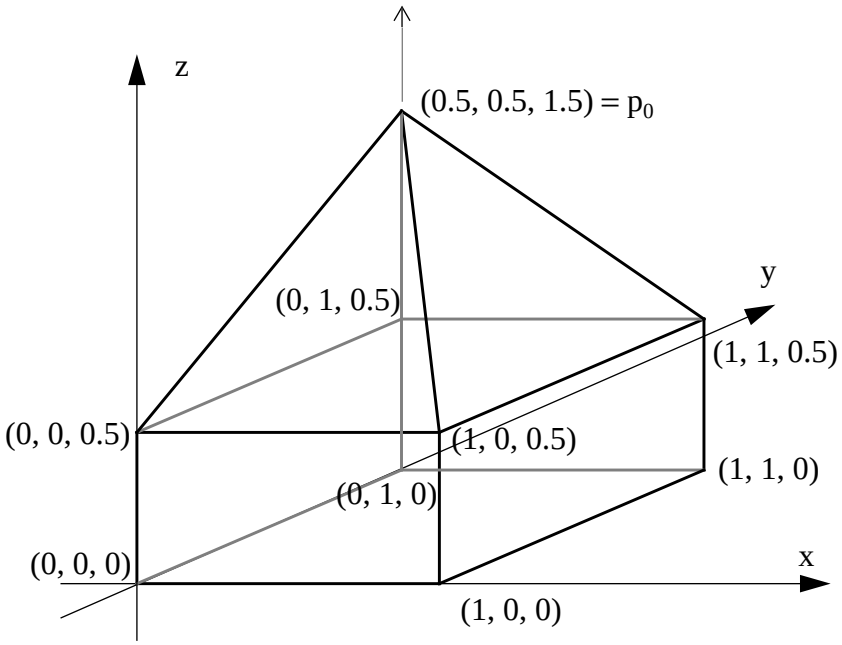


Abbildung 1.4: Pyramide und Quader –
zwei einfache geometrische Objekte

der Tabelle `Polyeder` fehlt die Hierarchie-Ebene „Kanten“, da diese implizit durch jeweils zwei aufeinanderfolgende Punkte der Liste `Eckpunkte` und der den ersten und letzten Punkt dieser Liste verbindenden Kante gegeben sind. ❒

Analysiert man die Wirkung der generischen Operationen auf die Tabelle `Polyeder`, so stellt sich heraus, daß so gut wie jede Operation die Konsistenz der in `Polyeder` enthaltenen geometrischen Objekte gefährdet. Lediglich die Modifikation des Attributes `Farbe` dürfte in jedem Falle erlaubt sein. Die *Delete*-Operation ist unproblematisch, falls der aktive Finger auf einem Tupel der Menge `Polyeder` positioniert ist. Ist dies etwa das erste Tupel in Abb. 1.5, so wird das Objekt „Pyramide1“ durch *Delete* komplett gelöscht, d.h. auch alle existenzabhängigen Teilobjekte (Flächen und Eckpunkte) werden gelöscht. Dagegen ist bereits das Erzeugen eines topologisch konsistenten Objektes mit den generischen Operationen nichttrivial! Das Löschen eines Punktes aus der Menge der Ecken einer Fläche, das Verändern der Koordinate eines Eckpunktes oder die Umordnung der Eckpunkte in einer Liste von Ecken kann unerwünschte Folgen haben. Es ist daher dafür zu sorgen, daß komplexe Objekte gegenüber modifizierenden Zugriffen, die die Konsistenz gefährden, abgekapselt werden.

Abbildung 1.5: Die ESCHER-Tabelle `Polyeder`

1.3 Defizite des bisherigen Datenmodells

In diesem Abschnitt diskutieren wir Probleme und Unzulänglichkeiten, die im Zusammenhang mit dem eNF^2-Datenmodell, das bisher die Grundlage des ESCHER-Prototyps bildete, genannt werden müssen und deren Überwindung die grundlegende Motivation für diese Arbeit sein soll.

Die gegenüber dem Relationen- und dem NF^2-Modell gesteigerten Ausdrucksmöglichkeiten des eNF^2-Datenmodells in *struktureller* Hinsicht sind ohne Zweifel äußerst wertvoll. Bei einer Fixierung auf den strukturellen Teil der Datenmodellierung werden jedoch andere wichtige Aspekte, die zur Erfassung der Semantik einer Anwendung in einem Datenbankschema notwendig sind, vernachlässigt. Es gilt daher, das bisherige Datenmodell des ESCHER-Prototyps semantisch anzureichern.

Die Notwendigkeit abstrakter Datentypen

Das CAD-Beispiel aus Abschnitt 1.2 hat deutlich gemacht, daß die „freie“ Anwendung generischer Update-Operationen die Konsistenz von Anwendungsobjekten sehr leicht zerstören kann. Für die Manipulation der Polyeder-Objekte aus Beispiel 1.4 haben daher die elementaren geometrischen Transformationsoperationen *Translation*, *Rotation* und *Skalierung* eine zentrale Bedeutung. Werden sie auf Polyeder-Objekte „als ganzes“ angewandt, kann topologische Konsistenz zugesichert werden. Es gilt also, diese anwendungsspezifischen Operationen in geeigneter Form an Polyeder-Tupel zu „binden“.

- Es ist deshalb unabdingbar, das Datenmodell um abstrakte Datentypen (ADTs) [Gut77, EGL89] zu ergänzen, damit konsistenzerhaltende Operationen definiert werden können und die notwendige Einkapselung komplexer Objekte gegenüber konsistenzgefährdenden Modifikationsoperationen erreicht wird.

Die Erweiterung des eNF^2-Modells um ADTs ist nicht neu, und es liegen bereits mehrere Vorschläge zur ihrer Integration in das eNF^2-Modell vor. (vgl. [KW87] bzw. [LKD+91]). Für ESCHER wurden in [WPC92, The93, Pau94] Typkonzepte vorgeschlagen. In [Pau94] wurden Operationen über die Einführung eines funktionalen Datentyps[6] in das Datenmodell integriert. Während dort der Struktur- und der Operationenteil einer Typdefinition vermischt wird, indem Operationen nichts anderes sind als funktionswertige Attribute, wollen wir in dieser Arbeit deutlich zwischen diesen beiden Aspekten trennen, wie es auch in der ADT-Theorie [EGL89] die Regel ist. In der in dieser Arbeit verwendeten Syntax lautet eine entsprechende Typdefinition für den benutzerdefinierten Typ `Polyeder` wie folgt:

```
define type Polyeder
  -struct [ Name: string,
            Flaechen:
             {[Name: string, Farbe: string,
               Ecken: <Punkt>]}
          ]
  -ops  trans(x: float, y: float, z: float -> void),
        rot(x: float, y: float, z: float, phi: float
            -> void),
        scale(Faktor: float -> void);
        setColor(Flaechenname: string, Farbe: Color
            -> void);
```

Die Typdefinition verwendet den bereits definierten Typ `Punkt`. Der erste Parameter („Receiver“) der Operationen ist implizit gegeben durch `self:` <*type*>.

Objekttypen oder Wertetypen?

Ein weiterer Problemkreis betrifft die Behandlung sog. gemeinsamer Teilobjekte (*shared subobjects*): Möchte man z.B. die Höhe der Pyramide in Abb. 1.4 ändern, so muß der Punkt p_0 – wie in der Abbildung angedeutet – parallel zur z-Achse verschoben werden. Dieser Punkt ist den vier Flächen F1, F2, F3, F4 gemeinsam. Werden die geometrischen Objekte als Instanzen

6. Werte dieses Datentyps sind Funktionen in der Syntax der in [Pau94] vorgestellten persistenten Programmiersprache SCRIPT.

des oben definierten Typs `Polyeder` dargestellt, so muß geklärt werden, ob es sich um einen *Werte-* oder *Objekttyp* handelt. Im letzteren Fall ist einer Instanz des Typs eine invariante (*Objekt-*)*Identität* (*OID*) und ein veränderbarer *Zustand* zugeordnet. Bei Instanzen von Wertetypen gibt es keine Trennung von Identität und Zustand [Bee90].

Sind alle benutzerdefinierte Typen Wertetypen, dann bedeutet dies, daß die Koordinaten des Punktes p_0 in allen Flächen, die ihn als Eckpunkt besitzen, gespeichert sind. Aufgrund dieser Redundanz muß eine Verschiebung des Punktes p_0 in ein Update an vier verschiedenen Stellen umgesetzt werden, damit das Objekt als ganzes topologisch konsistent bleibt. Dies zeigt, daß auch das um Wertetypen erweiterte eNF^2-Modell lediglich für *streng* hierarchische Beziehungen Redundanz vermeiden kann. Haben zwei Anwendungsobjekte jedoch gemeinsame Subobjekte (diese Situation wird in der englischsprachigen Literatur auch als *object sharing* bezeichnet), dann handelt es sich um eine „is part of"-Beziehung mit *m:n*-Kardinalität, und Redundanz läßt sich bei der Schachtelung gemäß der Hierarchie „Gesamtobjekt → Teilobjekt" nicht vermeiden.

Günstiger ist es, die benutzerdefinierten Typen als Objekttypen aufzufassen. Der Wert des Attributs `Ecken` ist dann einfach eine Liste von logischen Objektidentifikatoren. Konzeptuell repräsentiert eine OID das komplette Objekt, d.h. überall dort, wo die OID auftritt, ist gleichzeitig auch der an sie gekoppelte Zustand des Objektes verfügbar. Der Vorteil der Objekttypen liegt in dem direkten Bezug zu den Anwendungsobjekten des abzubildenden Umweltausschnitts: Auch in der Realität haben wir es mit Objekten zu tun, die bei sich verändernden Eigenschaften eine immer gleichbleibende Identität besitzen. Der Unterschied der Objekttypen zu den in vielen Programmiersprachen üblichen Zeigertypen (vgl. z.B. `^Punkt` in PASCAL) besteht darin, daß letztere als Instanzen *physische* Adressen haben. Auf konzeptueller bzw. logischer Ebene ist es jedoch adäquater, mit *logischen* Identifikatoren zu arbeiten, um weitestgehende Kongruenz mit den Anwendungsobjekten des darzustellenden Umweltausschnitts zu erlangen.

- Ein wesentlicher Beitrag dieser Arbeit soll die Entwicklung eines formalen Datenmodells für die *logische* Ebene sein, das abstrakte *Objekt*typen in das eNF^2-Datenmodell integriert.

Mit der Entscheidung für Objekttypen wird eine saubere Abgrenzung der logischen von der internen Ebene möglich, da zur Modellierung gemeinsamer Subobjekte nicht auf physische Identifikatoren zurückgegriffen werden muß.

In Abb. 1.6 ist veranschaulicht, wie sich die Daten der Tabelle `Polyeder` aus Abb. 1.5 als Instanzen der Objekttypen *Polyeder* und *Punkt* darstellen lassen. Die Pfeile geben an, welcher „Zustand" einer OID zugeordnet wird (nicht alle Pfeile sind gezeigt).

Man beachte, daß bei der konzeptuellen bzw. logischen Modellierung keine Entscheidung darüber gefällt wird, wie die Zustände der Objekttyp-Instanzen auf interner Ebene gespeichert werden. Allerdings ist die gewählte Schachtelungsstruktur als starker Hinweis auf die gewünschte Speicherung anzusehen: Die Koordinaten eines Punktes p_i sollten natürlich nach Möglichkeit „in der Nähe" aller p_i-Referenzen gespeichert sein, um das für geschachtelten Tabellen typische Clustering der Subobjekte in den sie umgebenden Objekten so weit wie möglich beizubehalten.

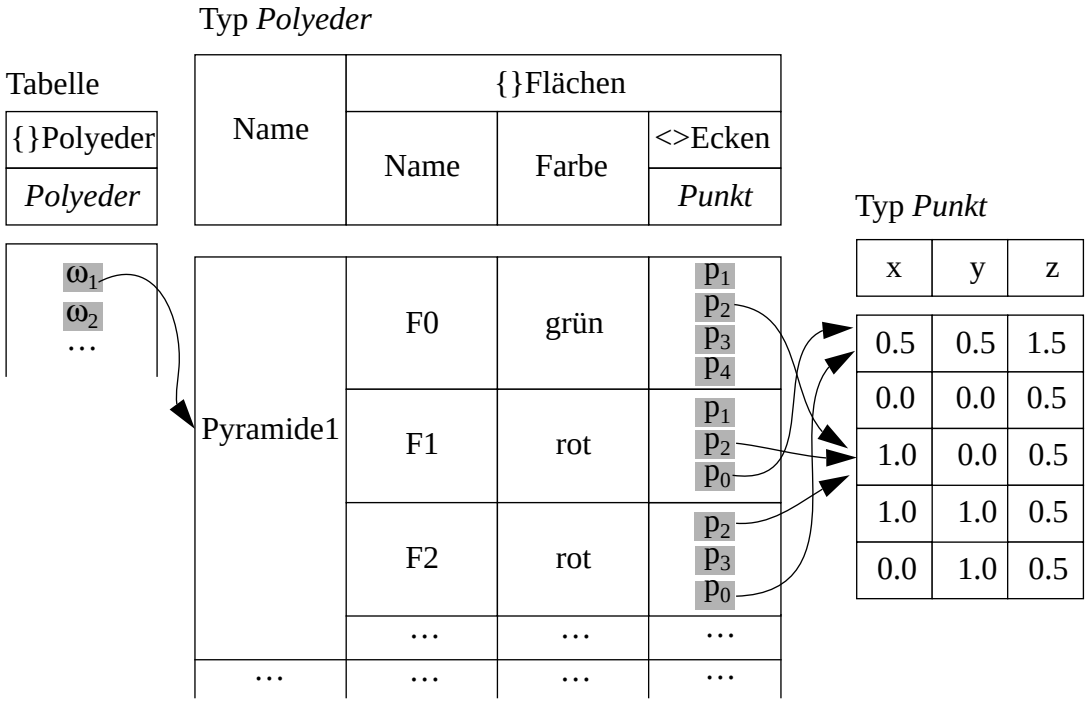


Abbildung 1.6: Modellierung des CAD-Beispiels mit Objekttypen

Variante Typen und dynamische Spezialisierung

Nach dem bisher Gesagten liegt der Nutzen, den man aus der Verwendung von Objekttypen ziehen kann, auf der Hand: Es wird eine größere Kongruenz des logischen Schemas mit dem konzeptuellen Schema auf der Basis eines semantischen Datenmodells erreicht. Gleichwohl gibt es nach wie vor einige Aspekte der Datenmodellierung mit Objekttypen, die der weiteren Auseinandersetzung bedürfen.

In einer CAD-Umgebung trifft man häufig auf die Situation, daß mehrere graphische Objekte zu einer 3D-Objektgruppe zusammengefaßt werden sollen. So wäre denkbar, die Pyramide und den Quader aus Abb. 1.4 zu einer Gruppe „Haus“ zusammenzufassen, um dieses dann als Einheit zu verschieben, zu drehen, zu skalieren usw. Die 3D-Objektgruppe soll rekursiv definiert sein, d.h. neben den „Basis“-Objekten (in unserem Beispiel die Polyeder) dürfen auch bereits existierende Gruppen Mitglied einer neuen Gruppe sein. Es liegt also die klassische „base parts/component parts“-Situation vor, bei der verschiedene Typen als Komponententypen eines zusammengesetzten Teils erlaubt sein sollen [AB87]. In [Pau94] wird das Problem, mehrere Typen als Elementtypen einer Gruppe zuzulassen, über eine speziellen *Varianten*-Konstruktor gelöst. Ein Typ `3DGruppe` könnte damit wie folgt definiert werden:

```
define type 3DGruppe
  -struct [ Name: string,
            Gruppe:{VAR(g: 3DGruppe, p: Polyeder)}
          ]
  -ops ...
```

Die Elemente des Attributs `Gruppe` sind Varianten. Nach Inspektion des *Tags* (`g` oder `p`) einer Variante kann auf ein Element bzgl. des „wirklichen“ Typs zugegriffen werden. Diese Lösung hat jedoch den Nachteil, daß sie nicht flexibel genug ist gegenüber nachträglichen Änderungen der Menge der möglichen Komponententypen einer Gruppe. Es ist denkbar, daß neben Polyedern zu einem späteren Zeitpunkt weitere 3D-Objekttypen hinzukommen (z.B. Kugeln, Zylinder oder durch Spline-Flächen begrenzte Körper), die ebenfalls als Mitglieder einer 3D-Objektgruppe erlaubt sein sollen.

- Es besteht Bedarf an der Modellierung von Anwendungsstrukturen mit varianten Teilen. Das entsprechende Modellierungskonzept muß aber auch so flexibel sein, daß es jederzeit um neue Typvarianten (Alternativen) erweiterbar ist.

Ein weiteres Problem ergibt sich aus dem traditionellen Umgang mit Objekttypen in objektorientierten Programmiersprachen. Dort werden Objekte als Instanzen eines bestimmten Typs *t* erzeugt, und ein Objekt behält bis zum Ende seines Lebenszyklus diesen Typ *t* als seinen Instanziierungstyp, der gleichzeitig der sog. *minimale Typ* des Objektes ist. Üblicherweise stehen Typen in einer Subtyp-Beziehung (*isa*-Beziehung) zueinander, und ein Objekt kann überall dort eingesetzt werden, wo ein Objekt des Typs *t* oder eines direkten oder indirekten Supertyps von *t* erwartet wird (Substituierbarkeit). Zur Laufzeit wird für das dynamische Binden (*late binding*) der tatsächliche Typ eines Objektes ermittelt, und dies ist gerade sein Instanziierungstyp. Die konstante Typzugehörigkeit eines Objektes zu seinem Instanziierungstyp ist jedoch problematisch zu bewerten. Es soll nun ein klassisches Beispiel angegeben werden, bei dem ein dynamischer Typwechsel wünschenswert ist: Die Typen `Student` und `Angestellter` sind voneinander unabhängige Spezialisierungen des Typs `Person` (vgl. Abb. 1.7).

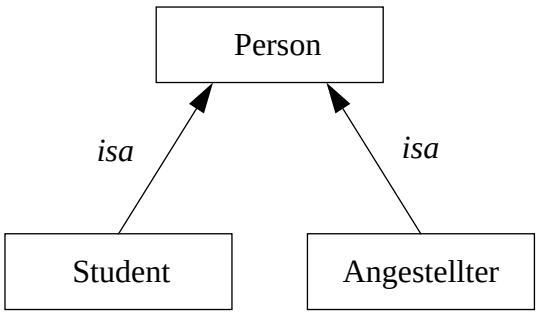


Abbildung 1.7: Zwei unabhängige Spezialisierungen des Typs `Person`

Es ist denkbar, daß eine Person, genauer ein als `Person` erzeugtes Objekt, aufgrund von Ereignissen in der realen Welt (Immatrikulation, Arbeitsvertrag) zu einem späteren Zeitpunkt den Typ `Student` und ggf. zu einem weiteren Zeitpunkt sogar noch den Typ `Angestellter` hinzugewinnt. Die verbreiteten objektorientierten Programmiersprachen sind darauf angewiesen, ein neues Objekt zu instanziieren, die Daten aus dem Zustand des bisherigen Objektes zu kopieren und danach das ursprüngliche Objekt zu löschen. Äußerst nachteilig ist an dieser Lösung, daß sie mit einem Wechsel der Objektidentität verbunden ist. Dieselbe Beobachtung trifft auch für die in [Pau94] vorgeschlagenen Erweiterungen des Datenmodells zu.

- Im Rahmen des in dieser Arbeit zu entwickelnden Datenmodells soll das dynamische Hinzufügen und Abgeben von Typen (in [WJS94] als *dynamische Spezialisierung* bezeichnet) unter Beibehaltung der Identität von Objekten möglich sein.

Die dynamische Spezialisierung wirft jedoch auch neue Fragen auf: Hat im obigen Beispiel ein Objekt sowohl den Typ `Student` als auch den Typ `Angestellter` hinzugewonnen, dann hat das Objekt offensichtlich keinen eindeutig bestimmten minimalen Typ mehr. Es ist dann auch fraglich, ob dynamisches Binden in dieser Situation sinnvoll ist.

Modellierung von Beziehungen

Es wurde im Zusammenhang mit dem Bibliotheksbeispiel bereits darauf hingewiesen, daß sich das eNF^2-Datenmodell gut zur Entwicklung von (Teil-)Schemata eignet, deren Schachtelungsstrukturen besonders gut an die Anforderungen eines jeweils gegebenen Verarbeitungsvorganges angepaßt sind. Die Erfahrung im Umgang mit eNF^2-Strukturen hat uns gezeigt, daß man bei der Erstellung eines Schemas oft dazu verleitet wird, eine auf einen bestimmten Verarbeitungsvorgang abgestimmte „Sicht“ auf das Gesamtschema in den Vordergrund zu stellen.

Betrachten wir dazu die Tabelle `Leser` aus Abb. 1.1. Die Daten über Vormerkungen sind in die Tabelle `Leser` integriert und werden deshalb „einseitig“ aus der Sicht des Lesers erfaßt. Ist man nun an einer Tabelle interessiert, in der zu allen Büchern die Vormerkungen als Paare aus Reservierungsdatum und Lesernummer zugeordnet werden, läßt sich dies auf der Grundlage der Tabellen `Bestand` und `Leser` nur über eine Anfrage erreichen. In HDBL [PT86, LPS91], der Anfragesprache von AIM-P, läßt sich die gewünschte Tabelle mit folgender Anfrage erzeugen:

```
SELECT
    [ ISBN: b.Signatur,
      Titel: b.Titel,
      Vormerkungen:
            SELECT
                [ RDatum: v.VDatum,
                  Leser: l.LeserNr,
                ]
            FROM l in Leser, v in l.Vormerkungen
            WHERE b.Signatur = v.Signatur
    ]
FROM b IN Bestand
```

Die Formulierung dieser Anfrage dürfte für Nicht-Spezialisten schwierig sein. Die einseitige Erfassung der Beziehung `Vormerkung` aus Abb. 1.8 in der Tabelle `Leser` hat demnach den schwerwiegenden Nachteil, daß eine Anfrage, in der dieselbe Beziehung lediglich in anderer Richtung traversiert werden soll, über eine vergleichsweise komplizierte Anfrage ausgedrückt werden muß.

Es ist deshalb mehr als fraglich, ob im logischen Schema, auf dessen Grundlage ja alle Anfragen formuliert werden, bereits eine solch starke Ungleichgewichtung von „Beziehungssichten“ verankert werden sollte.

Man beachte, daß die Schachtelungsstruktur der Tabelle `Leser` die Beziehung `Vormerkung` aus dem ER-Diagramm in Abb. 1.8 „richtet“, wie dies in Abb. 1.8 (a) durch die Pfeilrichtung

angedeutet ist. Die oben formulierte HDBL-Anfrage ist im wesentlichen nichts anderes als die Umkehrung der Pfeilrichung, wie dies in Abb. 1.8 (b) dargestellt ist.

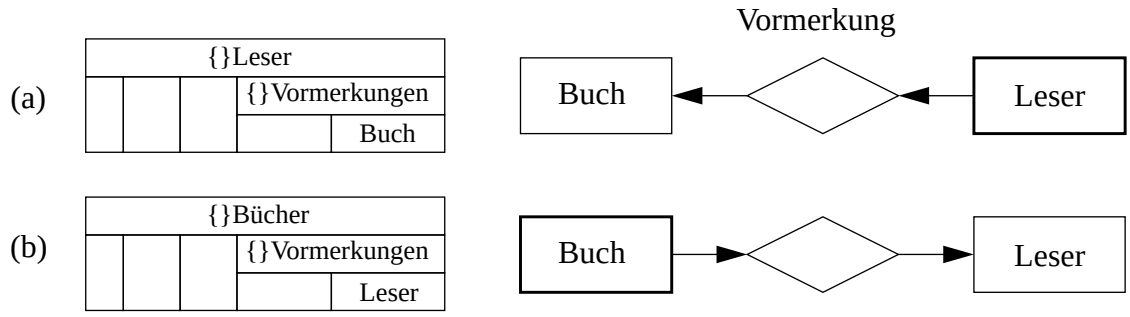


Abbildung 1.8: Schachtelungsstruktur als gerichteter Graph im ER-Schema

Neben einem Hinweis auf die Schwierigkeiten bei der Formulierung obiger Anfrage lohnt sich auch ein Blick auf die möglichen Ausführungspläne. Der „naive“ Ausführungsplan würde die Anfrage in drei geschachtelten Schleifen abarbeiten: In einer äußeren Schleife wird dabei über alle Tupel in `Bestand` iteriert, in der zweiten jeweils über *alle* Tupel in `Leser` und in der inneren über alle Reservierungen des jeweiligen Lesers. Es ist klar, daß dies äußerst ineffizient ist. Besser ist es, zunächst eine temporäre „flache“ Tabelle `Temp` als Ergebnis der Anfrage

```
SELECT
  [ Leser: l.LeserNr,
    VDatum: v.VDatum,
    Signatur: v.Signatur
  ]
FROM l in Leser, v in l.Vormerkungen
```

zu erzeugen, `Temp` nach `Signatur` zu sortieren und dann aus `Temp` und `Bestand`[7] in einem einzigen parallelen Durchlauf durch beide Tabellen das gewünschte Ergebnis zusammenzustellen.

Hier wird deutlich, daß durch die „einseitige“ Schachtelung bzgl. der m:n-Beziehung `Vormerkung` die Effizienz der Anfrageauswertung sogar noch hinter einer „flach“-relationalen Lösung zurückfällt, sofern auf interner Ebene nicht weitere Hilfsstrukturen (Indizes) existieren: Im Relationenmodell wird im Schema für eine m:n-Beziehungen eine „Verknüpfungsrelation“ definiert und instanziiert, während bei der Ausführung obiger Anfrage auf der Basis der Tabellen `Leser` und `Bestand` diese Verknüpfungsrelation jedesmal in der Gestalt von `Temp` erst erzeugt werden muß!

- Der zunächst zu beobachtende Vorteil geschachtelter Modelle wie (e)NF^2, daß die Schachtelungsstruktur von Tabellen gerade den benötigten Zugriffspfaden entspricht, wird also zu einem Nachteil, wenn der Datenbankentwurf so „unsymmetrisch“ ist, daß eine einseitige Bevorzugung bestimmter Richtungen beim Traversieren von Beziehungen vorliegt.

Will man etwa auch die Traversierung der Instanzen der Beziehung `Vormerkung` in der Richtung von Büchern zu Lesern in einem eNF^2-Datenbankschema adäquat unterstützen, so ist

7. Unter der Annahme, daß `Bestand` ebenfalls nach `Signatur` sortiert ist.

es angebracht, die in Abb. 1.8 (b) gezeigte Schachtelung ebenfalls in das logische Datenbankschema zu integrieren. Hierzu sind im Rahmen des eNF^2-Modells zwei Alternativen denkbar: Entweder gibt es eine separate Tabelle `Vormerkungen` mit der in Abb. 1.8 (b) angedeuteten Struktur, oder aber die Struktur wird als weiteres (mengenwertiges) Attribut den Bücher-Tupeln in der Tabelle `Bestand` hinzugefügt. In beiden Fällen führt dies auf Instanzenebene zu Redundanz, denn die Instanzen der Beziehung `Vormerkung` werden an verschiedenen Stellen repliziert. Auch hier gilt es, Inkonsistenzen infolge der Redundanz zu verhindern.

- Es besteht Bedarf an kontrollierter Redundanz auf der Instanzenebene: Soll in einem Datenbankschema mehr als eine „Sicht“ auf die Beziehung enthalten sein, dann liegt auf jeden Fall Redundanz auf der Instanzenebene vor, die zur Laufzeit besondere Kontrollmechanismen notwendig macht.

Wir plädieren dafür, im logischen Modell den „neutralen“ Charakter von Beziehungen zwischen Anwendungsobjekten beizubehalten. Wir werden dazu ein spezielles Beziehungskonstrukt einführen, das an die *relationship types* der ER-Modellierung erinnert, jedoch auch die Spezifikation verschiedener „Beziehungssichten“ einschließt und gleichzeitig für deren Synchronisation sorgt, so daß trotz Redundanz keine Inkonsistenzen auftreten. Jeder Komponente einer Beziehung kann eine individuelle Beziehungssicht zugeordnet werden. Für ein Objekt, das Komponente einer Beziehung ist, kann auf die Attribute der relevanten Beziehungssicht wie auf jedes andere Attribut seines Typs zugegriffen werden. Werden Bücher und Leser als Instanzen der Objekttypen `Buch` und `Leser` modelliert, dann werden wir für die Beziehung `Vormerkung` zwei Sichten definieren können, die jeweils ein Attribut `Vormerkungen` besitzen. Dieses Attribut kann dann für den Zugriff auf die Beziehungsinstanzen aus der jeweiligen Sicht verwendet werden:

Ist `b` ein Objekt vom Typ `Buch`, dann gibt `b.Vormerkungen` einen Wert vom Typ `{[L: Leser, V: date]}` zurück, der die Beziehungsinstanzen aus der Sicht eines Buches angibt. Ist umgekehrt `l` ein Objekt vom Typ `Leser`, dann gibt `l.Vormerkungen` einen Wert vom Typ `{[B: Buch, V: date]}` zurück, der die Beziehungsinstanzen aus der Perspektive des Lesers angibt.

Integritätsbedingungen

Eine weitere Gruppe von Problemen ergibt sich aus der mangelnden Berücksichtigung von *Integritätsbedingungen* in den bisherigen Arbeiten zum eNF^2-Modell. Lediglich in [PA86], einer frühen Veröffentlichung zu AIM-P, werden Schlüsselbedingungen kurz erwähnt. So werden auch in den Implementierungen des eNF^2-Modells – genannt seien wiederum AIM-P und der ESCHER-Prototyp –, keine Integritätsbedingungen unterstützt[8].

Dazu seien einige Beispiele, wiederum aus der Bibliotheksanwendung, genannt:

- Es ist dafür zu sorgen, daß keine Lesernummer doppelt vergeben wird, d.h. `LeserNr` muß Schlüssel in der Menge `Leser` sein. In ESCHER ist die Formulierung solcher *Schlüsselbedingungen* bisher nicht vorgesehen.
- Es darf nicht erlaubt sein, unter den Ausleih- oder Vormerkungsdaten in `Leser` eine Exemplarnummer oder Signatur anzugeben, die in `Bestand` nicht vorhanden ist. Dies

8. Laut [LPS91] können in AIM-P immerhin fixe oder maximale Kardinalitäten für Listen und Multimengen angegeben werden.

sollte wie beim Relationenmodell über eine *Fremdschlüsselbedingung* (bzw. *Inklusionsbedingung*) sichergestellt werden.

Schlüsselbedingungen sind unabdingbar für die eindeutige Identifikation von Anwendungsobjekten innerhalb der Datensammlung. Sie sollten für Kollektionen auf allen Ebenen formuliert werden können, so z.B. auch für das mengenwertige Attribut `Vormerkungen`, das in die Tabelle `Leser` „hineingeschachtelt" ist: Will man fordern, daß jedes Buch von einem Leser nicht mehrfach vorgemerkt werden darf, dann muß `Signatur` Schlüssel in jeder Menge `Vormerkungen` sein. Fremdschlüsselbedingungen sind essentiell für die Einhaltung der referentiellen Integrität. Beide Arten von Integritätsbedingungen haben sich bereits für das „flache" Relationenmodell als äußerst wertvolle Konzepte erwiesen, die maßgeblich zur Konsistenz der Datensammlung beitragen.

- Es fehlt bisher auch die Möglichkeit zur Spezifikation von Kardinalitätsbedingungen für Beziehungen. In die Tabelle `Bestand` ist die 1:n-Beziehung `Inventarisierung` integriert. Es wird verlangt, daß ein Exemplar (vernünftigerweise!) genau einem Buch zugeordnet werden kann. Jedoch hindert uns jedoch bisher nichts daran, dieselbe Exemplarnummer auch noch unter einem anderen Buch einzutragen. Dies würde aus der 1:n- eine m:n-Beziehung machen.

Es muß daher möglich sein, im Datenbankschema auszudrücken, daß es sich bei der Beziehung `Inventarisierung` um eine 1:n-Beziehung handelt, d.h. es ist die Gültigkeit der Bedingung

$$\forall\, b_1, b_2 \in \text{Buecher}: b_1 \neq b_2 \Rightarrow b_1.\text{Exemplare} \cap b_2.\text{Exemplare} = \varnothing$$

einzufordern. Analoges gilt für die 1:n-Beziehung „Ausleihe": Falls versucht wird, ein und dasselbe Buchexemplar mehrfach auszuleihen, sollte dies erkannt und der Entleihvorgang unterbrochen werden.

Schließlich ist auch die Unterstützung elementarer Wertebedingungen, die den Wertebereich von Attributen einschränken, wünschenswert. Einige Beispiele seien genannt:

- Die `LeserNr` jedes Lesers darf nicht nullwertig sein (vgl. Abb. 1.1)
- Der Nachname eines Lesers darf nicht der leere String sein
- Jeder Leser darf höchstens 10 Bücher vormerken
- Die Liste der Autoren eines Buches darf nicht leer sein (vgl. Abb. 1.3)

Im SQL/92Standard für das Relationenmodell können vergleichbare Integritätsbedingungen als sog. *column constraints* bereits ausgedrückt werden [DD93].

Alle genannten und vergleichsweise elementaren Integritätsbedingungen werden durch das bisher dem ESCHER-Prototypen zugrundeliegende Datenmodell nicht unterstützt. Es besteht daher Bedarf an der Spezifikation von elementaren Integritätsbedingungen für ein Datenbankschema und deren Überwachung zur Laufzeit. Wollte man weiterhin auf Integritätsbedingungen verzichten, liefe man Gefahr, in diesem Punkt noch hinter die Möglichkeiten des Relationenmodells zurückzufallen.

Eng im Zusammenhang mit der Integritätssicherung steht der Begriff der Transaktion. Bisher spielten Transaktionen in den Arbeiten zu ESCHER keine Rolle, auch nicht im Erweiterungsvorschlag [Pau94]. Wollte man weiterhin auf explizite, durch den Benutzer initiierbare Transaktionen verzichten, dann müßte zur Gewährleistung der Integrität jeder einzelne

modifizierende Zugriff auf die Datenbank wie eine Transaktion behandelt werden. Die Zusammenfassung einer ganzen Folge von Operationen zu einer Transaktion als einer neuen atomaren Einheit hat den Vorteil, daß während der Ausführung einer Transaktion Konsistenzverletzungen zugelassen sind. Wir können daher zwischen zwei Arten von Integritätsbedingungen unterscheiden:

- Die „strengen“ (bzw. „harten“) Integritätsbedingungen erfordern ihre sofortige Überprüfung, sobald der Verdacht einer Integritätsverletzung besteht.
- Die Überprüfung der „weichen“ Integritätsbedingungen wird dagegen bis zum Abschluß der Transaktion zurückgestellt.

Der SQL/92-Standard unterscheidet entsprechend zwischen *immediate* und *deferred constraints*.

1.4 Ziele und Aufbau der Arbeit

Die im vorangegangenen Abschnitt dargestellten Probleme machen die Defizite des eNF^2-Modells deutlich, das bislang als Datenmodell des ESCHER-Prototyps verwendet wurde. Das globale Ziel dieser Arbeit ist die Überwindung der oben aufgeführten Defizite, was wir durch *semantische Anreicherung des eNF^2-Datenmodells* erreichen werden. Wir legen dabei sehr großen Wert auf eine formale Beschreibung des Datenmodells und aller vorgestellten Konzepte. Einer der Hauptbeiträge dieser Arbeit soll – neben der Vorstellung der Modellerweiterungen „an sich“ – gerade eine stärkere Formalisierung der vorgestellten Konzepte sein, als dies in bisherigen Arbeiten zum eNF^2-Modell und seinen Erweiterungen der Fall war. Deshalb ist diese Arbeit den Forschungsaktivitäten zuzuordnen, die sich mit den Grundlagen der Datenmodellierung beschäftigen.

Das in dieser Arbeit zu entwickelnde Datenmodell nennen wir **ESCHER$^+$**.

Im einzelnen können die Beiträge dieser Arbeit in folgenden Punkten zusammengefaßt werden:

- Formale Beschreibung des Datenmodells ESCHER$^+$, das in seinem Strukturteil die folgenden Eigenschaften besitzt:
  - Für jedes Datenbankschema kann ausgehend von vordefinierten Basistypen und Konstruktoren ein anwendungsspezifischer Typvorrat definiert werden. Von zentraler Bedeutung sind dabei die Objekttypen, deren Instanzen Objekte im objektorientierten Sinn sind.
  - Aufzählungstypen und variante Typen sollen den benutzerdefinierbaren Typvorrat ergänzen.
  - Die Grundidee des eNF^2-Modells der Konstruktion komplexer Wertebereiche ausgehend von einer vorgegebenen Menge von Typen und Konstruktoren wird beibehalten, so daß ESCHER$^+$ tatsächlich das eNF^2-Datenmodell subsumiert. Neben Mengen, Listen und Tupeln soll ESCHER$^+$ auch Multimengen und Felder (Arrays) unterstützen.
  - Eine *isa*-Beziehung auf der Menge der Objekttypen steuert die Vererbung struktureller und operationaler Eigenschaften.
  - Multiple Typzugehörigkeit und dynamische Spezialisierung für Objekttypen: Eine

Instanz eines Objekttyps soll während seiner Lebensdauer unter Beibehaltung ihrer Identität weitere Typen annehmen und auch wieder abgeben können.

  - Der Zugriff auf die Instanzen soll wie beim ESCHER-Prototyp über Tabellen erfolgen. Persistenz in ESCHER$^+$ soll äquivalent zur Erreichbarkeit über eine Tabelle sein.
  - Metamodellierung: Jedes benutzerdefinierte Datenbankschema ist Instanz eines vordefinierten Metaschemas. Somit können Schemadaten wie gewöhnliche Datenobjekte behandelt werden („Uniformität“ aller Daten). Das Metaschema ist ebenfalls ein ESCHER$^+$-Datenbankschema mit der Eigenschaft, daß es sich selbst als Instanz enthält („Selbstreferenzierung“).

- Formale Einführung von Baumrepräsentationen für Datenobjekte:
  - Es werden beschriftete Bäume eingeführt, die als anschauliche und implementationsnahe Repräsentation komplexer Datenobjekte dienen. Damit wird eine Brücke zu rein graph-basierten Modellen geschlagen.
  - Eine Baumrepräsentation ist ein Paar aus einem Wertebaum und einem Strukturbaum, wobei letzterer die für den Wertebaum benötigte Typinformationen enthält.
  - Die Baumrepräsentationen bilden die Grundlage für die formale Semantik des operationalen Teils von ESCHER$^+$. Dem Datenmodell soll eine Menge vordefinierter Basisoperationen zugrundeliegen. Die Semantik aller Basisoperationen ist so zu wählen, daß durch die Ausführung einer Operation die Eigenschaft eines beschrifteten Baumes, ein Datenobjekt zu repräsentieren, nicht zerstört wird. Nur so bleiben Baumrepräsentationen und Datenobjekte jederzeit austauschbare „Sichtweisen“.
  - Eine Datenbankinstanz besteht aus einer Menge von Baumrepräsentationen. Für Knoten in beschrifteten Bäumen, die abstrakte Objekte repräsentieren, existieren „Interpretationen“ in Form von Querverweisen auf die Wurzelknoten anderer beschrifteter Bäume, welche die „konkreten“ Zustände des Objektes darstellen.
- Beiträge zum operationalen Teil des Datenmodells:
  - Einführung des Begriffs „Laufzeitinstanz“: In einer Laufzeitinstanz werden neben Baumrepräsentationen für die persistenten Datenobjekten (d.h. die zur Datenbankinstanz gehörenden Datenobjekte) auch Baumrepräsentationen für transiente Datenobjekte, die zur Laufzeit entstehen, sowie ein Stack zur Verwaltung der Prozedur- bzw. Funktionsaufrufe berücksichtigt. Laufzeitinstanzen bilden die Grundlage für die formale Beschreibung der Semantik aller Operationen.
  - Einführung eines internen Datentyps *Link*, dessen Werte Verweise auf Knoten in Baumrepräsentationen sind. Er bildet die Grundlage für die Semantik von Iteratoren auf Kollektionen.
  - Angabe der Syntax und Semantik der Sprache BASESCRIPT: Sie ist als „primitive“ Basissprache (Kernsprache) des Datenmodells konzipiert. Eine Anweisungsfolge in BASESCRIPT kann schrittweise abgearbeitet werden und besteht im wesentlichen aus Aufrufen von Basisoperationen oder benutzerdefinierter Operationen, die auf den Datenobjekten einer Laufzeitinstanz operieren.
    Benutzerdefinierte Operationen und Methoden von Objekttypen werden in einer „höheren“ persistenten Programmiersprache implementiert (kodiert). BASESCRIPT ist als Zielsprache der Übersetzung dieses Codes vorgesehen.

  - Als „höhere“ persistente Programmiersprache dient in dieser Arbeit die Sprache SCRIPT$^+$, die eine Erweiterung und an die Spezifika von ESCHER$^+$ angepaßte Variante der in [Pau94] vorgestellten persistenten Programmiersprache SCRIPT ist. Die Erweiterungen umfassen u.a. zusätzliche syntaktische Elemente zum dynamischen Hinzufügen und Entfernen von Typen eines Objektes sowie dynamisches Binden bei Methodenaufrufen oder Attributzugriffen für Instanzen varianter Typen.
  - Hinsichtlich der Integration von benutzerdefinierten Operationen in das Datenmodell wird in dieser Arbeit eine andere Strategie als in [Pau94] verfolgt: Benutzerdefinierte Operationen sind in [Pau94] nichts anderes als funktionswertige Attribute, deren Werte (d.h. die Implementationen) wie bei jedem anderen Attribut von Benutzern modifiziert werden können. In ESCHER$^+$ sollen benutzerdefinierte Operationen als „höhere Konzepte“ [Heu92] zum Datenbankschema gehören. Somit sind in ESCHER$^+$ die Implementationen *benutzer*definierter Operationen nur für bestimmte Benutzer (DBAs, Anwendungsprogrammierer) modifizierbar.
- Erweiterung des Datenmodells um Konzepte und Mechanismen zur Gewährleistung der Konsistenz einer Datenbank:
  - Ergänzung des formalen Modells für ESCHER$^+$ um (flache) Transaktionen
  - Einführung verschiedener deklarativer Integritätsbedingungen: Innerhalb eines Datenbankschemas können neben Schlüsselbedingungen auch Inklusions- und Disjunktheitsbedingungen sowie einfache Wertebedingungen ausgedrückt werden.
  - Die automatische Überwachung der Integritätsbedingungen zur Laufzeit soll Aufgabe einer speziellen Komponente eines DBMS sein, des sog. *Integritätsmonitors*. Es wird vorgeschlagen, alle Integritätsbedingungen in interne ECA(*Event-Condition-Action*)-Regeln zu transformieren, auf deren Grundlage die Überwachung stattfindet.
  - Selektive Einkapselung und Lese/Schreibrechte für Attribute von Objekttypen und selektive Einkapselung benutzerdefinierter Operationen als zusätzliches Mittel der Konsistenzkontrolle
- Erweiterung des Datenmodells um ein Beziehungskonstrukt, das folgende Eigenschaften besitzt:
  - Beziehungen zwischen Anwendungsobjekten werden – ähnlich wie die *relationship types* des ER-Modells – in einem ESCHER$^+$-Datenbankschema explizit definiert.
  - Als Komponententypen einer Beziehung sind die benutzerdefinierten Objekttypen zugelassen. Für Beziehungen in ESCHER$^+$ gelten keine Einschränkungen hinsichtlich der Anzahl ihrer Komponenten oder der Anzahl ihrer Attribute.
  - Für jede Beziehung können Integritätsbedingungen (Schlüsselbedingungen, Inklusionsbedingungen, Wertebedingungen) angegeben werden.
  - Mit jeder Beziehung ist die Definition von *Beziehungssichten* für die verschiedenen Komponenten verknüpft. Über die Attribute der Beziehungssichten erfolgt der Zugriff auf die Beziehungsinstanzen, wie er aus der Perspektive der jeweiligen Komponente erforderlich ist. Da für jede Komponente eine eigene Beziehungssicht existieren kann, ist man hinsichtlich der Abbildung einer Beziehung in ein Datenbankschema nicht mehr auf die oben kritisierte „einseitige“ Schachtelung angewiesen, wie es im eNF^2-Modell noch der Fall ist.

– Aus der Existenz verschiedener Beziehungssichten für eine Beziehung ergibt sich Redundanz auf der Instanzenebene, die jedoch zur Laufzeit kontrolliert werden kann, indem entsprechende ECA-Regeln für einen Integritätsmonitor erzeugt werden.

Die Arbeit ist wie folgt gegliedert:

In Kapitel 2 geben wir eine Übersicht über Entwicklungslinien der Datenmodellierung. Es werden die wesentlichen Konzepte der semantischen Datenmodellierung vorgestellt, die auch für logische Datenmodelle von Bedeutung sind. Danach wird auf das z.Z. im kommerziellen Bereich dominierende Relationenmodell und dabei insbesondere auf seine Nachteile eingegangen. Schließlich werden die bekannten (und eher konservativen) Strategien zur Erweiterung des Relationenmodells den sog. objektorientierten DBMS gegenübergestellt.

In Kapitel 3 wird der strukturelle Teil des ESCHER$^+$-Datenmodell formal beschrieben. Ebenfalls in diesem Kapitel erfolgt die Einführung von Baumrepräsentationen für Datenobjekte auf der Grundlage von beschrifteten Bäumen. Dies führt zu einer alternativen Sichtweise auf das Datenmodell in Form eines graphbasierten Ansatzes.

In Kapitel 4 behandeln wir den operationalen Teil des ESCHER$^+$-Datenmodells. Es geht zunächst um die Formalisierung der Semantik der vorgegebenen Basisoperationen. Das Datenmodell erlaubt die Definition benutzerdefinierter Operationen. Es wird die Sprache BASESCRIPT vorgestellt, die sich aufgrund ihrer Einfachheit für die Angabe der Semantik benutzerdefinierter Operationen eignet. Es werden Beispiele für die Übersetzung der „höheren“ persistenten Programmiersprache SCRIPT$^+$ nach BASESCRIPT gegeben.

In Kapitel 5 wird das Datenmodell zunächst um ein Transaktionskonzept erweitert. Es werden dann verschiedene Klassen von Integritätsbedingungen eingeführt. Ferner wird ein Konzept zur selektiven Einkapselung von Attributen und Operationen vorgestellt, das ebenfalls der Konsistenzerhaltung dient.

In Kapitel 6 motivieren wir zunächst die Notwendigkeit der Erweiterung des Datenmodells um ein weiteres Modellierungskonstrukt zur direkten Abbildung von Beziehungen zwischen Anwendungsobjekten in einem Datenbankschema. Dem formalen Datenmodell wird mit der *Beziehung* dann ein weiterer fundamentaler Baustein hinzugefügt. Gleichzeitig werden Beziehungssichten eingeführt, die jeweils mit einer Komponenten einer Beziehung verknüpft werden. Ferner wird die Frage erörtert, auf welche Weise die Redundanz in den Daten kontrolliert werden kann, die aus dem Vorliegen mehrerer Beziehungssichten für eine Beziehung resultiert.

Schließlich fassen wir in Kapitel 7 die Ergebnisse dieser Arbeit zusammen und geben einen Ausblick auf Ansatzpunkte für weitere Forschungsarbeiten.

Kapitel 2

Entwicklungslinien der Datenmodellierung

In diesem Kapitel wird ein Überblick über die relevanten Entwicklungslinien der Datenmodellierung gegeben. Dabei werden Stärken und Schwächen der einzelnen Modell-„Typen“ diskutiert sowie Gemeinsamkeiten unterschiedlicher Ansätze herausgestellt. In dieser Übersichtsdarstellung wird auf präzise Definitionen der verwendeten Begriffe verzichtet, da wir von einem Vorwissen des Lesers auf dem Gebiet der Datenmodellierung ausgehen. Dennoch soll der Überblick nicht zu knapp sein, da die Reflexion über bisherige Entwicklungen die Voraussetzung für weitere Fortschritte in der Datenmodellierung ist.

Wir definieren ein *Datenmodell* informell gemäß [Bro84, Nav92, Dat95] als eine Menge von formalen Konzepten, die aus folgenden Komponenten besteht:

- **Komponente 1**: Beschreibung von Anwendungsobjekten und ihren Beziehungen zueinander (*struktureller Teil*)
- **Komponente 2**: Operationen zur Manipulation von Anwendungsobjekten (*operationaler Teil*)
- **Komponente 3**: Integritätsbedingungen

Wie bereits im einleitenden Kapitel herausgearbeitet wurde, sollte ein logisches Datenmodell angestrebt werden, das tatsächlich alle drei Komponenten in geeigneter Weise unterstützt.

2.1 Semantische Datenmodelle

Zunächst soll mit den semantischen Datenmodellen (SDMs) eine Klasse von Datenmodellen vorgestellt werden, die historisch nach den in den darauffolgenden Abschnitten erwähnten implementationsnahen Datenmodellen, also auch nach dem Relationenmodell, aufkam. Die SDMs werden bereits jetzt besprochen, da sie im Zuge einer Anwendungsentwicklung bereits in der Phase der *Anforderungsanalyse* und des *konzeptuellen Designs* eine Rolle spielen, also *vor* den an ein konkretes DBMS gebundenen Datenmodellen in Erscheinung treten.

Die Aufgabe der semantischen Datenmodellierung liegt darin, ein *konzeptuelles Schema* als Instanz eines SDMs zu entwerfen, das ein möglichst kongruentes Abbild des relevanten Ausschnitt der „realen Welt“ („Miniwelt“, „Universe of Discourse (UoD)“) darstellt. Allerdings muß festgestellt werden, daß auch SDMs sich vielfach auf den strukturellen Aspekt konzentrieren und die anderen oben genannten Komponenten eines Datenmodells vernachlässigen. Anwendungsspezifische Operationen fehlen – wie beim Relationenmodell – oft völlig.

Obwohl einige SDMs sich für die konzeptuelle Phase als sehr nützlich herausgestellt haben gibt es nur sehr wenige direkte DBMS-Implementierungen auf der Grundlage eines SDMs, die zudem über das Prototypenstadium nicht hinauskamen (siehe [GKP92] für eine Übersicht). Der Nutzen bleibt oft auf die Verwendung einer intuitiv gut verständlichen *graphischen Notation* des Strukturteils des Datenmodells beschränkt. Die meisten SDMs sind zunächst nur informell beschrieben und haben deshalb keine formale Semantik. Dies hat sich durch Forschungsaktivitäten in diesem Sektor in den vergangenen Jahren verbessert (vgl. die Ausführungen zum Entity-Relationship-Modell weiter unten), führte jedoch auch zu einer Vielzahl von Modell-„Dialekten“.

Es ist gängige Praxis, zunächst ein konzeptuelles Schema als Instanz eines SDMs zu entwikkeln und dieses dann in einem Folgeschritt, dem *Datenbankentwurf*, in ein logisches Schema zu transformieren, das Instanz des von einem konkreten DBMS implementierten Datenmodells ist, wie dies weiter unten in Abb. 2.6 dargestellt ist. Liegt das logische Schema einmal fest, dann spielt das konzeptuelle Schema – abgesehen von Phasen der konzeptuellen Revision oder des Redesigns – keine Rolle mehr. Anfragen und Datenmanipulationen beziehen sich immer auf das i.d.R. abstraktere logische Schema.

Es stellt sich die Frage, wieso nicht bereits der konzeptuelle Entwurf auf der Grundlage eines von einem DBMS implementierten Datenmodells geschieht. Beispielsweise hat sich das Relationenmodell für die Phase des konzeptuellen Entwurfs als nicht geeignet erwiesen. Es ist in seinen Beschreibungsmitteln einfach zu „schlicht“, als daß es alle anwendungsspezifischen Semantiken in geeigneter Form (wenn überhaupt!) erfassen kann. Es bedarf einer größeren Vielfalt an Beschreibungsmitteln, um die geforderte Anwendungssemantik einfach, präzise und vollständig abbilden zu können. In den frühen Phasen steht noch sehr die Kommunikation mit den Anwendern im Vordergrund, und als Kommunikationsbasis eignen sich in erster Linier Datenmodelle mit „anschaulichen“ Konstrukten. Eine frühzeitige Modellierung auf der Basis des Relationenmodell ist für den Anwender zu abstrakt. SDMs zeichnen sich gerade dadurch aus, daß sie aufgrund einer größeren Anzahl differenzierter Modellierungskonstrukte Anwendungsobjekte direkter, anschaulicher und semantisch reichhaltiger repräsentieren können.

2.1.1 Abstraktionsmechanismen

In diesem Abschnitt werden die grundlegenden Abstraktionsmechanismen der semantischen Datenmodellierung stichpunktartig zusammengefaßt. Die Ausführungen sind bewußt allgemein gehalten, um verschiedene Datenmodelle auf dieser Basis vergleichbar zu machen.

- *Typisierung, Klassifikation*:
  Es wird die für eine Anwendung relevante Menge der Typen festgelegt. Alle Anwendungsobjekte müssen dann nach ihrer Typzugehörigkeit klassifiziert werden können. Sind zwei verschiedene Objekte gleichen Typs, dann werden sie durch dieselben statischen Eigenschaften (Attribute) beschrieben und haben dieselben dynamischen Eigenschaften (Anwendbarkeit derselben Operationen).

Ziel der Typisierung bzw. Klassifikation ist es, aus dem Kontinuum der Phänomene der realen Welt die für eine Applikation relevanten Anwendungsobjekte herauszufiltern. Oftmals wird die *Typextension* eines Typs, d.h. die Menge aller aktuell existierenden Objekte eines Typs, als *Klasse* bezeichnet[1]. Zur Beschreibung der statischen Eigenschaften werden vordefinierte *Basistypen* (*printable types* [AH87, HK87], z.B. integer, string), aber auch andere Anwendungsobjekte verwendet. Dies führt direkt zum nächsten Abstraktionsmechanismus, der Aggregation.

- *Aggregation*:
  Der Begriff Aggregation geht auf [SS77] zurück und bezeichnet auf Schemaebene eine Zusammenfassung von Attributen zu einer neuen Einheit. Eine Typdefinition ist deshalb immer mit einer Aggregation seiner statischen Eigenschaften verbunden. Auf der Instanzenebene findet eine Zusammenfassung der Werte bzw. Objekte, die den Attributen zugeordnet sind, zu einer neuen Einheit statt. Ist ein Objekt aus *Komponentenobjekten* zusammengesetzt, die oftmals von dem Objekt, das sie enthält, existenzabhängig sind, dann liegt eine „is part of“-Beziehung vor, die naheliegenderweise auch als Aggregation modelliert wird.

Von der Aggregation ist der nachfolgende Abstraktionsmechanismus sorgfältig zu unterscheiden.

- *Assoziation*:
  Während die „is part of“-Beziehung eine sehr enge Verknüpfung zwischen Objekten und ihren Komponentenobjekten darstellt, werden semantische Beziehungen, sofern es sich um eher „lockere“ Beziehungen zwischen Objekten handelt, über das Konstrukt der Assoziation ausgedrückt. Der Übergang zwischen Aggregation und Assoziation ist jedoch fließend, so daß es oft im Ermessensraum des Anwendungsentwicklers liegt, welches Konstrukt zum Einsatz kommt. Als „Faustregel“ kann angegeben werden, daß eine semantische Beziehung als Assoziation anzusehen ist, wenn die beteiligten Objekte auch unabhängig von der Beziehung existieren (z.B. Teilnahme einer Person an einem Kurs).

Die Assoziation ist am deutlichsten im Entity-Relationship-Modell [Che76] in Form des Relationship-Typs verwirklicht. Es wird häufig der Fehler gemacht, die Aggregation nicht von der Assoziation zu unterscheiden (die Relationship-Typen des Entity-Relationship-Modells werden dort unter Aggregation eingeordnet), vgl. z.B. [HK87].

1. In [AH87, Heu92] wird die Typextension als die *aktiven Domäne* des Typs bezeichnet.

- *Gruppierung*:
  Die Gruppierung faßt gleichartige Objekte oder Werte zusammen und ermöglicht so, mengenwertige Attribute adäquat abzubilden. Über die Gruppierung können auch benutzerdefinierte *Kollektionen* (Mengen, Listen oder Multimengen) gleichartiger Objekte gebildet werden.

In einigen Arbeiten (z.B. [EGH+92]) wird die Gruppierung mit der Assoziation gleichgesetzt, dem wir hier jedoch nicht folgen wollen.

- *Spezialisierung*:
  Unter Spezialisierung versteht man eine Beziehung zwischen Typen mit der Semantik, daß eine Instanz eines Typs gleichzeitig eine Instanz jeder ihrer *Supertypen* ist (*isa-Beziehung*). Demzufolge läuft die *isa*-Beziehung auf eine Teilmengenbeziehung zwischen den Typextensionen der an der *isa*-Beziehung beteiligten Typen hinaus. Gekoppelt wird mit der Spezialisierung i.d.R. ein *Vererbungsmechanismus*, der dafür sorgt, daß Eigenschaften (Attribute, anwendbare Operationen) sich automatisch von Instanzen des Supertyps an Instanzen des Subtyps vererben.

Vielfach wird die Generalisierung einfach als Spiegelbild der Spezialisierung angesehen. Einige Modelle verbinden mit der Generalisierung jedoch eine andere Semantik [AH87, Heu92, EGH+92, EN94]. In [EWH85] wird die Generalisierung auch als *Kategorisierung* bezeichnet, wohl um einer Verwechselung mit dem Spiegelbild der Spezialisierung aus dem Weg zu gehen. Diese Semantik der Generalisierung wird nun erläutert.

- *Generalisierung*:
  Während es bei einer *isa*-Beziehung sinnvoll ist, daß eine Instanz des Supertyps existiert, ohne daß sie gleichzeitig Instanz des Subtyps ist (Standardbeispiel: `Student ISA Person`), sind auch Beispiele denkbar, bei denen dies gerade nicht der Fall sein soll: Für verschiedenartige geometrische Objekte seien Typen definiert, etwa `Kreis`, `Polygon`, `Kurve` usw. Es ist nun sinnvoll, alle diese Typen als geometrische Objekte erkennen zu können und daher die genannten Typen zu einem Typ `GeoObject` zu generalisieren. Es macht aber keinen Sinn, `GeoObject`-Instanzen zu erzeugen, die *nicht* gleichzeitig einem der Subtypen `Kreis`, `Polygon`, `Kurve` usw. angehören. Die Generalisierung ist eine Beziehung zwischen einem Generalisierungstyp und einer Menge von Subtypen, die gerade diesen Sachverhalt ausdrückt.

Bei der Generalisierung steht das Herausfaktorisieren gemeinsamer Eigenschaften aus verschiedenen Typen im Vordergrund. Diese können bei den geometrischen Objekten z.B. die Koordinaten eines Ankerpunktes sein, der angibt, an welcher Position das Objekt gezeichnet werden darf. Dieses Attribut wird an die Subtypen des Generalisierungstyps vererbt.

Der Unterschied zwischen Spezialisierung und Generalisierung läßt sich auch mit dem Gedanken der *Typkonstruktion* erläutern, wie er in [AH87, EGH+92, Heu92] vertreten wird: Bezeichnet man mit σ (T) die Typextension eines Typs T, dann wird σ (`GeoObject`) aus σ (`Kreis`), σ (`Polygon`) usw. konstruiert:

$$\sigma(\texttt{GeoObject}) := \sigma(\texttt{Kreis}) \cup \sigma(\texttt{Polygon}) \cup \ldots \tag{1.1}$$

`GeoObject` ist dann gewissermaßen der nachgeordnete Typ, der aus den Typen `Kreis`, `Polygon` usw. entsteht. Bei genauerer Betrachtung der Generalisierung/Kategorisierung gemäß [EWH85, EGH+92, Heu92, EN94] stellt man fest, daß sie in (1.1) anstelle von „:=“ tatsächlich nur „$\subseteq$“ fordern. Für Gleichheit muß eine zusätzliche *Überdeckungsbedingung* ange-

geben werden. In einigen Situationen ist es auch sinnvoll, von der Überdeckungsbedingung in (1.1) abzugehen und „:=“ durch „⊆“ zu ersetzen: Definiert man in einer Bibliotheksanwendung den Typ `Ausleihobjekt` als Generalisierung der Typen `Buch` und `Zeitschrift`, dann ist die Bedingung

$$\sigma(\texttt{Ausleihobjekt}) \subseteq \sigma(\texttt{Buch}) \cup \sigma(\texttt{Zeitschrift}) \tag{1.2}$$

adäquat, wenn man neben den ausleihbaren Büchern und Zeitschriften auch weitere Bücher und Zeitschriften erfassen möchte. Man beachte, daß eine Modellierung durch Spezialisierung (`Buch ISA Ausleihobjekt`, `Zeitschrift ISA Ausleihobjekt`) die Teilmengenbeziehung in (1.2) genau umdrehen würde!

In objektorientierten Programmiersprachen besteht ein Zusammenhang zwischen der Generalisierung und dem sog. späten Binden (*late binding*). Durch Generalisierung zu `GeoObject` läßt sich auch der Name der Zeichenoperation, etwa `drawMe` herausfaktorisieren. Diese ist im Generalisierungstyp `GeoObject` eine virtuelle Operation mit leerer Implementation, und zur Laufzeit sorgt spätes Binden dafür, daß die korrekte Implementation zur Ausführung kommt. Voraussetzung ist dann aber eine zusätzliche Disjunktheitsbedingung, die besagt, daß eine Instanz eines Generalisierungstyps zu *genau einem* der generalisierten Typen gehört (d.h. ein `GeoObject` darf vernünftigerweise nicht gleichzeitig `Kreis` und `Polygon` sein). Diese *disjunkten Generalisierungen* haben viel Ähnlichkeit mit dem *Varianten*-Konstruktor, der in einigen Modellen zu finden ist [AH88, Pis89, WPC92, Pau94]. Die disjunkte Generalisierung geht auf [SS77], einer Pionierarbeit zum Thema Abstraktion, zurück.

- *Funktionen*:
  Einige SDMs verwenden Funktionen als Beschreibungsmittel. Im Functional Data Model (FDM) [Shi81] sind Entity-Typen und Funktionen sogar die einzigen Konstrukte des Modells, wobei noch einfache und mengenwertige Funktionen unterschieden werden. Ähnlich wie beim Relationenmodell zeigt sich jedoch, daß FDM zwar durch seine Einfachheit besticht, die Ausdrucksmittel aber zu abstrakt sind, um als Modell für die konzeptuelle Modellierung populär zu werden.

Abgeleitete Typen und abgeleitete Attribut gehören zwar nicht direkt zu den Abstraktionsmitteln, sind aber als weitere, wichtige Ausdrucksmittel zu nennen.

- *abgeleitete Typen und Attribute*:
  Einige Modelle erlauben die Definition abgeleiteter Typen aufgrund eines Prädikats (z.B. `TeureMitarbeiter` als alle diejenigen Mitarbeiter, deren Gehalt über einer gewissen Grenze liegt) oder die Definition abgeleiteter Attribute aufgrund einer Funktion (z.B. Alter berechnet aus Geburtsdatum und Tagesdatum).

Abgeleitete Typen und Attribute bilden einen ersten Schritt in Richtung *Sichtdefinition*.

2.1.2 Integritätsbedingungen

Um weitestgehende Kongruenz des konzeptuellen Schemas und der Instanzenmenge zur modellierten „realen“ Welt sicherzustellen, müssen Gesetzmäßigkeiten einer Anwendung, die nicht allein über den Strukturteil erfaßt werden können, in Form von (*semantischen*) *Integritätsbedingungen* formuliert werden. Sie bestimmen, welche Datenbankzustände bzw. welche Zustandsübergänge „zulässig“ sind. Nur bei Einhaltung aller Integritätsbedingungen zu

bestimmten Zeitpunkten (z.B. unmittelbar zu Beginn oder unmittelbar nach Abschluß einer Transaktion) ist die *Konsistenz* einer Datenbank zugesichert.

Nach [Nav92] werden folgende Arten von Integritätsbedingungen unterschieden:

- Die *inhärenten* Integritätsbedingungen ergeben sich unmittelbar aus im Datenmodell verankerten Regeln. So ist etwa die referentielle Integrität fest in das ER-Modell eingebaut: Komponenten einer Beziehung dürfen nur existierende Entitäten sein.
- Die *impliziten* Integritätsbedingungen sind im Rahmen der DDL ausdrückbare Bedingungen, die von einem implementierten System automatisch kontrolliert werden, wie etwa Schlüsselbedingungen oder Kardinalitätsbedingungen.
- Die *expliziten* Integritätsbedingungen sind allgemeinere anwendungsspezifische Integritätsbedingungen, die oft in einer eigens dafür konzipierten Subsprache der DDL in Form von Zusicherungen (*assertions*) ausgedrückt werden. In vielen Modellen können *Integritätsregeln* angegeben werden, die zur Laufzeit bei bestimmten Operationen eine Integritätsbedingung prüfen und bei Vorliegen einer Integritätsverletzung eine bestimmte Reaktion auslösen.

Auf einer anderen Ebene lassen sich Integritätsbedingungen in *statische* und *dynamische* Integritätsbedingungen unterteilen. Statische Integritätsbedingungen beziehen sich auf Datenbankzustände zu einem einzelnen Zeitpunkt. Dynamische Integritätsbedingungen lassen sich wiederum in *transitionale* Bedingungen, die die Zulässigkeit elementarer Zustandsübergänge Z_{alt} nach Z_{neu} definieren, und allgemeinere *temporale* Bedingungen einteilen [HS95]. Letztere kontrollieren die Zulässigkeit ganzer Zustandsfolgen. Eine deklarative Spezifikation dynamischer Integritätsbedingungen im Rahmen einer temporalen Logik wird in [Lip89, Saa91] vorgeschlagen. Daneben gibt es den Ansatz, *Transaktionen* als integritätserhaltende Transitionen zwischen Datenbankzuständen in den Mittelpunkt zu stellen und diesen deklarative Vor- und Nachbedingungen zuzuordnen [Ngu89, Lip89]. Nach [EGH+92] sind beide Ansätze für die konzeptuelle Modellierung einander ergänzend verwendbar.

Mit den genannten Arten von Integritätsbedingungen und der Spezifikation (konsistenzerhaltender) Transaktionen kann ein breites Spektrum anwendungsspezifischer Gesetzmäßigkeiten erfaßt werden. Gleichzeitig sind die oben genannten Vorschläge für implementierte Systeme in vielen Punkten zu ambitioniert, da eine effiziente automatische Überwachung allgemeiner Integritätsbedingungen noch jenseits der Möglichkeiten existierender DBMS liegt. Deshalb müssen weiterhin viele Integritätsbedingungen in den Anwendungsprogrammen „ausprogrammiert“ werden, wobei oft eine und dieselbe Integritätsbedingung redundant in verschiedenen Anwendungsprogrammen geprüft wird. Nach [Dit94] werden solche Integritätsbedingungen auch *extern* genannt, während die vom Datenmodell direkt unterstützten gerade die *internen* Integritätsbedingungen sind.

2.1.3 Beispiele für semantische Datenmodelle

Es sollen nun einige wichtige Vertreter unter den SDMs genannt werden. Ausführliche Übersichten zu SDMs sind in [HK87, PM88, GKP92] zu finden.

Das Entity-Relationship-Modell

Herausragender Vertreter unter den SDMs ist nach wie vor das hinlänglich bekannte Entity-Relationship-Modell (ER-Modell), das auf [Che76] zurückgeht. Eine aktuelle Monographie zur ER-Modellierung ist [BCN92]. In Abb. 1.8 wurde bereits ein typisches Beispiel eines ER-Diagramms gezeigt. Zentrale Konstrukte sind die *Entity*- und *Relationship*-Typen. Während Entity-Typen die realen Gegenstände abbilden, stellen Relationship-Typen alle Arten der Beziehungen zwischen Entitäten dar, also neben Assoziationen (als eher „lockeren“ Beziehungen) auch die „is part of“-Beziehungen (mit der starken Bindung zwischen der Entität und seinen Komponenten). Neben Schlüsselbedingungen, die die Identifizierung von Entitäten ermöglichen, sind *Kardinalitätsbedingungen* wichtige Integritätsbedingungen für Beziehungen. Dabei hat die sog. *min-max-Notation* in Form einer Teilnahmebedingung weite Verbreitung gefunden, die differenziertere Bedingungen zuläßt als die grobe Einteilung nach 1:1, 1:n und n:m-Beziehungen. Sie wurde bereits in Beispiel 1.3 erläutert (vgl. auch [LEW93] für eine erschöpfende Diskussion von Kardinalitätsbedingungen in SDMs).

Dem ER-Modell wird manchmal vorgeworfen, daß die Entscheidung zwischen einer Modellierung eines Phänomens der realen Welt als Entity oder als Relationship oft willkürlich ist. Beispielsweise ist `Ehe` sicher eine Beziehung zwischen zwei Personen, ebenso plausibel ist aber auch, eine `Ehe`-Instanz etwa mit der Hochzeitsurkunde zu identifizieren, also zu „verdinglichen“, wodurch `Ehe` einen Entity-Charakter bekommt. Letztendlich kann es keine festen Regeln geben, wann welches Konstrukt einzusetzen ist. Es ist aber nützlich, wenn Beziehungen in dem Sinne „verdinglicht“ werden können, daß sie selbst als Komponenten anderer Beziehungen einsetzbar sind.

Nachdem die ersten Arbeiten zum ER-Modell eher informeller Natur waren und sich zunächst sehr auf die beiden Konstrukte Entity und Relationship konzentrierten, wurde das ER-Modell im Laufe der Zeit um weitere Abstraktionsmechanismen erweitert, so daß heute viele *erweiterte* ER-Modelle existieren, die über vielseitigere strukturelle Ausdrucksmittel verfügen und für die auch formale Beschreibungen vorliegen. Zu nennen sind u.a. das EER-Modell von Teorey et.al. [TYF86], das Braunschweiger EER-Modell [EGH+92, GH91, Gog94], ECR [EWH85, EN94] und ERC+ [PS89, PS92]. Auch das aus der objektorientierten Modellierung bekannte OMT-Modell [BPR88, RBP+94] ist im Grunde ein erweitertes ER-Modell. Die wesentlichen Erweiterungen gegenüber [Che76] gehen in die Richtung Spezialisierung/Generalisierung sowie der direkten Modellierung komplex strukturierter Entity-Typen (ursprünglich hatten – ähnlich wie bei den Tupeln im Relationenmodell – alle Entity-Typen nur „flache“ Attribute). Letzteres bedeutet eine Entlastung der Relationships, die nicht länger auch noch die „is part of“-Beziehung abbilden müssen. In HERM (Higher-Order ER-Modell) [Tha90, Tha91] werden Relationships so verallgemeinert, daß auch andere Relationships als Komponenten zugelassen sind. In [TWB+89] werden Möglichkeiten untersucht, die oft sehr großen ER-Diagramme durch weitere Abstrahierung (Clusterbildung) dem Benutzer zugänglicher zu machen, indem ein ER-Diagramm unter einem gewünschten Abstraktionslevel betrachtet werden kann.

An dieser Stelle sei besonders auf das Projekt DAMOKLES [ADG+87, ADL+91] hingewiesen, da das zugrundeliegende Datenmodell EODM (*E*ntwurfs*o*bjekt-*D*aten*m*odell) ebenfalls ein ER-Modell darstellt, das um die Möglichkeiten der Modellierung komplex strukturierter

Entity-Typen und ein Versionskonzept erweitert wurde. DAMOKLES kann als eine der wenigen „größeren“ Implementationen eines ER-Modells angesehen werden.

Für die Transformation eines ER-Schemas in ein logisches Schema (Datenbankentwurf) gibt es in der Literatur eine Vielzahl von Vorschlägen. Eine Übersicht bietet [FV95]. Wegen der praktischen Relevanz befassen sich die meisten Arbeiten mit Transformationen in das Relationenmodell. Neben eher informellen oder heuristischen Verfahren [Lin85, TYF86, RBP+94, EN94], die zudem nicht informationserhaltend sind[2], gibt es auch formale und informationserhaltende Verfahren. Zu nennen sind u.a. die Algorithmen von Markowitz et al. [MMR86, Mar90, MS92] auf der Grundlage des EER-Modells aus [TYF86]. Erzeugt wird ein Relationenschema in Boyce-Codd-Normalform (BCNF) mit Inklusionsbedingungen und Einschränkungen für Nullwerte. Die Arbeiten von Markowitz et al. sind insbesondere deshalb interessant, weil es mit ERDRAW [SM93] ein frei verfügbares graphisches Tool gibt, mit dem EER-Schemata erzeugt und in Relationenschemata für die verbreiteten kommerziellen DBMS (Ingres, Sybase, Informix, ORACLE) übersetzt werden können. Informationserhaltend ist auch der Algorithmus aus [MR92], der ein ER-Schema zunächst in eine Normalform überführt und dann ebenfalls ein Relationenschema in BCNF liefert. Laut [FV95] wird lediglich in [SC88] ein weiterer Algorithmus zur informationserhaltenden Transformation in ein Relationenschema angegeben.

IFO

Das IFO-Modell [AH87] wird an dieser Stelle genannt, weil viele seiner Abstraktionsmitteln durchaus mit denen des eNF^2-Modells, das die Grundlage für ESCHER bildet, vergleichbar sind. Ferner hatte IFO großen Einfluß auf den strukturellen Teil vieler objektorientierter Datenbankmodelle. So baut z.B. das Datenmodell EXTREM [Heu89], das auch das Referenzmodell in [Heu92] ist, auf IFO-Konzepten auf. Das O_2-Datenmodell [LRV88, Deu91, BDK92] läßt ebenfalls IFO-Traditionen deutlich erkennen.

Die Ausdrucksmittel von IFO sollen an dem beliebten [KTW90, Heu92, Pau94, SP94] Bücherbeispiel veranschaulicht werden. Die Aggregation wird graphisch durch $\otimes$ ausgedrückt, die Gruppierung durch $\circledast$. Damit können hierarchische Strukturen, in IFO *Fragmente* genannt, aufgebaut werden, wie in Abb. 2.1 dargestellt.

IFO ist ein hybrides Modell in dem Sinne, daß es wertebasierte und objektbasierte Modellierung vereint. Eine Instanz des in Abb. 2.1 angegebenen Fragmentes ist *ein* komplexer *Wert* (eine Menge von Tupeln), wie er z.B. durch die ESCHER-Tabelle `Bestand` in Abb. 1.3 gegeben ist. In IFO lassen sich Anwendungsobjekte aber auch als *abstrakte* Objekte modellieren. Diese Alternative ist in Abb. 2.2 dargestellt. Die leeren Kreise symbolisieren die (abstrakten) Objekttypen `Buch` und `Autor` (Typnamen stehen immer unter den Knoten). Die Domäne eines Objekttyps ist eine abzählbar unendliche Menge $\{\alpha_1, \alpha_2, \ldots\}$ von Objektidentifikatoren. Der Zustand von Objekten eines Objekttyps ist gegeben durch eine Menge von Funktionen (in Abb. 2.2 durch die Pfeile repräsentiert), deren Wertebereich die Domäne des jeweiligen Objekttyps bildet. So liefert Autoren(α_i) etwa die Menge der Autoren des Buches α_i. Attribute werden also über Funktionen modelliert. Mit den OIDs steht eine über die gesamte Lebenszeit eines Objektes invariante Identifikation zur Verfügung. Gleichzeitig erlangt man eine Abstrak-

2. Bei der Transformation geht semantische Information verloren, z.B. weil ein bestimmtes Konstrukt oder eine Integritätsbedingung im Relationenmodell nicht wiedergegeben werden kann.

tion im Sinne von ADTs [EGL89]. Dagegen ist in Abb. 2.1 keine Form der Typisierung vorgesehen. Die Tatsache, daß der komplexe Wert eine Menge von *Büchern* darstellt, ergibt sich allein aus der Wahl des optionalen und rein dokumentarischen Zwecken dienenden Labels „Buch“ am entsprechenden Aggregationsknoten.

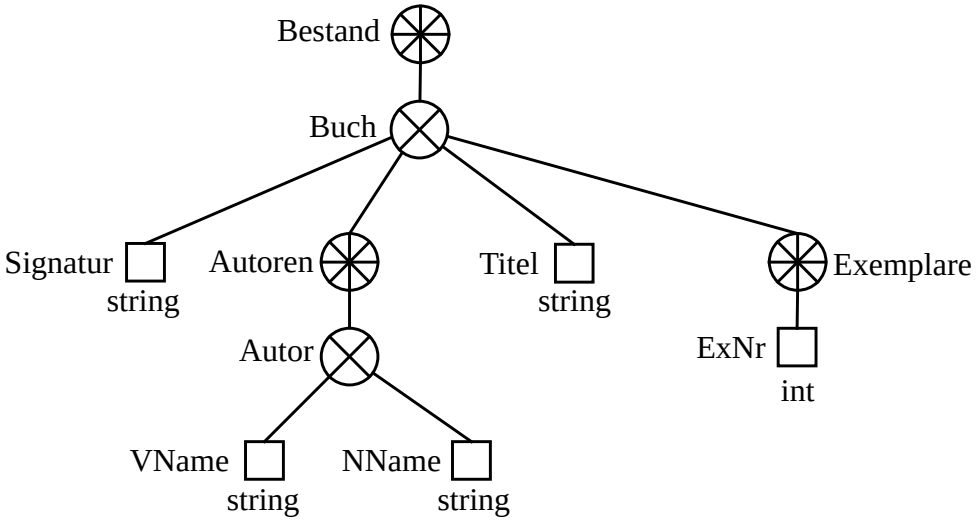


Abbildung 2.1: Ein rein wertebasiertes IFO-Schema

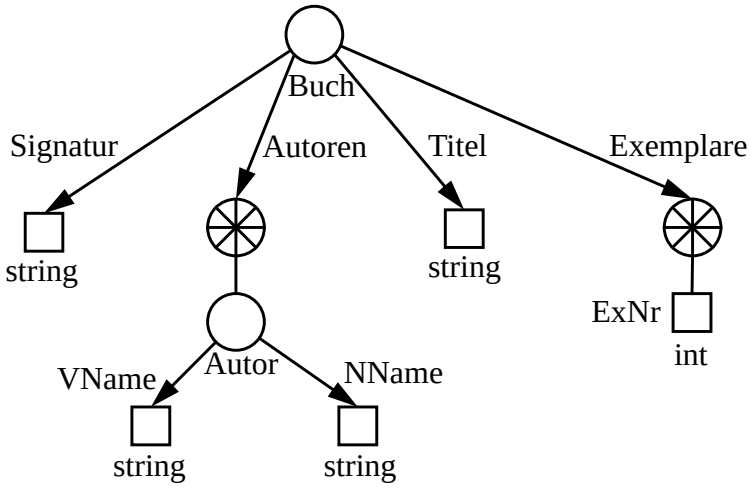
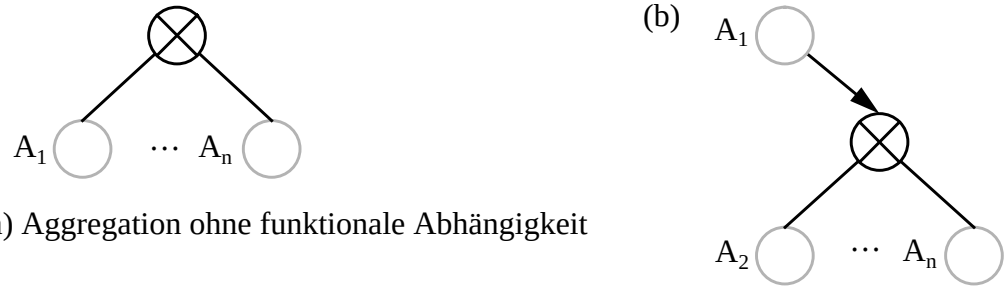


Abbildung 2.2: Ein alternatives IFO-Schema mit abstrakten Objekttypen

Schlüsselbedingungen als Spezialfall funktionaler Abhängigkeiten lassen sich in IFO über Funktionen direkt abbilden. In Abb. 2.3 (b) wird durch eine Funktion ausgedrückt, daß A_1 Schlüssel in der Aggregation $A_1 \times \ldots \times A_n$ ist (die schraffierten Knoten stehen dabei für beliebige IFO-Knoten). Allerdings legt man sich auf diese Weise auf eine bestimmte Schlüsselbedingung fest. Alternative Schlüssel lassen sich in IFO nicht erfassen.

Im Vergleich mit dem ER-Modell fällt auf, daß IFO – genauso wie das eNF^2-Modell – das Konzept der Assoziation zur Darstellung semantischer Beziehungen nur indirekt unterstützt. Die Assoziation wird deshalb vielfach durch Aggregation simuliert, wie dies in den Abbildungen 2.1 und 2.2 auch zu erkennen ist: Die Menge der Autoren ist in die Beschreibung eines

Buches „hineingeschachtelt“. Dies bedeutet, daß bereits bei der konzeptuellen Modellierung eine bestimmten Sichtweise auf die Beziehung „geschrieben von/schreibt“ zwischen Autoren und Büchern Vorrang vor alternativen Sichtweisen genießt.



(a) Aggregation ohne funktionale Abhängigkeit

Abbildung 2.3: Ausdruck der in (a) fehlenden funktionalen Abhängigkeit durch das Funktionskonstrukt in (b)

Binäre Beziehungen lassen sich in IFO auch über zwei Funktionen abbilden, wobei diese über eine zusätzliche Integritätsbedingung als invers zueinander gekennzeichnet werden müssen (vgl. Abb. 2.4).

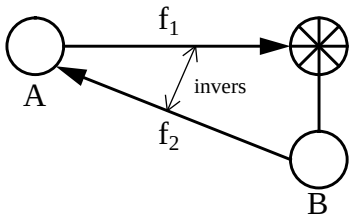


Abbildung 2.4: Modellierung einer 1:n Beziehung zwischen den abstrakten Typen A und B über zueinander inverse Funktionen f_1 und f_2

IFO macht auch die Unterscheidung zwischen Spezialisierung und Generalisierung, die in Abschnitt 2.1.1 beschrieben wurde. In IFO werden die Objekttypen, die bei der oben erwähnten Typkonstruktion durch Generalisierung entstehen, als *freie* Objekttypen bezeichnet.

In [Han95] wird zurecht bemängelt, daß in [AH87] eine systematische Behandlung von Integritätsbedingungen fehlt. Sie werden lediglich in eher beiläufigen Kommentaren zu den angegebenen Beispielen erwähnt. Hervorzuheben ist allerdings das Verdienst von [AH87] für die Formalisierung der semantischen Datenmodellierung. Insbesondere wird die Semantik generischer Update-Operationen unter Berücksichtigung der Spezialisierungs-/Generalisierungshierarchien untersucht, so daß auch der oft völlig vernachlässigte operationale Aspekt Berücksichtigung findet. Ferner werden genaue Konsistenzbedingungen für gültige Spezialisierungs- bzw. Generalisierungshierarchien angegeben (z.B. keine Zyklen in *isa*-Beziehungen).

Unlängst wurde IFO zu IFO_2 [PTC+93, TPC93] weiterentwickelt. Neben der Behandlung einiger Integritätsbedingungen steht dabei die Transformation eines IFO_2-Schemas in O_2-Klassendefinitionen und die Generierung von Integritätsüberprüfungsroutinen im Vordergrund.

weitere SDMs

RM/T [Cod79] ist eine Weiterentwicklung des Codd’schen Relationenmodells zu einem Modell, das nicht als Datenmodell für ein implementiertes DBMS, sondern für die konzeptu-

elle Modellierung gedacht ist [Dat83]. Nachteilig ist wie beim Relationenmodell weiterhin die Einschränkung auf „flache“ Relationen. Es wird zwischen E- und P-Relationen unterschieden. Die E-Relationen enthalten lediglich die system-erzeugten künstlichen Schlüssel (Surrogate) der aktuell existierenden Entitäten (vergleichbar also mit der *aktiven* Domäne eines Objekttyps). Die Attribute als Eigenschaften eines Objekttyps werden in eine oder mehrere P-Relationen (P wie „property“) ausgelagert. Nach [Dat83] ist vorgesehen, die Unterscheidung zwischen E- und P-Relationen sowie die Surrogate vor dem Anwender zu verbergen, indem Relationen im herkömmlichen Sinne als Sichten über einem RM/T-Schema definiert werden. Die E-Relationen werden unterteilt in

- *kernel entities*, die den „Kern“ einer Datensammlung bilden (z.B. Personen, Bauteile, Abteilungen usw.)
- *characteristic entities*, die existenzabhängige Entitäten (*weak entities* in ER-Modellen [EN94]) darstellen
- *associative entities* für m:n-Beziehungen

Über eine an eine E-Relation geknüpfte *designation*-Bedingung werden 1:n-Beziehungen ausgedrückt. Referentielle Integrität ist bei diesem Modell inhärent zugesichert. Ferner unterstützt RM/T die Spezialisierung (*unconditional generalization*) und die Generalisierung (*alternative generalization*), wobei der Wert eines klassifizierenden Attributs im Supertyp die Typzugehörigkeit in einer Menge von Subtypen bestimmt. RM/T hat jedoch keine Verbreitung gefunden, und die von Codd anvisierte Ablösung des klassischen Relationenmodells durch RM/T hat nicht stattgefunden, wofür Codd in [Cod90] die „Trägheit“ der kommerziellen DBMS-Anbieter verantwortlich macht.

NIAM [NH89], das seit einiger Zeit auch unter der Bezeichnung *ORM* (Object Role Model) [Hal95] bekannt ist, hat in der konzeptuellen Modellierung eine gewisse Beliebtheit erlangt. Das Modell verfolgt einen „bottom-up“-Ansatz, indem zunächst natürlichsprachliche Aussagen in logische Prädikate umgewandelt werden. Die „Zusammensetzung“ der Prädikate gibt dann die abzubildende „Miniwelt“ wieder. So werden etwa die Aussagen „Eine Person hat einen Namen“, „Eine Person fährt ein Auto“, „Ein Auto hat eine Kfz-Nummer“ in die in Abb. 2.5 gegebene graphische Notation umgesetzt.

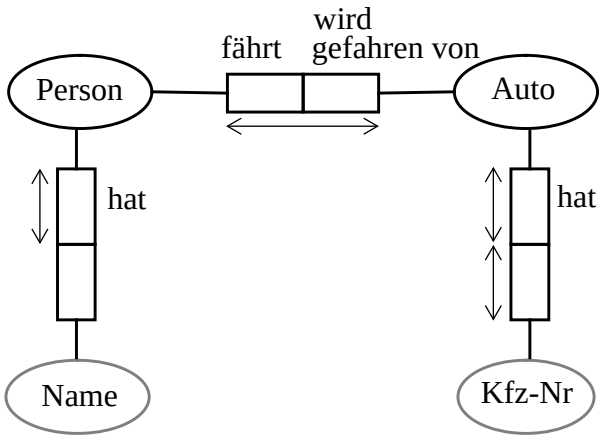


Abbildung 2.5: Ein NIAM (ORM)-Schema

Entitäten und ihre Attribute werden also durch eine binäre „hat die Eigenschaft/ist Eigenschaft von“-Beziehung miteinander in Verbindung gebracht. Das Modell zeichnet sich durch eine große Vielfalt an unterstützten Integritätsbedingungen aus (z.B. ist in Abb. 2.5 zu sehen, wie Schlüsselbedingungen für Fakten durch Doppelpfeile ausgedrückt werden).

Wie bereits erwähnt, unterstützen nur sehr wenige SDMs den operationalen (oder auch dynamischen) Teil der Modellierung. Eher eine Ausnahme ist in dieser Hinsicht TAXIS [MBW80], das auch eines der wenigen SDMs ist, für die eine Implementation (im Fall von TAXIS als Programmiersprache) vorliegt [NLL+87]. Es können Transaktionen als per definitionem konsistenzerhaltende Operationen sowie verschiedene Klassen von Ausnahmen und Ausnahmebehandler (*exception handler*) definiert werden. Auf der Basis einer *isa*-Hierarchie sind dann Verfeinerungen der Transaktionen und exception handler möglich. Für Transaktionen lassen sich Vor- und Nachbedingungen in Form von Zusicherungen angeben. Ein weiteres semantisches Datenmodell, das sich sehr um eine Verknüpfung struktureller und dynamischer Aspekte bemüht, ist SHM+ [BR84]: Es werden „behavior schemes“ definiert, die mit Datenflußdiagrammen vergleichbar sind und in die das Strukturmodell integriert wird.

2.2 Die Drei-Schichten-Architektur

2.2.1 Erfahrungen mit frühen DBMS

In den 60er Jahren entstanden die ersten Systeme, die als DBMS bezeichnet werden können. Die Datenmodelle, die diesen DBMS zugrundelagen, sind das *Hierarchische Modell* und das *Netzwerk-Modell*. Nach einer Schätzung von Navathe [Nav92] verwendeten 1992 noch über 70 % aller kommerziellen Datenbank-Anwendungen eines dieser frühen Systeme, die zunehmend als „Altlasten“ angesehen werden (*legacy systems*).

Beiden Datenmodellen gemeinsam ist, daß sie sog. *Records* verwalten, die – vergleichbar mit den (flachen) Tupeln des Relationenmodells – atomare Werte als Komponenten (auch *Felder* genannt) zu einer Einheit aggregieren.

Das Hierarchische Modell erlaubt zunächst eine streng hierarchische Anordnung von Records, d.h. die Bildung von Bäumen mit Records gewisser Recordtypen als Knoten. Auf diese Weise können ohne Probleme 1:n-Beziehungen mit Existenzabhängigkeit der Sohnknoten von ihren Vätern erfaßt werden. Die Vater-Sohn-Beziehung zwischen Records wird nicht wie beim Relationenmodell implizit über Wertegleichheit bestimmter Felder, sondern durch die physische Speicherung der Records realisiert. Der Zugriff erfolgt immer von einem Vaterknoten in Richtung der Söhne. Die bekannteste Implementation dieses Modells ist IMS (Information Management System) [IBM75]. IMS war lange Zeit bis zur Akzeptanz relationaler Systeme auf dem kommerziellen Markt das dominierende DBMS und ist auch heute noch im Bereich der Standardanwendungen sehr stark verbreitet. Ein guter Überblick über IMS ist in [SS83] zu finden. Die Darstellung in [Ull88] hat den Vorteil, daß sie frei von IMS-spezifischen Bezeichnungen bleibt und somit einen unmittelbaren Vergleich mit dem Netzwerk-Modell ermöglicht.

Obgleich rein hierarchisch strukturierte Daten in der Realität häufig anzutreffen sind, stellen sich die durch strenge Hierarchien gegebenen Möglichkeiten schnell als zu restriktiv heraus.

Nachteilig ist u.a. die Disjunktheit der Bäume. Als Erweiterung führte man die sog. *virtuellen Records* ein. Ein virtueller Record ist lediglich ein Verweis (*Link*) auf den „eigentlichen" Record, und ein Link ist nichts anderes als ein physischer Zugriffspfad auf einen Record. Auf diese Weise kann Redundanz in den Daten vermieden werden, da über virtuelle Records ein *sharing* von Records über Baumgrenzen hinweg realisiert werden kann. Ferner können innerhalb eines Records auch virtuelle Felder definiert werden.

Das Netzwerk-Modell, auch CODASYL-Modell genannt, wurde in [COD71] spezifiziert (vgl. auch den Übersichtsartikel [TF76] bzw. die entsprechenden Abschnitte in [Gra84, SS83, Ull88, Vos91]) . Es bietet ein sog. *Set*-Konstrukt an, über welches jeder Record als sog. *Owner* mit einer Menge von *Member*-Records verbunden werden kann. Ein Set-Typ legt die Record-Typen des Owners bzw. der Member fest. Ausgehend von einem Owner kann die Liste[3] der Member-Records durchlaufen werden. Gleichzeitig ist aber der direkte Zugriff von einem Member auf seinen Owner möglich. Eine Instanz eines Set-Typs wird über explizite physische Links implementiert, die den Owner-Record und die Member-Liste miteinander bzw. die Member-Records untereinander verbinden. Ein Record kann gleichzeitig Member verschiedener Set-Typen sein. Daher kann über eine streng hierarchische Anordnung von Records hinaus ein allgemeineres Netzwerk von 1:n-Beziehungen aufgebaut werden.

In IMS sind netzwerkartige Strukturen bedingt möglich, indem einem Record neben seinem *physischen* Vater auch ein *logischer* Vater zugeordnet wird, was wiederum mit Hilfe von virtuellen Records realisiert wird. Damit kann die Beschränkung auf den einseitigen Zugriff in (physischer) Vater-Sohn-Richtung aufgehoben werden. Die „aufgesetzten Features" in IMS (virtuelle Records und Felder, logische Vater-Sohn-Beziehung) bieten zwar eine Loslösung von rein hierarchischen Strukturen, führen aber sehr schnell zu unübersichtlichen Schemata.

Als wesentlicher Nachteil wird für die beiden genannten Modelle auch die Art des Zugriffs auf die Daten empfunden. Die DML (Data Manipulation Language) dieser Modelle ist rein *navigierend* und auf den Zugriff auf jeweils genau einen Record ausgerichtet. Ausgehend von einem Vater-Record (bzw. Owner-Record) werden nacheinander die einzelnen Sohn-Records (bzw. Member-Records) gelesen. Hierin manifestiert sich die für beide Modelle zu beobachtende starke Verflechtung logischer und physischer Aspekte sehr deutlich. Das Hierarchische und das Netzwerk-Modell sind als sehr implementationsnah zu bezeichnen. Unübersehbar ist die Nähe zur File-Verarbeitung, für die das sukzessive Lesen einzelner Records ebenfalls typisch ist. Anfragen in deklarativer Form sind in den genannten Modellen nicht möglich. Es ist im Netzwerk-Modell z.B. nicht möglich, aus einem gegebenen Instanzen-Netzwerk ein Unternetzwerk auf der Grundlage gewisser Selektionsbedingungen herauszufiltern. Das Anwendungsprogramm muß einen *Cursor* (beim Netzwerk-Modell *currency indicator* genannt) mit Hilfe von einfachen navigierenden Operationen durch die komplette Instanzenstruktur bewegen. Es kann lediglich die Records, die sich aufgrund eines Wertevergleichs mit den Komponenten der Records als nicht relevant herausstellen, „überspringen". Die Folge ist eine sehr große Belastung des Anwendungsprogrammierers, der jede Anfrage „voll" ausprogrammieren muß. Diese *prozedurale* Orientierung führt zu unübersichtlichen Programmen, die bei Modifikationen im Datenbankschema oftmals massiven Änderungen unterworfen werden müssen.

3. Der Begriff *Set* ist also irreführend!

2.2.2 Trennung der Ebenen

Nach den schlechten Erfahrungen in Hinblick auf Änderungsfreundlichkeit und Wartbarkeit von Anwendungsprogrammen auf der Basis der frühen, implementationsnahen Datenmodelle und bereits inspiriert von Implementationen des Relationenmodells, wurde Mitte der 70er Jahre eine Normierung für Datenbank-Architekturen vorangetrieben, die zur ANSI/X3/SPARC-3-Schichten-Architektur führte. [Bur+86] enthält den Abschlußbericht des zuständigen Normierungskommitees.

In Abb. 2.6 ist die 3-Schichten-Architektur graphisch veranschaulicht. Die verschiedenen Schichten werden durch die schattierten Rechtecke dargestellt. Zentral für die 3-Schichten-Architektur ist die *konzeptuelle Ebene*: Ihr liegt ein *logisches Schema* zugrunde, das Instanz eines geeigneten Datenmodells (z.B. des Relationenmodells) ist. Aspekte der physischen Realisierung, wie etwa Speicherorganisationsformen und Zugriffspfade, sind allein Sache der *internen Ebene* und werden im internen Schema festgelegt. Um unnötige Komplexität des logischen Schemas vor dem Benutzer zu verbergen, werden oberhalb der konzeptuellen Ebene auf der *externen Ebene* sog. *Sichten* definiert. Die Idee besteht darin, die für den jeweiligen Anwendungskontext relevanten Teile des logischen Schemas herauszufiltern und diese dem Benutzer als Sicht zu präsentieren. Dazu gehört aber auch, daß eine Sicht auch die Instanzenmenge einschränken kann, einerseits zum Ausblenden irrelevanter Instanzen, andererseits aber auch aus Gründen der Autorisierung, wenn Daten für bestimmte Benutzergruppen nicht zugänglich sein sollen. Die Definition von Sichten sollte daher über die alleinige Spezifikation von Subschemata hinausgehen. In relationalen Systemen sind Sichten bekanntlich virtuelle Relationen, die als SQL-Anfrageausdruck definiert werden. In diesem Fall bleiben Sichten im Rahmen desselben Datenmodells wie das logische Schema. Denkbar ist aber auch, daß der externen Ebene ein anderes Datenmodell zugrundeliegt.

Angestrebt wird mit der 3-Ebenen-Architektur die sog. *Datenunabhängigkeit*, die zu größerer Robustheit von Anwendungen gegenüber Änderungen in den Schemata führt: Die *physische* Datenunabhängigkeit fordert, daß das logische Schema von Änderungen am internen Schema unbeeinflußt bleibt, da dies nur eine Änderung der Abbildung zwischen den beiden Ebenen zur Folge hat. Die *logische* Datenunabhängigkeit bezieht sich auf den analogen Sachverhalt im Verhältnis zwischen logischer und externer Ebene: Führen Änderungen am logischen Schema zu einem äquivalenten Schema, dann sollen alle existierenden Sichten weiterhin ihre Gültigkeit haben und sich nur die Art ihrer Definition (als Abbildung zwischen logischer und externer Ebene) ändern.

Bereits in [COD71] wurde für das Netzwerk-Modell eine deutliche Trennung zwischen logischen und physischen Aspekten als wichtig herausgestellt, letztendlich aber auch in den Folgespezifikationen nicht konsequent umgesetzt [GKP92]. Dagegen gibt es beim Netzwerk-Modell bereits Ansätze zur Formulierung von Sichten. Es gibt eine *Subschema-DDL*, die es erlaubt, eine Projektion des globalen Schemas auf die interessierenden Felder, Records oder Sets durchzuführen. Allerdings gibt es keine Möglichkeit, für ein Subschema die Instanzenmenge durch eine Selektionsbedingung einzuschränken.

Während viele Autoren die Begriffe konzeptuelles und logisches Schema synonym verwenden, wollen wir in dieser Arbeit – ähnlich wie [Nav92] – auf eine Differenzierung beider Begriffe drängen. In Abschnitt 2.1 wurde bereits festgestellt, daß in der Praxis bei der Entwicklung einer Anwendung als erster Schritt ein konzeptuelles Schema erzeugt wird, das meist

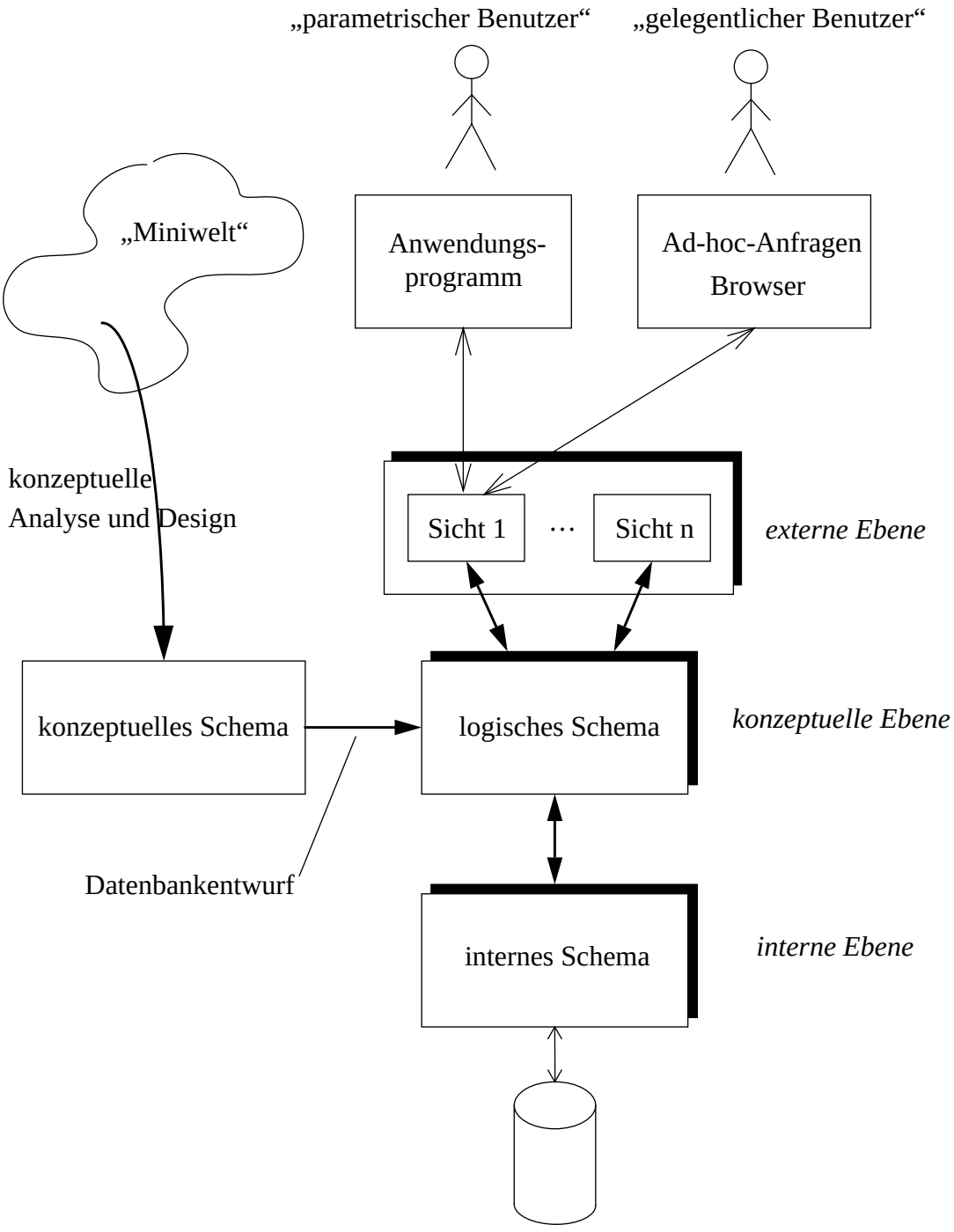


Abbildung 2.6: Drei-Schichten-Architektur nach ANSI/X3/SPARC im Kontext der Anwendungsentwicklung und der Schnittstelle zum Benutzer

Instanz eines SDMs ist. Die ersten Phasen einer Anwendungsentwicklung, die von einer „Miniwelt“ zu einem konzeptuellen Schema führen, sind unter der Bezeichnung „konzeptuelle Analyse und Design“ ebenfalls in Abb. 2.6 aufgenommen worden. Der Datenbankentwurf ist dann die Transformation des konzeptuellen Schemas in ein logisches Schema.

Schließlich zeigt Abb. 2.6 noch, wie die Benutzergruppen „parametrischer Benutzer“ bzw. „gelegentlicher Benutzer“ (vgl. Kapitel 1) auf die Daten zugreifen: Während erstere Anwendungsprogramme aufrufen, benutzen letztere anspruchsvollere Anfrage- und Browser-Tools. In beiden Fällen wird der Zugriff im Idealfall ausschließlich über Sichten geleitet. Die dritte Benutzergruppe, der „Datenbankspezialist“, ist für die Spezifikation der Schemata der verschiedenen Ebenen verantwortlich, wird aber in Abb. 2.6 aus Platzgründen nicht gezeigt.

2.3 Das Relationenmodell

Das Relationenmodell von Codd [Cod70] besticht durch die Einfachheit seiner formalen Beschreibung mit mathematischen Mitteln und bedeutet einen qualitativen Sprung in der Entwicklung von Datenmodellen. Zum ersten Mal kann von einem wirklich *logischen* Datenmodell gesprochen werden, auf dessen Grundlage sich die erwähnte 3-Schichten-Architektur realisieren läßt. Einziges Modellierungskonstrukt ist die Relation als eine Menge von Tupeln, deren Komponenten (auch Attribute genannt) *atomare* Werte sein müssen, d.h. es gilt die *erste Normalform*. Hinsichtlich genauerer Ausführungen zum Relationenmodell sei auf die Lehrbuchliteratur [Ull88, EN94, Dat95, Vos91, HS95] und auf [Cod90] verwiesen, wo Codd eine aktuelle Version „seines“ Relationenmodells ausführlich beschreibt.

Relationale Datenbankmanagementsysteme (RDBMS) haben sich mittlerweile in der Praxis als Standard für neu zu entwickelnde kommerzielle Datenbank-Anwendungen durchgesetzt. Exemplarisch seien als Vertreter die relationalen Systeme DB2, SQL/DS, Ingres, ORACLE, Informix und Sybase genannt. Der Erfolg des Relationenmodells in der Praxis läßt sich stichpunktartig auf folgende Vorteile gegenüber bisherigen Datenmodellen zurückführen:

- Einfachheit des Datenmodells: Die Relation ist einziger Baustein des Strukturteils
- Die Anfragesprache (Relationenalgebra) ist deklarativ und optimierbar
- Trennung der externen, konzeptuellen und internen Ebene und damit verbunden logische und physische Datenunabhängigkeit
- Möglichkeiten zur Formulierung von Integritätsbedingungen
- interaktive Komponenten: Schnittstelle für Ad-hoc-Anfragen (und Ad-hoc-Updates), Browser, Formulare

In den folgenden Abschnitten wird auf einige, für diese Arbeit relevante Aspekte des Relationenmodells genauer eingegangen.

2.3.1 Relationentheorie und Integritätsbedingungen

Im Rahmen des Relationenmodells konnten auch erstmals Integritätsbedingungen unter formalen Gesichtspunkten und in Hinblick auf den Datenbankentwurf untersucht werden. Integritätsbedingungen werden für das Relationenmodell in der Form von sog. *Abhängigkeiten* formuliert: Die wichtigsten Abhängigkeiten sind sicher die *funktionalen* Abhängigkeiten, die Schlüsselbedingungen als Spezialfall enthalten. Daneben spielen aber auch *mehrwertige*, *Verbund-* und *Inklusionsabhängigkeiten* eine Rolle. Die mathematisch fundierte Definition des Relationenmodells inklusive der genannten Integritätsbedingungen war der Startschuß zu

umfangreichen theoretischen Arbeiten. Unter den Lehrbüchern, die die formale Relationentheorie behandeln, seien [AD93, KK93, PDG+89] genannt. Angestrebt wird ein formaler, algorithmisch beschreibbarer Datenbankentwurf[4], der ausgehend von einer Menge von Attributen und Abhängigkeiten ein *Relationenschema* erzeugt, das gewissen *Normalformen* genügt. Ein Relationenschema ist durch eine Menge von Relationstypen[5] und eine Menge von Integritätsbedingungen gegeben. Die Normalformen zielen auf die Vermeidung von Redundanz ab, die durch Ausführung von Update-Operationen zu Inkonsistenzen in der Datenbank führen kann (*Update-Anomalien*). Die Varianten des vielfach zitierten *Dekompositionsalgorithmus* etwa gehen davon aus, daß anfangs eine Relation vorliegt, die *alle* Attribute enthält. Diese Relation wird dann schrittweise zerlegt, bis die gewünschte Normalform erreicht ist.

Nachteilig ist dabei jedoch, daß bei einem völlig automatischen Ablauf des Algorithmus' Attribute völlig verschiedener Entitäten in einer Relation zusammenbleiben, während eigentlich zusammengehörende Attribute auch auf verschiedene Relationen verteilt werden [Heu92]. Es tut sich also mitunter eine große Kluft zwischen dem resultierenden Relationenschema und der ursprünglichen Struktur der Anwendungsobjekte auf.

Problematisch ist ferner, daß das Relationenmodell sich nicht als Grundlage für die frühe Phase der konzeptuellen Modellierung eignet, die dem Datenbankentwurf unmittelbar vorangeht. Aufgrund der Beschränkung auf die Relation als einziges Modellierungskonstrukt müssen alle semantischen Konzepte (Entitäten, Beziehungen usw.) auf Relationen abgebildet werden. Das Phänomen, daß ein und dasselbe Konzept eines Modells für mehr als ein semantisches Konzept steht, wird als *semantische Überladung* [GKP92] bezeichnet. Letztendlich wird durch die semantische Überladung des Relationenkonstrukts eine für die konzeptuelle Modellierung zu frühe und zu hohe Abstraktion erreicht. Dies wirkt sich auch auf die Erfassung von Integritätsbedingungen in Form von Abhängigkeiten aus: Es besteht leicht die Gefahr, die eine oder andere Abhängigkeit ganz einfach zu übersehen! Aus diesem Grund wurden für die Phase der konzeptuellen Modellierung die semantischen Datenmodelle entwickelt, die semantisch reichhaltigere Konstrukte besitzen. Wie bereits erwähnt läßt sich ein ER-Diagramm relativ problemlos in ein Relationenschema in dritter oder vierter Normalform transformieren, wobei auch noch Fremdschlüsselbedingungen berücksichtigt werden, was bei den in den Lehrbüchern beschriebenen Normalisierungsverfahren der Relationentheorie nicht der Fall ist. Dabei sind gerade die Fremdschlüsselbedingungen unabdingbar zur Sicherung der der referentiellen Integrität [Dat81, Dat90, Rei93][6].

2.3.2 Relationale Anfragesprachen

Als Grundlage einer Anfragesprache für das Relationenmodell wurde – ursprünglich ebenfalls von Codd – ein Satz generischer Operationen (Selektion, Projektion, Kartesisches Produkt,

4. Oftmals auch als „Relationales Datenbankdesign" bezeichnet.
5. Wir unterscheiden zwischen einem Relationstyp und seiner Instanz, der „eigentlichen" Relation. Allerdings ist der Begriff „Typ" im Relationenmodell anders zu verstehen als man dies aus Programmiersprachen gewohnt ist. Typäquivalenz ist im Relationenmodell gleichbedeutend mit de sog. *Vereinigungskompatibilität*. Danach sind zwei Relationentypen äquivalent, wenn sie bis auf Umbenennung und Vertauschung von Attributen strukturgleich sind
6. Beim Netzwerk-Modell gibt es das Problem der *dangling references*, d.h. Referenzen „ins Leere", nicht, da beim Löschen eines Records dieser aus allen Sets, in denen er Mitglied ist, entfernt wird.

Vereinigung, Differenz, Umbenennung[7]) angegeben, die auf Relationen operieren und zusammengenommen die bekannte *Relationenalgebra* bilden. Das Ergebnis jeder dieser Operationen ist wiederum eine Relation, d.h. der Ansatz ist *mengenorientiert* im Unterschied zu der bis dahin vorherrschenden Record-Orientierung. Parallel dazu wurde ein an der Prädikatenlogik orientierter *Relationenkalkül* in zwei Varianten (tupelorientiert und domänenorientiert) entwikkelt, der gleichmächtig zur Relationenalgebra ist. Sowohl mit der Relationenalgebra als auch mit dem Relationenkalkül lassen sich Anfragen *deklarativ* formulieren.

In der Praxis hat sich SQL als Standard einer Sprachschnittstelle zu relationalen Datenbanken durchgesetzt. Bekanntlich ist SQL nicht nur eine deklarative Anfragesprache, sondern gleichzeitig auch eine DDL und DML. In diesem Abschnitt soll jedoch SQL als Anfragesprache im Mittelpunkt stehen. SQL wurde ursprünglich unter den Namen SEQUEL bzw. SEQUEL2 [Cha+76] für das IBM-Projekt System R [ABC+76], eine der ersten Implementierungen des Relationenmodells, entwickelt. Direkte Implementierungen der Relationenalgebra, wie z.B. ISBL [Tod76], oder des (tupelorientierten) Relationenkalküls, wie QUEL [SWK+76], konnten sich gegenüber SQL nicht behaupten. Aufgrund der Dominanz von SQL werden relationale Datenbanken zuweilen auch als SQL-Datenbanken bezeichnet.

Mittlerweile hat SQL mehrere Standardisierungen erfahren. Der aktuelle Standard ist SQL2 [ISO92a], auch SQL/92 genannt, der in [DD93] ausführlich beschrieben wird. Der in Arbeit befindliche SQL3-Standard bemüht sich vor allem um die Integration objektorientierter Konzepte (u.a. benutzerdefinierte Typen und Funktionen) in SQL [ISO92b, Pis93, Web93, DD93]. Trotz der Standardisierung des Sprachumfangs liegt jedoch bis heute keine „offizielle“ *formale* Semantik des Anfrageteils von SQL vor. Die Semantik einer SQL-Anfrage läßt sich in vielen Fällen durch einen Ausdruck der Relationenalgebra oder des Relationenkalküls erklären, und umgekehrt ist jeder Ausdruck der Relationenalgebra in eine SQL-Anfrage übersetzbar, d.h. SQL ist relational vollständig. Nicht erklären kann man im Rahmen der Relationenalgebra aber z.B. die Nichteinhaltung der Mengeneigenschaft (SELECT ohne DISTINCT), die Aggregatfunktionen und das Gruppieren mittels GROUP BY. Es gibt eine Reihe von Arbeiten zur formalen Semantik von SQL, darunter [CG85, NPS91]. In [Gog94] werden SQL-Anfragen in ein Kalkül eines erweiterten Entity-Relationship-Modells übersetzt, das dem tupelorientierten Relationenkalkül nahekommt.

Die Deklarativität einer Anfragesprache erfordert die Umsetzung einer Anfrage in einen effizienten prozeduralen Ausführungsplan für die interne Ebene. Eine formale Semantik auf der Grundlage einer Algebra bietet die Grundlage für die Optimierbarkeit von Anfragen. Neben der Optimierung durch äquivalenzerhaltende algebraische Umformungen spielt natürlich auch die interne Optimierung eine Rolle, die Parameter wie Zugriffspfade, Speicherstrukturen, vorhandene Indexe oder z.B. auch die aktuelle Größe der Relationen berücksichtigt. Näheres zur Anfrageoptimierung beim Relationenmodell ist z.B. [JK84] bzw. [Ull89] zu entnehmen. Die Leistungsfähigkeit der SQL-Anfrage-Optimierer ist von System zu System sehr unterschiedlich [Heu92].

Ein navigierender Zugriff auf die Tupel einer Relation ist im Relationenmodell gar nicht vorgesehen. Vom theoretischen Standpunkt ist dies auch nicht möglich, da keine Vorgänger-Nachfolger-Beziehung auf einer Menge von Tupeln definiert ist.

---

7. Die Umbenennungsoperation *rename* gehört in einigen Darstellungen zu den Grundoperationen der Algebra, vgl. [Mai83, Heu92].

2.3.3 Relationale Sichten und das View-Update-Problem

Das Relationenmodell ermöglicht die nach dem 3-Schichten-Modell angestrebte Trennung zwischen der logischen und der externen Ebene durch die Definition von Sichten (*Views*). Sie sind virtuelle Relationen, die durch Anfrageausdrücke über Basisrelationen oder bereits definierten Sichten definiert sind. Anfragen und Updates auf Views werden oftmals nach der Technik der *Query Modification* [Sto75] unter Verwendung des den View definierenden Anfrageausdrucks in Anfragen und Updates auf Basisrelationen übersetzt, d.h. ein View wird nicht einmal zur Laufzeit (temporär) instanziiert. Daneben gibt es auch den Ansatz, Views zu *materialisieren* und Updates auf den Basistabellen in inkrementelle Änderungen der materialisierten Views umzusetzen [BLT86, CW91].

Die Umsetzung eines View-Updates in Operationen auf den Basistabellen ist in relationalen DBMS nur in sehr beschränktem Umfang möglich. In SQL/92 sind – etwas vereinfacht gesagt – nur solche Sichten modifizierbar, die als Selektions- oder Projektionssichten auf einer einzigen Basistabelle oder einer anderen modifizierbaren Sicht definiert sind [DD93]. Es ist klar, daß z.B. Attribute einer Sicht, die den Wert einer Aggregatfunktion wiedergeben, nicht modifizierbar sind. Dagegen ist es in vielen Situationen wünschenswert, daß Sichten, die keine „neuen Werte“ berechnen, sondern allein mit den Operatoren der Relationenalgebra definiert sind, eine Semantik für Modifikationsoperationen zugeordnet werden kann. Problematisch ist bei View-Updates zweierlei:

- Es gibt i.a. keine eindeutig bestimmten Regeln zur Propagation von View-Updates (Ambiguität der Update-Propagation)
- Zu berücksichtigen ist, daß die Propagation von View-Updates möglichst wenig „Nebeneffekte“ erzeugen soll
  (Minimalität der Update-Propagation, Effektkonformität [HS95])

„Nebeneffekte“ treten insbesondere bei Join-Views auf: Das Löschen eines Tupels aus einem Join-View kann das Löschen weiterer Tupel des Join-Views zur Folge haben.

Eine Vielzahl von Arbeiten (u.a. [BS81, DB82, FC85, Kel85, Kel86, DM94a, DM94b]) hat sich mit dem View-Update-Problem auseinandergesetzt, ohne daß dies bislang zu wesenlichen Verbesserungen in existierenden relationalen DBMS geführt hätte.

Exemplarisch gehen wir auf die Arbeiten von Date und McGoveran [DM94a, DM94b] ein. Sie schlagen Regeln zur Propagation von View-Updates vor, die auf sog. *Tabellen-Prädikaten* basieren. Ein Tabellen-Prädikat P(A) für eine Tabelle A ist die Konjuktion aller Integritätsbedingungen, die sich lokal auf die Tabelle A beziehen (z.B. Schlüsselbedingungen, Einschränkungen des Wertebereichs für ein Attribut). Tabellen-Prädikate gibt es auch für Sichten (als virtuelle Tabellen). Eine Selektionssicht V, die als `SELECT * FROM A WHERE` *Bedingung* definiert wird, hat beispielsweise das Tabellenprädikat $P(V) = P(A) \wedge$ *Bedingung*.

Für das Einfügen in eine Sicht und Löschen von Tupeln aus einer Sicht V werden von Date und McGoveran für alle Arten von Views Regeln zur Propagation in Einfüge- und Löschoperationen in den Tabellen angegeben, die in der Definition von V vorkommen. Updates auf ein Tupel werden auf das Löschen des Tupels und Einfügen des modifizierten Tupels zurückgeführt, wobei zwischen Löschen und Einfügen kein Integritäts-Check durchgeführt wird. Dabei müssen folgende Prinzipien gelten:

- Die Tabellenprädikate aller Tabellen (inkl. Sichten!) müssen erfüllt bleiben
- Die Propagation darf nicht von der Syntax abhängen, die zur Definition einer Sicht definiert wurde. Die Regeln sollen Äquivalenzen bei einer Sichtdefinition berücksichtigen.

Es sind z.B.

```
SELECT * FROM Angest WHERE Abt = 'A1' OR Gehalt > 4000
```

und

```
(SELECT * FROM Angest WHERE Abt = 'A1') UNION
(SELECT * FROM Angest WHERE Gehalt > 4000)
```

zwei Varianten zur Definition desselben Views. Die Propagationsregeln sollen in beiden Fällen zum gleichen Resultat führen. Es sei am Rande erwähnt, daß in SQL/92 nur die erste Variante zu einem veränderbaren View führt. Letztendlich können aber auch unter Berücksichtigung der genannten Prinzipien gewisse Ambiguitäten beim View-Update nicht verhindert werden, wie das folgende Beispiel zeigt.

Beispiel 2.1 Es seien A und B zwei vereinigungskompatible Relationen, d.h. die Menge der Attribut/Wertebereich-Paare stimmt für A und B überein.

Die Sicht V1 sei definiert durch

```
(SELECT * FROM A) UNION (SELECT * FROM B)
```

Beim Einfügen eines neuen Tupels t in V1 können folgende Situationen vorliegen:

- Nach dem hypothetischen Einfügen von t in A und B sind weder P(A) noch P(B) erfüllt. Dann ist das Einfügen von t in V1 nicht erlaubt, da sonst das Tabellenprädikat P(V1) = P(A) $\vee$ P(B) nicht erfüllt wäre.
- Nach dem hypothetischen Einfügen von t in A und B ist genau eines der Tabellenprädikate P(A) oder P(B) erfüllt. Dann ist das Einfügen von t in V1 erlaubt und wird in das Einfügen in die Tabelle A oder B übersetzt, deren Tabellenprädikat erfüllt ist.
- Sind nach dem hypothetischen Einfügen von t in A und B beide Tabellenprädikate P(A) und P(B) erfüllt, dann ist unklar, ob das Einfügen von t in V1 tatsächlich, wie Date und McGoveran vorschlagen, in das Einfügen von t in sowohl A als auch B übersetzt werden soll. Das Einfügen in entweder A oder B wäre schon ausreichend!

Entsprechende Zweifelsfälle gibt es auch bei Sichten, die als Durchschnitt zweier Tabellen definiert sind. Die Sicht V2 sei definiert durch

```
(SELECT * FROM A) INTERSECT (SELECT * FROM B)
```

Beim Löschen eines Tupels t aus V2 würde es ausreichen, t aus A oder B zu löschen. In [DM94a] wird jedoch gefordert, t aus A und B zu löschen, falls t in beiden Tabellen vorkommt. ❐

Das Einfügen in sowohl A als auch B für den Fall, daß P(A) und P(B) erfüllt sind, wird in [DM94a] als eines der „slightly surprising results“ bezeichnet, die auch bei anderen Propagationsregeln aus [DM94a, DM94b] zu beobachten sind. Die Autoren sehen die Überraschungen in einem schlechten Datenbankdesign begründet, was unseres Erachtens eine unzutreffende und zu pauschale Argumentation darstellt. Vielmehr legen sie sich beim Vorliegen von sichtinhärenten Mehrdeutigkeiten auf eine bestimmte Entscheidung für die Art der Update-Propagation fest. Diese Entscheidung sollte aber nur eine Default-Regel darstellen, die bei Bedarf durch eine andere Regel ersetzt werden kann. Dazu machen Date und McGoweran keine Aus-

sagen. Die Konsequenzen einer unkritischen Annahme der Default-Regeln soll an folgendem Beispiel illustriert werden.

Beispiel 2.2 Wir betrachten das Bibliotheksbeispiel aus Abb. 1.8. Ein relationales Datenbankschema enthalte die Relationen

```
LESER (LeserNr, LName, Adresse) und
AUSLEIHE (ExNr, LeserNr, LDatum)
```

Schlüsselattribute sind unterstrichen. Die Sicht `ENTLEIHUNGEN` sei als einfache Join-Sicht gemäß

```
SELECT a.ExNr,a.LDatum,a.LeserNr,l.LName,l.Adresse
FROM AUSLEIHE a, LESER l WHERE a.LeserNr = l.LeserNr
```

definiert. Das Löschen des entsprechenden Tupels aus `ENTLEIHUNGEN` bedeutet die Rückgabe eines Buches. Die Regeln aus [DM94b] fordern, daß die entsprechenden Tupel aus `AUSLEIHE` *und* `LESER` (!) gelöscht werden, d.h. der Leser verschwindet ebenfalls. Dies ist natürlich nicht sinnvoll, da Leser unabhängig von Entleihungen existieren können. Falls im Schema weitere Lösch-Regeln angegeben sind, die eine der propagierten Löschoperationen verbieten, dann soll nach [DM94b] die Update-Operation auf der Sicht `ENTLEIHUNGEN` nicht erlaubt sein. Dies ist für das gegebene Beispiel unakzeptabel, da ein Verbieten des Löschens von Leser-Tupeln dazu führt, daß über `ENTLEIHUNGEN` gar keine Entleihungen rückgängig gemacht werden könnten. ❒

Es ließen sich noch viele Beispiele angeben, in denen ein Abweichen von starren Regeln für die Propagierung von Updates wünschenswert wäre, um anwendungsspezifische Semantiken ausdrücken zu können. Daher erscheint es uns sinnvoller, gemäß den Arbeiten von Keller [Kel85, Kel86] vorzugehen. In [Kel85] werden für eine große Klasse relationaler Views Regeln zur Update-Propagation angegeben, wobei beim Vorliegen von Mehrdeutigkeiten alle sinnvollen Alternativen aufgezählt werden. Als „sinnvoll“ gelten dabei solche Regeln, die einige grundsätzliche Forderungen erfüllen. Die wichtigste dieser Forderungen besagt, daß der „Nebeneffekt“, der durch ein View-Update ausgelöst wird, so gering wie möglich bleiben soll. Keller schlägt in [Kel86] vor, zum Zeitpunkt der View-Definition den Benutzer (i.d.R. den DBA) im Dialog die gültigen Alternativen anzubieten, aus denen er dann die gewünschte Regel auswählt.

In [DD93] wird erwähnt, daß der angekündigte SQL3-Standard die Modifizierbarkeit von Sichten durch die Verwendung spezieller Join-Operatoren (`JOIN USING FOREIGN KEY`, `JOIN USING PRIMARY KEY`, `JOIN USING CONSTRAINT`) wesentlich erweitern wird. Ferner ist für SQL3 angedacht, daß für jede Sicht angegeben werden kann, in welche sichtspezifischen Update-Operationen eine `INSERT`-, `DELETE`- oder `UPDATE`-Operation umgesetzt wird. Letzteres entspricht genau dem Vorschlag aus [SJG+90] für POSTGRES (ein DBMS auf der Basis eines erweiterten Relationenmodells), das Datenmodell um eine Regel-Komponente zu erweitern. Das *Ereignis* `DELETE` löst dann die gewünschte Aktion aus. Zu Beispiel 2.2 paßt die Regel

```
ON DELETE TO ENTLEIHUNGEN
THEN DO DELETE TO AUSLEIHE
WHERE AUSLEIHE.ExNr = ENTLEIHUNGEN.ExNr                    (2.1)
```

Schließlich ist zu berücksichtigen, daß Sichten auch aus Gründen des Datenschutzes verwendet werden, indem ein bestimmter Teil der Instanzenmenge unzugänglich gemacht oder gegen bestimmte Arten des Zugriffs geschützt wird. Es ist dann fraglich, ob die Propagation von Updates in den unzugänglichen Teil überhaupt erlaubt sein darf. Dies ist ein weiteres Argument dafür, daß die Propagation von View-Updates sehr stark von anwendungsspezifischen Semantiken abhängt, so daß die Suche nach allgemeingültigen Regeln zur Update-Propagation nicht sinnvoll erscheint.

2.3.4 Nachteile des Relationenmodells

Die Schlichtheit des Relationenmodells in seiner Grundkonzeption ist leider auch für eine ganze Reihe von Unzulänglichkeiten des Modells verantwortlich, die seine Verwendung im Zusammenhang mit den bereits erwähnten Nicht-Standard-Anwendungen in Frage stellen. Die wichtigsten dieser Unzulänglichkeiten werden in diesem Abschnitt zusammengefaßt.

Da das Relationenmodell per definitionem rein wertebasiert ist, müssen Verweise von einem Tupel auf ein anderes Tupeln implizit über Attributwerte ausgedrückt werden. Das verweisende Attribut hat ein korrespondierendes Attribut in der Zielrelation[8]. Im Relationenschema kann dies durch Fremdschlüsselbedingungen formuliert werden. Ein direkter physischer Link von einem Tupel auf ein anderes wie bei den recordbasierten Modellen ist nicht möglich. Explizit gemacht werden solche impliziten Beziehungen zwischen Tupeln erst bei der Auswertung einer Anfrage durch die Ausführung eines *Joins* (Verbundes), der formal als Hintereinanderausführung eines Kartesischen Produkts zweier Relationen und einer Selektion definiert ist. Die Ausführung der Joins ist ein Flaschenhals bei einer Anfrageauswertung, da große Zwischenergebnisse entstehen können und zahlreiche Wertevergleiche (falls zusammengesetzte Schlüssel vorliegen, dann sogar für Attributkombinationen!) notwendig sind. Viele Optimierungsverfahren bemühen sich deshalb, den Aufwand zur Ausführung der notwendigen Joins möglichst gering zu halten. Dabei helfen Indexe, die Anzahl der notwendigen Wertevergleiche klein zu halten.

Über Fremdschlüssel werden z.B. Assoziationen (d.h. die in Abschnitt 2.1.1 erwähnten „lockeren“ Beziehungen) zwischen Anwendungsobjekten in einem Relationenschema erfaßt. Da die Abbildung einer m:n-Beziehung in einem Relationenschema die Einführung einer „Verknüpfungsrelation“ notwendig macht, sind zum Verfolgen einer solchen Beziehung sogar zwei Joins notwendig.

Aber nicht nur für Assoziationen, sondern auch in vielen anderen Situationen ist man beim Relationenmodell darauf angewiesen, Beziehungen zwischen Tupeln über Fremdschlüssel aufzubauen:

- Die Beschränkung auf *flache* Relationen führt dazu, daß Redundanz nur vermieden werden kann, wenn mengenwertige Attribute wie bei einer 1:n-Beziehung in eine andere Relation ausgelagert werden. Das führt dazu, daß Attribute einer Entität auf mehrere Relationen verteilt werden und explizit durch Joins wieder zusammengeführt werden müssen. Dabei fallen Entitäten, bei denen ein mengenwertiges Attribut gerade die leere Menge ist,

8. Es können natürlich auch Attributkombinationen einen solchen Verweis ausdrücken.

aus dem Ergebnis heraus. Um dies zu vermeiden, müssen Nullwerte und eine weitere Operation, der OUTER JOIN, bemüht werden.

- Es gibt keine ausreichende Unterstützung *komplexer Objekte*. Während die flache Tupelstruktur für kaufmännische und administrative Anwendungen i.d.R. adäquat ist, tauchen in vielen technischen Anwendungen stark strukturierte Objekte auf, die aus vielen Subobjekten zusammengesetzt sind. Ein effizienter Zugriff auf ein komplexes Objekt mit allen seinen Subobjekten – quasi „auf einen Schlag“ – ist mit dem Relationenmodell nicht möglich, sondern die Subobjekte sind auf verschiedene Relationen verteilt und müssen mühsam durch Joins „eingesammelt“ werden [GS82, KM94].
- *isa*-Beziehungen müssen ebenfalls über Verweise durch Fremdschlüsselwerte ausgedrückt werden. Eine automatische Vererbung von Eigenschaften gibt es nicht, sondern es obliegt auch hier dem Anwender, vererbte Information durch explizite Joins wieder „einzusammeln“.

Die soeben genannten Punkte lassen sich unter *„Zerstückelung zusammengehöriger Information“* subsumieren. Dadurch wird der Join zur allgegenwärtigen Operation. Zudem bieten nur wenige relationale Systeme (z.B. Oracle [HKU95]) auf der internen Ebene eine Form von Clustering an, die zusammengehörige Information physisch benachbart speichert und somit die Kosten eines Joins niedrig halten kann.

Die Beschränkung auf die erste Normalform hat auch Konsequenzen für die Verständlichkeit der Anfragesprache:

- SQL ist als Ad-hoc-Anfragesprache für den gelegentlichen Benutzer nicht gut geeignet. SQL-Anfragen sind oft mit vielen Join-Bedingungen überfrachtet, die das Verständnis der Anfrage erschweren. Der Benutzer benötigt umfangreiche Kenntnis über die Verteilung der Information über die verschiedenen Relationen eines Schemas und über die Zusammenhänge zwischen den Relationen.
- Einige umgangssprachlich leicht zu formulierende Anfragen sind nur sehr umständlich in SQL-Anfragen umsetzbar: z.B. eine Anfrage nach allen Anwendungsobjekten, die die Bedingung „A = {a, b}“ erfüllen, wobei A ein (ursprünglich) mengenwertiges Attribut ist.

Um Berechnungsvollständigkeit zu erlangen, werden SQL-Anweisungen in eine Host-Sprache – z.B. Cobol, PL/1, C oder Pascal – eingebettet (*Embedded SQL*). Das Quellprogramm wird zunächst von einem Präprozessor bearbeitet, der die eingebetteten SQL-Statements in Funktionsaufrufe an das Datenbanksystem umgesetzt. Notwendige Berechnungen werden dann im Adreßraum des Programmes ausgeführt, nachdem die relevanten Daten mittels einer SQL-Anfrage dorthin transferiert wurden. Dies ist immer noch die Standard-Vorgehensweise bei der Erstellung kommerzieller relationaler Anwendungen.

- Bei dem Ansatz der SQL-Einbettung (*Embedded SQL*) trifft man auf das Problem des *impedance mismatch*, der ein Ausdruck der Tatsache ist, daß die Datenmodelle des DBMS und der Host-Sprache i.d.R. schlecht zueinander „passen“: Die verbreiteten imperativen Programmiersprachen kennen keinen Datentyp „Menge“ und können somit auch Anfrageergebnisse nicht direkt verarbeiten.

Bei der Einbettung von SQL in eine Hostsprache müssen spezielle Datenstrukturen und Kommunikationsvariablen definiert werden, um den Transfer zwischen den getrennten Programmiersprachen- und Datenbankadreßräumen zu ermöglichen. Die Weiterverarbeitung von Anfrageergebnissen erfolgt über die von Embedded SQL bereitgestellten *Cursor*, die von

Tupel zu Tupel bewegt werden können, also mit den Fingern in ESCHER [Weg91b] oder den *currency pointers* des Netzwerk-Modells vergleichbar sind. Während die Beweglichkeit des Cursor früher nur sehr eingeschränkt war (sequentieller Scan in einer Richtung), sind sie ab SQL/92 sehr frei bewegbar [DD93].

Um dem „impedance mismatch“ aus dem Weg zu gehen, wurden *Datenbank-Programmiersprachen* (DBPL) entwickelt, die das Relationenmodell in ihr Typsystem integrieren und entsprechende Konstrukte zur Verarbeitung von Mengen vorsehen. Als Vertreter relationaler DBPL, die auf der Basis verbreiteter Progammiersprachen entstanden, seinen genannt: PS/Algol [ACC81], Pascal/R [Sch77] und die auf Modula-2 aufbauende Sprache DBPL [SM92] (vgl. auch die Übersicht in [AB87]). In der Praxis haben sie jedoch wenig Verbreitung gefunden. Die Autoren von [SRL+90] sehen die Hauptschuld für das Festhalten an der Host-Einbettung mit den damit verbundenen Nachteilen in der fehlenden Zusammenarbeit von Compiler-Anbietern und DBMS-Entwicklern.

Im Verhältnis zu den auch in der Praxis immer mehr Verbreitung findenden objektorientierten Programmiersprachen (OOPL) tritt der „impedance mismatch“ noch gravierender zutage:

- Das Relationenmodell kennt kein ADT-Konzept, wie es in vielen Programmiersprachen heute zu finden ist. *Einkapselung* und *typspezifische Operationen*, die ggf. auch in Anfragen verwendet werden dürfen, fehlen im Relationenmodell völlig. Das ursprüngliche Relationenmodell ist im wesentlichen auf den Strukturteil, einige Integritätsbedingungen und generische Anfrage- und Updateoperationen fixiert. Erst im SQL3-Standard sind Verbesserungen in dieser Hinsicht zu erwarten.

Letztendlich gefährdet das Fehlen einer Abstraktion in Form von ADTs die Konsistenz einer Datenbank: Gerade durch die Einkapselung der inneren Struktur eines Objektes und die Spezifikation einer Schnittstelle konsistenzerhaltender Operationen kann das Objekt in einem konsistenten Zustand gehalten werden. Bei fehlender Einkapselung kann eine einzelne generische Update-Operation leicht zu einem inkonsistenten Zustand führen, wie es bereits in Abschnitt 1.2 an einem CAD-Beispiel deutlich gemacht wurde.

Ein weiteres Problem betrifft die Verwendung informationstragender Attribute als Schlüssel bzw. Fremdschlüssel:

- Die Identifikation von Anwendungsobjekten findet über sichtbare Schlüsselattribute statt. Da die Werte von Schlüsselattributen auch zum Herstellen aller Arten von Beziehungen zu anderen Tupeln verwendet werden, findet auf Instanzenebene zumindest eine Replizierung dieser Werte an mitunter sehr vielen Stellen statt. Soll ein Schlüsselattribut modifiziert werden, dann kann ein solches Update nur mit sehr großem Aufwand an alle Replikate propagiert werden. Falls man darauf verzichten will, ist die referentielle Integrität gefährdet.
  Probleme ergeben sich auch, wenn man zwar Werte für Schlüsselattribute nicht kennt (Nullwerte sind für Schlüsselattribute nicht zugelassen), jedoch die Existenz eines Objektes in der Datenbank festhalten will.

Konsequenz dieser Tatsache kann nur sein, daß informationstragende Attribute (wie z.B. Namen, Kfz-Nummern usw.), bei denen die Möglichkeit der Änderbarkeit besteht, nicht als Fremdschlüssel verwendet werden sollten. Es wird daher häufig ein *künstlicher Schlüssel* (ein *Surrogat*) eingeführt, der automatisch vom Anwendungsprogramm erzeugt wird. Jedoch ist auch dieser für den Benutzer sichtbar und modifizierbar! Besser ist es, Surrogate vor dem

Benutzer völlig zu verbergen. Dies führt uns direkt zum Prinzip der *Objektidentität* [KC86, KA90], wie es in OOPLs seit langem erfolgreich Anwendung findet: Objekte werden mit eindeutigen, invarianten, vom System erzeugten und für den Benutzer nicht sichtbaren *Objektidentifikatoren* (*OIDs*) versehen. Er kann lediglich zwei OIDs auf Gleichheit testen. Im Grunde hatte man beim Hierarchischen und Netzwerk-Modell bereits eine Form von „Objektidentität", nämlich in Form der physischen Adresse eines Records[9]. Im Unterschied dazu sollen OIDs jedoch rein *logischer* Natur sein und nicht von der physischen Realisierung abhängen. Gleichwohl ist es selbstverständlich, daß OIDs die Definition von Schlüsseln nicht überflüssig machen: Die Möglichkeit einer wertebasierten Identifizierung von Objekten hat für den Benutzer weiterhin eine zentrale Bedeutung.

2.4 Erweiterungen des Relationenmodells

Die Erweiterungen bestehender relationaler Systeme können grob in folgende Gruppen eingeteilt werden:

- Verallgemeinerung des Strukturteils
- Einführung eines Typkonzepts
- Integration weiterer Konzepte wie z.B. komplex strukturierte Sichten, Integritätsbedingungen und Regeln

2.4.1 Verallgemeinerungen des Strukturteils

Als eine wesentliche Schwäche des Strukturteils des Relationenmodells stellt sich die Forderung nach der Ersten Normalform heraus, die besagt, daß allen Attributen eines Tupels eine atomare Domäne (bzw. ein atomarer Typ) zugeordnet werden muß. Es wurde bereits von Makinouchi darauf hingewiesen [Mak77], daß sich in vielen Fällen eine natürlichere strukturelle Modellierung ergibt, wenn man nicht an der Ersten Normalform festhält.

In den 80er Jahren war die Loslösung von der „flachen" Struktur des Relationenmodells Gegenstand zahlreicher Forschungsarbeiten und Prototypen. Untersucht wurde das NF^2-Modell (*N*on *F*irst *N*ormal *F*orm), in dem die Attribute einer Relation entweder atomar oder selbst wieder NF^2-Relationen sind. Falls mehrere Attribute einer NF^2-Relation relationenwertig sind, dann spricht man auch von sog. *unabhängigen Wiederholungsgruppen*, die im „flachen" Relationenmodell zum Aufsplitten einer Relation führen, um Redundanz zu vermeiden.

In [JS82] wird zunächst von einer einstufigen Schachtelung ausgegangen, d.h. Attribute sind entweder atomar oder Relationen in 1. Normalform. Die Relationenalgebra wird um die Operationen *nest* und *unnest* erweitert, die eine Restrukturierung von geschachtelten Relationen erlauben, indem sie in eine „flache" Relation eine neue Schachtelung einfügen (*nest*) oder eine Schachtelungsebene entfernen (*unnest*). Auf die Definition dieser Operationen soll hier nicht näher eingegangen werden, es sei auf die Darstellungen zum NF^2-Modell in [KK89] bzw. die

9. Ullmann schreibt in [Ull88] dem Netzwerk-Modell gar das Prädikat „objektorientiert" zu. Hier geht Ullman jedoch wesentlich zu weit, indem er den Begriff „objektorientiert" auf die Formel „invariante Identität + veränderbarer Zustand" verkürzt.

entsprechenden Kapitel in [PDG+89, Heu92, AD93, KK93] verwiesen. Die Grundoperationen der „flachen“ Relationenalgebra können in naheliegender Weise auf NF^2-Relationen verallgemeinert werden.

In [TF86] und [SS86] wird beliebige Schachtelungstiefe zugelassen, d.h. Grundlage ist das NF^2-Modell. Während in [TF86] die Operationen der dort vorgestellten erweiterten Relationenalgebra nur auf der äußeren Schachtelungsebene anwendbar sind, erlauben Schek und Scholl [SS86] in ihrer *NF^2-Relationenalgebra* (*geschachtelten Relationenalgebra*) die Schachtelung von Algebraausdrücken, indem überall dort, wo Attributnamen stehen dürfen, nun auch Algebraausdrücke zugelassen sind.

Man beachte, daß der Begriff *nested relation* (geschachtelte Relation), der oft als Synonym für NF^2-Relation verwendet wird, in einigen Arbeiten eine speziellere Bedeutung hat: Es sind darunter nur solche NF^2-Relationen zu verstehen, die aus einer flachen Relation durch eine Folge von *nest*-Operationen entstanden sind (vgl. [FV85]).

Während die Hintereinanderausführung *unnest* o *nest* (wobei unnest genau die durch *nest* hergestellte Schachtelung wieder auflöst) die Identität liefert, ist dies für *nest* o *unnest* i.a. nicht der Fall. Die Frage, wann auch *unnest* gefolgt von *nest* die Identität liefert, wurde bereits in [JS82] untersucht. In [RKS88] wird eine Klasse von NF^2-Relationen identifiziert, für die die gewünschte Eigenschaft zutrifft: Es sind die NF^2-Relationen in sog. *Partitioned Normal Form* (kurz PNF-Relationen). Eine NF^2-Relation ist genau dann eine PNF-Relation, wenn es auf jeder Schachtelungsebene mindestens ein atomares Attribut gibt, das Schlüsseleigenschaft besitzt. Diese Eigenschaft ist in den meisten praktischen Fällen tatsächlich erfüllt. In [RKS88] werden die Operationen der geschachtelten Relationenalgebra so definiert, daß die Algebra abgeschlossen unter PNF-Relationen bleibt. Eine analoge Bedingung wird auch in [AB84] für das *Verso*-Datenmodell, einer Variante des NF^2-Modells, formuliert. Es wird jedoch oft übersehen, daß die PNF-Eigenschaft nicht hinreichend für *nest* o *unnest* = *id* ist. Falls ein relationenwertiges Attribut die leere Menge ist, dann gibt es für das Entnesten zwei Alternativen: (a) das entsprechende Tupel entfällt beim Entnesten, dann kann die folgende Nestung das „verlorengegangene“ Tupel nicht wiederherstellen, oder (b) für die Attribute der zu entnestenden leeren Relation werden im Ergebnis Nullwerte mit „does not exist“-Semantik eingesetzt. Die Integration von Nullwerten in das NF^2-Modell und ihre Berücksichtigung durch die Operationen der NF^2-Algebra werden in [RKS89] bzw. [Lev92] behandelt. Beide Arbeiten unterscheiden sich in ihren Grundannahmen zur Semantik von Nullwerten sehr stark, zudem sind ihre Vorschläge in vielen Punkten nur schwer nachvollziehbar und besitzen somit für die Praxis nur geringen Nutzen. Die bereits vom Relationenmodell bekannte Problematik der Integration von Nullwerten, ihren verschiedenen Semantiken und ihrer Behandlung in Anfragesprachen – es sei auf die z.T. heftigen Diskussionen in [Cod90, DD93, AD93, McG94] verwiesen –, entschärft sich durch den Übergang zum NF^2-Modell kaum.

Unter den Implementationen des NF^2-Modells ist insbesondere DASDBS (Darmstadt Database System) [SPS+90] zu nennen. Das NF^2-Modell spielt in DASDBS auf zwei Ebenen eine Rolle [SS89a]. Zunächst eignet es sich auf logischer Ebene als Ausgangsdatenmodell, das applikationsabhängig erweitert werden kann und zu sog. „DASDBS Frontends“ führt. Daneben werden auf der internen Ebene NF^2-Relationen als effiziente Speicherstrukturen verwendet, die hierarchisches Clustering realisieren. Clustering über NF^2-Speicherstrukturen wurde auch für die Speicherung relationaler Datenbanken [SPS87] untersucht. Auf interner Ebene bekommt jedes Tupel als weiteres „unsichtbares“ Attribut einen eindeutigen Tupelidentifikator

(ein TID) zugeordnet, der u.a. für Indexe verwendet wird (vgl. auch [KD91, Her93]), die wiederum der Anfrageoptimierung dienen.

Andere Implementationen des NF^2-Modells sind Verso [SAB+89] für das gleichnamige Modell [AB84] und Triton [HRS91]. Der letztgenannte Prototyp implementiert SQL/NF [RKB87], eine Erweiterung von SQL auf NF^2-Relationen. Ähnlich wie bei der geschachtelten Relationenalgebra läßt SQL/NF überall dort, wo gewöhnlich atomare Werte erwartet werden, auch NF^2-Relationen und Anfragen zu. Ferner wird das Nesten und Entnesten direkt über die syntaktischen Konstrukte `NEST (`*Anfrage*`) ON` *Attribut-Liste* und `UNNEST (`*Anfrage*`)ON` *Attribut-Liste* unterstützt.

Als weitere Verallgemeinerung des NF^2-Modells ist das *erweiterte* NF^2-Modell (kurz: eNF^2-Modell) zu nennen, das eng mit dem 1983 begonnenen „AIM-P"-Projekt (*A*dvanced *I*nformation *M*anagement *P*rototype) am Wissenschaftlichen Zentrum der IBM in Heidelberg verknüpft ist [DKA+86, PA86, PD89]. Dieses Modell liegt auch dem ESCHER-Prototyp zugrunde (vgl. Kapitel 1), dessen Wurzeln auf Arbeiten im Zusammenhang mit AIM-P zurückgehen.

Das eNF^2-Modell unterscheidet sich vom NF^2-Modell durch eine freiere Anwendung der sog. *Konstruktoren* zur Konstruktion beliebig strukturierter komplexer Strukturen. Instanzen dieser komplexen Strukturen werden *komplexe Objekte* genannt. Zunächst wird die Koppelung des Mengen- und des Tupelkonstruktors, wie man sie in der Relation antrifft, aufgegeben. Auf diese Weise läßt sich z.B. auch eine Menge von atomaren Werten definieren; im NF^2-Modell ist man auf die Darstellung durch eine Menge von Tupeln mit einem Attribut angewiesen. Ferner läßt sich der Tupelkonstruktor dazu verwenden, sinnvolle „Einheiten" innerhalb eines Tupels abzugrenzen (z.B. die tupelwertigen Attribute `Name` und `Adresse` in Abb. 1.1 und 1.3). Schließlich wird die *Liste* in das Datenmodell eingeführt, um eine benutzerdefinierte Ordnung von Elementen einer Menge zu unterstützen.

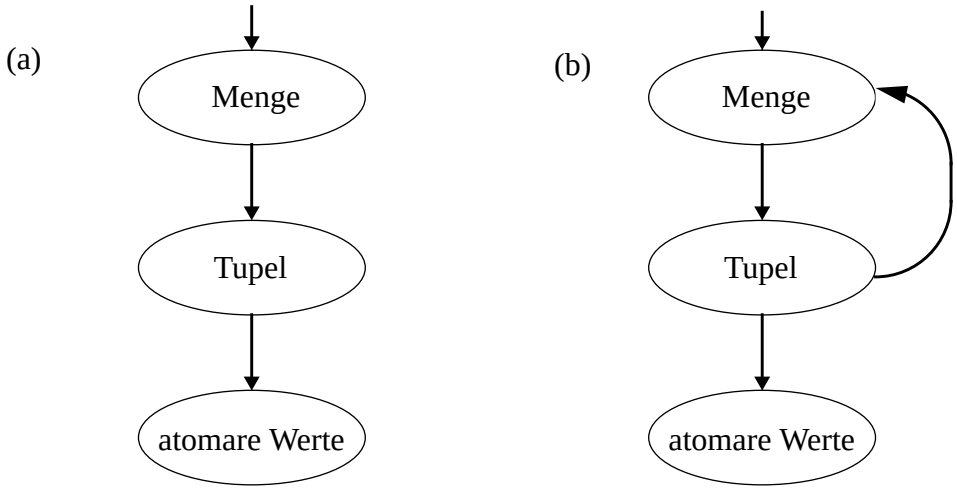


Abbildung 2.7: Konstruktoren (a) beim Relationenmodell, (b) beim NF^2-Modell

In den Abb. 2.7 und 2.8 sind die Möglichkeiten der Strukturbildung für das Relationenmodell, das NF^2-Modell und das eNF^2-Modell graphisch veranschaulicht. Man erkennt die beliebige Kombinierbarkeit der Konstruktoren beim eNF^2-Modell (Orthogonalität ihrer Anwendung). Interessant ist auch, daß in Abb. 2.8 jeder Knoten als Einstiegspunkt fungieren kann, so daß auch eine „Tabelle" definiert werden kann, die nur aus einem atomaren Wert besteht.

Ein Vergleich des eNF^2-Modells mit dem semantischen Datenmodell IFO zeigt, daß der Tupel- und der Mengenkonstruktor mit den ebenfalls orthogonal anwendbaren Abstraktionsmechanismen Aggregation bzw. Gruppierung in IFO korrespondieren.

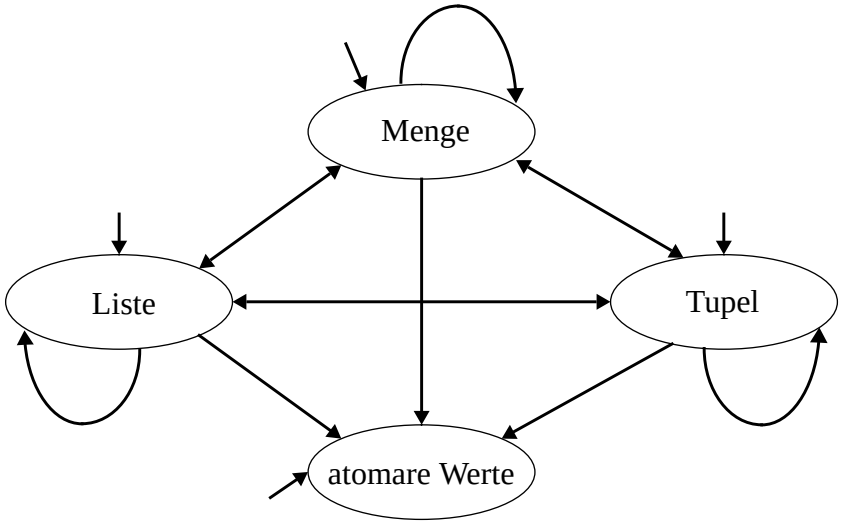


Abbildung 2.8: Konstruktoren beim eNF^2-Modell (vgl. [LKD+91])

In der Implementation von AIM-P ist allerdings keine vollständige Orthogonalität der Konstruktoranwendung erreicht worden [LPS91]. Die erlaubten Kombinationen sind der Abb. 2.9 zu entnehmen. In AIM-P wird bei Mengen auf die Forderung nach Duplikatfreiheit verzichtet, so daß besser von Multimengen gesprochen werden sollte. Duplikateliminierung sowohl für Multimengen als auch für Listen ist als separate Operation vorgesehen [PA86, KSW89, SLP+89].

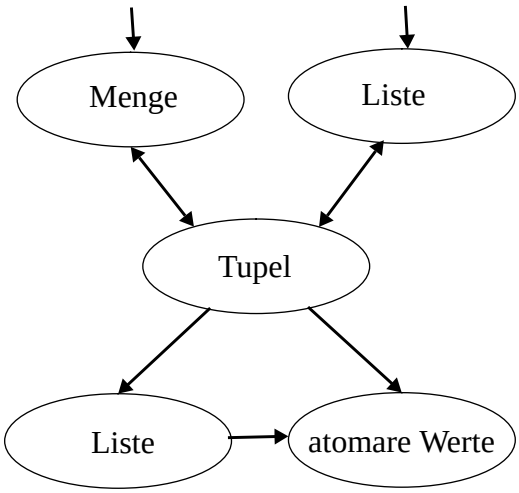


Abbildung 2.9: Für AIM-P implementiertes Modell

Für das eNF^2-Datenmodell ist eine eigene SQL-Erweiterung entwickelt worden, die Sprache HDBL (Heidelberg Database Language) [PT86]. Sie umfaßt die DDL, die DML und eine Anfragesprache. HDBL-Anweisungen und -Anfragen können über das Online Interface

[LPS91] eingegeben werden. Eine Programmiersprachenschnittstelle im traditionellen Einbettungsstil mit Pascal als Hostsprache ist ebenfalls dokumentiert [ESW88].

2.4.2 Erweiterungen um Typkonzepte

Die im vorangegangenen Abschnitt diskutierten strukturellen Erweiterungen kommen in erster Linie der direkten Modellierung streng hierarchisch strukturierter Objekte mit fester Schachtelungsstruktur und -tiefe zugute. Dagegen läßt sich eine rekursive „is part of“-Beziehung nicht ohne Probleme erfassen. Eine Lösung könnte die Definition einer NF^2-Relation `parts` mit der Struktur

```
(pnr: int, name: string, subparts:(fpnr: int))
```

sein, d.h. `parts` ist eine Relation mit den Attributen `pnr`, `name` und `subparts`, wobei `subparts` relationenwertig ist (mit genau einem Attribut `fpnr`). Die Werte von `fpnr` sollen Fremdschlüsselwerte sein, die auf Tupel in der Relation `parts` verweisen. Es muß also zunächst festgelegt werden, daß `pnr` tatsächlich ein Schlüssel für die Tupel in `parts` ist. Ferner muß eine Fremdschlüsselbedingung formuliert werden, die angibt, daß `fpnr` auf `pnr` „zeigt“. Nur dann kann auch die referentielle Integrität eingehalten werden. Es fällt sofort auf, daß mit dieser Art der Modellierung nahegelegt wird, daß *alle* Einzelteile eines komplexen Objektes Elemente von `parts` sind. Die für die „is part of“-Beziehung charakteristische Schachtelung der Komponentenobjekte „innerhalb“ des umgebenden Objektes ist damit aufgehoben.

Vom konzeptuellen Standpunkt viel eleganter ist die in vielen Programmiersprachen seit langem bewährte Definition von Typen. Es bietet sich an, einen Tupel-Typ `Part` zu definieren:

```
TYPE Part = TUPLE
              pnr: int;
              name: string;
              subparts: SET OF Part;
            END;                                          (2.2)
```

Eine NF^2-Relation läßt sich dann – ähnlich wie eine Variable – etwa in der Form

```
RELATION parts : SET OF Part;
```

definieren. Tatsächlich implementiert auch der bereits erwähnte Prototyp DASDBS nicht das klassische NF^2-Modell, sondern das soeben beschriebene, um Tupel-Typen erweiterte Modell [SPS+90]. Die Definition von Tupel-Typen ist in ihrem Nutzen nicht allein auf direkt rekursive Definitionen wie in (2.2) beschränkt: Die Verwendung von Typen macht Schemata übersichtlicher und ist konsistenzunterstützend. So ist z.B. die unsinnige Vereinigung zweier Relationen `Weine(Name: string, Jahrgang: integer)` und `Personen(Name: string, Jahrgang: integer)` beim Relationenmodell erlaubt, da lediglich Strukturgleichheit verlangt wird. Unter Verwendung von Typen kann im Zusammenhang mit der Vereinigung die Gleichheit der Typnamen verlangt werden.

In Abschnitt 1.2 wurde anhand des CAD-Beispieles deutlich gemacht, daß *shared subobjects* (z.B. Eckpunkte, die zwei aneinander grenzenden Flächen gemeinsam sind) eine adäquate Form der Modellierung erfordern, die das eNF^2-Modell nicht leistet. Hier bietet sich der Einsatz von Objekttypen an. Hat man lediglich Wertetypen zur Verfügung, dann kann – wie man

es aus vielen Programmiersprachen (z.B. Pascal, C) kennt – „object sharing“ über Pointertypen realisiert werden. In (2.2) würde man dann etwa „`subparts: SET OF REFTO Part`“ schreiben, und die Menge `subparts` wäre eine Menge von physischen Adressen von `Part`-Instanzen. Entspechende Vorschläge sind z.B. für LauRel [Lar88], eine weitere Variante des NF^2-Modells, und AIM-P [PD89] gemacht worden.

Um Typen im Sinne abstrakter Datentypen zu erhalten, muß der direkte Zugriff auf bzw. die direkte Manipulation von Typinstanzen über generische Operationen ausgeschlossen werden und die Typdefinition um eine Operationenschnittstelle ergänzt werden (Einkapselung). Ob dann zumindest noch ein *lesender* Zugriff über generische Operationen zugelassen sein sollte, ist strittig.

In AIM-P lassen sich Wertetypen definieren. Die strukturelle Seite einer Typdefinition wird in der Form „`DECLARE` <*Typname*> <*eNF²-Struktur*> `END`“ angegeben [LKD+91]. Der neue Typ kann dann als Parameter einer Funktionsdeklaration verwendet werden. Die Implementation der Funktionen erfolgt dabei in der Programmiersprache Pascal auf der Grundlage einer „Übersetzung“ der eNF^2-Struktur der Typdefinition in eine Pascal-Datenstruktur [DKS+88], da vor der Ausführung der Funktion die Daten aus der Datenbank in das Pascal-Format transformiert werden müssen. Mengen und Listen werden in Arrays übersetzt. Da Pascal keine dynamischen Arrays kennt, müssen Obergrenzen für die Kardinalität der Mengen bzw. Listen angegeben werden.

Einen anderen Ansatz verfolgt der Prototyp POSTGRES [SK91], der eine Erweiterung des relationalen DBMS Ingres [SWK+76] ist und mittlerweile unter dem Namen *Illustra* kommerziell vertrieben wird [Ill96a]. Neben einigen strukturellen Erweiterungen können dort benutzerdefinierte (Werte-)Typen als sog. POSTGRES-ADTs definiert werden. Die innere Struktur der Instanz eines POSTGRES-ADTs ist durch eine (beliebig komplexe) Datenstruktur der Programmiersprache C angegeben. Die Implementation der benutzerdefinierten Funktionen erfolgt konsequenterweise auch in C. Eine Instanz eines POSTGRES-ADTs ist somit – anders als bei AIM-P – für die Datenbank zunächst eine uninterpretierte Bytefolge, vergleichbar mit einem BLOB. Soll ein ADT für komplex strukturierte Objekte definiert werden, so liegt die Effizienz des Zugriffes auf Teilobjekte und der Verwaltung des kompletten komplexen Objekts allein in der Hand des C-Programmierers. Der Vorteil des AIM-P-Ansatzes liegt darin, daß die innere Struktur einer Instanz der Datenbank bekannt ist und auf interner Ebene diese Information ausgenutzt werden kann (z.B. für Indexe). Der Nutzen von POSTGRES-ADTs zur Definition beliebig komplex strukturierter Typen ist eher gering einzuschätzen. Dagegen eignen sich POSTGRES-ADTs gut zur nachträglichen Aufnahme neuer Basistypen, wie z.B. `date` oder `time`.

Sowohl bei AIM-P als auch bei POSTGRES wird davon abgesehen, benutzerdefinierte Funktionen bestimmten Typen zuzuordnen (vgl. den „Receiver“-Typ in objektorientierten Sprachen), wie es die Theorie abstrakter Datentypen verlangt [Gut77, EGL89].

Ein ähnlicher Ansatz wie bei POSTGRES wird von Starburst [LLP+91] verfolgt. Dort wird versucht, prinzipiell am Relationenmodell festzuhalten. Es gibt jedoch die Möglichkeit, ein komplexes Objekt als *long field* in einem Attribut einer (flachen) Relation abzuspeichern. Die Entscheidung über die interne Struktur eines *long field*-Wertes liegt wieder in der Hand des Implementierers benutzerdefinierter Funktionen. Daneben gibt es in Starburst die Möglichkeit, komplexe Objekte, die auf verschiedene Relationen verteilt sind, in Form einer sog. XNF-

Sicht (*eX*tended *N*ormal *F*orm) wieder zusammenzuführen [MPP+93, MP94], worauf wir im folgenden Abschnitt noch näher eingehen werden.

Durch die Aufnahme benutzerdefinierter Datentypen in das Datenmodell wird versucht, der Forderung nach *Erweiterbarkeit* von DBMS gerecht zu werden. Insbesondere ist die Eigenschaft, daß benutzerdefinierte Funktionen bzw. Operationen jetzt auch auf der Datenbankseite bekannt sind, ein entscheidender Fortschritt gegenüber bisherigen DBMS. Benutzerdefinierte Funktionen können in den oben genannten Prototypen direkt in Anfragen verwendet werden, z.B. kann der Ausdruck „`Abstand(p1, p2) <= 2`“ als Bedingung für den Abstand zweier Punkte `p1` und `p2` Bestandteil einer `WHERE`-Klausel sein.

Betrachtet man noch einmal POSTGRES als exemplarischen Vertreter einer eher konservativen Erweiterung des Relationenmodells, so erhält man den Eindruck, daß dort sehr viele verschiedene Konzepte integriert wurden und insgesamt ein etwas inhomogenes Datenmodell entstand. Neben der Definition von ADTs, von der bereits die Rede war, wird mit jeder Definition einer Relation gleichzeitig ein gleichnamiger *constructed type* (auch *class* genannt) definiert, wie z.B.

```
CREATE Manager
  (department: Dept, subordinates: Empl)
  INHERITS Empl;                                    (2.3)
```

Hierbei sind `Dept` und `Empl` bereits definierte Relationen/“konstruierte Typen“. Wie angedeutet, bietet POSTGRES an dieser Stelle auch einen Vererbungsmechanismus, was bei den ADTs nicht der Fall ist. Sehr unschön ist vor allem, daß die Attribute `department` und `subordinates` beide relationenwertig sind, was beim Attribut `department` nicht erwünscht ist. Genau denselben Fehler macht auch UniSQL [Kim93], ein mit POSTGRES vergleichbares System neueren Datums, das ein erweitertes Relationenmodell implementiert[10]. Parallel zu den in C zu implementierenden Funktionen gibt es noch sog. POSTQUEL-Funktionen, die parametrisierte Anfragen in der POSTGRES-spezifischen Anfragesprache POSTQUEL sind. Der Datentyp `postquel`, der es ermöglichte, beliebige Anfragen als Werte eines Attributs abzuspeichern, wurde aus einer früheren Version [RS87] nicht nach [SK91] übernommen.

2.4.3 Weitere Konzepte postrelationaler DBMS

In diesem Abschnitt gehen wir auf zwei weitere, für diese Arbeit relevante Aspekte der Erweiterung relationaler DBMS ein.

2.4.3.1 Komplexe Objekt-Sichten

Neben der starken Beschränkungen der Datenmanipulation über relationale Sichten ist die Tatsache, daß sie genauso wie Basistabellen der ersten Normalform genügen müssen, als ein weiteres Manko anzusehen. Die tatsächliche (Schachtelungs-)Struktur komplexer Anwendungsobjekten oder Anwendungsausschnitten kann also über Sichten nicht wiedergege-

10. Systeme wie POSTGRES (Illustra) oder UniSQL bezeichnen sich auch als *objekt-relationale* DBMS.

ben werden. Es ist i.a. unmöglich, ein komplexes Anwendungsobjekt in genau einer relationalen Sicht zusammenzufassen, ohne daß auf der Instanzenebene Redundanz entsteht (Wäre die Vermeidung von Redundanz möglich, dann hätte man die Sicht gleich als Basistabelle definieren können!).

Das DBMS Starburst [LLP+91] ermöglicht die Konstruktion komplex strukturierter „Sichten“, während es gleichzeitig an der relationalen Speicherung aller Daten festhält. Dazu wurde SQL zu SQL/XNF [MPP+93, MP94] erweitert. Mit SQL/XNF werden komplex strukturierte *XNF-Sichten* definiert, deren Instanzen sog. *composite objects* (kurz: COs) sind. Die grundlegende Idee ist, daß während jeder Verarbeitungsphase jeweils ein bestimmter Teil des gesamten Datenbestandes (in [MPP+93] „working set“ genannt) relevant ist. Diese Daten gilt es, in einem „Cache“ für die Applikation bereit zu halten, um so eine schnellere Verarbeitung zu ermöglichen. Ein CO stellt gerade eine strukturierte Verarbeitungseinheit dar. Der Aufbau einer XNF-Sicht orientiert sich dabei stark an der aus dem ER-Modell bekannten Unterscheidung von Entitäten und Beziehungen. Ein wesentlicher Unterschied ist jedoch, daß in XNF-Sichten Beziehungen gerichtet sind. Die Bestandteile einer XNF-Sicht sind:

- *XNF-Tabellen*: Sie sind flache relationale Tabellen, die jeweils das Resultat einer SQL-Anfrage über der zugrundeliegenden relationalen Datenbank sind
- *XNF-Beziehungen*: Diese setzen die XNF-Tabellen miteinander in eine *gerichtete* Beziehung. Instanzen von XNF-Beziehungen sind Tupel, die aus Fremdschlüsselwerten der miteinander in Beziehung stehenden Tupel aus den an der Beziehung beteiligten XNF-Tabellen bestehen.

Während ein gewöhnlicher relationaler View genau *einer* virtuellen Tabelle entspricht, besteht eine XNF-Sicht i.d.R. aus mehreren XNF-Tabellen. In [MPP+93] wird davon ausgegangen, daß jede XNF-Sicht mindestens eine XNF-Tabelle enthält, in die keine gerichtete Beziehung mündet. Solche XNF-Tabellen werden Wurzel-Tabellen genannt. Unklar bleibt jedoch, welches die Wurzel-Tabelle sein soll, wenn die gerichteten Beziehungen einen Kreis ergeben. Wir erläutern nun die Definition von XNF-Sichten anhand eines Beispiels.

Beispiel 2.3

Grundlage einer XNF-Sichtdefinition ist das `OUT OF…TAKE…`-Konstrukt. Mit

```
CREATE VIEW Sicht_1 AS
OUT OF
  Xabt          AS Abt,
  Xmitarb       AS Mitarb,
  Xproj         AS Proj,
  beschaeftigt  AS (RELATE Xabt, Xmitarb
                    WHERE Xabt.nr = Xmitarb.abtnr),
  fuehrt_durch  AS (RELATE Xabt, Xproj
                    WHERE Xabt.nr = Xproj.abtnr)
TAKE *
```

wird die XNF-Sicht `Sicht_1` definiert. Sie besteht aus den XNF-Tabellen `Xabt`, `Xmitarb` und `Xproj`, die jeweils genau einer Basisrelation entsprechen. Man hätte statt `Abt` auch `(SELECT * FROM Abt)` schreiben können oder die Menge der Tupel aus `Abt` einschränken können, z.B. mit `(SELECT * FROM Abt WHERE ort = "Kassel")`. Die beiden

gerichteten Beziehungen sind `beschaeftigt` und `fuehrt_durch`, die über ein Prädikat in der `WHERE`-Klausel des `RELATE`-Konstrukts spezifiziert werden. In Abb. 2.10 sind die Bestandteile dieser Sicht grau unterlegt. Hinter `TAKE` kann alternativ zu `*` auch eine Projektionsliste angegeben werden, um nur gewisse XNF-Tabellen oder Beziehungen in der Sicht zu halten oder um gewisse Spalten aus XNF-Tabellen wegzuprojizieren.

Die Sicht `Sicht_1` läßt sich zur Definition weiterer Sichten verwenden. So fügt

```
CREATE VIEW Sicht_2 AS
OUT OF
  Sicht_1,
  arbeitet_an   AS (RELATE Xmitarb, Xproj
                    USING MitarbProj mp
                    WHERE xproj.nr = mp.pnr
                    AND Xmitarb.nr = mp.mnr)
  WHERE Xmitarb m SUCH THAT m.gehalt < 5000
TAKE *
```

der Sicht `AlleAbt` die in Abb. 2.10 ebenfalls angegebene Beziehung `arbeitet_an` hinzu. Für die Definition dieser Beziehung wird die Basistabelle `MitarbProj` als „Verknüpfungstabelle“ bei einer m:n-Beziehung benötigt. Gleichzeitig wird eine Beschränkung auf diejenigen Angestellten angegeben, die weniger als 5000 DM verdienen („Ecken-Restriktion“). ❒

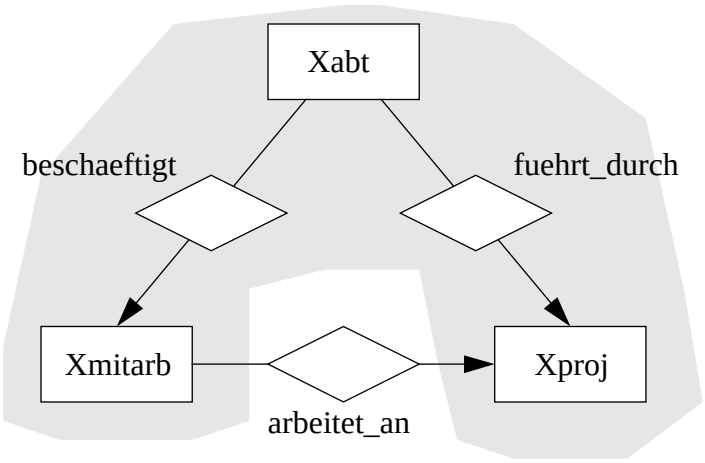


Abbildung 2.10: ER-artige Veranschaulichung einer XNF-Sicht

Anfragen auf XNF-Sichten werden ebenfalls über das `OUT-OF-TAKE`-Konstrukt gestellt. Die Anfrage

```
OUT OF Sicht_2
WHERE beschaeftigt(a, m) SUCH THAT m.gehalt < a.budget/100
TAKE *
```

bewirkt eine „Kanten-Restriktion“ und liefert wieder eine XNF-Sicht. Die Anfragesprache ist also abgeschlossen unter XNF-Sichten.

Die Instanziierung einer XNF-Sicht, d.h. der Aufbau der COs, erfolgt nach dem Prinzip der Erreichbarkeit: Zunächst sind alle Tupel der Wurzel-Tabellen erreichbar. Ferner sind alle diejenigen Tupel in anderen XNF-Tabellen erreichbar, die in einer XNF-Beziehung zu einem bereits erreichbaren Tupel stehen. In Abb. 2.11 ist eine Beispielinstanz angegeben. Die Pfeile entspechen dabei den Instanzen der gerichteten Beziehungen. Man beachte, daß XNF-Sichten das *object sharing* unterstützt, z.B. bearbeiten die Mitarbeiter m1 und m2 dasselbe Projekt p1.

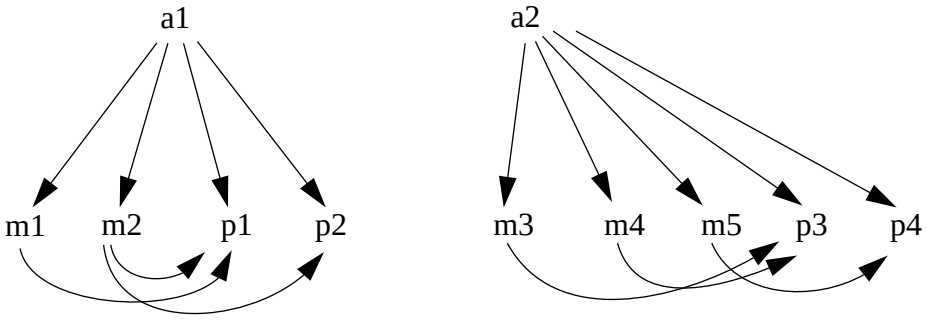


Abbildung 2.11: Instanziierung der XNF-Sicht `Sicht_2`

Zu `Sicht_2` könnte man noch eine zusätzliche Beziehung `geleitet_von` von `Xproj` in Richtung `Xmitarb` hinzufügen, die jedem Projekt genau einen Projektleiter zuordnet. Es entsteht eine *rekursive* XNF-Sicht, da der XNF-Graph nun einen Kreis enthält.

Innerhalb einer XNF-Sicht können auf diese Weise komplexe Objekte, die auf mehrere Relationen verteilt worden sind, als ein geschlossenes CO aufgebaut werden. Für die effiziente Verarbeitung eines COs wird vorausgesetzt, daß die Instanzen einer XNF-Sicht „komplett“ in den XNF-Cache im Hauptspeicher geladen werden, so daß danach kein *object faulting* auftritt. Die Verarbeitung erfolgt dann über eine navigatorische Cursor-Schnittstelle mit unabhängigen (für die Wurzel-Tabellen) und abhängigen Cursorn (für alle anderen XNF-Tabellen).

Leider werden Update-Operationen auf XNF-Sichten in [MPP+93, MP94] und ihre Propagation in die Basisrelationen nur sehr knapp besprochen. Neben *insert*, *update* und *delete* auf Tupeln aus XNF-Tabellen gibt es noch die Operationen *connect* und *disconnect* auf XNF-Beziehungen. Das Löschen von Projekt p1 in `Sicht_1` führt automatisch zu einem Löschen aller Kanten, die in p1 münden bzw. von p1 ausgehen. In der Folge müssen auch alle diejenigen Tupel und Beziehungen entfernt werden, die nun nicht mehr ausgehend von einem Tupel in einer Wurzel-Tabelle erreichbar sind. Es bleibt unklar, in welchem Umfang dies auf das Löschen von Tupeln der Basisrelationen durchschlagen muß. Hinsichtlich der Propagation von View-Updates bleiben auch die für XNF-Sichten in Abschnitt 2.3.3 diskutierten Probleme weiterhin zutreffend: Soll z.B. das Löschen von Abteilung a1 zu einem Löschen aller Mitarbeiter und Projekten in den Basisrelationen `Mitarb` und `Proj` führen oder sollen lediglich die Fremdschlüsseleinträge für die Abteilungen in den Angestellten- und Projekt-Tupeln auf `Null` gesetzt werden? Ein dem SQL/XNF-Ansatz sehr ähnliches Modell ist das *view object model* aus [BSK+91], für das Regeln zur Update-Propagation analog zu den Vorschlägen von Keller [Kel85, Kel86] für „flache“ Sichten diskutiert werden.

In [MPP+93, MP94] wird behauptet, daß XNF-Sichten beliebige n-stellige Beziehungen unterstützen können. Jedoch wird nicht näher erläutert, was Quelle und Ziel einer gerichteten n-stelligen Beziehung sein soll. Die Idee der gerichteten Beziehung paßt ohne weitere Annahmen nur zu binären Beziehungen.

Genauer betrachtet wird mit der Definition einer XNF-Sicht der Versuch gemacht, den Datenbankentwurf (Transformation eines (konzeptuelles) ER-Schemas in ein (logisches) Relationenschema) teilweise rückgängig zu machen. Dies ist vergleichbar mit dem „Reverse Engineering“, das gerade die Umkehrung des Datenbankentwurfs darstellt und momentan ein sehr aktives Forschungsgebiet ist [CBS94, FV95, HK95]. Trotz der Vorzüge von Starburst, bestehende relationale Anwendungen weiterverwenden zu können, halten wir es für besser, von vornherein ein semantisch reichhaltigeres logisches Datenmodell zugrundezulegen. Der Umweg des „Reverse Engineerings“ ist tatsächlich nur angebracht, wenn kein konzeptuelles Schema (mehr) vorliegt.

Es soll nun noch auf ein weiteres, mit dem SQL/XNF-Ansatz verwandtes Datenmodell eingegangen werden. Es handelt sich um das Molekül-Atom-Datenmodell (MAD) [Mit88], das die Grundlage des DBMS-Prototyps PRIMA ist. Die Grundelemente eines MAD-Schemas sind die sog. *Atomtypen*. Jeder Atomtyp hat genau ein atomares Attribut vom Typ `IDENTIFIER`, das jede Instanz eines Atomtyps durch ein Surrogat eindeutig identifiziert. Alle anderen Attribute können atomar, mengen- oder listenwertig sein. Zwischen den Atomtypen können binäre Beziehungen bestehen. Sie werden über Attribute vom Typ `REF_TO(...)` erfaßt, deren Wertebereich gerade die Menge der `IDENTIFIER`-Werte des referenzierten Atomtyps ist. In den folgenden Atomtyp-Definitionen wird auch die Angestelltenbeziehung, eine 1:n-Beziehung, zwischen Abteilungen und Mitarbeitern erfaßt:

```
CREATE ATOM_TYPE Abt
  ( nr: IDENTIFIER,
    name: CHAR_VAR,
    mitarb: SET_OF(REF_TO(Mitarb.abt))(0, VAR),
    ...
  );
CREATE ATOM_TYPE Mitarb
  ( nr: IDENTIFIER,
    vorname: CHAR_VAR,
    abt: REF_TO(Abt.mitarb),
    ...
  );
```

Das Beispiel zeigt, daß ähnlich wie bei COCOON [SLR+92] bzw. im ODMG’93-Standard [Cat+94] in der Syntax zum Ausdruck kommt, welche Attribute ein Referenz/Gegenreferenz-Paar bilden. `(0, VAR)` drückt eine Kardinalitätsbedingung aus. Die Syntax aus [Mit88] ist etwas unglücklich, da es so aussieht, als ob z.B. `REF_TO(Mitarb.abt)` einen Verweis auf das Attribut `abt` eines `Mitarb`-Atoms bezeichnet. Die „Dot“-Notation dient hier jedoch nur der Kennzeichnung von Referenz/Gegenreferenz-Paaren.

Aufbauend auf einer Menge von Atomtyp-Definitionen und den in ihnen definierten Beziehungen können sog. *Molekültypen* definiert werden. Ein Molekültyp ist das MAD-Pendant zu einer XNF-Sicht und kann genau wie diese über eine Anfrage in der MAD-spezifischen Anfragesprache MQL erzeugt werden. Ein Molekültyp besteht aus genau einem Wurzel- (oder Anker)-Atomtyp, weiteren Komponententypen (Atomtypen oder bereits definierte Molekültypen) und gerichteten Beziehungen (*Links*). Jeder Molekültyp wird durch einen zusammenhängenden und gerichteten Strukturgraphen beschrieben. Er darf genau einen Kreis enthalten

(dann liegt ein *rekursiver* Molekültyp vor), und es muß bei der Definition angegeben werden, welche Beziehung für das Schließen des Kreises verantwortlich ist. Molekültypen sind also wiederum „gerichtete Sichten“ auf das gegebene globale Schema. Ein zum grau unterlegten Graph in Abb. 2.10 analoger Strukturgraph wird in MAD durch

```
Abt-(.mitarb-Mitarb-.arbeitet_an-Proj, .proj-Proj)
```

definiert. Beziehungen werden in dieser Notation durch einen vorangestellten Punkt kenntlich gemacht. Durch Hinzufügen von

```
(RECURSIVE: Proj-.geleitet_von-Mitarb)
```

wird der Molekültyp rekursiv[11]. Die Instanz eines Molekültyps ergibt sich wieder durch Erreichbarkeit ausgehend von den Instanzen des Anker-Atomtyps. Neben Molekültypen, die erst bei Ausführung einer Anfrage instanziiert werden und die somit ein „logisches“ Cluster bilden, können in MAD im Rahmen einer Lastdefinitionssprache auch sog. statische Molekültypen definiert werden, die ein „physisches“ Clustering der beteiligten Atome zur Folge haben.

Da XNF-Beziehungen durch Prädikate definiert werden, kann man z.B. auch eine Beziehung „ist_mindestens_5_Jahre_älter_als“ zwischen Mitarbeitern definieren. Dies ist in MAD nicht möglich, da gerichtete Beziehungen dort immer auf bereits im globalen Schema angegebene binäre Beziehungen zurückgehen müssen. Allerdings hat dies den Vorteil, daß Modifikationen in einem Molekül-Instanzengraph ohne Schwierigkeiten in entsprechende Updates auf den originalen Atomen übersetzt werden können.

2.4.3.2 ECA-Regeln

In Abschnitt 2.1.2 wurde bereits erwähnt, daß Integritätsbedingungen oftmals in Form sog. *Integritätsregeln* formuliert werden. Ein früher Vorschlag für Integritätsregeln im Relationenmodell stammt von Eswaran [Esw76].

Integritätsregeln sind Spezialfälle sog. *ECA-Rules*[12]. Der Begriff „ECA-Regel“ geht auf [DBM88] zurück: Eine ECA-Regel hat die prinzipielle Gestalt

`on` *event* `if` *condition* `do` *action*

Die tatsächliche Syntax mag von System zu System unterschiedlich sein, jedoch bleibt die Grundstruktur „ECA“ immer gleich (und trifft bereits für [Esw76] zu!). Die Semantik einer ECA-Regel läßt sich vereinfacht wie folgt ausdrücken: Tritt das Eregnis *event* ein, dann wird die Bedingung *condition* geprüft. Trifft die Bedingung zu, dann wird die Reaktion *action* ausgeführt. Statt ECA-Rule findet man in der englischsprachigen Literatur auch den Begriff *production rule*. Wir verwenden ab sofort den Begriff „Regel“ immer im Sinne einer ECA-Regel.[13]

In einer (hypothetischen) Syntax kann die Schlüsselbedingung beim Einfügen eines neuen Tupels in die Relation `LESER` aus Beispiel 2.2 durch folgende ECA-Regel überwacht werden:

---

11. Wir zeigen hier nur die einfachste Form der Rekursionsbildung und verweisen ansonsten auf [Mit88].
12. ECA = *E*vent-*C*ondition-*A*ction
13. Regeln in unserem Sinne sind von Regeln abzugrenzen, wie sie in der Gestalt von Horn-Klauseln im Zusammenhang mit logischen bzw. deduktiven Datenbanken (siehe z.B. [Gra84, Das93]) vorkommen.

```
on before insert l to LESER
if EXISTS(SELECT * FROM LESER WHERE LName=l.LName)
do instead printf("Leser existiert bereits!");
```

Vor dem Einfügen wird die angegebene Bedingung getestet, und ggf. wird eine Meldung an den Benutzer ausgegeben. Die aktuelle Transaktion wird nicht abgebrochen, da durch `instead` angezeigt wird, daß die Aktion anstelle der ursprünglichen Operation ausgeführt wird.

Regeln fügen Datenbanksysteme eine *aktive* Komponente hinzu. Als *aktive Datenbanken* werden solche DBMS bezeichnet, die automatisch auf interne oder externe Eregnisse reagieren. Ende der 80er Jahre wuchs das Interesse an einer Nutzbarmachung von ECA-Regeln für Datenbanksysteme, die über die Überwachung von Integritätsbedingungen hinausgeht. Die bereits erwähnten Prototypen POSTGRES und Starburst unterstützen beide jeweils die Definition allgemeiner ECA-Regeln.

Für POSTGRES ist die Regel-Komponente mit ihren Anwendungen in [SJG+90, SK91] beschrieben. Eine POSTGRES-Regel wird mittels

`DEFINE RULE` *Name*
`ON` *Ereignis* [`TO`] *Objekt* [`WHERE` *Bedingung*]
`THEN DO` [`INSTEAD`] *Aktion*

definiert. Als Ereignisse kommen `retrieve`, `replace`, `delete`, `append`, sowie `new` (= `replace` $\vee$ `append`) und `old` (= `delete` $\vee$ `replace`) in Betracht. *Objekt* ist der Name einer Relation oder eine Liste von Attributnamen einer Relation. In Abschnitt 2.3.3 wurde im Zusammenhang mit der Propagation von View-Updates mit (2.1) der „Rumpf“ einer Regel angegeben, die in der POSTGRES-Syntax angegeben ist. Es fehlt dort die Angabe einer Bedingung, d.h. die Reaktion soll immer ausgeführt werden, wenn das Ereignis eintrifft (man könnte (2.1) um `WHERE TRUE` ergänzen).

In Starburst wird eine Regel definiert durch [WCL91]

`CREATE RULE` *Name* `ON` *Relation*
`WHEN` *Transitionsprädikat*
[`IF` *Bedingung* `,`]
`THEN` *Aktion*`,`
[`PRECEDES` *Regel-Liste* `,`]
[`FOLLOWS` *Regel-Liste* `,`]`;`

Das Transitionsprädikat ist eine nichtleere Menge von Ereignissen `inserted`, `deleted` oder `updated` [(*Attribut-Liste*)]. Die Regel muß ausgewertet werden, wenn ein Ereignis des Transitionsprädikates eintritt, d.h. alle Ereignisse sind durch Oder verknüpft. Stehen mehrere Regeln zur Auwertung an, dann muß es ein eindeutiges Kriterium für die Auswahl der nächsten auszuwertenden Regel geben, da sonst ein nichtdeterministisches Verhalten vorliegt. Dieses Kriterium ist durch eine totale Ordnung auf der Menge aller Regeln gegeben. Über die `PRECEDES`- bzw. `FOLLOWS`-Klauseln kann die auf der Basis der zeitlichen Reihenfolge der Regeldefinitionen gegebene Default-Ordnung modifiziert werden [ACL91].

Die Regel-Komponente für Starburst wurde Anfang der 90er Jahre sehr ausführlich dargestellt und untersucht. In [CW90] wird diskutiert, wie deklarativ formulierte Integritätsbedingungen in eine Menge von Regeln übersetzt werden können, welche die Einhaltung der Integritätsbedingungen gewährleisten. In [CW91] geht es um das Problem, auf welche Weise materiali-

sierte relationale Sichten inkrementell aktualisiert werden können, wenn Updates auf den Basisrelationen ausgeführt werden. Es wird gezeigt, daß für eine große Klasse relationaler Sichten eine effiziente inkrementelle Aktualisierung möglich ist, indem die Updates auf den Basisrelationen über Regeln in Updates auf der Sicht umgesetzt werden. In [AWH92] wird auch das schwierige Problem der Terminierung des kaskadierenden Triggerns von Regeln behandelt. Allerdings ist die Frage der Terminierung nach wie vor offen, so daß reale Systeme sich damit behelfen, eine maximale Triggertiefe vorzugeben.

Von Bedeutung sind auch die Zeitpunkte der Auswertung der Bedingung und der Ausführung der Aktion. Man unterscheidet verschiedene *coupling modes*: Wird sofort nach dem Auftreten eines Ereignisses die Bedingung ausgewertet und ggf. die Aktion ausgeführt, dann handelt es sich um den Modus *immediate*. Dies ist der Modus in POSTGRES. Wird dagegen die Auswertung der Bedingung und die Ausführung der Aktion bis zum Commit der aktuellen Transaktion verschoben, dann liegt der Modus *deferred* vor. Dies entspricht dem Vorgehen bei Starburst. Es muß darauf geachtet werden, daß zum Abschluß der Transaktion tatsächlich nur die für den „Netto-Effekt“ relevanten Regeln ausgewertet werden müssen. Löst z.B. das Einfügen eines Tupels die Auswertung einer Regel aus und wird später dasselbe Tupel wieder gelöscht, dann muß diese Regel zum Commit-Zeitpunkt nicht weiter verfolgt werden.

Regeln sind auch in viele objektorientierte DBMS integriert worden [Buc94]. Erwähnt seien HiPAC [DBM88, MD89], Ode [GJ91, GJS92], SAMOS [GGD91, GD95] und NAOS [CCS94]. Alle diese Prototypen geben dem Benutzer die Möglichkeit, zum Definitionszeitpunkt anzugeben, ob es sich um eine *immediate* oder eine *deferred* Regel handelt. In Ode und SAMOS werden auch zusammengesetzte und zeitabhängige Ereignisse ausführlich untersucht. Bei objektorientierten Datenmodellen stellen Methodenaufrufe weitere elementare Ereignisse dar. I.d.R. läßt sich für *immediate*-Regeln auch angeben, ob sie unmittelbar zu Beginn oder unmittelbar nach einem Methodenaufruf ausgewertet werden sollen (z.B. durch die Angabe `before` oder `after` in NAOS).

Regeln stellen ein vielseitig einsetzbares Konzept dar, dessen Nutzen weit über die Konsistenzüberwachung hinausgeht. Anwendungsspezifische Regeln finden als *Trigger* ihre Anwendung z.B. in der Prozeßsteuerung, bei Frühwarnsystemen oder auch im Wertpapierhandel (Auslösung des An- und Verkaufs von Aktien). Mit der Verwendung von Regeln ist ein Wandel des Programmierparadigmas verbunden: Es findet eine Hinwendung zu ereignisorientiertem Denken statt, wie es bei der Entwicklung graphischer Benutzeroberflächen bereits Standard ist. Dort werden an Ereignisse wie Tastendrücke, Mouseclicks usw. Callback-Routinen gebunden, die beim Eintreten der Ereignisse ausgeführt werden. Geht es wie bei ESCHER um eine enge Anbindung einer graphischen Benutzerschnittstelle an die Datenbank, dann ist es nur natürlich, auch ESCHER um eine Regelkomponente zu ergänzen.

2.5 Objektorientierte Datenmodelle und DBMS

2.5.1 Was ist „Objektorientierung“?

Neben dem *evolutionären* Ansatz einer Erweiterung bestehender, relationaler DBMS, wie er im vorangegangenen Abschnitt dargestellt wurde, spielen seit etwa Mitte der 80er Jahre

objektorientierte DBMS (OODBMS) in der Diskussion eine große Rolle. Sie verkörpern den *revolutionären* Ansatz, indem sie weitgehend unabhängig von existierenden DBMS einen „Neuanfang“ für Datenbanksysteme suchen.

Objektorientierte Datenmodelle sind seit langem aus objektorientierten Programmiersprachen (OOPLs) bekannt, deren Historie bis in die 60er Jahre zurückreicht (Simula [DN66]) und die im Software-Engineering immer größere Verbreitung finden. Die wohl mit Abstand populärste OOPL ist derzeit C++ [Str91]. Der Grund für den Erfolg der OOPLs im Software-Engineering ist darin zu sehen, daß objektorientierte Konzepte den Hauptanforderungen des Software-Engineerings nach Verständlichkeit, Modularität, Erweiterbarkeit und Wiederverwendbarkeit (in Form von Klassenbibliotheken) im Vergleich zu anderen Programmiersprachen besser gerecht werden.

Mit dem Einzug der Objektorientierung in die Software-Praxis ist eine Art Paradigmenwechsel verbunden: Währen bisher die prozedurale Beschreibung von Verarbeitungs- oder Geschäftsvorgängen, die passive Daten manipulieren, von zentraler Bedeutung war, steht jetzt die Kommunikation zwischen weitestgehend autonomen Objekten einer Anwendung im Mittelpunkt. Objekte sind dann mit Prozessen vergleichbar, die miteinander über Nachrichten kommunizieren [Bau93].

Es gibt bis heute keine allgemein akzeptierte Einigung über *das* objektorientierte Datenmodell und somit auch nicht über die Definition eines OODBMS. Mit dem „Object-Oriented Database System Manifesto“ [ABD+89] wurde ein Versuch untenommen, aus den existierenden OODBMS-Prototypen gewisse Gemeinsamkeiten herauszufiltern, die als grundlegende Konzepte objektorientierter Datenbanksysteme konsensfähig sind. Als unabdingbar sind nach [ABG+89] für objektorientierte Datenmodelle folgende Punkte:

(i) Unterstützung komplexer Objekte
(ii) Objektidentität
(iii) Einkapselung
(iv) Typen und Klassen
(v) Typ- und Klassenhierarchie
(vi) Overriding (Redefinition), Overloading (Überladen), spätes Binden
(vii) Berechnungsvollständigkeit
(viii) Erweiterbarkeit

Nicht direkt zum Datenmodell, sondern zum charakteristischen Leistungsumfang eines DBMS, gehören die Punkte

(ix) Persistenz
(x) effiziente Sekundärspeicherverwaltung für sehr große Datenmengen
(xi) Mehrbenutzerbetrieb, Nebenläufigkeit
(xii) Recovery
(xiii) Ad-hoc-Anfragen

Analysiert man die genannte Punkte, stellt man fest, daß sie viele der unter den Erweiterungen des Relationenmodells genannten Aspekte ebenfalls aufgreifen. Es werden nun einige der Punkte näher kommentiert.

Komplexe Objekte sollen analog zum eNF^2-Modell durch Anwendung von Konstruktoren aufgebaut werden können.

Die Punkte (ii) bis (iv) fordern die Definition benutzerdefinierter Typen, die jedoch – siehe Punkt (ii) – *Objekt*typen sein sollen, d.h. ihre Instanzen zeichnen sich dadurch aus, daß sie aus einer invarianten Objektidentität und einem veränderbaren Zustand bestehen [KC86, KA90].

Oftmals, vor allem im Zusammenhang mit OOPLs, werden die Begriffe *Typ* und *Klasse* synonym verwendet. In OOPLs hat eine Klasse `C` lediglich die Funktion einer „object factory" [Dit91] („Objektfabrik" [Heu92]), die zwar Objekte eines Typs erzeugt, aber die Menge der von ihr erzeugten und aktuell existierenden Objekte (d.h. die Typextension) nicht verwaltet. Gerade im Zusammenhang mit objektorientierten DBMS geht der Begriff „Klasse" jedoch über die Verwendung als „Objektfabrik" hinaus. Eine Klasse ist in objektorientierten DBMS ein „object warehouse" [Dit91] bzw. „Sammelbehälter" [Heu92] für eine Menge von Objekten. Auf diese Weise werden Anfragen der Gestalt `SELECT ... FROM C ...`, wobei `C` ein Klassenname sei, erst ermöglicht. Man findet zwei Alternativen vor:

- Für jeden Typ gibt es genau eine gleichnamige Klasse, die die gesamte Typextension verwaltet: Eine Klasse verwaltet *alle* Instanzen eines Typs, d.h. die Mitgliedschaft eines Objektes in einer Klasse ergibt sich automatisch aus der Typzugehörigkeit.
- Zu einem Typ können mehrere Klassen definiert werden. Jede Klasse beinhaltet nur eine Teilmenge der Typextension, und die Mitgliedschaft eines Objektes in einer Klasse ist benutzergesteuert.

Bei der ersten Alternative ist die Herkunft aus der semantischen Datenmodellierung unverkennbar. So wird z.B. im Entity-Relationship-Modell mit der Definition eines Entity-Typs die implizite Verwaltung einer Extension als Menge aller Entitäten dieses Typs auf der Instanzenebene verknüpft. Eine andere Gruppierung von Entitäten als über diese Extensionen ist nicht vorgesehen. Einige frühe objektorientierte Datenmodelle, die sich eng an diesem Konzept aus der semantischen Datenmodellierung orientierten, haben ebenfalls diesen Weg gewählt. Für diese Datenmodelle gilt, daß die Begriffe *Typ* und *Klasse* nach wie vor miteinander verschmelzen. Mittlerweile verfolgen viele Datenmodelle eine deutliche Trennung zwischen Typen und Klassen, indem sie sich für die zweite Alternative entscheiden.

Im Sinne abstrakter Objekttypen (AOTs)[14] fordert das Prinzip der *Einkapselung*, daß auf den Zustand eines Objektes nicht direkt zugegriffen werden darf, sondern ausschließlich über eine Menge von Operationen (*Methoden*), die die Schnittstelle des Objekttyps bilden. Nur innerhalb der Implementationen für diese Operationen ist es erlaubt, auf den Zustand von Objekten direkt zuzugreifen. Motivation dieser *strengen* Form der Einkapselung ist es einerseits, die Konsistenz eines Objektes zuzusichern, indem es nur durch Methoden, die *per se* als konsistenzerhaltende Operationen angesehen werden, manipuliert wird. Andererseits kann argumentiert werden, daß der Zustand eines Objektes und die Methodenimplementationen hinsichtlich der 3-Schichten-Architektur zur internen Ebene gehören [Sch93], so daß durch strenge Einkapselung „implementationsspezifische" Information versteckt (*information hiding*) und physische Datenunabhängigkeit erlangt wird. Die strenge Einkapselung war im Datenbankumfeld bisher unbekannt. In [GKP92] wird sogar von einem fundamentalen Unterschied zwischen Datenbank- und OOPL-Prinzipien gesprochen: „Basically, databases are

14. AOTs sind ADTs mit der *Objekt*-Eigenschaft der Instanzen (vgl. [SLR+92]).

about information sharing, whereas object orientation is partly about information hiding, and we need a compromise between these two aims.“ [GKP92, S.133]. Insbesondere für den lesenden Zugriff auf eine Objektkomponente ist es fraglich, ob dieser unbedingt über eine Methode abgekapselt werden muß. So wird auch in [ABD+89] vorgeschlagen, für Ad-hoc-Anfragen von der strengen Einkapselung abzugehen.

In OOPLs ist es vielfach üblich, gewisse Komponenten eines Objekttyps öffentlich zugänglich zu machen, wenn eine direkte Manipulation dieser Komponenten nicht zu Inkonsistenzen führen kann. Dieser groben Einteilung in *public*/*private* stehen in Datenbanksystemen in Form von Authorisierungskonzepten oft subtilere Möglichkeiten zur Verfügung, den direkten Zugriff auf Objektkomponenten oder die Ausführbarkeit von Methoden einzuschränken und dabei zwischen verschiedenen Benutzern oder Benutzergruppen differenzieren zu können [DD93, RBK+91, GGF93]. Als ein weiteres Argument für ein Abrücken von strenger Einkapselung kann angeführt werden, daß die Aufgabe der Konsistenzerhaltung bei Datenbanken zusätzlich von einem Subsystem zur Integritätskontrolle auf der Basis deklarativer Integritätsbedingungen übernommen werden kann. So ist z.B. der Versuch einer nicht zulässigen, direkten Änderung eines Attributwertes unproblematisch, wenn die Unzulässigkeit von der Integritätskontrolle erkannt wird.

Unter *Typ- und Klassenhierarchie* sind die bereits aus den semantischen Datenmodellen bekannten Spezialisierungs- und Generalisierungsbeziehungen zwischen Typen zu verstehen, wobei die meisten objektorientierten Datenmodelle sich auf die Spezialisierung in Verbindung mit Vererbung beschränken und Generalisierung in dem Sinne, wie der Begriff in Abschnitt 2.1.1 eingeführt wurde, nicht direkt unterstützen. Bei einer Verwendung von Klassen als „Sammelbehälter“ kann in einigen Modellen (z.B. EXTREM [Heu92], COCOON [SLR+92] oder das Modell aus [Bru94]) eine von der Typhierarchie unabhängige Klassenhierarchie definiert werden, die eine Teilmengenbeziehung der Klassen ausdrückt.

Zum Stichwort *Erweiterbarkeit* wird in [ABD+89] gefordert, daß kein Unterschied im Umgang mit vom System vorgegebenen und benutzerdefinierten Typen gelten soll. Ferner gehört zur Erweiterbarkeit das inkrementelle Hinzufügen weiterer Operationen.

Die Punkte (vi) und (vii) beziehen sich auf Eigenschaften der zu verwendenden Programmiersprache, über die auf die Datenbank zugegriffen wird. Vorbild sind natürlich die OOPLs. Unter „Overriding“ versteht man die Reimplementation einer Methode für einen Subtyp. Konsequenz davon ist das „Overloading“, nämlich die Tatsache, daß es zu einer Methode mehrere Implementationen gibt, was man auch als Ad-hoc-Polymorphie [CW85] bezeichnet. Spätes Binden ist der Mechanismus, der zur Laufzeit das „Overloading“ auflöst, indem die richtige Methodenimplementation zur Ausführung gebracht wird.

Nicht endgültig geklärt wird im „Manifest“ die Frage, ob ein objektorientiertes Modell volle *Uniformität* („alles ist ein Objekt“, siehe z.B. Cocoon [SLR+92], EXTREM [Heu89, Heu92], TIGUKAT [ÖPS+95]) vertreten sollte, d.h. auch „höhere“ Konzepte wie Typen, Klassen, Methoden sind Objekte. Diese sind dann Instanzen eines *Meta-Schemas*, das wiederum eine Instanz desselben Datenmodells ist, was eine Form der Selbstreferenzierung darstellt.

Es ist nicht zu übersehen, daß objektorientierte Datenmodelle sehr viele Konzepte, die bereits für die semantischen Datenmodelle ausgearbeitet wurden, übernommen haben, wie dies auch in [GKP92] ausführlich dargestellt wird. Der Zugewinn bei objektorientierten Datenmodellen liegt vor allem in der konsequenten Berücksichtigung des *Verhaltensteils*, wie der operationale

Teil eines Datenmodells auch genannt wird. Daß der Verhaltensaspekt im „Manifest“ [ABD+89] nicht als eigener Stichpunkt genannt wird, ist nur damit zu erklären, daß Methoden für OOPLs bereits immer eine Selbstverständlichkeit waren. Heute wird kaum noch von „semantischer“ Datenmodellierung gesprochen, aber ihre Tradition wird unter anderen Namen fortgesetzt: Es gibt für die konzeptuelle Phase eine Vielzahl objektorientierter Modellierungsverfahren. Exemplarisch seien OMT [RBP+94], OOD [Boo91], OSA [EKW92], TROLL [HSJ+94] genannt.

Die Verwendung objektorientierter Datenmodelle als Datenmodell der logischen Ebene eines implementierten DBMS führt zu einer Angleichung des konzeptuellen und des logischen Schemas, so daß der Informationsverlust bei der Transformation des konzeptuellen in das logische Schema vermindert wird.

2.5.2 Objektorientierte DBMS

Als OODBMS bezeichnet man ein Datenbanksystem, dem als logisches Datenmodell ein objektorientiertes Datenmodell zugrundeliegt.

Die ersten „objektorientierten DBMS“ entstanden als Erweiterungen von OOPLs zu *persistenten Programmiersprachen*, deren Errungenschaft oft einfach darin bestand, Objekte in irgendeiner Form persistent speichern zu können. Diese Erweiterungen verdienen nicht das Prädikat „Datenbanksystem“, wenn datenbankspezifische Aspekte – wie z.B. eine deklarative Anfragesprache, Mehrbenutzerzugriff, Transaktionen, Recovery – überhaupt nicht oder nur unzureichend unterstützt werden. In dieser anfänglichen Gleichsetzung objektorientierter persistenter Programmiersprachen mit OODBMS ist wohl auch ein Grund für das auch anzutreffende „Mißtrauen“ in die Funktionalität und Leistungsfähigkeit objektorientierter DBMS zu sehen[15].

Motivation für die Entwicklung objektorientierter DBMS war wiederum die Überwindung des „impedance mismatch“: Die Kluft zwischen den Typsystemen objektorientierter Programmiersprachen und relationaler DBMS ist tatsächlich so groß, daß man von vornherein eine relationale Speicherung von Objekten und den Zugriff über „Embedded SQL“ ausschließen kann. In [Loo94] wird anhand einer Fallstudie aus der Praxis berichtet, welche Schwierigkeiten auftreten, wenn eine objektorientiert implementierte Anwendung mit einer relationalen Datenbank arbeiten soll. In diesem Fall ist es günstiger, wenn dem DBMS selbst bereits ein objektorientiertes Datenmodell zugrundeliegt.

Es gibt heute eine Vielzahl von mit Recht als OODBMS bezeichneten Datenbanksystemen, die auf eine Erweiterung einer existierenden OOPL zu einer objektorientierten DBPL zurückgehen. Auf persistente Erweiterungen von C++ gehen u.a. ObjectStore[LLO+91] und ONTOS [AHS91] zurück. GemStone [MS90] ist ein OODBMS auf der Basis von Smalltalk [GR83]. Neben den Erweiterungen bekannter OOPLs sind auch einige neu entworfene objektorientierte DBPL wie z.B. Galileo [AGO90], Fibonacci [AGO95] oder FAD [DV92] zu erwähnen.

In den letzten Jahren sind aber auch völlige Neuentwicklungen objektorientierter DBMS entstanden, die nicht durch die persistente Erweiterung einer bekannten OOPL motiviert wurden. Diese entstanden zunächst meist als Forschungsprototypen, erfuhren dann aber eine kommer-

15. Darüber hinaus ist „Objektorientierung“ zu einem Modewort geworden [Kin89], was zu einem starken Mißbrauch des Begriffs als immer anwendbares Verkaufsargument führte.

zielle Umsetzung, wie dies auch bei relationalen DBMS der Fall war. Als prominente Vertreter seien O_2 [BDK92], ORION (jetzt Itasca) [BCG+87] und OpenODB [AD92] genannt. Das letztgenannte System ist eine direkte Weiterentwicklung des DBMS Iris [FAC+89], das eine der wenigen Implementationen semantischer Datenmodelle darstellt und Elemente von FDM [Shi81] und TAXIS [NLL+87] in sich vereint. Dieses Beispiel unterstützt die These, daß sich semantische und objektorientierte Datenmodellierung auf nahezu derselben Entwicklungslinie bewegen.

Die Diskussion in Abschnitt 2.4 hat gezeigt, daß einige der aus dem evolutionären Ansatz hervorgegangenen Erweiterungen relationaler DBMS auch eine Vielzahl der im „Manifest“ genannten Konzepte unterstützen, so daß auch einige dieser Systeme bedingt als objektorientiert bezeichnet werden können: Nach [Dit91] sind Modelle, die zumindest komplexe Objekte unterstützen, als *strukturell* objektorientiert einzuordnen.

Der so oft funktionierende „darwinistische“ Verdrängungsmechanismus, von dem in [ABD+89] gesprochen wird und der aus der Vielzahl von Vorschlägen einen oder wenige Sieger hervorbringen sollte, hat für OODBMS noch zu keinem Ergebnis geführt. Festzustellen ist auch, daß – trotz der starken Präsenz objektorientierter Konzepte in der wissenschaftlichen Diskussion – OODBMS einen verschwindend geringen Marktanteil unter den kommerziellen Datenbanken haben [Kim93]. Mit dem ODMG’93-Standard [Cat+94] wird nun aber von der Object Database Management Group (ODMG), einem Zusammenschluß von Herstellern, ein ernsthafter Versuch unternommen, eine Standardisierung voranzutreiben. Der Standard umfaßt die Bereiche Datenmodell, Object Definition Language (ODL), Object Query Language (OQL) sowie Anbindungen an C++ und Smalltalk.

2.5.3 Kritik an objektorientierten DBMS

Der Haupteinwand der Kritiker des revolutionären Ansatzes, der zu objektorientierten DBMS führen soll, lautet: Warum soll man alle Errungenschaften relationaler DBMS über Bord werfen und zu völlig neuen Systemen übergehen? Nur durch allmähliche Erweiterung relationaler DBS kann die Kompatibilität mit bestehenden, auf relationalen Datenbanken basierenden Anwendungen gesichert bleiben. Diese Position wird im „Third Generation Data Base System Manifesto“ [SRL+90] vertreten, das als Antwort auf das „Manifest“ [ABD+89] formuliert wurde. Den Autoren von [ABD+89] wird vorgeworfen, daß sie ihre Aussagen einseitig aus der Perspektive objektorientierter Programmiersprachen machen und die Diskussion wichtiger datenbankspezifischer Funktionalität vernachlässigen. In beiden Manifesten sind aber auch eine Reihe von Gemeinsamkeiten in zentralen Punkten auszumachen: Diese betreffen die Unterstützung von komplexen Objekten, abstrakte Datentypen und Vererbung.

Es sollen nun weitere Gegenpositionen aus [SRL+90] kommentiert werden.

- OIDs sind nach [SRL+90] unerwünscht, da sie keine für den Benutzer verständliche Information tragen und beim Austausch mit anderen Datenbanken unbrauchbar sind. Daraus ziehen die Autoren den Schluß, daß alle ADTs *Wertetypen* sein sollen.

Gleichzeitig wird aber – etwas widersprüchlich – von systemerzeugten, invarianten, künstlichen Schlüsseln (Surrogaten) gesprochen, die notwendig sind, wenn kein Primärschlüssel vorhanden ist. Dabei wird übersehen, daß diese aber gerade eine Implementation der Objektidentität darstellen [WJ91]. Semantische Beziehungen sollen also wieder durch Fremd-

schlüsselverweise ausgedrückt werden – mit den bereits in Abschnitt 2.3.4 geschilderten Nachteilen. In [SRL+90] wird ferner übersehen, daß die wertebasierte Identifikation von Objekten durch den Benutzer natürlich weiterhin eine zentrale Rolle spielen muß. Anfragen lauten eben nicht „Gib mir die Person mit der OID ω“, sondern „Gib mir die Person mit dem Namen …“. Man wird deshalb – auch in OODBs – den meisten benutzerdefinierten Typen auch Schlüssel zuordnen. Aber nicht für alle benutzerdefinierten Typen ist die Möglichkeit einer wertebasierten Identifikation zwingend. Es kann ausreichend sein, Objekte dadurch zu identifizieren, daß sie in einer bestimmten Beziehungskonstellation zu anderen Objekten auftreten, d.h. ihre „Umgebung“ trägt zu ihrer Identifizierung bei. Dies ist eine andere wichtige Form der Objektidentifizierung, auf die auch in [ST93] besonders hingewiesen wird. Dazu sei ein Beispiel genannt: Mehrere Komponenten eines Bauteils werden von Schrauben zusammengehalten. Es ist nicht notwendig, alle gleichartigen Schrauben durch eindeutige „Namen“ auseinanderhalten zu können. Vielmehr interessiert zweierlei: Erstens die Lage der Schraube relativ zu ihrer Umgebung (etwa „2. Bohrung von unten“) und zweitens die Information, daß zwei Bauteile durch *dieselbe* Schraube zusammengehalten werden.

- In [SRL+90] wird kritisiert, daß viele OODBMS über keine deklarative Anfragesprache verfügen, die jedoch als unabdingbar angesehen wird. Daneben sollen alle Zugriffe, also auch Update-Operationen, in deklarativer Form geschehen. Die Autoren von [SRL+90] setzen dabei auf SQL-Erweiterungen.

Tatsächlich steht in vielen frühen OODBMS die *Navigation* durch ein Instanzennetz im Mittelpunkt, wie dies bereits beim Netzwerk-Modell praktiziert wurde. In [ABD+89] wird etwas zurückhaltend und unspezifisch eine Ad-hoc-Anfragesprache gefordert, die „reasonable declarative“ sein müsse. Neuere OODBMS verfügen jedoch über deklarative Anfragesprachen. So ist z.B. OQL, die Anfragesprache des ODMG’93-Standards [Cat+94], eine an der geläufigen SELECT-FROM-WHERE-Syntax orientierte Anfragesprache.

An dieser Stelle soll auch auf die zwei Bedeutungen des Begriffs *Navigation* hingewiesen werden: Die erste Bedeutung bezieht sich auf die Verwendung navigatorischer Sprachelemente in Anwendungsprogrammen. Dazu gehört z.B. das Positionieren von Cursorn in Embedded SQL oder die Manipulation von currency pointers im Netzwerk-Modell. Die andere Bedeutung bezieht sich auf das „freie“ Navigieren in einem komplexen Datenobjekt über eine Benutzerschnittstelle, wie z.B. die graphische Benutzeroberfläche des ESCHER-Prototyps, in der man sich mit Fingern „frei“ in einer Baumstruktur bewegt, um z.B. durch das Ergebnis einer Anfrage zu browsen. Während ein Benutzer sich bei einer Änderung der Struktur einer Tabelle oder eines Anfrageergebnisses „selbständig“ auf die veränderte Situation einstellen kann, ist dies bekanntlich für Anwendungsprogramme, die auf Sprachkonstrukte zum mehr oder weniger freien Navigieren zurückgreifen, nicht der Fall. Wir stimmen daher mit [SRL+90] überein, daß navigatorische Elemente in einer Sprache auf das Notwendigste begrenzt bleiben sollten.

- Laut [SRL+90] muß ein DBMS ein *offenes System* sein, d.h. es muß von mehr als einer höheren Programmiersprache aus zugänglich sein. Der für viele OODBMS zutreffende Ansatz, das DBMS mit einer einzigen objektorientierten DBPL zu identifizieren und somit von dieser abhängig zu machen, hat zur Konsequenz, daß die Datenbank nicht mehr der integrative Mittelpunkt für verschiedene Anwendungen sein kann.
- In [SRL+90] wird darauf hingewiesen, daß die Integration von *Sichten* (*Views*) in ein Datenmodell vorangetrieben werden sollte, um auf diese Weise logische Datenunabhän-

gigkeit zu gewährleisten. Sichten sollten jedoch so weit wie möglich modifizierbar sein, da sonst der Nutzen von Sichten stark eingeschränkt ist. Bekanntlich sind relationale Sichten im Relationenmodell nur in wenigen Fällen modifizierbar (vgl. Abschnitt 2.3.3). Sichten wurden von keinem der frühen OODBMS unterstützt.

In neuester Zeit werden unterschiedliche Sichtenkonzepte für objektorientierte DBMS diskutiert. Zu nennen sind u.a. die Arbeiten [TYI88, SS89b, HZ90, AB91, SLT91, Ber92, Run92, BK93, Sch93, SAD94, San94]. Nach unserer Kenntnis ist O_2 das einzige kommerzielle OODBMS, das ein View-Konzept anbietet [SW93].

Vor kurzem erschien ein drittes „Manifest“ [DD95], das sich ebenfalls kritisch zur Objektorientierung in Datenbanksystemen äußert. Die Aussagen bewegen sich auf der Linie von [SRL+90], jedoch wird noch stärker am ursprünglichen Relationenmodell festgehalten. Die größten Unterschiede zu [SRL+90] liegen in der Ablehnung von NF^2-Strukturen (beliebig komplex strukturierte „skalare“ Typen, vergleichbar mit POSTGRES-ADTs, sind dagegen zugelassen!), in der Ablehnung der Syntax von SQL als Vorbild für eine deklarative Anfragesprache und in einer noch radikaleren Ablehnung navigatorischer Elemente.

Als Ergänzung zu den bereits in [SRL+90] enthaltenen Kritikpunkten wollen wir auf zwei weitere, bereits im einleitenden Kapitel angesprochene Unzulänglichkeiten objektorientierter Datenmodelle hinweisen:

- Bei einem Blick auf die Abstraktionsmechanismen aus Abschnitt 2.1 fällt auf, daß die meisten objektorientierten Datenmodelle die Assoziation nicht direkt unterstützen. Die Assoziation wird dann – ähnlich wie dies bereits für IFO beobachtet wurde – durch die Aggregation simuliert, indem die mit einem Objekt ω in Beziehung stehenden Objekte Komponentenobjekte im Zustand von ω werden. Dies ist eine im wahrsten Sinne des Wortes *objektorientierte* Perspektive, die die Beziehung nicht adäquat wiedergibt.
- In OOPLs werden Objekte als zu einem bestimmten Typ gehörig erzeugt und behalten diesen *Instanziierungstyp* während ihrer Lebenszeit. Ein Typwechsel zu einem spezielleren Typ ist in OOPLs nicht vorgesehen. Die Typzugehörigkeit ist somit *statisch*. Im Kontext von Datenbanken ist jedoch eine dynamische Veränderbarkeit der Typzugehörigkeit wichtig (*dynamische Spezialisierung* [16]), da aufgrund der möglicherweise sehr langen Lebensspanne einzelner Objekte eine statische Typzugehörigkeit nicht garantiert werden kann.

16. In [WJS94] wird im Zusammenhang mit dynamischer Spezialisierung auch von „dynamic subclasses“ gesprochen.

Kapitel 3

Der strukturelle Teil des Datenmodells

Ein wichtiges Anliegen dieser Arbeit soll es sein, das Datenmodell ESCHER$^+$ formal und präzise zu definieren. In diesem Kapitel beginnen wir mit dem Strukturteil von ESCHER$^+$.

Das Datenmodell ESCHER$^+$ ist eine Erweiterung des eNF^2-Datenmodells. Alle fundamentalen Grundbausteine des eNF^2-Datenmodells sind daher auch in ESCHER$^+$ enthalten. Dazu gehört eine a priori existierende Menge von Basistypen, die in Abschnitt 3.1 eingeführt wird. Wir gehen jedoch bereits in Abschnitt 3.1 über die Möglichkeiten des eNF^2-Modells hinaus, indem wir weitere Gruppen von Datentypen zulassen. Von zentraler Bedeutung für unser Datenmodell sind die sog. Objekttypen, die wir auch als benutzerdefinierte Typen bezeichnen. Ihre Instanzen sind Objekte im objektorientierten Sinn, d.h. sie besitzen eine unveränderliche Identität, der ein modifizierbarer Zustand zugeordnet ist. Die Anwendungsobjekte des zu modellierenden Umweltausschnittes werden i.d.R. bereits im konzeptuellen Schema als Objekttypen dargestellt, und dies soll auch im logischen Modell der Regelfall sein. Weitere Gruppen von Datentypen, die in Abschnitt 3.1 besprochen werden, bilden die Aufzählungstypen und die varianten Typen. Letztere sind vergleichbar mit `UNION`-Typen anderer Datenmodelle und ermöglichen es, inhomogene Gruppierungen zu bilden.

In diesem und den folgenden Kapiteln verwenden wir den Begriff *Wert* nur für die Instanzen eines Basistyps, zu denen auch die Aufzählungstypen gerechnet werden. Den Begriff *Objekt* reservieren wir für die Instanzen eines Objekttyps. Als generischen Begriff für eine Instanz eines beliebigen Wertebereiches benutzen wir den Begriff *Datenobjekt.*

Die für das eNF^2-Modell charakteristischen Konstruktoren als weitere fundamentale Bausteine des Datenmodells sind Gegenstand von Abschnitt 3.2. In Abschnitt 3.2.1 motivieren wir die Wahl der von ESCHER$^+$ unterstützten komplexen Wertebereiche. Im Vergleich zum eNF^2-

Datenmodell (vgl. Abb. 2.8) und zu dessen Erweiterungen in [Pau94] berücksichtigt ESCHER$^+$ zusätzlich noch Multimengen und Felder.

In Abschnitt 3.2.2 gehen wir unter formalen Gesichtspunkten auf die Konstruktion komplexer Typen und komplexer Wertebereiche ein. In der Regel sprechen wir nicht von einem komplexen Typ, den ein komplexes Datenobjekt besitzt, sondern von seiner *Struktur*. Durch die Verwendung des Begriffes „Struktur“ soll ein Hinweis darauf gegeben werden, daß ein komplexes Datenobjekt einen „inneren Aufbau“ besitzt. Die formale Einführung von Strukturen und komplexer Wertebereiche ist Voraussetzung für die formale Behandlung varianter Typen und der Semantik von Objekttypen.

In Abschnitt 3.3 wird die Definition varianter Typen gegenüber Abschnitt 3.1.4 weiter präzisiert. Ein varianter Typ besteht danach aus mehreren Alternativen mit jeweils beliebiger Struktur. Die tatsächlich zutreffende Alternative wird durch ein Label angezeigt, das dem „eigentlichen“ Datenobjekt beigefügt ist.

Auf alle Datenobjekte soll immer hinsichtlich eines bestimmten Typs bzw. einer bestimmten Struktur zugegriffen werden. In Abschnitt 3.4 formalisieren wir deshalb den Begriff des getypten Datenobjektes. Dies ist insbesondere für Instanzen von Objekttypen relevant, da beabsichtigt ist, daß diese gleichzeitig Instanzen weiterer Objekttypen sein dürfen. Es muß jedoch für den Zugriff auf ein Objekt klar sein, bzgl. welchen Typs zugegriffen wird.

In Abschnitt 3.5 greifen wird die intuitive und implementationsnahe Vorstellung eines komplexen Datenobjektes (z.B. einer Menge) auf und betrachten es als eine Art strukturiertes „Behälter-Objekt“, dessen Inhalt wiederum „Behälter-Objekte“ sind, was mit einer geschachtelten bzw. hierarchischen Anordnung von Speicherplätzen vergleichbar ist.

Wir führen beschriftete Bäume ein, die Strukturen und Datenobjekte repräsentieren. Getypte Datenobjekte werden durch ein Paar, bestehend aus einem Wertebaum und einem Strukturbaum, repräsentiert. Diese Baumrepräsentationen bilden die Grundlage für die Beschreibung der Semantik des operationalen Teils des Datenmodells, der Gegenstand von Kapitel 4 sein wird. Baumrepräsentationen eignen sich dazu wesentlich besser als eine strenge Interpretation komplexer Datenobjekte als *Werte* im Sinne von [Bee90].

Wichtig ist für ESCHER$^+$ jedoch, daß jederzeit ein Interpretationswechsel zwischen getypten Datenobjekten als *Werten* auf der einen Seite und Baumrepräsentationen auf der anderen Seite möglich ist. Aus diesem Grund gehen wir auch sehr detailliert auf die „Übersetzung“ getypter Datenobjekte in Baumrepräsentationen ein und definieren eine Äquivalenz auf Baumrepräsentationen, die mit dem Gleichheitsprädikat für getypte Datenobjekte übereinstimmt.

In Abschnitt 3.6 geht es um die noch ausstehende strukturelle Semantik von Objekttypen, die vorher als rein abstrakte Objekte ohne einen „beobachtbaren“ oder modifizierbaren Zustand eingeführt wurden. Jedem Objekttyp ist eine Tupelstruktur zugeordnet, die als „Schablone“ für die strukturelle Beschreibung seiner Instanzen dient. Beim Erzeugen einer neuen Instanz eines Objekttyps t wird dem neuen Objekt ω ein initialer Zustand zugeordnet, dessen Struktur gerade durch die „Schablone“ aus der Objekttypdefinition festgelegt ist. Auf formaler Ebene gibt es für jeden Objekttyp t eine Interpretationsfunktion $I(t)$, deren Wertebereich gerade die aktive Domäne von t ist und die mit dem Funktionswert $I(t)(\omega)$ gerade den Zustand von ω bzgl. t angibt. Die Definitionen sind von vornherein so angelegt, daß ein Objekt ω im Definitionsbereich mehrerer solcher Interpretationsfunktionen liegen darf, so daß für Objekte eine multiple Typzugehörigkeit möglich ist.

Die Beschränkung auf Tupelstrukturen, deren Komponenten jedoch beliebig strukturiert sein dürfen, ermöglicht eine einfache Integration einer Subtyp-Beziehung *isa* für Objekttypen in ESCHER$^+$, die in Abschnitt 3.7 eingeführt wird. Damit verbunden ist die Vererbung von Attributen und das Prinzip der Substituierbarkeit für Objekte. Für ESCHER$^+$ soll es ferner eine dynamische Typzugehörigkeit geben, d.h. ein Objekt ω kann während seiner Lebenszeit einen direkten oder indirekten Subtyp einer der Typen, in dessen Wertebereich ω bereits liegt, hinzugewinnen oder abgeben, was wir als dynamische Spezialisierung bezeichnen. Die grundlegenden Prinzipien dazu werden in Abschnitt 3.7.2 behandelt.

In Abschnitt 3.8 werden Tabellen eingeführt. Sie stellen benamte Datenobjekte dar, die i.d.R. Mengen oder Listen weiterer Datenobjekte sind. Der Zugriff auf eine Typextension für Objekttypen unabhängig von einer Tabelle ist in ESCHER$^+$ nicht möglich.

In Abschnitt 3.9 geben wir an, was unter einem ESCHER$^+$-Datenbankschema und einer (zulässigen) Datenbankinstanz zu einem gegebenen Datenbankschema zu verstehen ist. Eine Datenbankinstanz ist im wesentlichen durch eine Menge von Baumrepräsentationen und Interpretationsfunktionen I(t) für Objekttypen und Tabellen gegeben. Da Tabellen die einzigen „Einstiegspunkte“ zur Datenbankinstanz sind, darf eine zulässige Datenbankinstanz auch nur Information enthalten, die von einer Tabelle aus erreichbar ist.

In Abschnitt 3.10 wenden wir uns der Metamodellierung für ESCHER$^+$ zu. Es wird ein spezielles Datenbankschema entwickelt, das sog. *Meta-Schema*. Diese hat die Eigenschaft, daß alle benutzerdefinierten Datenbankschema Instanzen des Meta-Schemas sind. Damit können alle Schemadaten wie gewöhnliche Datenobjekte behandelt werden. Das Meta-Schema besitzt ferner die Eigenschaft der Selbstreferenzierung, denn es beschreibt sich selbst. Auf diese Weise kann das Prinzip der Selbstreferenzierung, das bereits für den ESCHER-Prototyp eine wichtige Rolle [KTW90] spielte, auch für ESCHER$^+$ realisiert werden.

Mit einer Zusammenfassung und Diskussion in Abschnitt 3.11 wird dieses Kapitel abgeschlossen.

In diesem und den folgenden Kapiteln werden einige Notationen und Begriffe aus der Mathematik, insbesondere aus der Graphentheorie, vorausgesetzt, die in Anhang A zusammengestellt sind.

3.1 Typen und einfache Wertebereiche

Die zur Verfügung stehenden „Arten“ oder „Sorten“ von Daten werden durch eine Menge von *Typen* angegeben, denen jeweils ein eindeutiger *Wertebereich* (auch *Domäne* genannt) zugeordnet ist und deren Elemente *Datenobjekte* genannt werden sollen. Für ESCHER$^+$ werden fünf Gruppen von Typen unterschieden:

- *Basistypen* als unzerlegbare Grundbausteine des Datenmodells
- *Aufzählungstypen* als spezielle, benutzerdefinierte Basistypen
- *Objekttypen* als benutzerdefinierte Typen, deren Instanzen Objekte im objektorientierten Sinne sind
- *variante Typen*, deren Instanzen mit einem Label versehene Werte aus einer endlichen Menge von Wertebereichen sind (vergleichbar z.B. mit dem `UNION`-Typ in C)

- *parametrisierte Typen* (*Konstruktoren*), die der Gruppierung und Aggregation von Datenobjekten dienen und deren Instanzen komplexe Datenobjekte sind.

In diesem Abschnitt geht es zunächst um die vier erstgenannten Gruppen, da ihnen direkt ein Wertebereich zugeordnet wird, während Konstruktoren *per se* keinen Wertebereich haben, sondern erst in Abhängigkeit von den aktuellen (Typ-)Parametern einen solchen konstruieren.

Die Menge $\{t_1, \ldots, t_n\}$ aller Typen bezeichnen wir mit TYPES. Es gilt

$$\text{TYPES} = \text{BASETYPES} \cup \text{ENUMTYPES} \cup \text{VARTYPES} \cup \text{DEFTYPES} \cup \text{CONSTR}, \tag{3.1}$$

d.h. TYPES wird gemäß obiger Unterteilung in Teilmengen zerlegt, die paarweise disjunkt sein sollen, d.h. sie partitionieren die Menge TYPES.

Auf TYPES sei eine injektive Funktion

$$\mathit{name} : \text{TYPES} \rightarrow \mathcal{N}^1 \tag{3.2}$$

definiert, die jedem Typ einen eindeutigen Namen zuordnet.

Alle Typen werden im folgenden als „abstrakte Objekte“ t_i eingeführt, die gewisse Eigenschaften haben. Ist mit $T = \{t_1, \ldots, t_n\}$ eine Menge von Typen gegeben, dann sind auf T Funktionen definiert, deren Funktionswerte gerade die Eigenschaften der Typen angeben. Die Funktion *name* ordnet jedem Typ also einen Namen zu. Aufgrund der Injektivität kann für jeden gültigen Typnamen das zugehörige „abstrakte Objekt“ t_i ermittelt werden, d.h. der Name ist Schlüssel für die Menge TYPES. Der Ansatz, Typen als „abstrakte Objekte“ aufzufassen, wird sich als vorteilhaft erweisen, wenn die Typen selbst Instanzen eines Meta-Schemas sein sollen. Darauf wird in Abschnitt 3.10 näher eingegangen.

3.1.1 Basistypen und atomare Werte

Das Fundament jedes Datenmodells bilden a priori gegebene Wertebereiche, die gewissen vordefinierten atomaren Typen (*Basistypen*) zugeordnet sind und deren Elemente nicht weiter zerlegbar sind.

Definition 3.1 (Basistypen)

Es sei *a priori* eine nicht-leere, endliche Menge $\text{BASETYPES} = \{\beta_1, \ldots, \beta_n\} \subseteq \text{TYPES}$ gegeben, $n \in \text{IN}$. Die Elemente von BASETYPES werden *Basistypen* genannt.
Auf BASETYPES sei neben der Funktion *name* aus (3.2) eine Funktion

$$\mathit{dom} : \text{BASETYPES} \rightarrow \mathcal{S}\mathit{et} \tag{3.3}$$

definiert die jedem Basistyp einen *Wertebereich* (eine *Domäne*) zuordnet. Es soll gelten:

$$\forall\, \beta \in \text{BASETYPES}: \mathit{null} \in \mathit{dom}(\beta)$$

d.h. die Wertebereiche der Basistypen haben den Nullwert *null* gemeinsam. ❒

Der Nullwert *null* soll, ähnlich wie in SQL/92 [DD93], in Ausnahmefällen als Platzhalter für existierende, nicht bekannte Information fungieren („unknown-Semantik“) oder aber anzeigen, daß eine Information gar nicht existiert („does not exist“-Semantik).

1. Mit $\mathcal{N}$ bezeichnen wir in dieser Arbeit eine Menge von Namen, wie sie z.B. in Anhang B.1 angegeben ist.

Wir betrachten für ESCHER$^+$ nur die Basistypen `integer`, `boolean` und `string` mit den bekannten Wertebereichen als verbindlich. Dieser Minimalsatz ergibt sich zwingend aus der Forderung nach Abgeschlossenheit des Datenmodells[2]. In Tabelle 3.1 wird ein Beispiel für die Menge BASETYPES angegeben, auf die in dieser Arbeit immer wieder zurückgegriffen wird und die ein „Standardrepertoire“ an Datentypen, wie es etwa aus den verbreiteten Programmiersprachen bekannt ist, inklusive des gerade bestimmten Minimalsatzes abdeckt. Es handelt sich gerade um die implementierten Datentypen des ESCHER-Prototyps.

Basistyp	*name*	*dom*
β_{int}	`integer`	$\mathbb{Z} \cup \{null\}$
$\beta_{boolean}$	`boolean`	$\{true, false, null\}$
β_{string}	`string`	$\Sigma^* \cup \{null\}$
β_{float}	`float`	$\mathbb{R} \cup \{null\}$
β_{char}	`char`	$\Sigma \cup \{null\}$

Tabelle 3.1: Beispiel für eine a priori vorhandene Menge von Basistypen

In Tabelle 3.1 sei Σ eine Menge druckbarer Zeichen. Auf die detaillierte syntaktische Beschreibung der Literaldarstellung der Elemente der atomaren Wertebereiche wird verzichtet. Folgende Beispiele mögen genügen: `4711` $\in$ *dom* (integer)[3], `3.1415` $\in$ *dom* (float), "`Hallo`" $\in$ *dom* (string), '`a`' $\in$ *dom* (char).

3.1.2 Aufzählungstypen

Ein sinnvoller Schritt zur Erweiterbarkeit des Angebots an Basistypen stellt die Möglichkeit dar, auf eine Anwendung zugeschnittene Aufzählungstypen zu definieren. In der Praxis trifft man häufig auf den Fall, daß für ein Attribut nur einige, wenige Werte sinnvoll sind. Bei der Beschreibung von Personen trifft dies etwa für die Unterscheidung nach Geschlecht bzw. nach dem aktuellen Familienstand zu. Es ist in solchen Situationen nicht angebracht, das dafür vorgesehene Attribut als `string`-Attribut zu definieren. Das würde bedeuten, daß bei jeder Zuweisung eines Wertes an dieses Attribut getestet werden muß, ob ein zulässiger String (z.B. "`m`" bzw. "`f`" für das Geschlecht) zugewiesen wird. Beschränkt sich die Auswahl auf nur zwei verschiedene Werte, könnte man auf den Datentyp `boolean` zurückgreifen. Dies ist jedoch ebenfalls eine mehr als unbefriedigende Lösung, da sie die tatsächliche Semantik verschleiert (Ist „männlich“ als `true` oder `false` kodiert?) und spätere Erweiterungen der Auswahl um weitere Werte (was zugegebenermaßen für das Beispiel des Geschlechtes unwahrscheinlich ist) erschwert. Aufzählungstypen sind aus zahlreichen Programmiersprachen, z.B. Pascal oder

2. Man beachte, daß überraschenderweise erst für den SQL3-Standard das Vorhandensein des Datentyps `boolean` zwingend vorgeschrieben ist [DD93]! Somit war das Typsystem von SQL bis dato nicht abgeschlossen, da das Ergebnis der Auswertung von Selektionsbedingungen kein „first-class“-Datenobjekt war.

3. Statt $\beta_{integer}$ schreiben wir im folgenden auch einfach integer. Analoges gilt auch für alle anderen Typen.

C, bekannt. Einige semantische Datenmodelle bieten ebenfalls Aufzählungstypen an, vgl. etwa die *value sets* im EER-Modell aus [MS92].

Für einen Aufzählungstyp müssen Literale für seine Elemente festgelegt werden, die Namen aus $\mathcal{N}$ sein sollen. Für einen Aufzählungstyp `Familienstand` sind beispielsweise `ledig`, `verheiratet`, `geschieden` und `verwitwet` geeignete Literale.

Definition 3.2 (Aufzählungstyp)

Es sei eine Menge ENUMTYPES = $\{\varepsilon_1, \ldots, \varepsilon_n\} \subseteq$ TYPES, $n \in \mathrm{IN}_0$, gegeben. Die Elemente von ENUMTYPES heißen *Aufzählungstypen*. Auf ENUMTYPES seien die Funktionen *name* und *dom* analog zu (3.2) und (3.3) definiert. Für die Wertebereiche der Basis- und Aufzählungstypen gelte:

$$\forall\, t_i, t_j \in \text{BASETYPES} \cup \text{ENUMTYPES} : t_i \neq t_j \Rightarrow \mathit{dom}\,(t_i) \cap \mathit{dom}\,(t_j) = \{\mathit{null}\},$$

d.h. die Wertebereiche verschiedener Typen haben nur den Nullwert *null* gemeinsam.
Ferner sei für jedes $\varepsilon \in$ ENUMTYPES eine injektive Funktion

$$\mathit{lit}\,(\varepsilon) : \mathit{dom}(\varepsilon) \rightarrow \mathcal{N} \tag{3.4}$$

definiert, die jedem $v \in \mathit{dom}(\varepsilon)$ ein Literal zuordnet. ❒

Mit der DDL-Anweisung

```
define enumtype
  -name Familienstand
  -enum ledig,
        verheiratet,
        verwitwet;
end define;
```

wird ein Aufzählungstyp `Familienstand` erzeugt. Sie hat folgende Wirkung: Es wird ein neues Element ε zu ENUMTYPES hinzugefügt. Die `name`-Klausel legt *name* (ε) fest. Außerdem wird ein neuer Wertebereich *dom* (ε) = { $v_1, \ldots, v_{n(\varepsilon)}$, *null*} generiert, wobei $n(\varepsilon)$ die Anzahl der Elemente der `enum`-Klausel ist. Die `enum`-Klausel legt auch die Funktion *lit*(ε) aus (3.4) fest.

Der Wertebereich eines Aufzählungstyps kann über ein die DDL-Anweisung `modify` verändert werden. Über eine `add`-Klausel wird ein Element hinzugefügt. Der oben zunächst „vergessene“ Wert `geschieden` kann später durch

```
modify
  -enumtype Familienstand
  -add  geschieden
end modify;
```

hinzugefügt werden. Ferner ist über die `drop`-Klausel bzw. die `rename`-Klausel (vgl. die Syntax in Anhang B.2) das Löschen eines Elementes aus dem Wertebereich bzw. die Umbenennung eines Literals möglich. Allerdings sind diese Schemamodifikationen problematisch, da sie nicht prüfen, ob das zu löschende Element in einer Datensammlung noch zu finden ist bzw. ob ein Anwendungsprogramm ein nicht mehr gültiges Literal verwendet.

Aufzählungstypen und Basistypen haben die gemeinsame Eigenschaft, daß sie Typen mit einer dem Anwender direkt zugänglichen Darstellung sind („printable types“, vgl. [AH87, HK87]),

indem sie (mindestens) eine Literaldarstellung besitzen. Für die Basistypen haben wir in Def. 3.1 auf die Angabe einer Funktion *lit*(β) im Sinne von (3.4) verzichtet, da „Literaldarstellungen“ für Basistypen weit über ASCII-Repräsentationen hinausgehen können: Für einen Basistyp `audio` ist eine „Literaldarstellung“ etwa durch das Abspielen der Audio-Sequenz über geeignete Ausgabegeräte gegeben.

Die Typen $t \in$ BASETYPES $\cup$ ENUMTYPES nennen wir auch *atomare Typen*, da die Elemente nicht weiter zerlegbar sind[4]. Ihre Wertebereiche *dom* (t) sind dementsprechend die *atomaren Wertebereiche*, ihre Elemente die *atomare Werte* des Datenmodells.

3.1.3 Benutzerdefinierte Typen und Objekte

Objekte im Sinne der Objektorientierung sind im eNF^2-Datenmodell und somit auch im Datenmodell des ESCHER-Prototyps nicht vorgesehen. Wie jedoch bereits in den vorangegangenen Kapiteln deutlich gemacht wurde, ist es vorteilhaft, den Objektbegriff nicht nur für das konzeptuelle Schema, sondern auch für das logische Schema im Zusammenhang mit benutzerdefinierten Typen zur Verfügung zu haben. In [The93] und [Pau94] wurde deshalb vorgeschlagen, Objekttypen in das ESCHER-Datenmodell zu integrieren. Die Instanzen von Objekttypen haben insbesondere eine über ihre Lebensdauer invariante Identität. In diesem Kapitel ist allein die invariante Identität eines Objektes von Interesse. Dem „Zustand“ eines Objektes wenden wir uns später zu. Wir gehen von einer abzählbar unendlichen Menge $\Omega = \{ \omega_1, \omega_2, \ldots \}$ aus, deren Elemente wir *Objektidentifikatoren* (*OIDs*) nennen. Es soll gelten

$$\forall\, t \in \text{BASETYPES} \cup \text{ENUMTYPES}: \Omega \cap \mathit{dom}\,(t) = \emptyset,$$

d.h. die Menge Ω soll disjunkt sein zu den Wertebereichen aller Basis- und Aufzählungstypen. Wir kommen nun zu einer (vorläufigen) Definition von Objekttypen.

Definition 3.3 (benutzerdefinierte Typen, Objekttypen)

Es sei eine Menge DEFTYPES $= \{t_1, \ldots, t_k\} \subseteq$ TYPES gegeben, $k \in \mathbb{N}_0$. Die Elemente von DEFTYPES heißen *benutzerdefinierte Typen* oder auch *Objekttypen*. Auf DEFTYPES sind die Funktionen *name* und *dom* analog zu (3.2) und (3.3) für BASETYPES definiert. Es gelte:

Die Wertebereiche *dom* (t) für $t \in$ DEFTYPES liegen im Unterschied zu den Basis- und Aufzählungstypen nicht a priori fest, sondern sind ebenfalls benutzerdefiniert.

Für alle $t \in$ DEFTYPES gilt zu jedem Zeitpunkt:
Es gibt eine endliche Teilmenge $\Omega_t \subseteq \Omega$, so daß

$$\mathit{dom}\,(t) = \Omega_t \cup \{\mathit{null}\} \tag{3.5}$$

Die Elemente von *dom* (t) für $t \in$ DEFTYPES heißen (*abstrakte*) *Objekte*. ❒

Es sei darauf hingewiesen, daß sich die Wertebereiche verschiedener Objekttypen überlappen dürfen. Eine Disjunktheitsbedingung wie bei den Basistypen gibt es für Objekttypen nicht. Dies ist auch beabsichtigt, um Objekten eine Typzugehörigkeit zu mehreren Typen zu ermöglichen. In jeder Domäne *dom* (t) eines Objekttyps t ist wie bei den Basistypen das generische Nullobjekt *null* enthalten, das wiederum als Platzhalter für ein (noch) unbekanntes Objekt dient.

4. Zuweilen werden Basistypen auch als *Sorten* bezeichnet, vgl. etwa [Gog94].

Diese noch unvollständig erscheinende Einführung benutzerdefinierter Typen läßt abstrakte Objekte zunächst als „atomar“, d.h. als unzerlegbare Einheit erscheinen, genauso wie dies für atomare Werte gilt. In Abschnitt 3.6 wird abstrakten Objekten ihre eigentliche Semantik zugeordnet. Zu einem Objekt gehören ein variabler Zustand und eine Operationenschnittstelle, über die es manipuliert werden kann. Somit wird ein „Hineinschauen“ in ein abstraktes Objekt und dessen kontrollierte Manipulation möglich.

Es ist klar, daß mit Def. 3.3 noch keine Aussage darüber gemacht ist, in welcher Form der Benutzer Objekttypen erzeugen kann, d.h. der Menge DEFTYPES hinzufügt. Ebenso ist noch unklar, was der genaue Wertebereich eines Objekttyps sein soll. In Definition 3.3 wird lediglich gefordert, daß ein Wertebereich aus einer gewissen Teilmenge von Ω und dem Null-Objekt besteht. Auf diese offenen Punkte wird in Abschnitt 3.6 detailliert eingegangen. Es reicht zunächst aus, daß über das Datenbankschema eine Menge DEFTYPES spezifiziert ist, deren Elemente zu jedem Zeitpunkt einen wohldefinierten Wertebereich haben. Wir können daher jeden Objekttyp bis auf weiteres wie einen Basistyp behandeln. Objekttypen werden deshalb bereits an dieser Stelle mit rein „abstrakten“ Instanzen eingefügt, um sie für die Konstruktion komplexer Werte, die im nächsten Abschnitt beschrieben wird, zur Verfügung zu haben.

Die atomaren Domänen und die Objekt-Domänen Ω_t für $t \in$ DEFTYPES fassen wir unter dem Begriff *einfache* Domänen zusammen. Elemente der einfachen Domänen nennen wir *einfache Datenobjekte*.

3.1.4 Variante Typen

In diesem Abschnitt wird die Menge VARTYPES der *varianten Typen* auf zunächst informelle Art eingeführt.

Eine der Hauptaufgaben eines DBMS liegt in der Verwaltung großer Kollektionen von Anwendungsobjekten. In einem auf Typisierung aufbauenden Datenmodell haben Mengen (oder auch andere Kollektionen, wie z.B. Listen) einen bestimmten Elementtyp, was dazu führt, daß lediglich homogene Gruppierungen möglich sind. In einigen Situationen ist man auch an der Gruppierung von Elementen aus endlich vielen, unterschiedlichen Wertebereichen interessiert. In ESCHER$^+$ sollen *variante Typen* definiert werden können, mit denen die Bildung inhomogener Gruppierungen ermöglicht wird, indem man einen varianten Typ als Elementtyp wählt. Für ESCHER$^+$ soll jedoch das Prinzip der Typisierung aufrechterhalten bleiben: Um weiterhin Kontrolle über den Typ der Elemente zu haben, werden die Elemente mit einem *Label* versehen, der die Information über ihren genauen Typ enthält. An die Stelle von Werten treten somit Label-Werte-Paare.

Beispiel 3.1 Abb. 3.1 zeigt ein Beispiel für einen varianten Typ `Verkehrsmittel`: Verschiedene Arten von Verkehrsmitteln werden als unterschiedliche Objekttypen modelliert. Wir verwenden das Symbol $\oplus$, wie es auch in [AH88] zur Darstellung einer Variante verwendet wird. Jede Instanz von `Verkehrsmittel` ist ein geordnetes Paar bestehend aus einem Label (`a`, `s` oder `f`) und einer Instanz einer der Objekttypen `Auto`, `Schiff` oder `Flugzeug`.

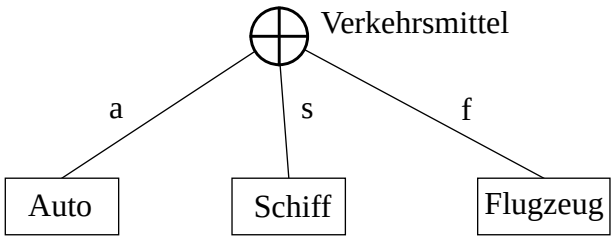


Abbildung 3.1: `Verkehrsmittel` als varianter Typ

Eine Instanz des Typs `Verkehrsmittel` ist dann ein Paar
$(l, \omega) \in \{\mathtt{a}, \mathtt{s}, \mathtt{f}\} \times dom(t_{Auto}) \cup dom(t_{Schiff}) \cup dom(t_{Flugzeug})$. ❒

Sind alle möglichen Alternativen eines varianten Typs durch Objekttypen gegeben, so entspricht der variante Typ offensichtlich einer disjunkten Generalisierung mit Überdeckungseigenschaft, wie sie unter den Abstraktionsmechanismen der semantischen Datenmodellierung (vgl. Abschnitt 2.1.1) beschrieben wurde.

Die möglichen Alternativen eines varianten Typs sollen jedoch nicht nur Objekttypen sein, sondern beliebige komplexe Typen (Strukturen), die formal erst im folgenden Abschnitt eingeführt werden. Deshalb verschieben wir die genaue Definition varianter Typen auf Abschnitt 3.3. Jedem varianten Typ wird dort neben einem Namen als weitere Eigenschaft die eigentliche Typspezifikation in Form einer Menge von (Label, Alternative)-Paaren zugeordnet, woraus sich dann auch der Wertebereich eines varianten Typs ergibt, der wie bei den anderen Typen durch eine auf VARTYPES definierte Funktion *dom* gegeben ist.

3.2 Strukturen und komplexe Wertebereiche

Das Datenmodell ESCHER$^+$ soll das eNF^2-Datenmodell subsumieren. Daher erfolgt in diesem Abschnitt die formale Beschreibung der für das eNF^2-Datenmodell charakteristischen Konstruktion *komplexer Datenobjekte* mit Hilfe von (Typ-)Konstruktoren. Wie in Abschnitt 2.1.3 bei der Besprechung des semantischen Datenmodells IFO deutlich wurde, entsprechen die Konstruktoren des eNF^2-Modells den aus der semantischen Datenmodellierung bekannten Abstraktionsmechanismen Gruppierung und Aggregation. Sie bilden daher ein ebenso grundlegendes „Rüstzeug“ für das Datenmodell wie die bereits eingeführten elementaren Datenobjekte und sind Voraussetzung dafür, daß Datenobjekte zu den sinnvollen komplexen Einheiten „zusammengesetzt“ werden können.

3.2.1 Komplexe Wertebereiche

Während in semantischen Datenmodellen i.d.R. allein die Gruppierung und die Aggregation als Konstruktoren zur Verfügung stehen, wird für das Datenmodell von ESCHER$^+$ ein differen-

zierterer Konstruktorensatz vorgeschlagen. Das Datenmodell soll folgende komplexe Datenobjekte unterstützen:

- die *Menge*
- die *Liste*
- das *Tupel*
- die *Multimenge* (*Bag*)
- das *Feld* (*Array*)

Es folgen nun in knapper Form einige Definitionen im Zusammenhang mit Mengen, Listen und Tupeln, den komplexen Datenobjekten, die bereits aus dem eNF^2-Modell bekannt sind. Danach wenden wir uns den Multimengen und Feldern zu.

3.2.1.1 Die komplexen Wertebereiche des eNF^2-Modells

Das wichtigste Strukturierungskonstrukt, das der Gruppierung in der semantischen Datenmodellierung entspricht, ist die *Menge*. Dabei verstehen wir unter einer Menge[5] über $M \in \mathcal{S}et$ eine *endliche* Teilmenge von M.

Definition 3.4 (Menge)

Eine *Menge* über einer (Grund-)Menge $M \in \mathcal{S}et$ ist ein Element von

$$\mathcal{F}\mathcal{S}et(M) := \{ S \mid S \subseteq M \wedge S \text{ endlich} \}.$$

$\mathcal{F}\mathcal{S}et(M)$ ist die Menge aller endlichen Teilmengen von M. ❒

Listen stellen ein weiteres Mittel zur Gruppierung gleichartiger Datenobjekte dar. Sie liefern zusätzlich noch die Information über eine benutzerdefinierte lineare Ordnung der Elemente innerhalb der Gruppierung.

Definition 3.5 (Liste)

Eine *Liste* über einer (Grund-)Menge $M \in \mathcal{S}et$ ist ein Element aus M*, der Menge der Wörter über dem „Alphabet“ M. Das leere Wort wird auch die *leere Liste* genannt. ❒

Notation: Ist $e_1e_2\ldots e_n$ ein Wort aus M*, dann schreiben wir dieses Wort als Liste i.d.R. in der Form $<e_1, \ldots, e_n>$. Die leere Liste, die dem leeren Wort ε entspricht, wird – konsequenterweise – mit <> bezeichnet. Ist $L = <e_1, \ldots, e_n>$, dann setzen wir $L[k] := e_k$ für $1 \leq k \leq n$, d.h. wir fassen L als Funktion $L : IN \cdots> M$ mit $Def(L) = \{1 , \ldots, n\}$ auf.

Man beachte, daß per definitionem alle Listen endlich sind. Im Unterschied zur Menge darf eine Liste Duplikate enthalten.

Schließlich entspricht der Aggregation der konzeptuellen Modellierung in unserem Datenmodell das *Tupel*. Ein Tupel besteht aus mehreren Komponenten, die auch *Attribute* genannt werden. Wir gehen von einer zunächst beliebigen abzählbaren Menge $\mathcal{A}$ aus, die als Attributvorrat fungiert und deren Elemente sozusagen als Identifikatoren der Tupelkomponenten verwendet werden. Ein Tupel läßt sich dann als Abbildung definieren, die eine Menge von Attributen auf

5. Im Unterschied zu einer beliebigen Menge aus $\mathcal{S}et$, die nicht endlich sein muß.

Werte abbilden. Auf diese Weise wird auch der Begriff des Tupels im Relationenmodell definiert [Vos91, Ull88, EN94, SS83, Dat95].

Definition 3.6 (Tupel)

Ein *Tupel* t ist eine partielle Funktion

$$t : \mathcal{A} \dashrightarrow \bigcup_{M \in \mathcal{S}et} M$$

mit endlichem Definitionsbereich Def(t). Die Elemente der Menge Def(t) werden als *Attribute* des Tupels bezeichnet. Mit $\mathcal{T}up$ bezeichnen wir die Menge aller Tupel t.
Für $\{a_1, \ldots, a_n\} \subseteq \mathcal{A}$, $n \geq 0$, und gegebenen Mengen $M_i \in \mathcal{S}et$ setzt man

$$\mathcal{T}up\,(a_1 : M_1, \ldots, a_n : M_n) :=$$
$$\{\, t \mid t \in \mathcal{T}up \text{ mit } Def(t) = \{a_1, \ldots, a_n\} \wedge t\,(a_i) \in M_i \text{ für } 1 \leq i \leq n\}.$$

❒

Man beachte, daß die angegebene Definition für Tupel zunächst keine Aussage über die genaue Beschaffenheit der Menge $\mathcal{A}$ macht. Ein Tupel kann theoretisch über jede beliebige Attributmenge $\mathcal{A}$ „attributiert“ werden. Damit jedoch für einen Benutzer über eine textuelle oder graphische Schnittstelle der Umgang mit Tupeln ermöglicht wird, ist es sinnvoll, die Menge $\mathcal{A}$ mit der Menge $\mathcal{N}$ von Namen zu identifizieren. In den Beispielen werden wir deshalb durchgängig Namen anstelle der „abstrakten“ a_i gebrauchen.

Notation:

Ist $t \in \mathcal{T}up\,(a_1 : M_1, \ldots, a_n : M_n)$ mit $t\,(a_i) = v_i$ für $1 \leq i \leq n$, dann schreiben wir wie allgemein üblich

$$t = [\, a_1 : v_1, \ldots, a_n : v_n]. \tag{3.6}$$

Dabei wird also vorausgesetzt, daß die Angabe der „Wertebereiche“ M_i, in denen die v_i liegen, aus dem Kontext ersichtlich ist.

Anstelle der funktionalen Notation $t\,(a_i) = v_i$ verwenden wir für den Zugriff auf eine Tupelkomponente die übliche „Dot“-Notation $t.a_i = v_i$. Ist π eine Permutation der Menge $\{1, \ldots, n\}$, dann bezeichnet auch $[a_{\pi(1)} : v_{\pi(1)}, \ldots, a_{\pi(n)} : v_{\pi(n)}]$ dasselbe Tupel t wie in (3.6). Dies unterscheidet die hier angegebene Definition des Tupels von dem aus der Mathematik bekannten Tupelbegriff, der ein Tupel als Element des kartesischen Produktes $M_1 \times \ldots \times M_n$ definiert und wo die Position der v_i wesentlich für die Gleichheit zweier Tupel ist. In der Literatur werden Tupel, wie sie hier definiert wurden, deshalb auch als *labeled cartesian products* bezeichnet [Car84].

Gilt $n = 0$ in Def. 3.6 , so erhalten wir das *leere Tupel* [ ], das im „reinen“ eNF^2-Datenmodell nicht zugelassen ist. Das leere Tupel spielt eine Rolle bei der Definition von Objekttypen, speziell im Zusammenhang mit Vererbung: Ein Objekttyp erbt Teile seiner strukturellen Beschreibung von seinen direkten oder indirekten Supertypen. Über das leere Tupel läßt sich dann ausdrücken, daß ein Objekttyp selbst keinen eigenen Beitrag zu seiner strukturellen Beschreibung hinzufügt, sondern nur seine operationale Schnittstelle (siehe Abschnitt 4.5.2) erweitert.

3.2.1.2 Multimengen

Eine *Multimenge* (*Bag*) B über einer Grundmenge M soll eine Struktur ähnlich einer Menge sein, in der Elemente jedoch mehr als einmal aufgeführt werden dürfen und in der die Anzahl ihres Auftretens signifikant ist für die Gleichheit zweier Multimengen.

Definition 3.7 (Multimenge, Bag)

Eine *Multimenge* (*Bag*) B über einer (Grund-)Menge M ist ein Paar (M, *anzahl*), bestehend aus einer beliebigen Menge $M \in \mathcal{S}et$ und einer Funktion *anzahl*: $M \rightarrow IN_0$.
Die Funktion *anzahl* soll gerade angeben, wie oft ein Element $e \in M$ in der Multimenge B vorkommt. Jede Multimenge soll *endlich sein*, d.h. $\{ e \in M \mid anzahl(e) > 0 \}$ muß endlich sein.
Ist *anzahl* (e) = 0 für alle $e \in M$, dann ist B die *leere Multimenge.* Die Menge aller endlichen Multimengen über der Menge M wird mit $\mathcal{B}ag(M)$ bezeichnet. ❒

Notation: Man schreibt $B = \{* e_1 / a_1, \ldots, e_n / a_n *\}$, wenn B eine endliche Multimenge über einer Menge M mit *anzahl* $(e_i) = a_i$ für $e_i \in \{e_1, \ldots, e_n\} \subseteq M$ und *anzahl* (e) = 0 für $e \in M - \{e_1, \ldots, e_n\}$ ist. Die leere Multimenge wird mit {**} bezeichnet.
Alternativ zur oben angegebenen Notation $B = \{* e_1 / a_1, \ldots, e_n / a_n *\}$ läßt sich B auch in der Form

$$B = \{* \underbrace{e_1, \ldots, e_1}_{a_1\text{-mal}}, \underbrace{e_2, \ldots, e_2}_{a_2\text{-mal}}, \ldots, \underbrace{e_n, \ldots, e_n}_{a_n\text{-mal}} *\}$$

oder auch in einer Permutation dieser Aufzählung angeben. Jedes $e \in B$ wird also genau *anzahl* (e)-fach aufgeführt.

Vielfach werden Multimengen als überflüssiges Konstrukt angesehen und tauchen in vielen Datenmodellen gar nicht auf. Gerade Verfechter eines „puristischen“ Zugangs zur semantischen Datenmodellierung sehen allein in der Menge *das* Mittel zur Gruppierung von Objekten (und lehnen auch die Liste ab!). Es muß aber festgestellt werden, daß Multimengen auf jeden Fall für die adäquate Darstellung der (Zwischen-)Ergebnisse von Anfragen eine wesentliche Rolle spielen. So wird jede formale Beschreibung der SQL-Semantik nicht ohne Multimengen auskommen (vgl. [Gog94]), da eine `SELECT`-Anweisung ohne `DISTINCT` immer eine Multimenge liefert (zudem erzwingt SQL nicht die Mengeneigenschaft von Basistabellen!). Ein weiteres Argument für die Berücksichtigung von Multimengen ergibt sich durch statistische Auswertungen: So ist z.B. für die Bestimmung eines Mittelwertes über einem Anfrageergebnis die Anzahl des Auftretens der Elemente durchaus relevant und eine Duplikateliminierung sogar unerwünscht.

Der Unterschied zwischen Mengen und Multimengen manifestiert sich auf der Seite der Operationen natürlich insbesondere in der jeweils unterschiedlichen Semantik der Einfügeoperation, auf die wir in Abschnitt 4.3.2 noch genau zu sprechen kommen werden.

Leider gelten für Multimengen viele im Zusammenhang mit der Vereinigungs-, Durchschnitts- und Differenzbildung gewohnte algebraische Eigenschaften nicht mehr.
Für $B_1 = (M, anzahl_1) \in \mathcal{B}ag(M)$ und $B_2 = (M, anzahl_2) \in \mathcal{B}ag(M)$ definieren wir

$$B_1 \cup B_2 := (M, anzahl) \text{ mit}$$
$$anzahl(e) := \max(anzahl_1(e), anzahl_2(e)) \text{ für } e \in M, \qquad (3.7)$$

$B_1 \cap B_2 := (M, anzahl)$ mit
$anzahl\ (e) := \min\ (anzahl_1\ (e), anzahl_2\ (e))$ für $e \in M$, (3.8)

$B_1 \setminus B_2 := (M, anzahl)$ mit
$anzahl\ (e) := \max\ (anzahl_1\ (e) - anzahl_2\ (e), 0)$ für $e \in M$. (3.9)

Die Definitionen sind gerade so gewählt, daß gilt: Die Vereinigung zweier Multimengen ist die kleinste Multimenge, die beide Ausgangsmultimengen enthält. Entsprechend ist der Durchschnitt die größte Multimenge, die in beiden Ausgangsmultimengen enthalten ist. Außerdem stimmen die Operationen auf Mengen (als Spezialfall der Multimenge mit *anzahl* (e) $\in \{0, 1\}$) mit den entsprechenden Mengenoperationen überein. In [Alb91] wird u.a. gezeigt, daß keine Komplementbildung für Multimengen angegeben werden kann, die zusammen mit $\cup$ und $\cap$ eine Boolesche Algebra bilden, wenn man gleichzeitig fordert, daß die Operationen angewandt auf Multimengen mit *anzahl* (e) $\in \{0, 1\}$ dieselbe Semantik haben wie die entsprechenden Mengenoperationen. In [Alb91] wird neben der Vereinigung noch eine *bag concatenation* $\cup^*$ angegeben, die *anzahl* durch $anzahl\ (e) := anzahl_1\ (e) + anzahl_2\ (e)$ definiert. Es gilt die Gleichheit $(A \setminus B) \cup^* (A \cap B) = A$, aber nach dem Ersetzen von $\cup^*$ durch $\cup$ wird die Aussage i.a. falsch. Ebenso ist die Aussage $(A \setminus B) \cap (A \cap B) = \{**\}$ für Multimengen i.a. falsch, was sich z.B. mit $A = \{* x, y, y *\}$ und $B = \{* y, z, z, z *\}$ verifizieren läßt. In diesem Fall ist nämlich $A \setminus B = \{* x, y *\}$, $A \cap B = \{* y *\}$ und somit
$(A \setminus B) \cap (A \cap B) = \{* y *\} \neq \{**\}$!

3.2.1.3 Felder

In den meisten Programmiersprachen gibt es die Möglichkeit, Felder (Arrays) zu definieren. Es fällt auf, daß viele Datenmodelle für Datenbanksysteme Felder gar nicht berücksichtigen. Dies erklärt sich aus dem traditionellen Anwendungsbereich für DBMS, für den besonders wichtig ist, eine große und in ihrer Mächtigkeit *variable* Menge gleichartiger Daten verwalten und manipulieren zu können.

Nicht-Standard-Anwendungen sind jedoch häufig im technischen Sektor bzw. im Ingenieur-Bereich angesiedelt, wo mathematische Berechnungen eine große Rolle spielen. Insbesondere benötigt man eine geeignete Modellierung der häufig benötigten Vektoren und Matrizen. Die Verwendung von Listen ist dazu nicht geeignet, da ihre Länge über Einfüge- und Löschoperationen variiert werden kann. Felder sind dagegen vergleichbar mit Listen fester Länge. Der wesentliche Unterschied manifestiert sich bei den Operationen: Für Felder gibt es keine Einfüge- oder Löschoperation, aber eine Änderungsoperation bzgl. einer Feldposition, letztere finden wir wiederum bei Listen nicht. Hinsichtlich der genauen Definition dieser Operationen sei auf Abschnitt 4.3.2 verwiesen.

Es werden nun k-dimensionale Felder definiert.

Definition 3.8 (k-dimensionales Feld)

Für ein $k \in$ IN seien $\{n_1, \ldots, m_1\} \times \ldots \times \{n_k, \ldots, m_k\} \in \mathbb{Z}^k$ mit $n_i \leq m_i$ für $1 \leq i \leq k$ gegeben. Ein *Feld* A über dem Indexbereich $\{n_1, \ldots, m_1\} \times \ldots \times \{n_k, \ldots, m_k\}$ und einer Grundmenge M ist eine Funktion

$$A : \{n_1, \ldots, m_1\} \times \ldots \times \{n_k, \ldots, m_k\} \rightarrow M \quad (3.10)$$

Die Menge aller Felder über einer Grundmenge M werde mit $\mathcal{A}rray(M)$ bezeichnet. Die Menge aller Felder A über einer Grundmenge M und der Indexmenge Def(A) = $\{n_1, \ldots, m_1\} \times \ldots \times \{n_k, \ldots, m_k\}$ werde mit $<M \mid n_1 : m_1, \ldots , n_k : m_k>$ bezeichnet. ❒

Eine lineare Notation für ein k-dimensionales Feld A erhalten wir dadurch, daß wir die Feldelemente gemäß der *lexikographischen* Ordnung der zugehörigen Indizes aus der Indexmenge $\{n_1, \ldots, m_1\} \times \ldots \times \{n_k, \ldots, m_k\}$ anordnen. Die bijektive Abbildung

$$lexord_{Def(A)} : \bigtimes_{i=1}^{k} \{n_i, \ldots, m_i\} \rightarrow \{1, \ldots, p\}, \tag{3.11}$$

ordne jedem Index seine Ordnungszahl bezüglich der lexikographischen Ordnung zu, wobei

$$p = \prod_{i=1}^{k} (m_i - n_i + 1) \tag{3.12}$$

Ist $e_i = A(lexord_{Def(A)}^{-1}(i))$, $1 \leq i \leq p$, dann ist $< e_1, \ldots, e_p \mid n_1 : m_1, \ldots , n_k : m_k>$ die textuelle Notation des Feldes A.

Offensichtlich sind Listen und Felder Spezialfälle von materialisierten Funktionen. Beim Einfügen und Entfernen in einer Liste ändert sich der Definitionsbereich der Liste, wenn diese als Funktion aufgefaßt wird. Ein Feld hat einen konstanten Definitionsbereich, der eine Teilmenge von $\mathbb{Z}^k$ ist. Denkbar wäre auch, einen Konstruktor anzubieten, der materialisierte Funktionen mit beliebigem Definitionsbereich zuläßt. Das OODBMS Ontos [AHS91] bietet mit *Dictionary* einen solchen Konstruktor an. Für ESCHER$^+$ bleiben wir bei den genannten Spezialfällen. Beliebige Funktionen können in ESCHER$^+$ über eine Menge von Tupeln zusammen mit einer Schlüsselbedingung (vgl. Abschnitt 5.3.1) simuliert werden, die sich gerade auf die Tupelkomponenten für die Argumente der Funktion bezieht.

Beispiel 3.2

Eine 3×3-Matrix A, die im $\mathbb{R}^3$ eine Drehung um φ Grad um die z-Achse beschreibt, ist gegeben durch

$$\begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Als Element von $<\mathbb{R} \mid 1 : 3, 1 : 3 >$ hat A die lineare (zeilenorientierte) Darstellung $A = <\cos\varphi, -\sin\varphi, 0, \sin\varphi, \cos\varphi, 0, 0, 0, 1 \mid 1 : 3, 1 : 3>$. Man vergleiche diese Darstellung einer Matrix mit dem Vorschlag aus [KM94], einen Vektor als einen Objekttyp mit der Strukturbeschreibung [x: real, y: real, z: real] zu modellieren. Eine Matrix ist dann wiederum durch einen Objekttyp mit der strukturellen Beschreibung [s1: vector, s2: vector, s3: vector] definiert. In ESCHER$^+$ werden Felder direkt als komplexe *Werte* spezifiziert und brauchen nicht über Objekttypen vergleichsweise umständlich simuliert werden. Zudem wird es in ESCHER$^+$ möglich sein, eine generische Funktion z.B. zur Multiplikation zweier Matrizen zu definieren, die nicht von einer mehr oder weniger zufälligen Wahl der Namen der Komponenten (bei einem Vektor im $\mathbb{R}^3$: `x`, `y`, `z` oder `x1`, `x2`, `x3` usw.) abhängt. ❒

POSTGRES [SK91] bietet zwar einen Array-Konstruktor an, jedoch ist dieser nicht orthogonal zu den anderen Typen und Konstruktoren, sondern Elemente eines Arrays können in Postgres nur Basistypen sein, was allerdings für die häufig vorkommenden Spezialfälle von reellwerti-

gen Matrizen und Vektoren ausreicht. Anders als bei POSTGRES wird für ESCHER$^+$ keine Einschränkung hinsichtlich der erlaubten Feldelemente gemacht. In einigen Modellen verbirgt sich hinter der Bezeichnung „Array“ nichts anderes als eine Liste, wie z.B. in GemStone [MS90]. Objectivity/DB bietet das sog. *VArray* an, ein eindimensionales Feld, dessen Anfangslänge festliegt, dessen Länge jedoch implizit durch Zugriff auf Feldkomponenten jenseits der aktuellen Feldobergrenze verändert werden kann. Interessanterweise stand das *VArray* offensichtlich Pate bei der Einführung des Array-Konstruktors in den ODMG’93-Standard [Cat+94]. Neben dem Zugriff auf eine Feldkomponente jenseits der aktuellen Feldobergrenze erlaubt der Standard auch, die Länge explizit über eine *resize*-Operation zu verändern.

3.2.2 Strukturen und Typausdrücke

Im vorangegangenen Abschnitt wurden komplexe Wertebereiche definiert, deren Konstruktion von einer Grundmenge M (bzw. mehreren Grundmengen M_i beim Tupel) ausging. Für M bzw. M_i können zunächst die elementaren Wertebereiche eingesetzt werden. Die so entstehenden Mengen können wiederum als Grundmengen bei einer Konstruktion fungieren, wodurch erneut komplexe Wertebereiche entstehen. Dieser Prozeß läßt sich beliebig fortsetzen, wobei bei den Konstruktionen keine weiteren Voraussetzungen an die Beschaffenheit der Grundmengen M bzw. M_i gestellt werden. Dies bezeichnet man als *Orthogonalität* in der Konstruktion. Parallel zur Konstruktion komplexer Wertebereiche sollen auf Schemaebene *komplexe Typen* konstruiert werden können. Dazu führen wir die Menge

$$\text{CONSTR} := \{c_{set}, c_{bag}, c_{list}, c_{tuple}, c_{array}\} \tag{3.13}$$

als Menge *parametrisierter Typen* (*Konstruktoren*) ein[6]. Auf CONSTR definieren wir die Funktion *name* aus (3.2), indem wir den jeweiligen Index aus der Menge in (3.13) als Namen verwenden. Es wurden nun alle zu Beginn dieses Abschnitts genannten Teilmengen der Menge TYPES aller Typen eingeführt.

In der nächsten Definition wird angegeben, welche komplexen Typen, die wir auch *Strukturen* nennen, auf der Grundlage der Menge TYPES konstruiert werden können. Jeder Konstruktor aus CONSTR wird dabei als Operator auf bereits konstruierten Strukturen sowie ggf. weiteren Parametern (bei Tupeln und Feldern) aufgefaßt.

Definition 3.9 (Strukturen)

Es sei $\text{CONSTR} = \{c_{set}, c_{bag}, c_{list}, c_{tuple}, c_{array}\} \subseteq \text{TYPES}$ die gegebene Menge von parametrisierten Typen (Konstruktoren).

Es sei $S_0 := \text{BASETYPES} \cup \text{ENUMTYPES} \cup \text{VARTYPES} \cup \text{DEFTYPES}$.
Ferner seien $s, s_1, \ldots, s_k \in \bigcup_{0 \le j \le i} S_j$. Dann ergibt sich S_{i+1} wie folgt:

(i) $c_{set}(s) \in S_{i+1}$

(ii) $c_{bag}(s) \in S_{i+1}$

(iii) $c_{list}(s) \in S_{i+1}$

6. Im Gegensatz zu dem bei den Basistypen eingeschlagenen Weg, wo bis auf einen Mindestsatz von Datentypen der Inhalt der Menge BASETYPES nicht näher spezifiziert sein muß, soll also hinsichtlich der Konstruktoren genau festgelegt sein, welche Konstruktoren in ESCHER$^+$ zur Verfügung stehen.

(iv) $c_{tuple}((a_1, s_1), \ldots, (a_k, s_k)) \in S_{i+1}$,
mit $a_i \in \mathcal{A}$ paarweise verschieden für $0 \le i \le k$

(v) $c_{array}(s, (n_1, m_1), \ldots, (n_k, m_k)) \in S_{i+1}$,
mit $n_i, m_i \in \mathbb{Z}$, $n_i \le m_i$ für $1 \le i \le k$, $k \in \mathrm{IN}_0$.

(vi) nichts anderes liegt in S_{i+1}

Es ist dann $\mathcal{S}(\mathrm{TYPES}) := \bigcup_{i \ge 0} S_i$ die Menge aller *Strukturen* über TYPES. Die Strukturen aus $\mathcal{S}(\mathrm{TYPES}) - S_0$ heißen *komplexe* Strukturen. ❒

Genauso, wie jedem Typ aus TYPES ein ihn identifizierender Name zugeordnet ist, soll auch für jede Struktur ein Literal existieren, das diese Struktur beschreibt. Wir setzen daher die Abbildung *name* auf $\mathcal{S}$(TYPES) fort.

Definition 3.10 (Typausdrücke)

Die Funktion *name*: $\mathcal{S}(\mathrm{TYPES}) \to \Sigma^*$ ist definiert durch

(i) *name* (s) := *name* (t), falls $s = t \in$ TYPES – CONSTR

(ii) *name* (c_{set} (s)) := { *name* (s) }

(iii) *name* (c_{bag} (s)) := {* *name* (s) *}

(iv) *name* (c_{list} (s)) := < *name* (s) >

(v) *name* ($c_{tuple}((a_1, s_1), \ldots, (a_k, s_k))$) := [a_1: *name* (s_1), …, a_k: *name* (s_k)],
mit $a_i \in \mathcal{A}$ paarweise verschieden für $1 \le i \le k$

(vi) *name* ($c_{array}(s, (n_1, m_1), \ldots, (n_k, m_k))$) := < *name* (s) | $n_1 : m_1, \ldots, n_k : m_k$ >,
mit $n_i, m_i \in \mathbb{Z}$, $n_i \le m_i$ für $1 \le i \le k$, $k \in \mathrm{IN}_0$.

Für $s \in \mathcal{S}$(TYPES) nennen wir *name* (s) den zu s gehörenden *Typausdruck*. ❒

Notation: Für die in technischen Anwendungen häufiger anzutreffenden Spezialfälle eines reellwertigen Vektors bzw. einer reellwertigen Matrix setzen wir als syntaktischen Zucker

`Vector(n)` := `< float| 1 : n >`

`Matrix(n, m)` := `< float | 1 : n, 1 : m >`

Die Definition 3.10 läßt sich offensichtlich auch als Sammlung von Grammatikregeln zur Erzeugung von Typausdrücken auffassen. Die Menge aller Typausdrücke über TYPES sei die ausgehend von dem Nichtterminal *struct* erzeugte Sprache $\mathcal{L}$(*struct*) (vgl. auch das gleichnamige Nichtterminal in Anhang B.2).

Beispiel 3.3

Es soll eine Liste von Schülern zusammen mit ihren jeweiligen Zeugnissen angegeben werden. Ein Zeugnis ist eine Menge von Fach-Note-Kombinationen. Die dazu passende Struktur aus $\mathcal{S}$(TYPES) ist gegeben durch

$$c_{list}(c_{tuple}((\mathrm{Name}, \beta_{string}), (\mathrm{Zeugnis}, c_{set}(c_{tuple}((\mathrm{Fach}, \beta_{string}), (\mathrm{Note}, \beta_{integer}))))))) \qquad (3.14)$$

Der zugehörige Typausdruck lautet

```
<[Name: string, Zeugnis: {[Fach: string, Note: integer]}]>
```

Im folgenden wird nicht mehr streng zwischen einer Struktur aus $\mathcal{S}$(TYPES) und dem „lesbareren“ Typausdruck unterschieden. ❒

Wir schaffen nun die Verbindung zwischen den Strukturen und den komplexen Wertebereichen, indem wir die Abbildung *dom* : TYPES – CONSTR → $\mathcal{S}et$ zu einer Abbildung

$$dom : \mathcal{S}(\text{TYPES}) \rightarrow \mathcal{S}et \tag{3.15}$$

fortsetzen. Dazu wird lediglich der durch die Konstruktoren aus CONSTR vorgegebene rekursive Aufbau einer Struktur s nachverfolgt.

Definition 3.11 (Wertebereich von Strukturen)

Die Funktion *dom* aus (3.15) wird wie folgt rekursiv definiert:

(i) *dom* (s) = *dom* (t), falls s = t ∈ TYPES – CONSTR

(ii) *dom* (c_{set} (s)) = $\mathcal{FS}et$ (*dom* (s))

(iii) *dom* (c_{bag} (s)) = $\mathcal{B}ag$ (*dom* (s))

(iv) *dom* (c_{list} (s)) = (*dom* (s))*

(v) *dom* (c_{tuple} ((a_1, s_1), …, (a_k, s_k))) = $\mathcal{T}up$ (a_1: *dom* (s_1), …, a_k: *dom* (s_k))

(vi) *dom* (c_{array} (s, (n_1, m_1), …, (n_k, m_k))) = <*dom* (s') | n_1 : m_1, …, n_k : m_k>

Die Menge *dom* (s) heißt *Wertebereich* der Struktur s. Ist s ∉ TYPES – CONSTR, dann ist *dom*(s) ein *komplexer* Wertebereich.

Wir setzen $\mathcal{V}al$ (TYPES) := $\bigcup_{s \in \mathcal{S}(\text{TYPES})}$ *dom* (s).

Ferner sei $\mathcal{V}al_0$(TYPES) := $\bigcup_{s \in \text{BASETYPES} \cup \text{DEFTYPES}}$ *dom* (s).

Somit ist $\mathcal{V}al$ (TYPES) – $\mathcal{V}al_0$ (TYPES) die Menge aller *komplexen Datenobjekte*, die über der Menge TYPES gebildet werden können. ❒

In der nachfolgenden Definition werden einige nützliche Prädikate und Funktionen auf Strukturen definiert, die Strukturen als Parameter haben und unerläßlich zur Durchführung von Typprüfungen und Laufzeit-Checks sind.

Definition 3.12 (Prädikate und Funktionen auf Strukturen)

(i) Mengen, Multimengen, Listen und Felder werden auch als *Kollektionen* bezeichnet. Wir definieren daher
isCollection (s) :⇔ s = { s' } ∨ s = {* s' *} ∨ s = < s' > ∨ s = < s' | n_1 : m_1, … >,
Ferner sei
isSimple (s) :⇔ s ∈ BASETYPES ∪ ENUMTYPES ∪ DEFTYPES

(ii) Für Kollektionstypen liefert die Funktion *substruct* : $\mathcal{S}$ (TYPES) ···> $\mathcal{S}$ (TYPES) die *Unterstruktur* einer gegebenen Struktur:
Def(*substruct*) = { s ∈ $\mathcal{S}$(TYPES) | *isCollection* (s) } und
substruct (s) = s', falls s = { s' } ∨ s = {* s' *} ∨ s = < s' > ∨ s = < s' | n_1 : m_1, … >

(iii) Es seien s_1 = [a_{11}: s_{11}, …, a_{1k}: s_{1k}] und s_2 = [a_{21}: s_{21}, …, a_{2l}: s_{2l}] zwei Tupel-Strukturen. Wir definieren
$s_1 \leq_{tup} s_2 \Leftrightarrow$
∃ eine Menge {i_1, …, i_l } ⊆ {1, …, k} mit (a_{1i_ν}, s_{1i_ν}) = ($a_{2\nu}$, $s_{2\nu}$) für ν∈ {1, …, *l*}.
Das bedeutet gerade, daß s_1 alle Attribut/Struktur-Paare von s_2 enthält.

❒

3.3 Die Semantik varianter Typen

Wir kommen nun noch einmal auf die varianten Typen zurück, die bereits in Abschnitt 3.1.4 informell eingeführt wurden. Es steht noch die Spezifikation der Labels und Alternativen eines varianten Typs aus, die seinen Wertebereich bestimmen. Als Alternativen einer Variante kommen alle Strukturen s ∈ $\mathcal{S}$(TYPES) in Frage. Außerdem gehen wir von einer nicht näher spezifizierten Menge $\mathcal{L}$ von Labels aus.

Definition 3.13 (variante Typen)

Es sei VARTYPES = $\{t_1, \ldots, t_k\} \subseteq$ TYPES, k ∈ IN_0, gegeben. Die Elemente von VARTYPES heißen *variante Typen*. Auf VARTYPES sei eine Abbildung

$$variants: \text{VARTYPES} \rightarrow \mathcal{FS}et\,(\mathcal{L} \times \mathcal{S}(\text{TYPES}))$$

gegeben mit der Eigenschaft

$$\forall\, t \in \text{VARTYPES}: variants(t) = \{(l_1, s_1), \ldots, (l_n, s_n)\} \Rightarrow l_i \neq l_j \text{ und } s_i \neq s_j \text{ für } i \neq j$$

Ferner sei die Abbildung *dom* : VARTYPES → $\mathcal{S}et$, die den Wertebereich des varianten Typs angibt, definiert durch

$$dom\,(t) := \{\, (l, v) \mid (l, s) \in variants(t), v \in dom\,(s)\,\} \cup \{null\}, \tag{3.16}$$

Die Elemente aus *dom* (t) für t ∈ VARTYPES heißen *Varianten*.

Ist (*l*, v) eine Variante, dann ist die Komponente *l* das *Label* der Variante, während v ihren Wert darstellt. Ist (*l*, s) ∈ *variants*(t), dann nennen wir s die *Alternative* zum Label *l*.

Für t ∈ VARTYPES sei *labels*(t) := {*l* | (*l*, s) ∈ *variants*(t)}.

Falls für eine Variante nicht bekannt ist, welche Alternative für sie zutrifft, soll sie den Wert *null* haben.

Als *Objekttypvariante* bezeichnen wir einen varianten Typ, bei dem alle Alternativen Objekttypen oder andere Objekttypvarianten sind. Die Teilmenge von VARTYPES, die aus allen Objekttypvarianten besteht, bezeichnen wir mit OBJVAR. ❒

Mit varianten Typen vergleichbare Konzepte gibt es in vielen Datenmodellen. Aus der Programmiersprache C sind beispielsweise die `UNION`-Typen bekannt. In [Pau94] wurde für ESCHER ein Variantenkonstruktor vorgeschlagen. Dieser ist orthogonal zu allen anderen Konstruktoren einsetzbar. Vergleichbare Varianten- oder *„discriminated union"*-Konstruktoren findet man u.a. auch in der DBPL Nuovo Galileo [AGO90], in dessen Nachfolger Fibonacci [AGO95], in FAD [DV92] (dort *disjunct* genannt) und in IFO_2 [PTC+93].

In allen diesen Datenmodellen können variante Strukturen definiert werden, die jedoch nicht benamt sind. In [Pau94] ist etwa die Definition der Struktur

```
{VAR(a: Auto, s: Schiff, f: Flugzeug)}
```

möglich (vgl. Beispiel 3.1), die eine Menge von Varianten der angegebenen Alternativen bezeichnet. Nachteilig ist dabei, daß die gesamte Variantenspezifikation `VAR(...)` an allen relevanten Stellen textuell wiederholt werden muß, falls eine Variante mit denselben Alternativen auch an anderer Stelle benötigt wird. Zudem ist in keinem der bisherigen Vorschläge das nachträgliche Hinzufügen weiterer Alternativen vorgesehen. Die genannten Nachteile können

vermieden werden, indem wie in unserem Datenmodell Variantenspezifikationen *benamt* werden.

Für die Spezifikation eines varianten Typs steht die DDL-Anweisung `define variant` zur Verfügung. Der variante Typ `Verkehrsmittel` aus Beispiel 3.1 läßt sich wie folgt spezifizieren:

```
define variant
  -name     Verkehrsmittel
  -variants a: Auto, f: Flugzeug, s: Schiff
end define;
```

Falls es sich bei den Alternativen ausschließlich um Objekttypen handelt, dann gibt es für die `variants`-Klausel auch die Möglichkeit, die Labels wegzulassen. In diesem Fall sind die Labels implizit durch die Typnamen gegeben. Im obigen Beispiel ist somit auch

```
  -variants Auto, Flugzeug, Schiff
```

möglich. Das nachträgliche Hinzufügen einer weiteren Alternative erfolgt über eine `modify`-Anweisung:

```
modify
  -variant  Verkehrsmittel
  -add      Bahn
end modify;
```

Für das Löschen einer Alternative ist die `modify`-Anweisung mit einer `drop`-Klausel vorgesehen.

Es müssen natürlich geeignete Sprachmittel zur Verfügung stehen, um eine Instanz eines varianten Typs auf sein Label und somit auf seine „tatsächliche“ Struktur zu inspizieren und einen „type cast“ auf diese Struktur durchzuführen. Wir verweisen dazu auf die Ausführungen in Abschnitt 4.7.4.

Mit den Objekttypvarianten haben wir nun auch ein Konstrukt gefunden, um die in Abschnitt 2.1.1 erwähnten *disjunkten Generalisierungen* in ESCHER$^+$ in geeigneter Form modellieren zu können.

Es soll nun noch ein Beispiel für die Verwendung eines varianten Typs gegeben werden, der keine Objekttypvariante ist, d.h. dessen Alternativen durch Strukturen $s \in \mathcal{S}(\text{TYPES}) - \text{DEFTYPES}$ gegeben sind.

Beispiel 3.4 Für die Erfassung einer Adresse wählen wir die Tupelstruktur

```
[PLZ: integer, Ort: string, Details: AdrVar]
```
(3.17)

Dabei ist `AdrVar` ein varianter Typ mit den Alternativen

(`postf`, `integer`) für das Postfach und
(`str`, `[Strasse: string, HNr: integer]`) für die Straße

Die in (3.17) angegebene Tupelstruktur ist mit einem varianten Record, wie er in der Programmiersprache Pascal definiert werden kann, vergleichbar. ❐

Mit den varianten Typen steht in ESCHER$^+$ ein Konstrukt zur Verfügung, mit dem auf einheitliche Weise sowohl die sog. disjunkte Generalisierung von Objekttypen als auch sonstige alternative Strukturen modelliert werden können.

3.4 Getypte Datenobjekte

Im ESCHER$^+$-Datenmodell sind grundsätzlich nur getypte Datenobjekte die Gegenstände unserer Betrachtung, d.h. zu jedem Datenobjekt soll jederzeit der Typ verfügbar sein, unter dem auf ein Datenobjekt zugegriffen wird. Dies wird in folgender Definition formalisiert.

Definition 3.14 (getypte Datenobjekte)
Ein Paar (v, s) $\in$ $\mathcal{V}al$ (TYPES) $\times$ $\mathcal{S}$ (TYPES) mit der Eigenschaft v $\in$ *dom* (s) heißt *getyptes Datenobjekt*. Die Menge aller getypten Datenobjekte wird mit $\mathcal{V}al^*$(TYPES) bezeichnet. ❒

Es ist notwendig, Datenobjekte mit Strukturinformation zu paaren, da i.a. aus einem Wert v nicht auf dessen Struktur geschlossen werden kann:

- Das Datenobjekt *null* liegt im Durchschnitt der Domänen aller Typen, ist also *generisch*. Soll durch eine Update-Operation das Datenobjekt *null* durch ein anderes Datenobjekt ersetzt werden, dann muß angegeben werden, von welchem Typ das neue Datenobjekt ist.
- Die leeren Kollektionen {}, {**} und <> benötigen Information über die Struktur ihrer Elemente, um das erste Element typgerecht einfügen zu können.
- Objekte $\omega \in \Omega$ sollen gleichzeitig im Wertebereich von mehr als einem Objekttyp liegen können. Der Zugriff soll aber immer relativ zu einem bestimmten Objekttyp t erfolgen. Dies kann nur geschehen, indem ω mit einem Objekttyp t gepaart wird.

3.5 Repräsentation von Datenobjekten durch Bäume

3.5.1 Motivation

Es stellt sich die Frage, wie die Semantik von Update-Operationen auf komplexen Datenobjekten formal erfaßt werden kann. Dies hängt maßgeblich von der Sichtweise auf komplexe Datenobjekte ab, wobei es nach [Bee90] zwei Alternativen gibt.

Die erste Sichtweise betrachtet komplexe Datenobjekte als *Werte*. In [Cat+94] wird dies als *extensionale* Semantik komplexer Datenobjekte bezeichnet. Da für Werte charakteristisch ist, daß bei ihnen Identität und Zustand zusammenfallen, ist ein komplexes Datenobjekt in dieser Interpretation auch nicht „veränderbar“ in dem Sinne, daß es Operationen gibt, die Teile des Zustandes modifizieren können, ohne die Identität des komplexen Datenobjektes zu verändern.

Betrachten wir dazu ein Beispiel: In dem komplexen Wert {[a: "Hans", b: 3]}, der an den Namen `aSet` gebunden sei, soll ein weiteres Tupel [a: "Gretel", b: 4] eingefügt werden. Dazu steht die Funktion *insert* : $\mathcal{FS}et$ (M) $\times$ M $\rightarrow$ $\mathcal{FS}et$ (M) zur Verfügung, die durch

$$\textit{insert}\ (S, e) := \begin{cases} S & \text{, falls } e \in S \\ S \cup \{e\} & \text{, falls } e \notin S \end{cases}$$

definiert sei (wobei M eine beliebige Menge sei).
Der Funktionswert *insert*(`aSet`, [a: "Gretel", b: 4]) ist der komplexe Wert

{[a: "Hans", b: 3], [a: "Gretel", b: 4]}, jedoch ist `aSet` nach wie vor an {[a: "Hans", b: 3]} gebunden, d.h. die Funktion *insert* hat keinen „Nebeneffekt“. Erst durch eine zusätzliche Modifikation der Bindung für `aSet` erreichen wir das gewünschte Update. Diese Interpretation spiegelt sich z.B. in der Syntax und Semantik der Spache SCRIPT aus [Pau94] wieder, wo für das entsprechende Update

```
aSet := aSet add [a: "Gretel", b: 4]
```

geschrieben werden muß. Soll anschließend die b-Komponente des gerade eingefügten Tupels von 4 auf 2 modifiziert werden, so muß dies unter der extensionalen Semantik durch Binden des Resultates von *insert* (*remove*(S, [a: "Gretel", b: 4]), [a: "Gretel", b: 2])[7] an `aSet` durchgeführt werden. Bei einem großen, stark strukturierten komplexen Wert würde eine noch so kleine Modifikation im „Inneren“ zumindest auf der logischen Ebene zu der recht absurden Situation führen, daß der gesamte komplexe Wert durch einen neuen komplexen Wert „ausgetauscht“ wird – eine mögliche, aber sicherlich nicht adäquate Vorstellung, die auch in keinem Verhältnis zu einer anzustrebenden Lokalität des Updates steht.

Aus diesem Grund betrachtet die zweite Sichtweise ein komplexes Datenobjekt als ein „Behälter-Objekt“, auf dessen Bestandteile (Komponenten, Elemente) gezielt zugegriffen werden kann und die auch modifiziert werden können, ohne die Identität des umgebenden „Behälter-Objektes“ zu verändern. In [Cat+94] wird dies als *intensionale* Semantik bezeichnet. Diese Sicht auf komplexe Datenobjekte erklärt auch, wieso für sie in der Literatur auch die Bezeichnung komplexe *Objekte* zu finden ist.

Die intuitive (und implementationsnahe!) Vorstellung einer Menge ist die eines „Behälters“ oder „Containers“, der „Plätze“ für ihre Elemente enthält. In Abb. 3.2 wird die intensionale Semantik für das Einfügen in Mengen anhand einer Baumdarstellung veranschaulicht. Der Knoten r_1 entspricht dem „Container“ für die Menge. Die Sohnknoten sind die „Plätze“ in r_1. Das Einfügen eines neuen Elementes bedeutet dann, daß in dem „Container“ ein neuer „Platz“ für das neue Element bereitgestellt wird. Dies ist der Effekt einer Prozedur `insert`, deren Argumente gerade die „Identitäten“ der relevanten Knoten sind. Ebenso läßt sich das Update einer Tupelkomponente (z.B. durch Modifikation des Inhaltes des Knotens r_3) durchführen, ohne daß davon andere Knoten betroffen wären.

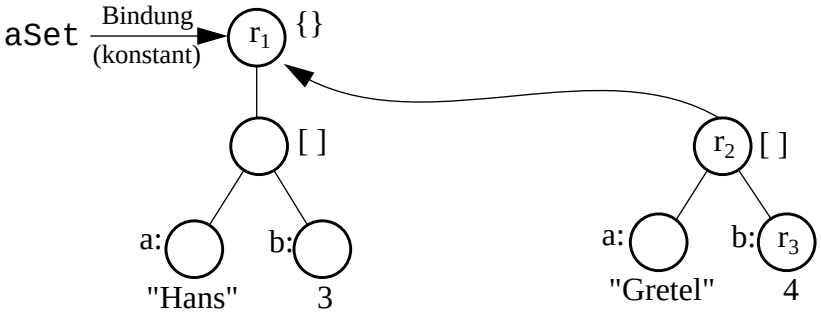


Abbildung 3.2: Veranschaulichung der intensionalen Semantik der Update-Operation `insert`(r_1, r_2)

7. Dabei habe die Funktion *remove* die gleiche Signatur wie *insert* und die naheliegende Semantik.

Für die Durchführung von Updates benötigt man jetzt keine *Funktionen*, wie dies unter der extensionalen Semantik der Fall war, sondern *Prozeduren*, die unter Beibehaltung der Identität der Knoten deren „Zustand“ bzw. „Inhalt“ modifizieren.

Für ESCHER$^+$ wählen wir die intuitivere intensionale Interpretation komplexer Datenobjekte als Grundlage für die Semantik aller Operationen. Eine Datenbank zu einem gegebenen Datenbankschema ist danach im wesentlichen durch eine Menge von Baumrepräsentationen gegeben, die gewissermaßen einen „Gesamtzustand“ verkörpert, der durch Update-Operationen modifiziert wird.

Die extensionale Semantik komplexer Datenobjekte erscheint unnatürlich und implementationsfern. Tatsächlich gibt es aber Beispiele aus der Praxis, die belegen, daß es sich dabei nicht allein um ein rein theoretisches Gedankenspiel handelt. Zum Lieferumfang des relationalen DBMS Ingres gehört auch die sog. Object Management Extension [Ing91], die es ermöglicht, einem Laufzeitsystem benutzerdefinierte ADTs inkrementell hinzuzufügen. Dabei ist das Vorgehen genauso wie bei den POSTGRES-ADTs, die bereits in Abschnitt 2.4.2 diskutiert wurden: Ein Element eines ADTs wird als uninterpretiertes BLOB in einer Tupelkomponente abgespeichert. Sind die Elemente eines solchen ADTs z.B. Mengen, so führt ein Einfügen eines weiteren Elementes in eine Menge zu einem Update des gesamten BLOBS, was gerade der extensionalen Semantik entspricht und nur bei kleinen BLOBS als tolerabel erscheint. Man beachte auch die Konsequenzen für das Setzen von Sperren im Mehrbenutzerbetrieb: Ein Element eines ADTs kann nur komplett gesperrt werden. Diese Granularität wird für möglicherweise stark geschachtelte Strukturen zu grob sein.

Im Datenmodell des ODMG’93-Standards [Cat+94] findet die Unterscheidung zwischen extensionaler bzw. intensionaler Semantik ihren direkten Niederschlag, indem z.B. zwischen den generischen Typen `Set<T>` und `Immutable_Set<T>` unterschieden wird. Die unveränderbaren (*immutable*) komplexen Datenobjekte entsprechen einem konstanten komplexen Datenobjekt im Sinne der extensionalen Semantik, der höchstens gegen ein anderes unveränderbares Datenobjekt „ausgetauscht“ werden kann: Ist etwa eine Variable an einen unveränderbaren komplexen Wert gebunden, kann durch eine Zuweisung die Variable an ein anderes unveränderbares Datenobjekt gleichen Typs gebunden werden. In den „Bestandteilen“ eines unveränderbaren komplexen Datenobjektes können - wie der Name schon sagt - jedoch keine lokalen Änderungen gemacht werden. Eine Einfüge-Operation ist auf `Immutable_Set<T>` demzufolge gar nicht definiert. Der ODMG’93-Standard selbst schreibt den unveränderbaren komplexen Datenobjekten nur geringe praktische Bedeutung zu. Laut [Cat+94] können sie zur Anzeige eines nicht-editierbaren Anfrageergebnisses oder zur Einschränkung der Update-Möglichkeiten bei Views eingesetzt werden. Im Gegensatz dazu sind wir der Überzeugung, daß die im ODMG’93-Standard gemachte strenge Unterscheidung zwischen unveränderbaren und veränderbaren (*mutable*) komplexen Datenobjekten übertrieben ist. Die Einschränkung der erlaubten Updates innerhalb eines komplexen Datenobjektes läßt sich über das Setzen von entsprechenden Update-Rechten, ggf. in Verbindung mit einem Autorisierungskonzept [RBK+91, GGF93, Brü94], ebenfalls erreichen.

3.5.2 Konstruktion von Baumrepräsentationen

In diesem Abschnitt soll die im vorangegangenen Abschnitt informell eingeführte „Behälter"-Metapher mit Hilfe von *beschrifteten Bäumen* formalisiert werden. Dabei setzen wir die Kenntnis der in Anhang A.2 aufgeführten Grundlagen und Notationen aus der Graphentheorie voraus. Die Beschriftungen eines Knotens in einem Baum stellen dabei gerade den „Zustand" bzw. „Inhalt" des Knotens dar. Auf diese Weise führen wir Baumrepräsentationen für komplexe Datenobjekte ein, die die adäquate Grundlage für die Spezifikation der Semantik aller noch zu definierenden Operationen bilden werden.

3.5.2.1 Baumrepräsentationen für Strukturen

Wir beginnen mit Baumrepräsentationen für $s \in \mathcal{S}$ (TYPES). Wir benötigen zur Konstruktion beschrifteter Bäume für Strukturen die Menge $L_{struct} = \{type, attr\_pos, domain\}$ von expliziten Beschriftungen, deren Elemente folgende Funktionen sind:

- *type* : $\mathcal{V} \cdots>$ TYPES
- *attr_pos* : $\mathcal{V} \cdots> \mathcal{PF}un(\mathcal{A}, \text{IN})$
- *domain* : $\mathcal{V} \cdots> \bigtimes_{i=1}^{k} \mathbb{Z}^k$.

Für die Beschriftungen *attr_pos* soll gelten: Ist *attr_pos*(v) für $v \in \mathcal{V}$ definiert, dann ist Def(*attr_pos*(v)) endlich und Bild(*attr_pos*(v)) = {1, …, |Def(*attr_pos*(v))| }, d.h. *attr_pos*(v) legt eine lineare Ordnung für eine endliche Menge von Attributen aus $\mathcal{A}$ fest.

Wir definieren im folgenden eine injektive Abbildung

$$\varphi_{struct}: \mathcal{S}(\text{TYPES}) \rightarrow \wp(\mathcal{T}ree(L_{struct})), \tag{3.18}$$

die jeder Struktur $s \in \mathcal{S}$ (TYPES) eine Menge $\varphi_{struct}(s)$ von beschrifteten Bäumen zuordnet, die paarweise „isomorph" sind, d.h. $\varphi_{struct}(s)$ besteht aus allen Bäumen, die ausgehend vom Knotenvorrat $\mathcal{V} = \{r_0, r_1, \dots\}$ konstruiert werden können und die Struktur s repräsentieren. Bei der Konstruktion muß wiederum lediglich der rekursive Aufbau von s nachverfolgt werden:

(i) Ist $s = t \in$ TYPES – CONSTR, dann ist $\varphi_{struct}(s)$ die Menge der beschrifteten Bäume B = (V, E) mit
V = {r} für $r \in \mathcal{V}$ beliebig, $E = \emptyset$ und mit *type* (r) = t.

(ii) Ist s eine Kollektion, d.h. es gilt *isCollection* (s), dann besteht $\varphi_{struct}(s)$ aus allen Bäumen B = (V, E), die wie folgt konstruiert werden:

Es sei ein beliebiger Baum $B' \in \varphi_{struct}(substruct\ (s))$ gegeben.

- $V := V(B') \cup \{r\}$, wobei $r \notin V(B')$
- $E := E(B') \cup \{(r, root\ (B'))\}$.
- Der Knoten r erhält die Beschriftung *type* (r) = *type* (s).
  Falls $s = < s' \mid n_1 : m_1, \dots, n_k : m_k >$, dann erhält r zusätzlich die Beschriftung
  domain $(r) = \{n_1, \dots m_1\} \times \dots \times \{n_k, \dots, m_k\}$.
- Die Unterbäume B_i behalten ihre Beschriftungen.

(iii) Ist $s = [a_1: s_1, \dots, a_k: s_k]$, dann besteht $\varphi_{struct}(s)$ aus allen Bäumen B = (V, E), die wie folgt konstruiert werden:

Für $1 \leq i \leq k$ seien beliebige, paarweise disjunkte $B_i \in \varphi_{struct}(s_i))$ gegeben.

- $V := \bigcup_{i=1}^{k} V(B_i) \cup \{r\}$, wobei $r \notin \bigcup_{i=1}^{k} V(B_i)$
- $E := \bigcup_{i=1}^{k} E(B_i) \cup \{(r, \mathit{root}(B_i)) \mid 1 \leq i \leq k\}$
- Der Knoten r erhält die Beschriftung *type* (r) = c_{tuple} und *attr_pos* (r) = $\{(a_1, \pi(1)), \ldots, (a_k, \pi(k))\}$, wobei π eine Permutation von {1, …, k} ist[8].
- Die Knoten $r_i = \mathit{root}(B_i)$ werden als Söhne von r so angeordnet, daß *ord* (r_i) = *attr_pos*(r)(a_i) gilt $(1 \leq i \leq k)$.
- Die Unterbäume B_i behalten ihre Beschriftungen.

Über die Beschriftung *attr_pos* wird also eine lineare Ordnung der Tupelkomponenten festgelegt. Der Zugriff auf den Strukturbaum zum Attribut a_i erfolgt dann über *get_child* (r, *attr_pos*(r)(a_i)).

Es ist $\mathcal{STree}$ (TYPES) := $\bigcup_{s \in \mathcal{S}(\text{TYPES})} \varphi_{struct}$ (s) die Menge aller *Strukturbäume* über TYPES.

Für die Struktur aus (3.14) wird in Abb. 3.3 ein Strukturbaum wiedergegeben. Die Identitäten der Knoten werden innerhalb der Knoten notiert. Der Wurzelknoten wird in der graphischen Darstellung besonders gekennzeichnet, um die Richtung der Kanten festzulegen.

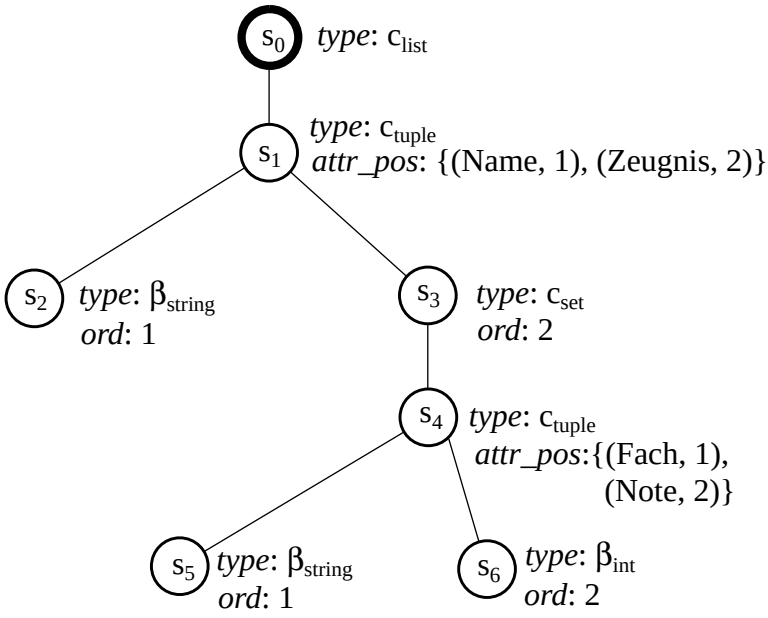


Abbildung 3.3: Struktur des komplexen Wertes aus Beispiel 3.3

Es folgt nun die Definition der Äquivalenz zweier Strukturbäume, aus der sich unmittelbar die noch ausstehende formale Definition eines Gleichheitsprädikates auf Strukturen ergibt, die für Tupelstrukturen eine mögliche Permutation der Tupelkomponenten berücksichtigt.

8. Wir wollen davon ausgehen, daß in einer realen Implementierung die Reihenfolge der Attribute durch eine DDL-Anweisung oder sonstige Strukturdefinition festgelegt ist.

Definition 3.15 (Äquivalenz von Strukturbäumen)

Es seien zwei Elemente $B_1, B_2 \in \mathcal{STree}$ (TYPES) gegeben. Man setze $r_i := root\ (B_i)$, i = 1, 2.

Es gilt $B_1 \cong_{struct} B_2 \Leftrightarrow$

$type\ (r_1) = type\ (r_2)$

$\wedge\ (isCollection\ (r_1) \Rightarrow Tree\ (get\_child\ (r_1, 1)) \cong_{struct} Tree\ (get\_child\ (r_2, 1))$

$\wedge\ (type\ (r_1) = c_{tuple} \Rightarrow$

$(\mathrm{Def}(attr\_pos\ (r_1)) = \mathrm{Def}(attr\_pos\ (r_2)) \wedge \forall\ a \in \mathrm{Def}(attr\_pos\ (r_1))$:

$Tree(get\_child\ (r_1, attr\_pos\ (r_1)(a))) \cong_{struct} Tree\ (get\_child\ (r_2, attr\_pos\ (r_2)(a))))$

Gilt $B_1 \cong_{struct} B_2$, dann nennen wir B_1 und B_2 äquivalent oder *isomorph*. ❒

Offensichtlich ist $\cong_{struct}$ eine Äquivalenzrelation, und die Mengen $\varphi_{struct}(s)$ sind gerade die Äquivalenzklassen dieser Relation. Die Gleichheit von Strukturen läßt sich nun wie folgt definieren:

$\forall\ s_1, s_2 \in \mathcal{S}$ (TYPES): $s_1 = s_2 :\Leftrightarrow \varphi_{struct}(s_1) = \varphi_{struct}(s_2)$.

Der Test auf Äquivalenz zweier Strukturbäume entspricht also gerade dem Test auf Gleichheit der durch sie repräsentierten Strukturen und umgekehrt.

Für variante Typen führen wir sog. *Variantenbäume* ein. Sie sollen gerade die Alternativen eines varianten Typs darstellen, wie sie durch die Funktion *variants* aus Definition 3.13 gegeben ist. Wir benötigen dazu eine zusätzliche Beschriftung

- $label\_pos : \mathcal{V} \cdots> \mathcal{PF}un\ (\mathcal{L}, \mathrm{IN})$

Diese Beschriftung habe die zur Beschriftung *attr_pos* analogen Eigenschaften, d.h. jede Funktion *label_pos*(r) lege eine lineare Ordnung für eine endliche Teilmenge von Labels fest.

Ist $t \in$ VARTYPES mit $variants(t) = \{(l_1: s_1), \ldots, l_k: s_k)\}$, dann wird ein Variantenbaum zu t analog zur Konstruktion einer Tupelstruktur (vgl. (iii) oben) konstruiert. Dabei ist auf folgende Unterschiede zu achten:

- Der Knoten r erhält die Beschriftung *type* (r) = t und *label_pos* (r) = $\{(l_1, \pi(1)), \ldots, (l_k, \pi(k))\}$, d.h. an die Stelle der Beschriftung *attr_pos* tritt nun *label_pos*.

Die Menge der auf diese Weise konstruierbaren Variantenbäume für einen varianten Typ t werde mit $\varphi_{variants}$ (t) bezeichnet. Wir nehmen im folgenden grundsätzlich an, daß für alle $t \in$ VARTYPES der Funktionswert *variants* (t) durch einen eindeutig bestimmten Baum aus $\varphi_{variants}(t)$ gegeben ist, den wir mit $B_{variants(t)}$ bezeichnen.

Unsere Strategie ist es also, die Definition der Alternativen eines varianten Typs nicht in die Strukturbäume an allen Stellen „hineinzuexpandieren“, an denen ein varianter Typ verwendet wird. Vielmehr ist jeder Knoten r in einem Strukturbaum mit $type(r) \in$ VARTYPES ein Blatt. Die Alternativen ergeben sich aus dem Variantenbaum $B_{variants(t)}$. Dieses Vorgehen steht im Einklang mit der Idee, die varianten Typen zu benamen und somit „zentral“ definieren und modifizieren zu können.

3.5.2.2 Baumrepräsentationen für Datenobjekte

Es gilt nun, für ein getyptes Datenobjekt (v, s) ∈ $\mathcal{V}al^*$(TYPES) geeignete Baumrepräsentationen anzugeben. Die Menge $\varphi_{struct}(s)$ für die „Typkomponente“ s können wir dabei als gegeben annehmen. Wir benötigen im folgenden die Menge $L_{val} = \{value, struct, label\}$ von expliziten Beschriftungen, deren Elemente folgende Funktionen sind:

- *value* : $\mathcal{V} \cdots> \mathcal{V}al_0$ (TYPES)
- *struct* : $\mathcal{V} \cdots> \mathcal{V}$
- *label* : $\mathcal{V} \cdots> \mathcal{L} \cup \{null\}$.

Wir definieren nun eine injektive Abbildung

$$\varphi_{val}: \mathcal{V}al^*(\text{TYPES}) \rightarrow \wp(\mathcal{T}ree(L_{val}) \times \mathcal{T}ree(L_{struct})), \tag{3.19}$$

die jedem getypten Datenobjekt (v, s) ∈ $\mathcal{V}al^*$(TYPES) eine Menge φ_{val} ((v, s)) von Paaren $(B_v, B_s) \in \mathcal{T}ree(L_{val}) \times \mathcal{T}ree(L_{struct})$ zuordnet. Die Menge φ_{val} ((v, s)) umfaßt dann alle „isomorphen Kopien“ des getypten Wertes (v, s). Wir werden eine Äquivalenzrelation ≅ auf $\varphi_{val}(\mathcal{V}al^*(\text{TYPES}))$ definieren, die gerade der Gleichheitsrelation auf $\mathcal{V}al^*$(TYPES) entspricht.

Die Beschriftung *struct* drückt einen Verweis von einem Knoten r in einem *Wertebaum* B_v auf einen entsprechenden Knoten r' im Strukturbaum B_s aus, so daß $(Tree(r), Tree(struct(r)))$ wiederum ein Element aus φ_{val} ((v, s)) ist.

Baumrepräsentation eines einfachen Datenobjektes:

Die Menge φ_{val}((v, s)) der Baumrepräsentation eines getypten Datenobjektes (v, t) mit t ∈ TYPES − CONSTR, d.h. eines atomaren Wertes oder Objektes, ist gegeben durch die Menge der Baumpaare (B_v, B_s), die wie folgt konstruiert werden:

- B_s ist ein beliebiger Baumes aus $\varphi_{struct}(s)$
- $B_v = (V, E)$ mit $V = \{r\}$ für r ∈ $\mathcal{V}$ beliebig, $E = \emptyset$ und mit den Beschriftungen $value(r) = v$ und $struct(r) = root(B_s)$.

Baumrepräsentation von Kollektionen:

Es sei (v, s) ∈ $\mathcal{V}al^*$(TYPES) eine getypte Kollektion, d.h.

- $v = \{e_1, \ldots, e_n\}$, n ∈ IN_0, falls $s = c_{set}(s')$
- $v = \{*e_1, \ldots, e_n*\}$, n ∈ IN_0, falls $s = c_{bag}(s')$
- $v = <e_1, \ldots, e_n>$, n ∈ IN_0, falls $s = c_{list}(s')$
- $v = <e_1, \ldots, e_n \mid n_1 : m_1, \ldots, n_k : m_k>$, falls $s = c_{array}(s')$
  (Dabei ist n = p mit p aus (3.12) und $e_i = v(lexord_{Def(v)}^{-1}(i))$ für $1 \le i \le n$)

Die Menge φ_{val}((v, s)) besteht aus allen Baumpaaren (B_v, B_s), die wie folgt konstruiert werden:

- B_s ist ein beliebiger Baum aus $\varphi_{struct}(s)$.
- $B_v = (V, E)$ ergibt sich wie folgt:
  Für $1 \le i \le n$ seien beliebige Baumpaare $(B_i, B_{s'}) \in \varphi_{val}((e_i, s'))$ gegeben, wobei die B_i paarweise disjunkt seien und $B_{s'}$ derart, daß $parent(root(B_{s'})) = root(B_s)$, d.h. $B_{s'}$ ist der unterhalb des Wurzelknotens von B_s beginnende Unterbaum. Wir setzen dann:

$V := \bigcup_{i=1}^{n} V(B_i) \cup \{r\}$, wobei $r \notin \bigcup_{i=1}^{n} V(B_i)$

$E := \bigcup_{i=1}^{n} E(B_i) \cup \{(r, root(B_i)) \mid 1 \le i \le n\}$

Die Knoten $r_i = root(B_i)$ werden als Söhne von r so angeordnet, daß sie die implizite Beschriftung $ord(r_i) = \pi(i)$ tragen ($1 \le i \le n$), wobei π eine Permutation der Menge $\{1, \ldots, n\}$ ist. Dabei ist $\pi = id$[9], falls $type(s) \in \{c_{list}, c_{array}\}$. Für $type(s) \in \{c_{set}, c_{bag}\}$ kann π beliebig gewählt werden.
Der Knoten r erhält die Beschriftung $struct(r) = root(B_s)$.
Die Unterbäume B_i behalten ihre Beschriftungen.

In Abb. 3.4 ist die Baumrepräsentation einer Kollektion graphisch dargestellt.

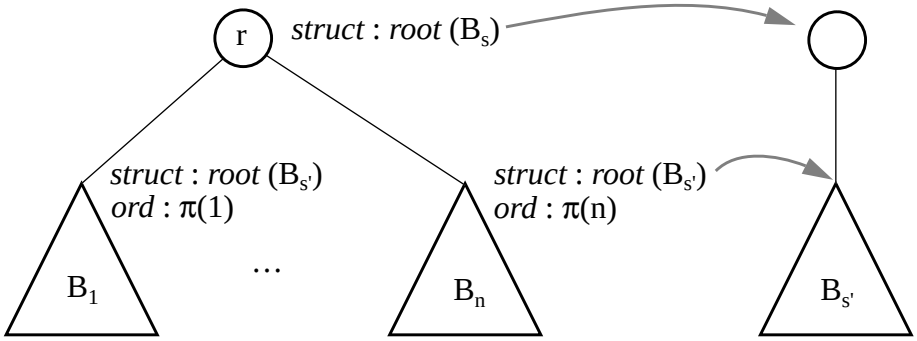


Abbildung 3.4: Baumrepräsentation einer Kollektion

Die Beliebigkeit in der Wahl der Permutation π für Mengen und Multimengen drückt aus, daß die lineare Anordnung der Elemente von Mengen oder Multimengen keine Rolle spielt. In einer Implementation ist die Wahl der Permutation π bei Mengen und Multimengen nicht immer zufällig, sondern kann von einem Sortierkriterium abhängen. Dies soll jedoch an dieser Stelle für das formale Modell keine Rolle spielen, so daß π als „zufällig“ gewählt erscheint.

Baumrepräsentation von Tupeln:

Es sei $(v, s) = ([a_1: v_1, \ldots, a_k: v_k], [a_1: s_1, \ldots, a_k: s_k]) \in \mathcal{V}al^*(\text{TYPES})$ ein getyptes Tupel. Die Menge $\varphi_{val}((v, s))$ besteht aus allen (B_v, B_s), die wie folgt konstruiert werden können:

- B_s ist ein beliebiger Baum aus $\varphi_{struct}(s)$.

  B_v ergibt sich wie folgt:
  Für $1 \le i \le k$ seien beliebige Baumpaare $(B_i, B_{s_i}) \in \varphi_{val}((v_i, s_i))$ gegeben, wobei die B_i paarweise disjunkt sind und $parent(root(B_{s_i}) = root(B_s)$ gilt. Wir setzen:

  $V := \bigcup_{i=1}^{k} V(B_i) \cup \{r\}$, wobei $r \notin \bigcup_{i=1}^{k} V(B_i)$

  $E := \bigcup_{i=1}^{k} E(B_i) \cup \{(r, root(B_i)) \mid 1 \le i \le k\}$

  Die Knoten $root(B_i)$ werden dabei so angeordnet, daß $ord(root(B_i)) = attr\_pos(root(B_s))(a_i)$ gilt, d.h. die Tupelkomponenten sind dann gemäß der im zugehörigen Strukturbaum festgelegten Reihenfolge angeordnet.

9. *id* sei die identische Abblidung, d.h. $id(i) = i$ für alle i.

Man beachte, daß bei der soeben beschriebenen Konstruktion die Baumrepräsentationen für eine Menge von Tupeln die Eigenschaft haben, daß alle Tupel im Wertebaum über die Beschriftung *struct* auf denselben Knoten im Strukturbaum „zeigen“. Das bedeutet, daß für alle Tupel dieselbe lineare Anordnung der Attribute relevant ist. Dies ist das „Zugeständnis“ des formalen Modells an eine effiziente Implementation: Nur durch homogene Speicherung aller Tupel ist ein effizienter Zugriff auf ein bestimmtes Attribut in allen Tupeln dieser Menge möglich.
Ist r ein Tupelknoten in einem Wertebaum, dann erfolgt der Zugriff auf die Tupelkomponente zum Attribut a_i über *get_child* (r, *attr_pos* (*struct* (r)(a_i)).

Baumrepräsentation einer Instanz eines varianten Typs

Eine besondere Behandlung ist für die Instanzen von varianten Typen notwendig. Hier kommt zum Tragen, daß die Information über die Alternativen nicht in den Strukturbäumen auftaucht, sondern für jeden varianten Typ t durch seinen eindeutig bestimmten Variantenbaum bestimmt ist. Das bedeutet, daß die Baumrepräsentationen für Instanzen von varianten Typen auf der Strukturseite eine Art „Sprung“ von einem Blatt eines Strukturbaums auf den Wurzelknoten des Strukturbaums der relevanten Alternative innerhalb des Variantenbaumes gemacht wird. Dies gilt es nun zu formalisieren:

Das getypte Datenobjekt (v, t) sei Instanz eines varianten Typs t $\in$ VARTYPES.
Es sei v $\neq$ *null*, d.h. v= (*l*, v') und (*l*, s) $\in$ *variants*(t). Die Menge φ_{val}(((*l*, v'), t)) der Baumrepräsentation für ((*l*, v'), t) mit t $\in$ VARTYPES ist gegeben durch die Menge der Baumpaare (B_v, B_t), die wie folgt konstruiert werden:

- Es sei r'' $\in$ $\mathcal{V}$ der Wurzelknoten des Variantenbaumes von t
- es ist $B_t \in \varphi_{struct}(t)$ beliebig, d.h. $B_t = (\{r'\}, \emptyset)$ mit der Beschriftung *type* (r') = t
- ($B_{v'}$, B_s) sei ein beliebiges Element aus $\varphi_{val}((v', s))$ mit der Eigenschaft
  struct (*root*($B_{v'}$)) = *get_child* (r'', *label_pos*(r'')(*l*))
- B_v = (V, E) ergibt sich wie folgt:
  $V = \{r\} \cup V(B_{v'})$, wobei $r \notin V(B_{v'})$, $E = E(B_{v'}) \cup \{(r, root(B_{v'}))\}$,
  label(r) = *l* und *struct* (r) = *root* (r').

In Abb. 3.5 ist graphisch veranschaulicht, wie eine Baumrepräsentation zu einer Instanz ((*l*, v), t)) eines varianten Typs t konstruiert wird.

Falls v = (*null*, t), dann besteht φ_{val}((*null*, t)) aus allen Baumpaaren (B_v, B_t), die wie folgt konstruiert werden können:

- B_t sei ein beliebiger Baum aus $\varphi_{struct}(t)$
- $B_v = (\{r\}, \emptyset)$ mit r $\in$ $\mathcal{V}$ beliebig, *label*(r) = *null* und *struct* (r) = *root* (B_t).

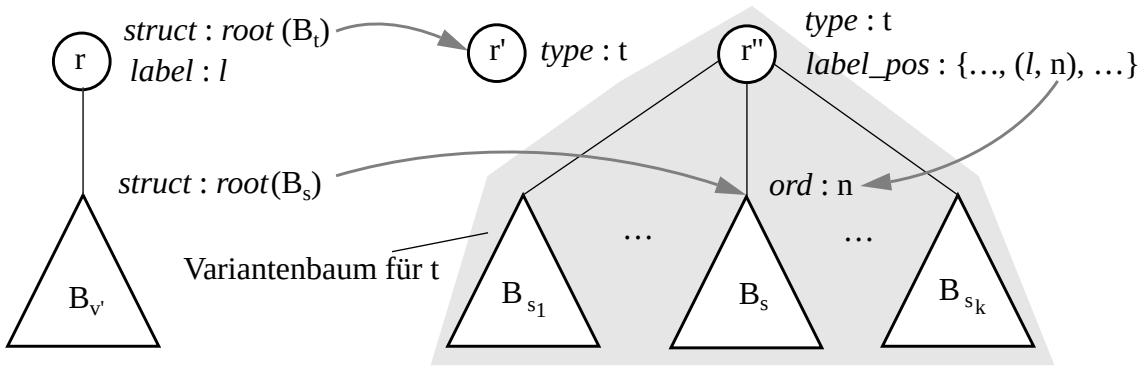


Abbildung 3.5: Baumrepräsentation für eine Instanz eines varianten Typs

Wir setzen nun

$$\mathcal{ValTree}(\text{TYPES}) := \bigcup_{(v,\, s) \in \mathcal{Val}^*(\text{TYPES})} \varphi_{val}((v, s))$$

$\mathcal{ValTree}$ (TYPES) ist die Menge aller *Baumrepräsentationen* (B_v, B_s), die auf die oben beschriebene Art und Weise konstruiert werden können. Die Bäume B_v nennen wir *Instanzenbäume*.

Beispiel 3.5

Eine Baumrepräsentation des getypten komplexen Datenobjektes (v, s) mit s aus (3.14) und

v = < [Name: "Hugo", Zeugnis: {[Fach: "Mathe", Note: 2],
[Fach: "Musik", Note: 3]}], [Name: "Anna", Zeugnis : {}]> (3.20)

ist in Abb. 3.6 gezeigt. Man beachte , daß die Ordnung der Knoten r_7 und r_8 als Söhne von r_4 „zufällig“ ist, da r_4 ein Mengenknoten ist. Der Knoten r_6 hat keine Söhne und repräsentiert daher die leere Menge. Die Beschriftung *struct* verweist auf die gleichnamigen Knoten in Abb. 3.3. Die Attributnamen werden der Übersichtlichkeit halber zusätzlich in Klammern als Kantenbeschriftungen angegeben.

In Abb. 3.7 wird dasselbe komplexe Datenobjekt in der Standard-Darstellung für ESCHER$^+$ in tabellarischer Form veranschaulicht. Der untere Teil enthält das Datenobjekt. Man erkennt die horizontale Anordnung der Tupelkomponenten und die vertikale Anordnung der Mengen- bzw. Listenelemente. Darüber befindet sich das *„Schema“*, das die Strukturinformation enthält. ❒

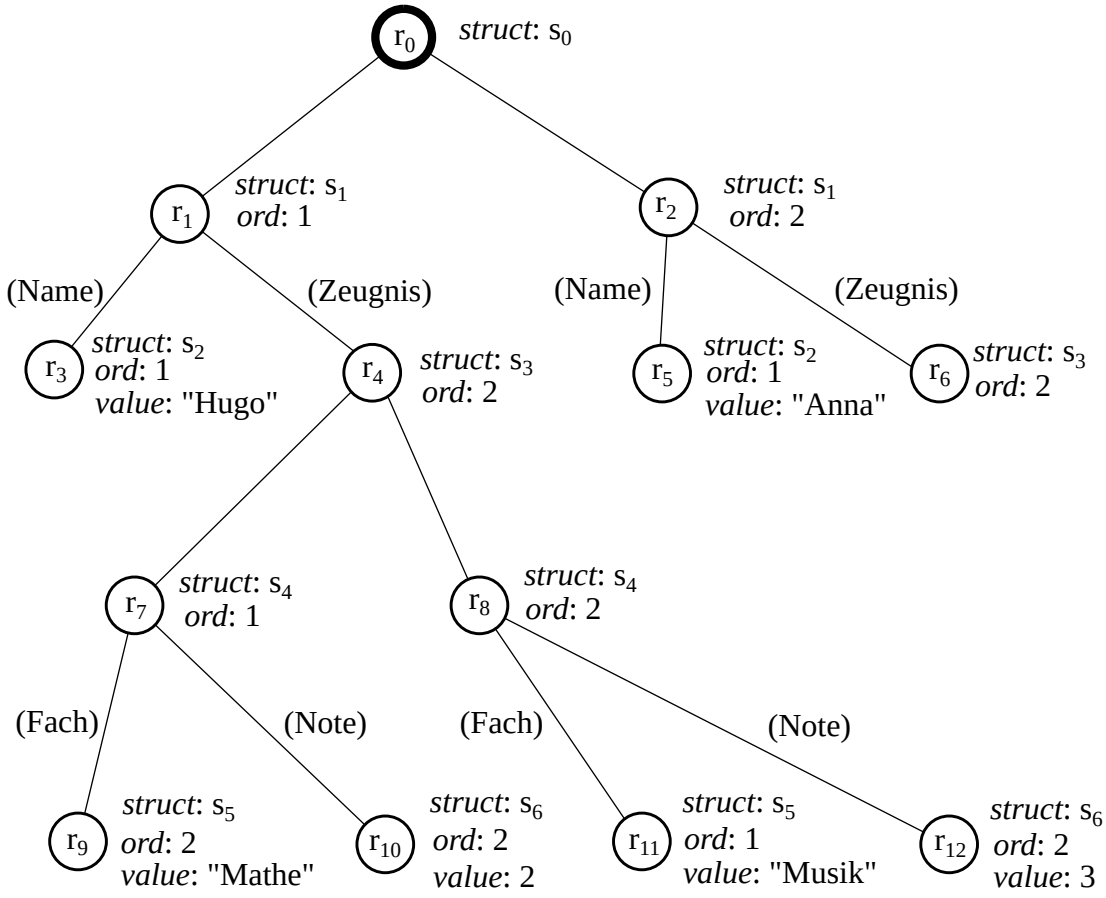


Abbildung 3.6: Beispiel für eine Baumrepräsentation

< >		
Name: string	Zeugnis: { }	
	Fach: string	Note: int
Hugo	Mathe	2
	Musik	3
Anna	{}	

Abbildung 3.7: Darstellung des komplexen Datenobjektes aus Abb. 3.6 in tabellarischer Form

Die gegenwärtige Implementation des ESCHER-Prototyps geht noch ein Stück über den gerade beschriebenen Ansatz der „Verweise“ in einen Strukturbaum hinaus: Während nach dem bisher Gesagten bei jedem Knoten in einem Wertebaum ein Verweis auf den entsprechenden Knoten im Strukturbaum gespeichert ist, ist dies im ESCHER-Prototyp nicht nötig: Die zentrale Rolle für den Zugriff auf persistente Objekte und deren Manipulation übernehmen die sog.

Finger. Sie sind nichts anderes als Zeiger auf Knoten einer Baumrepräsentation. Ein Zugriff auf einen Knoten ist immer durch den Pfad bestimmt, der ausgehend vom Wurzelknoten des Wertebaumes zu dem gewünschten Knoten führt. Die zu einem Knoten gehörende Strukturinformation ergibt sich aus einem in der Datenstruktur des Fingers integrierten *Schemafinger*, der parallel zur Navigation des „eigentlichen" Fingers (im ESCHER-Prototyp auch *Tabellenfinger* genannt, weil dort Instanzenbäume immer einer sog. Tabelle entsprechen) entlang des gewünschten Pfades im Wertebaum dem entsprechenden Pfad im Strukturbaum folgt. Das bedeutet, daß im ESCHER-Prototyp der oben angesprochene Verweis erst zur Laufzeit aufgebaut wird und nicht bei jedem Instanzenknoten abgespeichert werden muß.

Es stellt sich die Frage, wann zwei Elemente aus $\mathcal{ValTree}$ (TYPES) dasselbe getypte Datenobjekt (v, s) spezifizieren. Dies ist Gegenstand der folgenden Definition.

Definition 3.16 (Äquivalenz, Disjunktheit von Baumrepräsentationen)

Es seien zwei Elemente $(B_{v_1}, B_{s_1}), (B_{v_2}, B_{s_2}) \in \mathcal{ValTree}$ (TYPES) gegeben.
Es seien $r_i = root(B_{v_i})$, $s_i = struct(r_i)$, $i = 1, 2$.
Es gilt $(B_{v_1}, B_{s_1}) \cong (B_{v_2}, B_{s_2}) \Leftrightarrow$

$type(s_1) = type(s_2)$

$\wedge$ ($type(s_1) \in \{c_{set}, c_{bag}\} \Rightarrow$
$(\exists \alpha \in \mathcal{Fun}^{bij}(children(r_1), children(r_2)): \forall r' \in children(r_1): Tree(r') \cong Tree(\alpha(r'))))$

$\wedge$ ($type(s_1) \in \{c_{list}, c_{array}\} \Rightarrow (degree(r_1) = degree(r_2) \wedge$
$\forall r' \in children(r_1)\ \forall r'' \in children(r_2): (ord(r') = ord(r'') \Rightarrow Tree(r') \cong Tree(r'')))$

$\wedge$ ($type(s_1) = c_{tuple} \Rightarrow$
$(\mathrm{Def}(attr\_pos(s_1)) = \mathrm{Def}(attr\_pos(s_2)) \wedge \forall a \in \mathrm{Def}(attr\_pos(s_1)):$
$Tree(get\_child(r_1, attr\_pos(s_1)(a))) \cong Tree(get\_child(r_2, attr\_pos(s_2)(a))))$

$\wedge$ ($type(s_1) \in$ VARTYPES $\Rightarrow$
$(label(r_1) = label(r_2) \wedge$
$(label(r_1) \neq null \Rightarrow Tree(get\_child(r_1, 1)) \cong Tree(get\_child(r_2, 1)))))$

$\wedge$ ($type(s_1) \in$ BASETYPES $\cup$ ENUMTYPES $\cup$ DEFTYPES
$\Rightarrow value(r_1) = value(r_2)$)

Gilt $(B_{v_1}, B_{s_1}) \cong (B_{v_2}, B_{s_2})$, dann nennen wir (B_{v_1}, B_{s_1}) und (B_{v_2}, B_{s_2}) äquivalent oder *isomorph*.
(B_{v_1}, B_{s_1}) und (B_{v_2}, B_{s_2}) heißen *disjunkt*, wenn B_{v_1} und B_{v_2} disjunkt sind. ❐

Aufgrund der Art der Konstruktion der Mengen φ_{val} ((v, s)) und der darauf abgestimmten Definition der Relation $\cong$, die eine Äquivalenzrelation ist, ist klar, daß φ_{val} die (einelementigen) Äquivalenzklassen der Relation = auf $\mathcal{Val}^*$(TYPES) auf Äquivalenzklassen der Relation $\cong$ auf $\mathcal{ValTree}$ (TYPES) abbildet. Der Test auf Gleichheit zweier getypter Datenobjekte läßt sich daher auf die Isomorphie von Baumrepräsentationen zurückführen und umgekehrt. Ferner sei bemerkt, daß aus $(B_{v_1}, B_{s_1}) \cong (B_{v_2}, B_{s_2})$ offensichtlich $B_{s_1} \cong_{struct} B_{s_2}$ folgt.

Natürlich ist auch jeder Knoten r in einem Wertebaum wiederum Wurzel eines Wertebaumes *Tree* (r), und *struct* (r) ist Wurzel eines Strukturbaumes *Tree* (*struct* (r)). Das Paar (*Tree*(r), *Tree* (*struct* (r))) ist dann auch ein Element von $\mathcal{Val}^*$(TYPES). Wir definieren die Abbildung *val* : $\mathcal{V} \cdots > \mathcal{Val}^*$(TYPES), indem wir für einen Knoten r eines Wertebaumes setzen:

val(r) := dasjenige getypte Datenobjekt (v, s) ∈ $\mathcal{V}al^*$(TYPES)
mit (*Tree* (r), *Tree* (*struct* (r))) ∈ φ_{struct}((v, s)).

Notation: Um die Gleichberechtigung bzw. Austauschbarkeit zwischen getypten Werten und ihren Baumrepräsentationen zu unterstreichen, schreiben wir für
val (r) = (v, s) auch noch kürzer r = (v, s) bzw. *struct* (r) = s.

3.6 Die strukturelle Beschreibung von Objekttypen

In Abschnitt 3.1.3 wurden benutzerdefinierte Objekttypen eingeführt, deren Instanzen abstrakte Objekte aus Ω sind. In diesem Abschnitt wird an die Identität eines Objektes ein veränderbarer Zustand gebunden, den wir als die strukturelle Semantik eines abstrakten Objektes bezeichnen und der durch eine Tupel-Struktur aus $\mathcal{S}$(TYPES) gegeben ist. Wir setzen daher die in Def. 3.3 gegebene Definition für Objekttypen fort.

Definition 3.17 (Objekttypen, Fortsetzung der Definition 3.3)
Auf der Menge DEFTYPES der *Objekttypen* wird zusätzlich folgende Funktionen definiert:

$$struct: \text{DEFTYPES} \rightarrow \mathcal{S}(\text{TYPES}), \tag{3.21}$$

die jedem Objekttyp eine Struktur zuordnet. Dabei fordern wir:

$$\forall\, t \in \text{DEFTYPES}: type(struct(t)) = c_{tuple}{}^{10} \tag{3.22}$$

❒

Die Funktion *struct* in (3.21) beschreibt den *Strukturteil* eines Objekttyps.
Ist $struct(t) = [a_1{:}s_1, \ldots, a_n{:}s_n]$, dann sei

$$Attrs(t) := \{a_1, \ldots, a_n\} \tag{3.23}$$

die Menge der für t definierten *strukturellen* Eigenschaften. Später werden Objekttypen noch um *operationale* Eigenschaften (Methoden) ergänzt.

Die durch die Bedingung (3.22) gegebene Einschränkung des Strukturteils auf Tupel-Strukturen ist Voraussetzung für eine einfache Beschreibung der Vererbung auf der Grundlage der Subtyp-Beziehung auf Objekttypen, die in Abschnitt 3.7 eingeführt wird. Dann wird auch deutlich, wann es sinnvoll ist, einen Typ mit *struct* (t) = [ ] zu definieren.

Wir nehmen im folgenden an, daß für jeden Objekttyp t ∈ DEFTYPES ein eindeutiger Strukturbaum aus $\varphi_{struct}(struct(t))$ festgelegt ist, den wir mit $B_{struct(t)}$ bezeichnen. Die Funktion *struct* aus (3.21) läßt sich dann auch als Funktion auffassen, die den Wurzelknoten $root(B_{struct(t)})$ liefert.

Für jedes Datenbankschema gibt es eine Menge DEFTYPES, die anfangs leer ist. Der Schema-Designer definiert nach und nach neue Objekttypen, die in DEFTYPES eingefügt werden. Folgendes DDL-Statement definiert einen neuen Objekttyp:

10. Es sei daran erinnert, daß das leere Tupel [ ] mit eingeschlossen ist.

```
define type
  -name    identifier
  -struct '[' attrList ']'
end define;
```

mit

attrList ::= [*attrName* ':' *struct* { ',' *attrName* ':' *struct* }] [11]

Mit der Abarbeitung einer `define type`-Anweisung wird ein neuer Typ t_{neu} als weiteres Element zu DEFTYPES hinzugefügt. Die verschiedenen Klauseln innerhalb einer `define type`-Anweisung legen die Eigenschaften des neuen Typs fest:

- Die `name`-Klausel bestimmt den Funktionswert *name* (t_{neu}).
- Es wird *dom* (t_{neu}) := {*null*} gesetzt. Der dynamisch veränderbare Wertebereich des Typs t_{neu} enthält also zunächst nur das „Null-Objekt“.
- Die `struct`-Klausel setzt *struct* (t_{neu}) := [a_1: s_1, …, a_n: s_n], falls *attrList* = a_1: s_1 `;` … `;` a_n: s_n mit a_i: $s_i \in \mathcal{L}$ (*identifier* : *struct*), $n \geq 0$.

Beispiel 3.6 Es soll ein Objekttyp `Buch` definiert werden. Dies geschieht durch folgende DDL-Anweisung:

```
define type
  -name   Buch
  -struct [Signatur: string,
          Autoren: <Autor>,
          Titel: string,
          Exemplare: {integer}]
end define;
```

Diese Objektypdefinition erzeugt einen neuen Typ $t_{Buch} \in$ DEFTYPES, wie er in Abb. 2.2 als IFO-Fragment veranschaulicht wurde. Die Elemente der *Liste* von Autoren sollen Instanzen eines weiteren Objekttyps t_{Autor} sein. ❒

Bis hierhin ist die Frage offengeblieben, auf welche Weise Instanzen eines Objekttyps erzeugt werden und wie ein „abstraktes“ Objekt $\omega \in \Omega$ eine „konkrete“ strukturelle Semantik zugeordnet bekommt. Dies wird nun vorbereitet, indem für jeden Objekttyp t $\in$ DEFTYPES eine partielle Funktion

$$I\,(t) : \Omega \dashrightarrow \mathcal{V}al^*(\text{TYPES}) \tag{3.24}$$

definiert wird, die für ein $\omega \in \Omega$ ein zur Struktur des Objekttyps passendes, getyptes Datenobjekt (v, *struct* (t)) liefert. Wir nennen *I* (t) die (*strukturelle*) *Interpretation* von abstrakten Objekten aus Ω *relativ zum Typ* t. Sie ordnet jedem Objekt ω aus dem Definitionsbereich von *I* (t) einen Wert zu, der die strukturelle Beschreibung des Objekts hinsichtlich des Typs t angibt.

Ein $\omega \in \Omega$ nennen wir eine *Instanz* des Typs t oder auch kurz t-Objekt, wenn $\omega \in$ Def(*I* (t)).

Für jedes getypte Datenobjekt (v, *struct* (t)) $\in$ Bild(*I*(t)) läßt sich eine Baumrepräsentation (B_v, $B_{struct(t)}$) $\in \varphi_{val}$((v, *struct* (t))) angeben. Dabei ist $B_{struct(t)}$ wieder der für den Objekttyp t besonders ausgezeichnete Strukturbaum, der die Struktur *struct*(t) aus (3.21) repräsentiert, d.h. alle

11. Hinsichtlich unserer Notation von Grammatikregeln sei auf Anhang B.1 verwiesen.

Interpretationen beziehen sich auf denselben „zentralen“ Strukturbaum $B_{struct(t)}$. Dies ist also analog zu unserem Vorgehen bei varianten Typen: Dort beziehen sich alle Instanzen eines varianten Typs auf seinen „zentralen“ Variantenbaum.

Wegen der Austauschbarkeit getypter Datenobjekte mit ihren Baumrepräsentationen können wir $I(t)$ auch als Abbildung

$$I(t) : \Omega \dashrightarrow \mathcal{V} \tag{3.25}$$

auffassen, die den Wurzelknoten eines Wertebaumes B_v mit
$(B_v, B_{struct(t)}) \in \varphi_{val}((v, struct(t)))$ liefert.

Unmittelbar nach der Definition eines neuen Objekttyps t ist $I(t) = \emptyset$, d.h. $I(t)$ ist die überall undefinierte Funktion. Der Definitionsbereich von $I(t)$ wird nach und nach erweitert, indem Objekte aus Ω „initialisiert“ und gleichzeitig in die Domäne des Objekttyps t aufgenommen werden. Dies geschieht über eine Operation `initObject`, die zu den sogenannten Basisoperationen von ESCHER$^+$ gehört, auf die in Abschnitt 4.3.4 ausführlich eingegangen wird.

Die Semantik eines Aufrufs `initObject(`tname`)` läßt sich wie folgt angeben[12]:

- Es wird ein getyptes Datenobjekt (ω, t) zurückgegeben, wobei ω ein „neues“ Objekt aus Ω ist (d.h. ω ist von allen bisher mittels `initObject` erzeugten Objekten verschieden).
- Der Aufruf hat den „Nebeneffekt“ $I(t)(\omega) := initVal(struct(t))$

Dabei ist

$$initVal : \mathcal{S}(\text{TYPES}) \rightarrow \mathcal{V}al^*(\text{TYPES}) \tag{3.26}$$

wie folgt definiert:

$$initVal(s) := \begin{cases} (\{\} \text{ bzw. } <> \text{ bzw. } \{**\}, s) & \text{, falls } s = \{s'\} \text{ bzw. } <s'> \text{ bzw. } \{*s'*\} \\ ([a_1: initVal(s_1), \ldots, a_k: initVal(s_k)], s) & \text{, falls } s = [a_1: s_1, \ldots, a_k: s_k] \\ (<initVal(s'), \ldots, initVal(s') \mid n_1 : m_1, \ldots, n_k : m_k>, s) & \\ \qquad \text{, falls } s = < s' \mid n_1 : m_1, \ldots, n_k : m_k> & \\ (null, s) & \text{, sonst} \end{cases}$$

Der strukturelle Zustand des neuen Objektes wird bzgl. des Typs t also zunächst mit einem Default-Wert belegt.

Es zeigt sich, daß Ω als unbeschränkter „Objektvorrat“ fungiert. Es muß dafür gesorgt werden, daß durch einen Aufruf von `initObject` kein $\omega \in \Omega$ doppelt vergeben wird, damit die Eindeutigkeit der ω in ihrer Rolle als Objektidentifikatoren gewährleistet ist. Wir könnten dies in unserem Modell dadurch beschreiben, daß statt Ω immer ein Paar (Ω, i) betrachtet wird, wobei i angibt, welches ω_i beim nächsten Aufruf von `initObject` zurückgegeben wird, d.h. $\{\omega_1, \ldots, \omega_{i-1}\}$ sind die bereits „verbrauchten“ Objekte. Wir wollen an dieser Stelle jedoch von einer Überformalisierung des Datenmodells absehen und bleiben bei der gegebenen Definition von `initObject`.

Es wird nun auch die verbliebende „Lücke“ im Datenmodell geschlossen, indem die Funktion *dom* auf DEFTYPES durch $dom(t) := Def(I(t)) \cup \{null\}$ definiert wird.

Im Unterschied zu den Basistypen liegt der Wertebereich für Typen aus DEFTYPES also nicht

12. Es soll t der eindeutig bestimmte Typ in DEFTYPES mit $name(t) = tname$ sein.

a priori fest, sondern ändert sich dynamisch. Mittels der Operation `initObject` können weitere Objekte dem Wertebereich *dom* (t) hinzugefügt werden. Die Aussage, daß der Wertebereich eines definierten Typs eine bestimmte Teilmenge von Ω ist, gilt also nur für einen bestimmten Zeitpunkt im „Leben“ einer Datenbank.

Es reicht jedoch nicht aus, ein Objekt über einen Aufruf von `initObject` zu erzeugen, um es persistent zu machen. Dazu muß es über eine sog. Tabelle „erreichbar“ sein, d.h. es soll das Prinzip „Persistenz durch Erreichbarkeit“ gelten. Tabellen werden im nächsten Abschnitt eingeführt, jedoch bereits hier einige Vorbereitungen durchgeführt. Wir betrachten die Funktionen $I(t)$ gemäß (3.25), d.h. $I(t)(\omega)$ liefert den Wurzelknoten r eines Wertebaumes, der den „Zustand“ des Objektes relativ zu t repräsentiert. Für diese Wurzelknoten wird eine weitere Beschriftung

$$pref : \mathcal{V} \cdots > \mathrm{IN}_0 \tag{3.27}$$

eingeführt, die als Zähler für die Anzahl der „*p*ersistenten *Ref*erenzen“ fungiert. Durch einen Aufruf von `initObject` wird für das neue Objekt ω mit dem Typ t zunächst $pref(I(t)(\omega)) = 0$ gesetzt. Später bewirkt z.B. jedes Einfügen von ω in eine Tabelle ein Inkrementieren von *pref* um 1. Entsprechend führt ein Entfernen zu einem Dekrementieren von *pref* um 1.

In einigen Situationen ist man an einer anderen Initialisierung der Attribute eines neu erzeugten Objekts interessiert, als sie durch die Funktion *initVal* gegeben ist. Initialisierungswerte lassen sich in einer Objekttypdefinition in der `default`-Klausel angeben. Sie hat den Aufbau

`-default` *attrName* '=' *expr* { ',' *attrName* '=' *expr* }

Für einen oder mehrere Attributnamen wird jeweils ein Ausdruck *expr* angegeben, mit dessen Wert das Attribut initialisiert wird. In vielen Fällen wird *expr* eine Konstante sein. Analog zur DDL aus [Pau94] lassen wir jedoch allgemeinere Ausdrücke zu. Es handelt sich dabei um Ausdrücke einer geeigneten persistenten Programmiersprache, wie z.B. die Sprache SCRIPT aus [Pau94], auf die im folgenden Kapitel noch eingegangen wird.

3.7 Die Subtyp-Beziehung auf Objekttypen

3.7.1 Grundlagen

Von den in Abschnitt 2.1.1 genannten Abstraktionsmechanismen für die semantische Datenmodellierung hat sich insbesondere die Spezialisierung als charakteristischer Bestandteil objektorientierter Datenmodelle etabliert. Die Spezialisierung wird über eine sog. Subtyp- oder *isa*-Beziehung zwischen Objekttypen ausgedrückt.

Wir führen mit der folgenden Definition auch für ESCHER$^+$ eine Beziehung *isa* zwischen Objekttypen im Sinne der Spezialisierung ein.

Definition 3.18 (Subtyp-Beziehung, *isa*-Beziehung, isa-Wurzeltyp)

Es sei eine Relation *isa* : DEFTYPES × DEFTYPES gegeben. Mit *isa** werde die reflexive und transitive Hülle von *isa* bezeichnet. Es sei *isa* derart, daß *isa** eine Halbordnung[13] auf DEFTYPES ist.

Wir nennen die Relation *isa* die *Subtyp-Beziehung* (*isa-Beziehung*) auf DEFTYPES.

Gilt t_1 *isa* t_2 (bzw. t_1 *isa** t_2, aber $t_1 \neq t_2$ und $(t_1, t_2) \notin$ *isa*), dann ist t_1 eine *direkte* (bzw. *indirekte*) *Spezialisierung* von t_2. Dann ist t_1 ein *direkter* (bzw. *indirekter*) *Subtyp* von t_2 und t_2 ein *direkter* (bzw. *indirekter*) *Supertyp* von t_1.

Gibt es für t ∈ DEFTYPES keinen Typ t' ∈ DEFTYPES mit t *isa* t', dann ist t ein *isa-Wurzeltyp*. ❒

Die Eigenschaft der Halbordnung für eine Subtyp-Beziehung *isa* ist sehr wichtig: Faßt man (t, t') ∈ *isa* als Kanten eines gerichteten Graphen mit der Knotenmenge DEFTYPES auf, so darf dieser keine Kreise enthalten. Zirkuläre Abhängigkeiten, die eine wohldefinierte Vererbung von Eigenschaften unmöglich machen, werden somit ausgeschlossen.

Ein klassisches Beispiel einer *isa*-Beziehung wurde bereits in Abb. 1.7 angegeben. Die beiden Typen `Student` und `Angestellter` sind Subtypen des Typs `Person`, der somit Supertyp der beiden erstgenannten Typen ist. Es handelt sich dabei um zwei parallele Spezialisierungen, die voneinander unabhängig sind.

Beispiel 3.7 Bei Typdefinitionen werden in der `isa`-Klausel eine Subtyp-Beziehung festgelegt. Zunächst definieren wir einen Objekttyp `Person`:

```
define type
  -name   Person
  -struct [Name:[VName:string, NName: string],
           GebDat: date,
           Adresse: [ PLZ: integer, Ort: string,
                      Strasse: string],
           Hobbies: { string }]
end define;
```

In der folgenden DDL-Anweisung wird der Typ `Student` als Subtyp von `Person` definiert:

```
define type
  -name   Student
  -isa    Person
  -struct [MatNr: int,
          FB: int,
          Studiengang: string,
          Leistungen: {[Veranst: string, Note: int]}]
end define;
```

Es werden nur die zusätzlichen Eigenschaften, d.h. der „Differenztyp“ zu `Person` angegeben. ❒

Der Nutzen von *isa*-Beziehungen läßt sich wie folgt zusammenfassen:

- An eine *isa*-Beziehung ist ein Mechanismus für die *Vererbung* von strukturellen und operationalen Eigenschaften gekoppelt: Einen Subtyp hat mindestens die strukturellen Eigenschaften des Supertyps, d.h. für alle t, t' ∈ DEFTYPES gilt:
  t *isa* t' ⇒ *Attrs*(t') ⊆ *Attrs* (t)

13. Eine reflexive und transitive Relation ρ heißt *Halbordnung*, falls sie antisymmetrisch ist, d.h. es gilt: t ρ t' ∧ t' ρ t ⇒ t = t'.

Eine analoge Aussage gilt für die operationale Schnittstelle von Objekttypen, auf die wir jedoch erst im folgenden Kapitel eingehen.

- *Substituierbarkeit* [CW85]: Überall dort, wo eine Instanz eines bestimmten Typs t erwartet wird, darf auch eine Instanz einer seiner Subtypen t' eingesetzt werden, ohne daß dies zur Laufzeit zu einem Typfehler führt. Dies setzt natürlich voraus, daß sich im Subtyp t' alle strukturellen und operationalen Merkmale des Typs t finden, was gerade durch den an die Subtyp-Beziehung gekoppelten Vererbungsmechanismus gewährleistet ist.
- Durch die Beschränkung auf die Angabe des *„Differenztyps“* zum Supertyp sind redundanzfreie und übersichtliche Objekttypdefinitionen möglich.

Im Hinblick auf Anforderungen aus dem Software-Engineering hat die Einbeziehung von *isa*-Beziehungen in ein Datenmodell letztendlich das Ziel, der Forderung nach Wiederverwendbarkeit und einfacher Erweiterbarkeit von Softwaremoduln (und als solche sind Objekttypdefinitionen aufzufassen) zu entsprechen. Die Möglichkeit, Objekttypen und Subtyp-Beziehungen spezifizieren zu können, liefert natürlich keine Wiederverwendbarkeit und Erweiterbarkeit „frei Haus“. Es kommt auf den Aufbau langfristig Gültigkeit besitzender *isa*-Hierarchien an.

Man spricht von einfacher Vererbung (*single inheritance*), falls es zu jedem Typ t höchstens einen Typ t' geben darf, so daß t *isa* t' gilt. Darf es zu einem Typ t mehr als einen Typ t' geben mit der Eigenschaft t *isa* t', dann liegt dem Modell Mehrfachvererbung (*multiple inheritance*) zugrunde. Die Mehrfachvererbung ist problematisch, da es zu Namenskonflikten kommen kann [KA90]: Gilt t *isa* t_1 und t *isa* t_2 mit $t_1 \neq t_2$, und werden in t_1 und t_2 gleichnamige Eigenschaften (Attribute bzw. Operationen) deklariert oder von wiederum anderen Typen geerbt, dann ist unklar, von welchem Typ der Typ t die betreffende Eigenschaft erbt. Die gleichnamigen Eigenschaften können natürlich völlig unterschiedlich typisiert sein. Namenskonflikte entstehen i.d.R. dadurch, daß zum Zeitpunkt der Definitionen der Supertypen eine Mehrfachvererbung noch nicht abzusehen war. Für ESCHER$^+$ wollen wir uns – genauso wie bereits der Vorschlag aus [Pau94] – auf einfache Vererbung beschränken und gehen Namenskonflikten auf diese Weise aus dem Weg.

Gilt t *isa* t' und gibt es einen Attributnamen $a \in \mathit{Attrs}(t) \cap \mathit{Attrs}(t')$, dann wird a im Subtyp t von t' redefiniert. Semantisch soll dies eine Nicht-Vererbung des Attributwertes an das gleichnamige Attribut des Subtyps bedeuten. Ein Zugriff auf a bzgl. t liefert dann i.d.R. einen anderen Wert als der Zugriff auf a bzgl. t'. Es stellt sich nun die Frage, ob mit einer Redefinition eines Attributes a auch die ihm zugeordnete Struktur verändert werden darf. Streng typisierte OOPLs erlauben zwar die Redefinition von Attributen, d.h. die Nicht-Vererbung des Attributwertes, aber der Typ des Attributes darf nicht verändert werden. Sogar der Übergang zu einem Subtyp kann zu Typfehlern zur Laufzeit führen [Kem91, KM94 (Kapitel 10)]. Die Veränderung des Typs eines redefinierten Attributs muß verboten werden, wenn die OOPL zur Laufzeit nicht den deklarierten Typ eines Bezeichners für den Zugriff auf ein Attribut verwendet, sondern den tatsächlichen Typ des an den Bezeichner gebundenen Objekts.

Beispiel 3.8 (vgl. auch ein ähnliches Beispiel in [KM94 (Kapitel 10)])
Es seien S und T zwei Objekttypen mit S *isa* T, wobei S das Attribut a, das in Typ T selbst wieder den Typ T hat, mit dem Typ S redefiniert. Ferner seien `einS` und `einT` zwei Variablen, die an Objekte vom Typ S bzw. T gebunden sind.

Die Anweisungsfolge `einT.a := einS; einT.a.a := einT` führt zu einem Typfehler zur Laufzeit: `einT.a.a` hat den statischen Typ `T`, so daß die zweite Zuweisung typkorrekt ist. Nach der ersten Zuweisung hat `einT.a` jedoch den tatsächlichen Typ `S`, und somit erwartet die linke Seite `einT.a.a` der zweiten Zuweisung auf der rechten Seite ein Objekt vom Typ `S` oder eines Subtyps von `S`, bekommt aber ein Objekt des Supertyps `T` zugewiesen! Ein späterer Zugriff auf `einT.a.a.b` für ein in `S`, aber nicht in `T` vorkommendes Attribut `b` führt zu einem Laufzeitfehler. ❒

Der tatsächliche Typ eines Ausdrucks ist in den verbreiteten OOPLs immer der minimale Typ eines Objektes, unter dem das Objekt auch initialisiert wurde. In ESCHER$^+$ wollen wir jedoch dynamische Typzugehörigkeit ermöglichen (siehe folgenden Abschnitt). Für ein Objekt läßt sich dann kein minimaler Typ angeben, da es zu mehreren Subtypen eines Typs gehören darf, ohne daß diese wiederum einen gemeinsamen Subtyp haben müssen. In ESCHER$^+$ wird deshalb für Attributzugriffe immer der statische Typ eines Ausdrucks zugrundegelegt, und damit sind keine Restriktionen bei der Redefinition von Attributen notwendig.

Es folgt nun noch die genaue Beschreibung des Vererbungsmechanismus hinsichtlich der strukturellen Eigenschaften von Objekttypen. Die Subtyp-Beziehung *isa* bildet die Basis für die Vererbung von Eigenschaften von einem Supertyp an einen Subtyp. Die Interpretationsfunktionen I (t) für Objekttypen t berücksichtigen jedoch keine Vererbung, sondern geben nur die „typ-lokalen“ Eigenschaften des „Differenztyps“ wieder. Bei einem Zugriff auf ein Attribut könnte prinzipiell folgendermaßen vorgegangen werden: Zunächst wird das Attribut in I (t) gesucht. Wird es dort nicht gefunden, so wird die *isa*-Relation verfolgt und in I(t') gesucht, wobei t *isa* t' gilt. Die Suche wird entlang der *isa*-Relation so lange fortgesetzt, bis man fündig wird.

Um jedoch immer gleiche Suchvorgänge zu vermeiden, ist es natürlich günstiger, für jeden Objekttyp t eine Funktion zur Verfügung zu haben, die zu einem Attributnamen die notwendige Information für den Attributzugriff liefert. Für jeden Objekttyp t $\in$ DEFTYPES sei eine Funktion

$$\mathit{AttrAccess}\ (t) : \mathcal{N} \dashrightarrow \mathrm{DEFTYPES} \times \mathrm{IN} \tag{3.28}$$

definiert. Wird auf ein Attribut a eines getypten Objektes (ω, t) zugegriffen, dann gibt $\mathit{AttrAccess}(t)(a) = (t', n)$ zweierlei an:

- t' ist der Objekttyp mit t *isa** t', der das Attribut a in seinem Zustand I (t') enthält
- n $\in$ IN ist der Index, mit dem durch Aufruf von *get_child* (I (t') (ω), n) auf denjenigen Knoten r $\in$ *children* (I (t') (ω)) zugegriffen wird, der dem Attribut a zugeordnet ist.

Es folgt nun eine informelle Angabe des Algorithmus zur Bestimmung der Funktion *AttrAccess* (t) für einen Objekttyp t:

1. Setze $f_t : \mathcal{N} \dashrightarrow \mathrm{DEFTYPES} \times \mathrm{IN}$ mit
   $\mathrm{Def}(f_t) := \mathit{Attrs}(t)$ und $f_t\ (a) := (t, \mathit{attr\_pos}\ (\mathit{struct}\ (t))\ (a))$ für a $\in \mathrm{Def}(f_t)$.
2. Falls t *isa* t' für ein t' $\in$ DEFTYPES, dann führe durch:
   2.1. Bestimme *AttrAccess* (t') und setze $f_{t'} := \mathit{AttrAccess}\ (t')$
   2.2. Berücksichtige Redefinitionen von Attributen im Subtyp:
   $f_{t'}\ (a) := \perp$ für alle a $\in$ D := $\mathrm{Def}(f_t) \cap \mathrm{Def}(f_{t'})$.
   2.3. Setze $f_t := f_t \cup f_{t'}$

3. f_t ist die gesuchte Funktion *AttrAccess*(t).

In Schritt 1 werden die Funktionswerte *AttrAccess*(t)(a) für alle im Typ t definierten Attribute festgelegt. Schritt 2 sorgt dafür, daß auch die von den Supertypen geerbten Attribute berücksichtigt werden. Die Redefinition eines Attributes a im Typ t soll das gleichnamige Attribut eines Supertyps t' überdecken, deshalb wird in Schritt 2.2. das Attribut a aus dem Definitionsbereich von $f_{t'}$ herausgenommen, da im Fall der Redefinition f_t (a) der korrekte Wert für *AttrAccess*(t)(a) ist.

Für den schnellen Zugriff auf ein Attribut eines getypten Objektes (ω, t) zur Laufzeit ist es sinnvoll, für jedes t ∈ DEFTYPES die Funktion *AttrAccess*(t) in Form einer Tabelle bereitzuhalten.

3.7.2 Dynamische Spezialisierung

Bereits in Abschnitt 1.3 wurde darauf hingewiesen, daß die Typzugehörigkeit von Objekten in vielen Datenmodellen rein statisch ist: Der Typ jedes Objekts ist dann sein Instanziierungstyp, der auch als sein minimaler Typ bezeichnet wird. Da ein Objekt ω vom Typ t überall dort eingesetzt werden kann, wo ein Objekt vom Typ t' mit t *isa* t' erwartet wird, besitzt ω gleichzeitig die Typen aus { t' | t *isa* t' }, d.h. wir können bereits von multipler Typzugehörigkeit von ω sprechen. Diese Menge liegt für ω jedoch ein für alle mal fest, da sie durch den Instanziierungstyp bestimmt ist.

Wir wollen in ESCHER$^+$ die multiple Typzugehörigkeit jedoch wesentlich flexibler gestalten. In vielen Fällen ist eine dynamische Veränderung der Typzugehörigkeit wünschenswert. Wir verweisen wiederum auf das Beispiel aus Abb. 1.7: Für ein Objekt vom Typ `Person` soll es die Möglichkeit geben, später noch den Typ `Student` und/oder `Angestellter` zu aquirieren, so daß sich die Menge

$$types(\omega) := \{t \in \text{DEFTYPES} \mid \omega \in dom(t)\} \tag{3.29}$$

dynamisch ändert. Genauer unterscheiden wir nach *persistenten* und *transienten* Typen eines Objektes ω. Wir definieren daher zusätzlich:

$$pers\_types(\omega) := \{\, t \in types(\omega) \mid pref(I(t)(\omega)) > 0 \,\}$$
$$trans\_types(\omega) := \{\, t \in types(\omega) \mid pref(I(t)(\omega)) = 0 \,\}$$

Das Hinzufügen eines weiteren Typs t' zu einem bereits initialisierten Objekt ω geschieht über den Aufruf einer Basisoperation namens `addType`. Ihre Parameter sind ω und der Name des Typs t'. Die Semantik des Aufrufes von `addType` mit den Parametern ω und t' läßt sich in verbaler Form wie folgt angeben:

- Zunächst wird getestet, ob ω ∈ Def(I (t')) gilt. Falls dies der Fall ist (d.h. es gilt bereits t' ∈ *types*(ω)), dann wird eine Ausnahme ausgelöst und der Aufruf beendet.
- Für alle t'' ∈ DEFTYPES mit t' *isa** t'' und t'' ∉ *types*(ω) wird
  I (t'') (ω) := *InitVal* (*struct* (t'')) und *pref*(I(t'')(ω)) := 0 gesetzt.
- Zurückgegeben wird das getypte Datenobjekt (ω, t').

Damit Typkorrekheit zur Laufzeit bei Substituierbarkeit zugesichert werden kann, muß natürlich für alle ω ∈ Ω gelten:

$$t \in types(\omega) \Rightarrow t' \in types(\omega) \text{ für alle } t' \in \text{DEFTYPES mit } t \; isa^* \; t' \tag{3.30}$$

Ein Objekt vom Typ `Student` *muß* also gleichzeitig vom Typ `Person` sein. Die Bedingung (3.30) wird durch einen Aufruf von `addType` nicht verletzt, wenn man gleichzeitig fordert, daß `initObject` nur für *isa*-Wurzeltypen aufgerufen wird. Die Operation `addType` gehört genauso wie `initObject` zu den „primitiven“ Basisoperationen von ESCHER$^+$, die im folgenden Kapitel detailliert besprochen werden. Sie werden nicht direkt vom Anwender aufgerufen, sondern dieser formuliert Anweisungen und Ausdrücke einer „höheren“ Sprache (wie z.B. der Sprache SCRIPT aus [Pau94]), die dann in entsprechende Aufrufe von `initObject` bzw. `addType` umgesetzt werden. Auf die notwendigen höhersprachlichen Konstrukte gehen wir in Abschnitt 4.7.4 näher ein.

Man beachte auch, daß – ähnlich wie bei `initObject` – die durch `addType` hinzugefügten Typen zunächst nur transiente Typen sind, d.h. sie liegen in *trans_types* (ω).

3.8 Tabellen

Die Instanzen im eNF^2-Modell bilden die sog. *Tabellen*, die bei AIM-P über eine DDL-Anweisung `CREATE` s `END` mit s ∈ $\mathcal{L}$(*struct*) definiert werden [LPS91]. Eine Tabelle ist in unserer Terminologie nichts anderes als ein benamtes getyptes Datenobjekt (v, s). Dies ist eine direkte Verallgemeinerung der `CREATE TABLE`-Anweisung von SQL.

Tabellen fungieren in den genannten Modellen als Einstiegspunkte in die Datenbank. Beim Relationenmodell erfolgt über eine Tabelle die Manipulation der in ihr enthaltenen Tupel, die die „eigentlichen Objekte“ einer Anwendung darstellen. Auch im eNF^2-Datenmodell ist eine Tabelle in den meisten Fällen eine Kollektion von Datenobjekten.

Für Modelle wie ESCHER$^+$, die die Definition von Objekttypen ermöglichen, muß insbesondere geklärt werden, wie die existierenden Objekte *logisch* verwaltet werden sollen. Zu dieser Fragestellung können zwei grundsätzliche Alternativen unterschieden werden, die bereits in Abschnitt 2.5.1 im Zusammenhang mit dem Begriff „Klasse“ erläutert wurden:

- **Alternative 1**: Das Datenmodell ermöglicht den Zugriff auf die komplette Typextension eines Objekttyps, die automatisch verwaltet wird. Der Name der Typextension stimmt mit dem Namen des Typs überein.
- **Alternative 2**: Das Datenmodell ermöglicht den Zugriff auf benamte Kollektionen, die Mengen von Objekten eines bestimmten Typs beinhalten und deren Verwaltung (d.h. Einfügen und Entfernen von Objekten) unter Benutzerkontrolle steht.

Für ESCHER$^+$ greifen wir die zweite Alternative auf, weil sie besser im Einklang mit der Tabellen-„Philosophie“ des eNF^2-Modells bzw. des ESCHER-Prototyps steht. Dort werden Anwendungsobjekte meist durch Tupel repräsentiert, die der Benutzer selbst in Tabellen einfügen und auch wieder löschen muß. Dieses Prinzip soll auch für die Instanzen von Objekttypen beibehalten werden. Damit schließen wir uns auch der Meinung des „Manifests“ [ADB+89] an, das automatisch verwaltete und für den Benutzer zugängliche Typextensionen nicht für notwendig hält.

Der ESCHER-Prototyp trennt zwischen sog. *Schemata* einerseits und *Tabellen* andererseits. *Schemata* sind benamte Strukturen s ∈ $\mathcal{S}$ (TYPES), wobei hier TYPES = BASETYPES ∪ CONSTR gilt. Zu einem Schema lassen sich im ESCHER-Prototyp beliebig viele *Tabellen* anle-

gen. Eine Schemadefinition kann mit einer Objekttypdefinition verglichen werden, und jede Tabelle ist dann ein über `initObject`(s, v_i) initialisiertes Objekt ω_i. Ein Zugriff auf die einzelnen Tabellenwerte *I* (s) (ω_i) = v_i geschieht im Prototyp über Tabellennamen n_i, die den Tabellen ω_i eindeutig zugeordnet sind. Allerdings ist die Gleichsetzung der Schemata des ESCHER-Prototyps und der Objekttypen, wie sie für ESCHER$^+$ definiert wurden, problematisch. Ein ESCHER-Schema ist – als Struktur einer Tabelle – typischerweise eine Kollektion von Datenobjekten. Die Objekttypen wurden jedoch gerade in der Absicht eingeführt, daß sie die unterschiedlichen „Sorten“ von Anwendungsobjekten wiedergeben und nicht *Kollektionen* von Anwendungsobjekten. Die Wiederverwendbarkeit z.B. einer Schemadefinition `Buecher` zur Struktur `{[Signatur: string, Autoren: …, …]}` ist als gering einzustufen, während ein Objekttyp `Buch` mit der Struktur `[Signatur: string, Autoren: …, …]` tatsächlich den Typ des Anwendungsobjektes „Buch“ wiedergibt, der an verschiedenen Stellen im Datenbankschema Verwendung finden kann. Für ESCHER$^+$ wird im Gegensatz zum ESCHER-Prototyp deshalb auf die Einführung benamter Strukturen verzichtet.

Mit der folgenden Definition werden für ESCHER$^+$ Tabellen als *die* „Einstiegspunkte“ in die Datenbank eingeführt.

Definition 3.19 (Tabelle)

Es sei eine endliche Menge TABLES $= \{\tau_1, \ldots, \tau_n\}$ gegeben, deren Elemente *Tabellen* genannt werden. Auf TABLES seien folgende Funktionen definiert:

$$\textit{name}: \text{TABLES} \rightarrow \mathcal{N},$$

die jedem $\tau \in$ TABLES einen eindeutigen Namen zuordnet.

$$\textit{struct}: \text{TABLES} \rightarrow \mathcal{S}\,(\text{TYPES}), \tag{3.31}$$

wobei gelten soll:

$$\forall\, \tau \in \text{TABLES}\textit{: struct}\,(\tau) \neq [\,] \tag{3.32}$$

Mit der Definition einer Tabelle soll gleich eine Instanz in Form eines zu ihrer Struktur passend getypten Wertes verbunden werden. Um dies formal zu erfassen, sei

$$\textit{inst}: \text{TABLES} \rightarrow \mathcal{V}al^*(\text{TYPES}) \tag{3.33}$$

eine *Instantiierungsfunktion*, die jeder Tabelle τ ein getyptes Datenobjekt (v, *struct* (τ)) als ihre *Instanz* zuordnet. ❐

Wie bereits für alle Objekttypen, so soll auch für alle $\tau \in$ TABLES ein eindeutig bestimmter Strukturbaum aus $\varphi_{\text{struct}}(\textit{struct}(\tau))$ festgelegt sein, den wir mit $B_{\textit{struct}(\tau)}$ bezeichnen. Die Funktion *struct* aus (3.31) läßt sich somit auch als Funktion auffassen, die den Wurzelknoten $\textit{root}(B_{\textit{struct}(\tau)})$ liefert. Analoges gilt auch für die Funktion *inst* aus (3.33): Sie kann auch so interpretiert werden, daß sie den Wurzelknoten r eines Wertebaumes B_v mit $(B_v, B_{\textit{struct}(\tau)}) \in \varphi_{\text{val}}((v, \textit{struct}\,(\tau)))$ liefert.

Beispiel 3.9 Unter Verwendung des Objekttyps `Buch` aus Beispiel 3.6 wird mit

```
define table
  -name   Bestand
  -struct {Buch}
end define;
```

eine Tabelle `Bestand` definiert, mit der eine Menge von `Buch`-Objekten verwaltet werden kann. Die Ausführung dieser DDL-Anweisung führt dazu, daß TABLES um ein neues Element $\tau_{Bestand}$ erweitert wird, wobei *name* ($\tau_{Bestand}$) = "`Bestand`", *struct* ($\tau_{Bestand}$) = $c_{set}(t_{Buch})$ und *inst* ($\tau_{Bestand}$) = *initVal* (*struct* ($\tau_{Bestand}$)) gesetzt wird. ❒

Eine Tabellendefinition unterscheidet sich offensichtlich nur wenig von einer Objekttypdefinition. Gegenüber einer Typdefinition haben wir die Bedingung (3.22) zu (3.32) geändert, d.h. Tabellen sind nicht auf Tupelstrukturen beschränkt, und ihre Struktur darf nicht das leere Tupel sein[14]. Fassen wir eine Tabelle τ nun als „Typdefinition“ auf, dann haben wir mit *inst* (τ) bereits einen Zustand eines Objektes dieses „Typs“. Das zugehörige Objekt ist natürlich die Tabelle selbst! Wir können also eine Interpretationsfunktion $I(\tau)$ für den „Typ“ τ definieren, indem wir $Def(I(\tau)) = \{\tau\}$ und $I(\tau)(\tau) := inst(\tau)$ setzen. Ferner setzen wir $pref(I(\tau)(\tau)) := 1$. Da später genau diejenigen Objekte ω als persistent gelten, für die $pers\_types(\omega) \neq \emptyset$ ist, sind Tabellen bereits per definitionem persistent.

Die Tabellen in ESCHER$^+$ haben gegenüber den Klassen aus objektorientierten Datenmodellen den Vorteil, daß für Tabellen variablere Strukturierungsmöglichkeiten vorliegen. Während Klassen immer nur *Mengen* von *Objekten* repräsentieren, sind Tabellen weitaus flexibler: Mit der Freiheit bei der Wahl der Struktur ergibt sich auch die Möglichkeit differenzierterer Gruppierung, Assoziierung und Aggregation von Objekten und Werten. Denkbar ist z.B. eine Tabelle `Angestellte`, die die Struktur

```
{[Mitarb: Person, Beurteilungen:
 <[Datum: date, Text:string]>]}
```

besitzt und in der neben den „offiziellen“ Mitarbeiterdaten, die durch den Objekttyp `Person` gegeben sind, „private“ Beurteilungen erfaßt werden können, ohne daß dazu der Objekttyp `Person` erweitert werden muß.

Es ist auch möglich, eine Tabelle zu definieren, deren Wert keine Kollektion ist, sondern z.B. ein „einfaches“ Tupel oder nur ein integer-Wert. Obwohl der Begriff „Tabelle“ in diesem Fall nicht angemessen erscheint, wollen wir unter Berücksichtigung des Regelfalles (eine Tabelle als Kollektion von Datenobjekten) an der Bezeichung Tabelle festhalten.

In einer `define table`-Anweisung kann optional eine `init`-Klausel verwendet werden (vgl. Anhang B.2). Diese dient – vergleichbar mit der `default`-Klausel für Objekttypen – der Initialisierung einer Tabelle mit einem anderen Wert als den durch *initVal* gegebenen. Dies kann notwendig sein, wenn für eine Tabelle Integritätsbedingungen (vgl. Kapitel 5) definiert werden, die bei der Initialisierung der Tabelle sofort verletzt wären. Ist beispielsweise `eineZahl` eine Tabelle mit der Struktur `integer` und soll der Wert von `eineZahl` größer oder gleich 0 sein, dann könnte bei der Definition der Tabelle `-init 0` angegeben werden, um die Integritätsbedingung zu erfüllen.

Es gibt einige Datenmodelle, die beide zu Beginn dieses Abschnittes angesprochenen Alternativen in sich vereinen. So kann z.B. im Datenmodell von O_2 [Deu91, BDK92] bei einer Typdefinition[15] explizit angegeben werden, daß eine benamte Typextension verwaltet wird („`with`

14. Das leere Tupel ist nur im Zusammenhang mit Objekttypdefinitionen relevant (vgl. die Abschnitte 3.6 und 3.7)

15. Man beachte, daß die O_2-Begriffe class bzw. type den ESCHER$^+$-Begriffen Objekttyp bzw. Struktur entsprechen.

`extension`“-Klausel). Daneben gibt es in O_2 jederzeit die Möglichkeit, weitere Namen einzuführen, an die Datenobjekte gebunden werden können, und alle über einen solchen Namen erreichbaren Datenobjekte sind ebenfalls persistent.

3.9 Datenbankschema und Datenbankinstanz

In diesem Abschnitt geben wir eine erste Definition der Begriffe Datenbankschema und Datenbankinstanz. Die Definition ist insofern noch unvollständig, als der operationale Aspekt und Integritätsbedingungen bislang ausgespart blieben.

Wir gehen von der a priori-Existenz der Mengen BASETYPES und CONSTR aus.

Definition 3.20 (Datenbankschema, Datenbankinstanz)

Ein *Datenbankschema* DS ist gegeben durch

- eine Menge ENUMTYPES von Aufzählungstypen,
- eine Menge DEFTYPES von Objekttypen
- eine Menge VARTYPES von varianten Typen
- eine nicht-leere Menge TABLES von Tabellen
- eine Subtyp-Beziehung $isa \subseteq \text{DEFTYPES} \times \text{DEFTYPES}$

Eine *Datenbankinstanz* INST(DS) zu einem Datenbankschema DS ist gegeben durch

- eine Instanzenfunktion *inst* : TABLES $\cdots>$ $\mathcal{V}al^*$(TYPES), wobei gelten muß:
  $\forall\, \tau \in$ TABLES: *inst* $(\tau) = (v, struct\,(\tau))$ für ein $v \in dom(struct(\tau))$.
- eine Menge $\{I\,(t) : \Omega \cdots> \mathcal{V}al^*(\text{TYPES}) \mid t \in \text{DEFTYPES}\}$ von Interpretationsfunktionen, wobei gelten muß:
  $\forall\, t \in$ DEFTYPES $\forall\, \omega \in$ Def(I (t)): I (t) $(\omega) = (v, struct\,(t))$ für ein $v \in dom(struct(t))$.

❒

Da jedes getypte Datenobjekt aus $\mathcal{V}al^*$(TYPES) äquivalent durch eine Baumrepräsentation angegeben werden kann und umgekehrt jede Baumrepräsentation aus $\mathcal{V}al\mathcal{T}ree$(TYPES) ein getyptes Datenobjekt repräsentiert, kann eine Datenbankinstanz INST(DS) auch auf der Grundlage einer Menge von Baumrepräsentationen definiert werden. Dazu müssen wir die Sichtweise der Funktionen I(t) bzw. *inst* ändern: Anstelle getypter Datenobjekte $(v,s) \in \mathcal{V}al^*$(TYPES) sollen ihre Funktionswerte nun Wurzelknoten von Wertebäumen B_v mit $(B_v, B_s) \in \varphi_{val}((v, s))$ liefern. Dies führt uns zu folgender **alternativen** Definition.

Definition 3.21 (Datenbankinstanz – alternative Definition)

Eine *Datenbankinstanz* INST(DS) zu einem Datenbankschema DS ist gegeben durch

- eine endliche Teilmenge $D \subseteq \mathcal{V}al\mathcal{T}ree$(TYPES) von paarweise disjunkten Baumrepräsentationen getypter Datenobjekte.
- eine Instanzenfunktion *inst* : TABLES $\rightarrow \mathcal{V}$, wobei gelten muß:
  - Jeder Tabelle τ wird eine Baumrepräsentation aus D zugeordnet, d.h. $inst(\tau)$ „zeigt“ auf den Wurzelknoten des Wertebaumes einer Baumrepräsentation

aus D. Formal:
Bild(*inst*) $\subseteq$ {*root* (B_v) | (B_v,B_s)$\in$ D}

– Für jede Tabelle τ „zeigt“ *struct*(*inst*(τ)) auf den Wurzelknoten des zu τ gehörenden, eindeutig bestimmten Strukturbaumes $B_{struct(\tau)}$. Formal:
$\forall \tau \in$ TABLES: *struct*(*inst*(τ)) = *root*($B_{struct(\tau)}$)

- eine Menge {*I* (t) : Ω ···> $\mathcal{V}$ | t $\in$ DEFTYPES} von Interpretationsfunktionen, wobei gelten muß:
  – Für jeden Objekttyp t und jedes t-Objekt ω ist die Interpretation *I*(t)(ω) durch eine Baumrepräsentation aus D gegeben, d.h. *I*(t)(ω) „zeigt“ auf den Wurzelknoten des Wertebaumes einer Baumrepräsentation aus D. Formal:
  $\bigcup_{t \in DEFTYPES}$ Bild(*I* (t)) $\subseteq$ {*root* (B_v) | (B_v,B_s)$\in$ D},
  – An jeden Wertebaum, auf den *I*(t)(ω) „zeigt“, ist immer derselbe eindeutig bestimmte Strukturbaum $B_{struct(t)}$ „gekoppelt“. Formal:
  $\forall$ t $\in$ DEFTYPES $\forall$ $\omega \in$ Def(*I* (t)): *struct* (*I* (t) (ω)) = *root*($B_{struct(t)}$)
- Wir lassen kein „Wertebaum-Sharing“ zu: Die Funktionswerte der Instanzenfunktionen *inst*(τ) und der Interpretationsfunktionen „zeigen“ auf paarweise verschiedene Wertebäume.
  Setzt man *I*(τ)(τ) := *inst*(τ) für $\tau \in$ TABLES, dann läßt sich dies formal wie folgt ausdrükken:
  – $\forall$ t, t' $\in$ DEFTYPES $\cup$ TABLES, $\forall$ ω, ω' $\in$ Ω:
  $\omega \in$ Def(*I*(t)) $\wedge$ ω' $\in$ Def(*I*(t')) $\wedge$ (t $\neq$ t' $\vee$ $\omega \neq \omega$') $\Rightarrow$ *I* (t) (ω) $\neq$ *I* (t') (ω')

Die letzte Bedingung stellt sicher, daß jedem Funktionswert *I*(t)(ω) eine eigene, separate Baumrepräsentation zugeordnet wird, wodurch unerwünschte „Nebeneffekte“ bei Update-Operationen verhindert werden.

❒

Die zweite Definition einer Datenbankinstanz wird die Grundlage für alle weiteren Ausführungen sein, insbesondere für die Beschreibung des operationalen Teils des Datenmodells im folgenden Kapitel. In Abb. 3.8 ist eine Datenbankinstanz graphisch veranschaulicht. Die von den Tabellen τ_i ausgehenden Pfeile führen zu den Funktionswerten *inst*(τ_i). Alle sonstigen Pfeile haben ihren Ausgangspunkt bei einem getypten Datenobjekt (ω, t) mit t $\in$ DEFTYPES und verweisen auf den Baum mit dem Wurzelknoten *I* (t) (ω), der den strukturellen Zustand von ω relativ zu t repräsentiert. Grau unterlegt sind alle Bäume aus D, die eine Tabelleninstanz repräsentieren. Alle anderen Bäume sind „Zustände“ von Objekten.

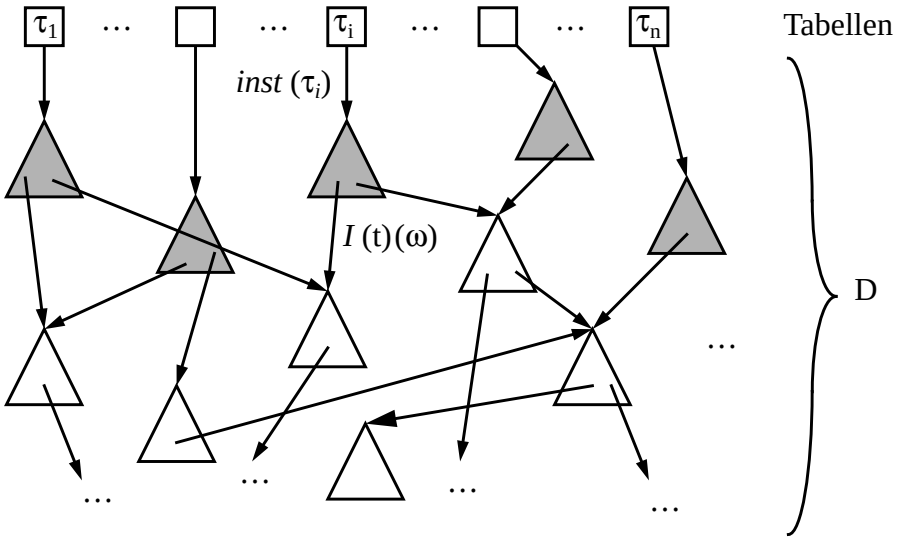


Abbildung 3.8: Schematische Darstellung einer ESCHER$^+$-Datenbankinstanz

Es wurde bereits erwähnt, daß wir in ESCHER$^+$ Persistenz eng an den Begriff der Erreichbarkeit über eine Tabelle koppeln wollen. Wir kommen deshalb zur Definition des Begriffs „Erreichbarkeit“. Die Erreichbarkeit ist ein wichtiges Kriterium, das uns ermöglicht, die Menge aller möglichen Datenbankinstanzen zu einem gegebenen Schema auf die „sinnvollen“, d.h. zulässigen einzuschränken.

Definition 3.22 (Erreichbarkeit, zulässige Datenbankinstanz)

Es sei $D \subseteq \mathcal{ValTree}$ (TYPES) die Menge der Baumrepräsentationen einer Datenbankinstanz INST(DS) gemäß Def. 3.20. Ferner sei $V(D) := \{r \in B_v \mid (B_v, B_s) \in D\}$.

Die *Erreichbarkeit* von Knoten $r \in V(D)$ ist wie folgt definiert:

(i) Alle Knoten $r \in \bigcup_{\tau \in \text{TABLES}} V(\mathit{Tree}(\mathit{inst}(\tau)))$ sind erreichbar[16]

(ii) Wenn ein Knoten r' erreichbar ist, der ein getyptes Datenobjekt (ω, t) mit $t \in$ DEFTYPES repräsentiert, dann kann ω relativ zu t interpretiert werden, d.h. alle Knoten in dem Wertebaum, auf den $I(t)(\omega)$ „zeigt“, sind ebenfalls erreichbar.
Formal:
$\exists\, r' \in V(D) : r'$ ist erreichbar $\wedge\ r' = (\omega, t)$ mit $t \in$ DEFTYPES $\wedge\ \omega \in \mathrm{Def}(I(t))$
$\Rightarrow$ alle Knoten $r \in \mathit{Tree}(I(t)(\omega))$ sind erreichbar

Eine Datenbankinstanz INST(DS) eines Datenbankschemas DS heißt *zulässig*, wenn gilt:

(i) Alle Knoten in V(D) sind erreichbar

(ii) Falls $r' \in V(D)$ erreichbar $\wedge\ r' = (\omega, t)$ mit $t \in$ DEFTYPES
$\Rightarrow t' \in \mathit{types}(\omega)$ für alle t' mit $t\ \mathit{isa}^*\ t'$

(iii) $\mathit{pref}(I(\tau)(\tau)) = 1$ für $\tau \in$ TABLES,

16. Dies sind gerade alle Knoten in einem der grau eingefärbten Bäume in Abb. 3.8.

$$pref(I(t)(\omega)) = |\{r \in V(D) \mid r = (\omega, t') \wedge t \; isa^* \; t'\}|,$$
falls $t \in$ DEFTYPES und $\omega \in Def(I(t))$

Eine Baumrepräsentation $(B_v, B_s) \in D$ und alle Knoten in $V(B_v)$ gelten als *persistent*, wenn $pref(root(B_v)) > 0$. ❒

Die Bedingung (i) für Zulässigkeit einer Datenbankinstanz besagt, daß eine Datenbankinstanz keine unzugängliche Information enthalten darf.

Bedingung (ii) besagt, daß es keine *dangling* (*object*) *references* gibt: Ist ein Knoten $r = (\omega, t)$ erreichbar, so ist durch (ii) garantiert, daß von diesem Knoten aus ω bezüglich t und aller Supertypen t' von t interpretiert werden kann.

Bedingung (iii) macht genaue Aussagen über den Zusammenhang zwischen den vorhandenen „Objekt-Knoten“ $r = (\omega, t)$ und den „Zählern“ $pref(I(t)(\omega))$: Ihr Wert ist für Tabellen konstant 1. Für sonstige Objekte ω und Objekttypen t soll $pref(I(t)(\omega))$ tatsächlich die Anzahl der persistenten Referenzen auf $I(t)(\omega)$ angeben. Da bei einem Zugriff auf ω bezüglich t auch auf alle Supertypen t' von t zugegriffen werden darf, berücksichtigen die „Zähler“ auch die *isa*-Hierarchie.

Ist D eine zulässige Datenbankinstanz, dann sind alle Elemente aus D persistent: Ist r der Wurzelknoten eines Tabellenbaumes, dann gilt $pref(r) = 1$. Ansonsten folgt aus Bedingung (i), daß es einen erreichbaren Knoten $r' = (\omega, t)$ gibt mit $I(t)(\omega) = r$. Dann ist $pref(r) > 0$.

Im folgenden gehen wir immer von einer zulässigen Datenbankinstanz INST(DS) aus und schreiben statt D auch D_{pers}, um auszudrücken, daß alle Elemente von D_{pers} persistente Datenobjekte repräsentieren.

Ein mittels `initObject` erzeugtes Objekt ist noch nicht persistent, da der *pref*-Zähler zunächst mit 0 initialisiert wird und das Objekt nicht automatisch über eine Tabelle erreichbar ist. Ein z.B. mittels `initObject(Buch)` initialisiertes `Buch`-Objekt wird persistent, wenn es etwa in die Tabelle `Bestand` aus Beispiel 3.9 eingefügt[17] und somit der *pref*-Zähler um 1 erhöht wird. Analoges gilt auch für das Hinzufügen weiterer Typen zu einem existierenden Objekt mittels `addType`.

Während in ESCHER$^+$ jedes Objekt eines Typs aus DEFTYPES potentiell persistent ist und Persistenz *implizit* über Erreichbarkeit definiert wird, muß Persistenz in vielen Datenmodellen, die als persistente Erweiterung einer OOPL entstanden sind, *explizit* spezifiziert werden.

Beim kommerziellen OODBMS VERSANT sind nur solche Typen persistent, die als Subtyp von *Persistent* deklariert werden. Ähnlich verhält es sich bei ONTOS [AHS91]. POET [Poe96], eine verbreitete objektorientierte DBPL auf C++-Basis, verlangt die Deklaration eines Typs als `PERSISTENT CLASS`, um Persistenz für die Instanzen des Typs zu erlangen. Die Object Definition Language (ODL) des ODMG’93-Standards [Cat+94] verlangt bei jeder Typdeklaration ebenfalls die Angabe `transient` oder `persistent`. Transiente Typen sind dort als Komponententypen eines persistenten Typs zugelassen. Allerdings ist dies mit einem unerfreulichen Nebeneffekt verbunden: Nach dem Commit einer Transaktion werden die Referenzen zu transienten Objekten durch Nullwerte ersetzt, d.h. ein transientes Objekt kann nie persistent werden.

---

17. In der Sprache SCRIPT aus [Pau94] geschieht dies über die Anweisung
`Bestand add Buch(`*Initialisierungswerte*`)`

Bei einer weiteren Gruppe sind zwar die Instanzen aller Typen potentiell persistent, aber der Benutzer muß beim Erzeugen eines Objektes angeben, ob dieses persistent oder transient sein soll. Dazu gehören z.B. die Datenmodelle der OODBMS ObjectStore [LLO+91] und ODE [AG89]. Persistente Objekte werden in ObjectStore mit `new(db)` erzeugt, wobei `db` auf die aktuell geöffnete Datenbank verweist. In ODE gibt es die Funktion `pnew` zur Erzeugung persistenter Objekte.

Persistenz über Erreichbarkeit ist auch im OODBMS O_2 [Deu91, BDK92] realisiert, wobei die Menge der „Einstiegspunkte“ durch Bindungen von Namen an Datenobjekte jederzeit vergrößert werden kann.

3.10 Metamodellierung

In diesem Abschnitt wollen wir für ESCHER$^+$ ein spezielles Datenbankschema entwickeln, das wir als *Meta-Schema* bezeichnen und das folgende Eigenschaft besitzt:

- Jedes benutzerdefinierte Datenbankschema (siehe Def. 3.20) läßt sich als Instanz des Meta-Schemas interpretieren
- Das Meta-Schema ist ebenfalls eine Instanz des ESCHER$^+$-Datenmodells und läßt sich somit als Instanz von sich selbst auffassen

Die erste Eigenschaft fordert, daß das Meta-Schema gewissermaßen die Schablone des ESCHER$^+$-Datenmodells ist, in die sich jedes anwendungsspezifische Datenbankschema einfügen muß. Die zweite Eigenschaft stellt eine Form der Selbstreferenzierung dar, die eine besondere Bedeutung für die Implementation des Datenmodells hat: Schemadaten, die ebenfalls persistente Daten sind, werden hinsichtlich ihrer persistenten Speicherung genauso behandelt wie die Daten einer Datenbankinstanz zu einem benutzerdefinierten Datenbankschema. Schemainformation kann dann über die Anfragesprache des Datenmodells erfragt und über die DML modifiziert werden. Es ist mittlerweile in vielen Implementationen von DBMS üblich, Schemainformation in einem Meta-Schema abzulegen. Dies wird von den meisten relationalen DBMS bereits seit längerem praktiziert, indem Schemainformation in speziellen Systemtabellen abgelegt wird. Im Kontext objektorientierter Datenmodelle wird z.B. für das Datenmodell COCOON [SLR+92] in [TS92] ein Meta-Schema angegeben.

Im folgenden geben wir für das in diesem Kapitel entwickelte Datenmodell ein Meta-Schema an.

Die Objekttypen des Meta-Schemas

Alle Elemente der Mengen BASETYPES, CONSTR, ENUMTYPES, DEFTYPES, VARTYPES und TABLES sollen Instanzen von Objekttypen des Meta-Schemas und somit Objekte aus Ω sein. Im Meta-Schema werden deshalb die Objekttypen $t_{basetype}$, t_{constr}, $t_{enumtype}$, $t_{deftype}$, $t_{vartype}$ und t_{table} definiert.

Die Struktur dieser Objekttypen erhält man, wenn man die auf den Elementen der oben genannten Mengen definierten Funktionen zu Komponenten einer Tupelstruktur macht. So setzen wir z.B. $struct(t_{table})$ = [name: β_{string}, struct: $t_{structure}$]. Die Instanzen des Typs $t_{structure}$ repräsentieren

sentieren dabei die Strukturen $s \in \mathcal{S}(\text{TYPES})$. Auf diesen Objekttyp $t_{structure}$ gehen wir weiter unten detaillierter ein.

Die Tabellen des Meta-Schemas

Jede Tabelle ist eine Instanz des Objekttyps t_{table}. Auf der Meta-Ebene sind genau drei Tabellen $\tau_{BASETYPES}$, τ_{CONSTR} und $\tau_{DBSCHEMATA}$ definiert. Für sie gilt:

$I(t_{table})(\tau_{BASETYPES})$ = [name: "BASETYPES", struct: $\{t_{basetype}\}$]

$I(t_{table})(\tau_{CONSTR})$ = [name: "CONSTR", struct: $\{t_{constr}\}$]

$I(t_{table})(\tau_{DBSCHEMATA})$ = [name: "DBSCHEMATA", struct: $\{t_{dbschema}\}$]

Die Instanzen der beiden erstgenannten Tabellen stimmen mit den gleichnamigen Mengen aus Def. 3.20 überein.

Die Instanz $inst(\tau_{DBSCHEMATA})$ der Tabelle $\tau_{DBSCHEMATA}$ soll gerade die Menge aller Datenbankschemata darstellen. Auch das Meta-Schema selbst ist in dieser Menge enthalten! Es werde durch das Objekt $\omega_{Meta} \in \Omega$ repräsentiert. Für die Tabelle $\tau_{DBSCHEMATA}$ ist also $inst(\tau_{DBSCHEMATA}) = \{\omega_{Meta}, \omega_1, \ldots, \omega_n\}$, wobei die $\omega_i \neq \omega_{Meta}$ gerade die anwendungsspezifischen Datenbankschemata sind. Das bedeutet aber auch, daß alle $\omega \in inst(\tau_{DBSCHEMATA})$ Instanzen eines weiteren Objekttyps $t_{dbschema}$ der Meta-Ebene sein sollen.

Der Objekttyp $t_{dbschema}$

Jedes Datenbankschema soll eine Instanz des Objekttyps $t_{dbschema}$ sein. Für diesen Objekttyp soll gelten:

- Es ist $struct(t_{dbschema})$ =
  [name: string, DEFTYPES: $\{t_{deftype}\}$, SUBTYPES: $\{t_{subtype}\}$,
  ENUMTYPES: $\{t_{enumtype}\}$, VARTYPES: $\{t_{vartype}\}$, TABLES: $\{t_{table}\}$]
- Die Tupelkomponente SUBTYPES aus der Struktur von $t_{dbschema}$ hat keine Entsprechung in Def. 3.20. Sie gibt für jedes Datenbankschema die Menge aller derjenigen Objekttypen an, die Subtyp eines anderen Objekttyps sind.
- $Def(I(t_{dbschema})) = \{\omega_{Meta}, \omega_1, \ldots, \omega_n\} = inst(\tau_{DBSCHEMATA})$
- $I(t_{dbschema})(\omega_{Meta})$ =
  [name: "META",
  DEFTYPES: $\{t_{type}, t_{basetype}, t_{constr}, t_{enumtype}, t_{vartype}, t_{deftype}, t_{subtype}, t_{table}, t_{dbschema}, t_{struct}\}$,
  SUBTYPES: $\{t_{basetype}, t_{constr}, t_{enumtype}, t_{vartype}, t_{deftype}, t_{subtype}\}$
  ENUMTYPES: {}, VARTYPES: {},
  TABLES: $\{\tau_{BASETYPES}, \tau_{CONSTR}, \tau_{DBSCHEMATA}\}$].
- Ein neues Datenbankschema ω_i wird initialisiert mit $I(t_{dbschema})(\omega_i)$ =
  [name: *null*, DEFTYPES: {}, SUBTYPES: {}, ENUMTYPES: {},
  VARTYPES: {}, TABLES: {}].
  Die Definition eines Objekttyps, einer Tabelle usw. führt zur Erzeugung eines neuen Objekts, das in die Menge der relevanten Tupelkomponente eingefügt wird.

Subtyp-Beziehungen im Meta-Schema

Die *isa*-Beziehung zwischen Objekttypen wollen wir über multiple Typzugehörigkeit modellieren: Ist t direkter Subtyp eines anderen Objekttyps, dann soll t eine Instanz des Objekttyps $t_{subtype}$ sein. Für diesen sei $struct(t_{subtype}) = [isa: t_{deftype}]$.
Genauer fordern wir: Falls t *isa* t', dann ist $I(t_{subtype})(t) = [isa: t']$.

Da $t_{subtype}$ selbst Subtyp von $t_{deftype}$ ist, gilt $t_{subtype} \in Def(I(t_{subtype}))$ und
$I(t_{subtype})(t_{subtype}) = [isa: t_{deftype}]$.

Jeder Typ besitzt einen Namen. Dies ist die einzige Eigenschaft, die bei allen Arten von Typen (also den Elementen der Menge TYPES aus (3.1)) zu finden ist. Es wird ein weiterer Objekttyp t_{type} auf der Meta-Ebene eingeführt, für den $struct(t_{type}) = [name: \beta_{string}]$ ist. Die Objekttypen $t_{basetype}$, t_{constr}, $t_{enumtype}$, $t_{vartype}$, $t_{deftype}$ des Meta-Schemas werden zu Subtypen des Objekttyps t_{type}, indem wir setzen:

$$I(t_{subtype})(t) = [isa: t_{type}] \text{ für } t \in \{t_{basetype}, t_{constr}, t_{enumtype}, t_{vartype}, t_{deftype}\}.$$

Für ein beliebiges Datenbankschema-Objekt ω gilt:

$$\omega.SUBTYPES = \omega.DEFTYPES \cap Def(I(t_{subtype})),$$

wobei ω.SUBTYPES bzw. ω.DEFTYPES den Wert der entsprechenden Tupelkomponente von $I(t_{dbschema})(\omega)$ bezeichnen soll.

Die Interpretationen für die Objekttypen des Meta-Schemas

Alle Objekttypen t des Meta-Schemas, d.h. alle Elemente $t \in \omega_{Meta}.DEFTYPES$, sind Instanzen der Objekttypen t_{type} und $t_{deftype}$, d.h. für diese Typen ist sowohl $I(t_{type})(t)$ als auch $I(t_{deftype})(t)$ definiert. Insbesondere ist gilt dies auch für t_{type} und $t_{deftype}$ selbst!

Es ist $struct(t_{deftype}) = [struct: t_{structure}]$, d.h. die Tupelkomponente struct repräsentiert die Funktion *struct* aus (3.21). Im nächsten Kapitel wird die Definition von Objekttypen um Operationen (Methoden) ergänzt. Dann muß $struct(t_{deftype})$ zu $[struct: t_{structure}, ops: \{t_{defop}\}]$ erweitert werden (wobei dann t_{defop} ein weiterer Objekttyp des Meta-Schemas ist, dessen Instanzen die benutzerdefinierten Operationen sind). Wir wollen uns in dieser Arbeit hinsichtlich der Entwicklung eines Meta-Schemas auf den strukturellen Teil beschränken.

Für $t \in \omega_{Meta}.DEFTYPES$ können die Interpretationen $I(t_{type})(t) = [name: n]$, $I(t_{deftype})(t) = [struct: s]$ und $I(t_{subtype})(t) = [isa: t']$ der Tabelle 3.2 entnommen werden.

Zum Inhalt der Tabelle 3.2 sei angemerkt, daß wir die Festlegung $I(t_{deftype})(t_{enumtype}) = [struct: '<[value: \beta_{integer}, literal: \beta_{string}]>']$ vor dem Hintergrund einer bestimmten Implementation von Aufzählungstypen getroffen haben. Jedes Element eines Aufzählungstyps, gegeben durch ein Literal, wird intern durch einen Integer-Wert repräsentiert. Die Ordnung innerhalb der Liste bestimmt eine lineare Ordnung der Elemente des Aufzählungstyps. Auf diese Weise ist eine Modifikation der Literale und eine Veränderung der linearen Ordnung der Elemente eines Aufzählungstyps möglich, ohne daß dies zu Änderungen in einer Datenbankinstanz führt.

Bei genauerer Betrachtung der Tabelle 3.2 fällt auf, daß die Einträge in der Spalte für *s* in Anführungszeichen stehen. Würden wir die Anführungszeichen weglassen, dann wäre beispielsweise $I(t_{deftype})(t_{table}) = [struct: [name: \beta_{string}, struct: t_{structure}]]$, jedoch ist $[name: \beta_{string}, struct: t_{structure}]$ kein Element aus $dom(t_{structure}) \subseteq \Omega$! Wie sind nun die in Anführungsstrichen stehenden Ausdrücke der Spalte *s* zu interpretieren? Diese Frage beantworten wir im Zusammenhang mit den folgenden Ausführungen zum Objekttyp $t_{structure}$.

	n	*s*	*t'*
t_{type}	"type"	'[name: β_{string}]'	—
$t_{basetype}$	"basetype"	'[ ]'	t_{type}
t_{constr}	"constructor"	'[ ]'	t_{type}
$t_{enumtype}$	"enumtype"	'[enum: <[value: $\beta_{integer}$, literal: β_{string}]>]'	t_{type}
$t_{vartype}$	"vartype"	'[variants: {[label: β_{string}, struct: $t_{structure}$]}]'	t_{type}
$t_{deftype}$	"deftype"	'[struct: $t_{structure}$]'	t_{type}
$t_{subtype}$	"subtype"	'[isa: $t_{deftype}$]'	$t_{deftype}$
t_{table}	"table"	'[name: β_{string}, struct: $t_{structure}$]'	—
$t_{dbschema}$	"dbschema"	'[name: β_{string}, DEFTYPES: {$t_{deftype}$}, SUBTYPES: {$t_{subtype}$}, ENUMTYPES: {$t_{enumtype}$}, VARTYPES: {$t_{vartype}$}, TABLES: {$t_{deftype}$}]'	—
$t_{structure}$	"structure"	'[type: t_{type}, domain: <[lower: $\beta_{integer}$, upper: $\beta_{integer}$]>, substruct: {[attr: β_{string}, struct: $t_{structure}$]}]'	—

Tabelle 3.2: Übersicht über die Interpretationen der Objekttypen des Meta-Schemas

Der Objekttyp $t_{structure}$

Für $t_{structure}$ soll *struct* ($t_{structure}$) = [type: β_{string}, domain: <[lower: $\beta_{integer}$, upper: $\beta_{integer}$]>, substruct: {[attr: β_{string}, child: $t_{structure}$]}] sein. Dies entspricht auch genau dem Eintrag in Tabelle 3.2. Die Struktur ist so gewählt, daß Instanzen von $t_{structure}$ beliebige Knoten eines Strukturbaumes beschreiben können. Die Beschriftungen *type* und *domain* finden ihre Entsprechung in den gleichnamigen Tupelkomponenten. Die Beschriftung *attr_pos* wird zusammen mit der impliziten Beschriftung *children* in der Tupelkomponente "substruct" erfaßt.

Jeder Strukturbaum und somit jede Struktur wird durch eine Teilmenge $S \subseteq dom(t_{structure})$ angegeben. In der Spalte *s* der Tabelle 3.2 bezeichnet ein in Anführungszeichen stehender Ausdruck dasjenige Objekt $s \in S$, das dem Wurzelknoten des zugehörigen Strukturbaumes entspricht.

	type: t_{type}	domain: <>		substruct: {}	
		lower: $\beta_{integer}$	upper: $\beta_{integer}$	attr: β_{string}	child: $t_{structure}$
s_0	c_{tuple}	< >		"name" "struct"	s_1 s_2
s_1	β_{string}	< >		{}	
s_2	$t_{structure}$	< >		{}	
	…	…		…	
s_3	c_{tuple}	< >		"type" "domain" "substruct"	s_4 s_5 s_6
s_4	t_{type}	< >		{}	
s_5	c_{list}	< >		*null*	s_7
s_6	c_{set}	< >		*null*	s_8
s_7	c_{tuple}	< >		"lower" "upper"	s_9 s_{10}
s_8	c_{tuple}	< >		"attr" "struct"	s_{11} s_{12}
s_9	$\beta_{integer}$	< >		{}	
s_{10}	$\beta_{integer}$	< >		{}	
s_{11}	β_{string}	< >		{}	
s_{12}	$t_{structure}$	< >		{}	

Tabelle 3.3: Ein Auszug aller Interpretationen $I(t_{structure})(s)$ für Instanzen s des Objekttyps $t_{structure}$

In Tabelle 3.3 wird ein Ausschnitt aller Objekte $s \in \text{Def}(I(t_{structure}))$ mit der zugehörigen Interpretation $I(t_{structure})(s_i)$ angegeben.
Der in Tabelle 3.2 für t_{table} angegebene Ausdruck '[name: β_{string}, struct: $t_{structure}$]' soll das Objekt s_0 in Tabelle 3.3 bezeichnen. Der gesamte Strukturbaum, dessen Wurzel s_0 ist, wird durch die Menge $\{s_0, s_1, s_2\}$ „aufgespannt“.

Die Struktur des Objekttyps $t_{structure}$ selbst wird durch die Menge $\{s_3, \ldots, s_{12}\}$ „aufgespannt“, wobei s_3 die Wurzel ist, was durch einen Blick auf Tabelle 3.3 verifiziert werden kann. Wir begegnen hier somit erneut einer Form von Selbstreferenzierung, da die Struktur des Objekttyps $t_{structure}$ durch Objekte dieses Typs beschrieben wird.

Beim Blick auf die Tabelle 3.4 fällt auf, daß der Wert für das Attribut `domain` durchgängig die leere Liste ist. Tatsächlich ist dieses Attribut nur dann relevant, wenn `type` den Wert c_{array} hat, was vergleichsweise selten der Fall sein wird. Wir schlagen deshalb als Alternative zu dem oben geschilderten Vorgehen vor, den Typ $t_{structure}$ als einen varianten Typ mit

$$variants(t_{structure}) =$$
$$\{(simple, t_{simple\_struct}), (coll, t_{coll\_struct}), (tuple, t_{tuple\_struct}), (array, t_{array\_struct})\}$$

zu definieren. Die Alternativen sind Objekttypen

- t_{simple_struct} mit der Struktur [type: t_{type}]
- t_{coll_struct} mit der Struktur [type: t_{type}, child: $t_{structure}$],
- t_{tuple_struct} mit der Struktur [type: t_{type}, substruct:{[attr:β_{string}, child: $t_{structure}$]} und
- t_{array_struct} mit der Struktur [type: t_{type}, domain: <[lower: …, upper: …]>, child: $t_{structure}$]

Lediglich der Einfachheit halber haben wir hier $t_{structure}$ als Objekttyp definiert und gleich mit *allen* Attributen versehen, die zur Beschreibung eines beliebigen Strukturbaumknotens notwendig sein können, aber nicht müssen.

Mit dem Objekttyp $t_{structure}$ und insbesondere der Menge $\{s_3, …, s_{12}\}$ seiner Instanzen haben wir für ESCHER$^+$ ein Analogon zum sog. *BOOT-Schema* geschaffen, das die Rolle des Meta-Schemas im ESCHER-Prototyp einnimmt [KTW90].

Eine Abschlußeigenschaft des Meta-Schemas

In Tabelle 3.4 ist eine beispielhafte Instanz von `DBSCHEMATA` nach der Definition zweier anwendungsspezifischer Datenbankschemata `Uni` und `Bibliothek` skizziert.

	name	DEFTYPES	…	TABLES	…
ω_{META}	`"META"`	$\{t_{deftype}, t_{subtype}, t_{dbschema}, \ldots\}$	…	$\{\tau_{DBSCHEMATA}, \tau_{BASETYPES}, \tau_{CONSTR}\}$	…
ω_{Uni}	`"Uni"`	$\{t_{Person}, t_{Student}, \ldots\}$	…	$\{\tau_{Personal}, \tau_{Studenten}, \tau_{Vorlesungen}, \ldots\}$	…
ω_{Bibl}	`"Bibliothek"`	…			
…	…				

Tabelle 3.4: $I(t_{dbschema})(\omega)$ für $\omega \in inst(\tau_{DBSCHEMATA})$

Eine gewünschte Abschlußeigenschaft bei Datenmodellen besteht darin, daß jedes benutzerdefinierte Datenbankschema als eine Instanz des Meta-Schemas interpretiert werden kann und dieses wiederum Instanz von sich selber ist (Selbstreferenzierung). Dabei soll ein Datenbankschema ω_{DS} eine Instanz eines Datenbankschemas $\omega_{DS'}$ genannt werden, wenn ω_{DS} über eine Tabelle des Datenbankschemas $\omega_{DS'}$ erreichbar ist. Dies ist bei dem soeben geschilderten Vorgehen tatsächlich der Fall: $\tau_{DBSCHEMATA}$ ist eine Tabelle des Meta-Schemas ω_{Meta}, und *alle* Schemata – auch ω_{Meta} – sind Elemente von $inst(\tau_{DBSCHEMATA})$, d.h. über $\tau_{DBSCHEMATA}$ erreichbar. In Abb. 3.9 ist die Situation noch einmal veranschaulicht.

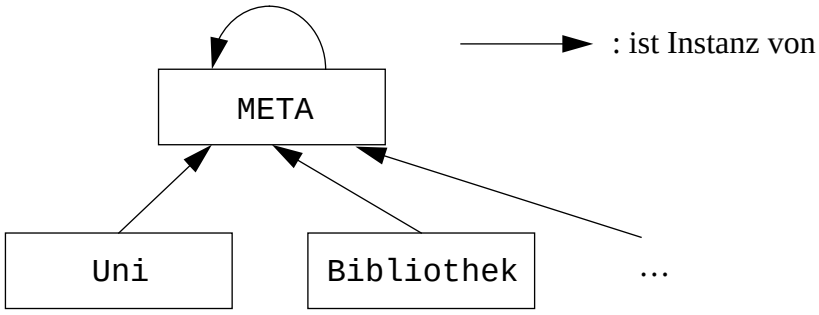


Abbildung 3.9: Benutzerdefinierte Schemata als Instanzen des Meta-Schemas

3.11 Zusammenfassung und Diskussion

In diesem Kapitel wurde der Strukturteil des Datenmodells ESCHER$^+$ unter formalen Gesichtspunkten beschrieben. Dabei ist die formale Integration von Objekttypen in das Modell von großer Bedeutung. Diese nehmen in einem Datenbankschema eine Schlüsselstellung ein, da durch sie eine direkte Umsetzung der Entitäten des zu modellierenden Umweltausschnittes in Instanzen eines Objekttyps erfolgen kann. Aus diesem Grund kann ESCHER$^+$ auch als strukturell objektorientiertes Datenmodell bezeichnet werden. Tatsächlich hat ESCHER$^+$ sehr viele Gemeinsamkeiten z.B. mit dem Datenmodell des ODMG'93-Standards [Cat+94].

In ESCHER$^+$ besteht die Möglichkeit der Definition benutzerdefinierter Aufzählungstypen. Damit ist eine zunächst sehr bescheiden anmutende, für die Praxis dennoch sehr relevante Erweiterung der Menge der Basistypen möglich.

ESCHER$^+$ erlaubt auch die für das eNF^2-Modell charakteristische Konstruktion komplexer Typen (Strukturen) und dazugehöriger komplexer Wertebereiche. Neben (endlichen) Mengen, Listen und Tupeln erlaubt unser Datenmodell auch die Definition von Multimengen und Feldern (Arrays) mit beliebiger Elementestruktur. Multimengen sind für die Abgeschlossenheit des Datenmodells notwendig: Soll z.B. die Projektion von Tupeln in einer Menge auf eine integer-wertige Komponente durchgeführt und anschließend der Mittelwert aller dieser Komponenten berechnet werden, dann ist nach der Projektion keine Duplikateliminierung erwünscht, d.h. die Multimenge ist die geeignete Struktur für das Zwischenergebnis! Felder sind für viele technische und wissenschaftliche Anwendungen relevant (Matrizen, Vektoren) und sollen deshalb auch auf einfache Art und Weise im ESCHER$^+$-Datenmodell abgebildet werden können.

Wie viele andere Datenmodelle [Pis89, AGO90, DV92, WPC92, PTC+93, Pau94, AGO95] erlaubt auch ESCHER$^+$ die Modellierung varianter Typen. Ein varianter Typ ist durch *n* Strukturen bzw. Typen als Alternativen definiert, von denen für jede Instanz genau eine zutrifft: Die Instanz eines varianten Typs „kapselt" das eigentliche Datenobjekt, dessen Typ oder Struktur durch Inspektion eines Labels ermittelt wird. Gegenüber anderen Modellen, die variante Typen über einen eigenen Konstruktor realisieren, sind variante Typen in ESCHER$^+$ immer benamte Typen. Gegenüber Vorschlägen, die lediglich einen Varianten-Konstruktor anbieten, wird ein

größerer Grad an Wiederverwendbarkeit und eine konsistentere Änderbarkeit (Erweiterung um neue Alternativen, Wegnahme von Alternativen) erreicht.

Variante Typen erlauben die direkte Modellierung der disjunkten Generalisierung mit Überdeckungseigenschaft, die wir bereits in Abschnitt 2.1.1 als in Anwendungen häufig anzutreffenden, wichtigen Spezialfall der Generalisierung herausgestellt hatten.

Eine neuere Arbeit, die den Begriff des varianten Typs erweitert, ist [KD95]. Dort wird das Relationenmodell erweitert, indem sog. *flexible Relationen* eingeführt werden. Grundlage des erweiterten Modells bilden die sog. *choice schemes*, die die Gestalt

$$<k_{min}, k_{max}, \{A_1, \ldots, A_n\}>$$

haben. Von den *n* Alternativen $A_1, \ldots, A_n$ sind für eine Instanz eines choice schemes mindestens k_{min} und höchstens k_{max} Alternativen zutreffend. Jede Alternative kann ein atomares Attribut oder wiederum ein choice scheme sein, d.h. die choice schemes lassen sich beliebig schachteln. Choice schemes sind die einzigen Konstruktoren des Modells aus [KD95]. Ein varianter Typ im Sinn von ESCHER$^+$ ist durch ein choice scheme mit $k_{min} = k_{max} = 1$ gegeben. Eine konventionelle Relation läßt sich mit $k_{min} = k_{max} = n$ modellieren, ein einzelnes optionales Attribut mit $k_{min} = 0$, $k_{max} = 1$ und $n = 1$.

Wenn auch der Vorschlag aus [KD95] eine sehr allgemeine und flexible Modellierung varianter Strukturen ermöglicht, so darf nicht übersehen werden, daß vom programmiersprachlichen Standpunkt flexible Relationen eher unhandlich sind. Vor jedem Zugriff auf eine Instanz einer flexiblen Relation muß die Zulässigkeit des Zugriffs geprüft werden, da nicht feststeht, welche Alternative für die Instanz zutrifft. Es soll ja gerade der Vorteil eines auf Typisierung aufbauenden Datenmodells sein, daß von gewissen Eigenschaften der Instanzen *sicher* ausgegangen werden kann. Trotzdem tragen wir für ESCHER$^+$ dem Bedarf an varianten Typen im Sinne einer 1-aus-*n*-Auswahl Rechnung. Typischerweise wird die Manipulation der Instanzen varianter Typen in einem Anwendungsprogramm mit Hilfe eines `case`-Konstrukts vollzogen. Im folgenden Kapitel werden wir die Definition von Objekttypvarianten noch einmal erweitern, indem gewisse Eigenschaften (Attribute oder Methoden) als für alle Alternativen verbindlich vorgeschrieben werden können (siehe Abschnitt 4.7.4). Der Zugriff auf diese Eigenschaften kann dann ohne `case`-Konstrukt erfolgen. Für die Ausführung von Methoden bedeutet dies, daß für variante Typen dynamisches Binden realisiert wird.

Die strukturelle Semantik von Objekttypen wurde mit Hilfe von Interpretationsfunktionen $I(t)$ formalisiert. Ein Objekt ω kann relativ zu einem Objekttyp t interpretiert werden, falls ω im Definitionsbereich der Interpretationsfunktion $I(t)$ liegt. Falls $\omega \in \mathrm{Def}(I(t))$, dann nennen wir ω auch eine Instanz von t. Der Funktionswert $I(t)(\omega)$, ein Tupel, gibt den strukturellen Zustand des Objektes ω relativ zum Typ t an. Ein Objekt ω kann gleichzeitig Instanz mehrerer Objekttypen sein und hat dann auch mehr als einen Zustand $I(t)$. Den äußeren Rahmen der multiplen Typzugehörigkeit bildet die *isa*-Beziehung auf Objekttypen. In ESCHER$^+$ liegt die Typzugehörigkeit eines Objektes nicht statisch fest, wie dies in den meisten objektorientierten Datenmodellen der Fall ist, sondern ist veränderbar. Einem Objekt ω können weitere Typen hinzugefügt werden, sofern der neue Typ ein Subtyp einer der Typen t ist, in dessen Wertebereich ω bereits liegt. In diesem Kapitel haben wir zunächst die Notwendigkeit dieser dynamischen Spezialisierung motiviert und einen ersten informellen Eindruck über den Mechanismus gegeben. Im folgenden Kapitel, das sich speziell mit dem dynamischen Teil des Datenmodells beschäftigt, präzisieren wir unseren Ansatz.

Bei der Einführung von Objekttypen fällt auf, daß bisher unklar ist, in welchem Umfang die Einkapselung von Objekttypen in ESCHER$^+$ eine Rolle spielen soll. Bei strenger Einkapselung – wie etwa in der OOPL Smalltalk [GR83] – ist gar kein direkter lesender oder schreibender Zugriff auf die Attribute eines Objektes möglich. Alle Zugriffe müssen dann über Methodenaufrufe abgewickelt werden, bei denen das betreffende Objekt der Receiver ist. Diese „reine Lehre“ der Einkapselung halten wir für zu restriktiv (vgl. auch die Diskussion in Abschnitt 2.5.1). Viele OOPLs, darunter C++ [Str91], lassen eine Einteilung in öffentliche und nichtöffentliche Attribute zu, wobei lediglich auf die öffentlichen direkt zugegriffen werden darf. Wir verschieben die Behandlung der Einkapselung von Objekttypen in ESCHER$^+$ auf Abschnitt 5.5.1. Damit schließen wir uns der Auffassung von [Nav92] an, wonach die Restriktion der Zugriffsrechte vor allem der Konsistenzerhaltung dient. Dort wollen wir für ESCHER$^+$ eine selektive Einkapselung im Stil von C++ einführen.

In [Pau94] gibt es neben der Definition von Objekttypen noch die Möglichkeit, Wertetypen zu definieren, d.h. die Instanzen dieser Typen sind *Werte* im Sinne von [Bee90]. Wertetypen ermöglichen eine differenzierte Definition von Typäquivalenz bzw. Typkompatibilität. Angenommen, es seien zwei Tabellen `Personen` und `Weine` definiert, die beide die Struktur `{[Name: string, Jahrgang: integer]}` haben, dann ist auf der Basis von Strukturgleichheit die Vereinigung beider Mengen erlaubt, obwohl dies semantisch keinen Sinn ergibt. Werden jedoch über

```
type PName = string, WName = string,
     PJahrg = integer, WJahrg = integer;
```

Wertetypen definiert und sind die Strukturen der Tabellen

`{[Name: PName, Jahrgang: PJahrg]}` bzw.
`{[Name: WName, Jahrgang: WJahrg]}`,

dann wird die Vereinigung als unzulässig zurückgewiesen, da keine Typkompatibilität vorliegt[18]. Stehen jedoch – wie in ESCHER$^+$ und auch in [Pau94] – Objekttypen zur Verfügung, dann können obige Tabellen gleich mit den Strukturen `{Person}` bzw. `{Wein}` spezifiziert werden, wobei `Person` und `Wein` entsprechende Objekttypen sind. Somit entfällt der Bedarf an Wertetypen an dieser Stelle. Ferner können laut [Pau94] Wertetypen auch die Rolle nachträglich hinzugefügter Basistypen übernehmen. Ist z.B. `date` nicht als vordefinierter Basistyp verfügbar, dann kann nach [Pau94] ein Wertetyp `date` mit der Struktur `[day: integer, month: string, year: integer]` spezifiziert werden. Damit hat man jedoch kaum etwas gewonnen, da nach [Pau94] keine wertetyp-spezifischen Operationen definiert werden können, was jedoch für neue Basistypen unabdingbar ist. Wir haben daraus die Konsequenz gezogen, für ESCHER$^+$ auf Wertetypen zu verzichten.

Für ESCHER$^+$ wurde eine Subtyp-Beziehung *isa* auf Objekttypen festgelegt. Es fällt auf, daß jedoch keine Subtyp-Beziehung $\preceq$ auf anderen Strukturen aus $\mathcal{S}$(TYPES) spezifiziert wurde, wie dies beispielsweise in [Pau94] der Fall ist. Wir wollen an dieser Stelle kurz auf die Definition der Relation $\preceq$ eingehen, wie sie bereits in [CW85] zu finden ist: Zunächst gilt $\beta \preceq \beta$ für alle Basistypen β. Sind s und s' zwei Tupelstrukturen, dann gilt $s \preceq s'$ gdw. s mindestens alle Attribut-Struktur-Paare (a_i, s_i) von s' enthält, und die Komponentenstrukturen gemeinsamer

18. Typkompatibilität basiert in [Pau94] auf Namensgleichheit. Dies unterscheidet Wertetypen von den in SQL [DD93] mittels `CREATE DOMAIN` erzeugten Domänen: Im SQL-Standard gilt (leider) Typkompatibilität aufgrund von Strukturgleichheit, d.h. die Domäne `PJahrg` ist dort kompatibel zu `integer`!

Attributnamen in der Relation $\preceq$ zueinander stehen. Bei varianten Typen ist es genau umgekehrt: Sind t und t' zwei variante Typen, dann gilt t $\preceq$ t' gdw. *variants*(t) $\subseteq$ *variants* (t'), d.h. t' hat mindestens die Label-Alternative-Paare von t', und die Alternativen gemeinsamer Label stehen in der Relation $\preceq$ zueinander. Für Kollektionen überträgt sich die Relation $\preceq$ von den Komponentenstrukturen, z.B. gilt z.B. {s} $\preceq$ {s'} gdw. s $\preceq$ s'. Für Objekttypen t, t' definiert man t $\preceq$ t' gdw. t *isa** t'. Es läßt sich leicht nachweisen, daß $\preceq$ eine Halbordnung auf $\mathcal{S}$(TYPES) ist. Motivation der Relation $\preceq$ ist die Substituierbarkeit zur Laufzeit: z.B. darf an eine Variable ein Datenobjekt gebunden werden, dessen Struktur in $\preceq$-Beziehung zur erwarteten Struktur steht. Wir hatten jedoch bereits argumentiert, daß Substituierbarkeit aufgrund der (nach rein strukturellen Kriterien definierten) Relation $\preceq$ zu semantisch unzulässigen Substitutionen führen kann (vgl. das obige Beispiel mit `Personen` und `Weinen`). Daher soll für ESCHER$^+$ allein die semantisch begründete *isa*-Beziehung auf Objekttypen Relevanz haben.

In diesem Kapitel wurde ausführlich der Übergang von Strukturen bzw. getypten Datenobjekten zu Baumrepräsentationen behandelt. Jeder Struktur s $\in$ $\mathcal{S}$(TYPES) läßt sich eine Menge von beschrifteten Bäumen zuordnen, die alle äquivalent sind in dem Sinne, daß sie gerade die Struktur s repräsentieren. Sie unterscheiden sich nur in den verwendeten Knoten des Knotenvorrates $\mathcal{V}$. Jedem getypten Datenobjekt (v, s) wird eine Menge von Baumpaaren (B_v, B_s) zuordnen, wobei B_v der Wertebaum mit den eigentlichen Daten ist, während B_s die Strukturinformation enthält. Jeder Knoten in B_v verweist mit der Beschriftung *struct* auf einen entsprechenden Knoten in B_s, so daß jeder Knoten von B_v selbst der Wurzelknoten eines Teilbaumes von B_v ist, der ein getyptes Datenobjekt repräsentiert. Die Menge aller Baumrepräsentationen für getypte Datenobjekte haben wir $\mathcal{ValTree}$(TYPES) genannt. Motiviert wurde der Übergang von einem datenobjekt-bezogenen zu einem graph-basierten Modell damit, daß letzteres eine intuitive (und implementationsnahe!) Vorstellung komplexer Datenobjekte und der Semantik der auf sie anwendbaren Operationen bietet. Das mentale Baum-Modell soll sich daher auch in der formalen Semantik der Operationen ausdrücken.

Im folgenden Kapitel werden Baumrepräsentationen zentrale Bedeutung für alle Ausführungen zum operationalen Teil des Datenmodells haben. Deshalb erfolgte auch die Definition einer Datenbankinstanz zu einem gegebenen Datenbankschema auf der Grundlage von Baumrepräsentationen. Es wird später darauf geachtet, daß die Menge $\mathcal{ValTree}$(TYPES) unter allen Operationen abgeschlossen ist, d.h. jede Operation transformiert ein Element aus $\mathcal{ValTree}$(TYPES) wiederum in ein Element aus $\mathcal{ValTree}$(TYPES). Aus diesem Grund sind beide Sichtweisen, die datenobjekt-bezogene und die graph-basierte, jederzeit austauschbar.

Es liegt nun ein Vergleich mit anderen graph-basierten Modellen nahe. Viele semantische Datenmodelle, wie z.B. das ER-Modell oder auch IFO, sind als graph-basiert einzustufen. Allerdings beschränken sich diese Modelle auf die Schema-Ebene. Als in engerem Sinne graph-basiert wollen wir solche Modelle bezeichnen, bei denen auf der Instanzen-Ebene elementare Operationen wie „Knoten einfügen/löschen“, „Kante einfügen/löschen“ zur Verfügung stehen. Zu diesen Modellen zählen z.B. GOOD [GPB+94] und das Modell von Levene/Loizou [LL95]. Letzeres basiert auf einem *Hypernode*-Ansatz: Ein Hypernode ist vergleichbar mit einem Objektidentifikator, und er hat einen „inneren Zustand“, der Knoten und Kanten enthält. Ein großer Nachteil von [LL95] ist, daß das Modell ungetypt ist. Mit Knoten, Kanten und den oben genannten Operationen als fundamentale Bausteine sind graph-basierte Modelle wie [GPB+94] und [LL95] auf sehr elementarem Niveau angesiedelt, das wir für die logische Ebene als nicht geeignet erachten.

Kapitel 4

Der operationale Teil des Datenmodells

In diesem Kapitel wenden wir uns dem operationalen Teil des Datenmodells ESCHER$^+$ zu. Es werden vordefinierte Operationen (die sog. Basisoperationen) sowie benutzerdefinierte Operationen eingeführt.

Als Grundlage für die Semantik des operationalen Teils verwenden wir die im vorangegangenen Kapitel eingeführten Baumrepräsentationen. Wir gehen von einem Datenbankschema DS und einer dazugehörenden zulässigen Datenbankinstanz INST(DS) aus. Letztere besteht gemäß Def. 3.21 und Def. 3.22 im wesentlichen aus einer Menge D_{pers} von paarweise disjunkten Baumrepräsentationen. Wir führen in Abschnitt 4.1 den Begriff des Laufzeitzustandes ein, der auch transiente Datenobjekte berücksichtigt. Grundsätzlich sollen alle Operationen als Funktionen definiert sein, deren aktuelle Parameter Knoten von Wertebäumen sind. Sie geben entweder einen Knoten r' oder einen besonders ausgezeichneten Wert *void* zurück. Im ersten Fall handelt es sich um eine Funktion im eigentlichen Sinn, im zweiten Fall handelt es sich um eine Prozedur.

In Abschnitt 4.2 führen wir Signaturen ein, die sich für alle Operationen eignen, die in diesem Kapitel behandelt werden. Die Signaturen können auch sog. Strukturvariablen enthalten, was die Möglichkeit der Spezifikation generischer Operationen ermöglicht, wie wir sie bei den Basisoperationen benötigen.

Gegenstand von Abschnitt 4.3 sind die bereits erwähnten Basisoperationen, die die „primitiven“ Operationen in ESCHER$^+$ sein sollen. Unter ihnen finden sich insbesondere die generischen Update-Operationen, über die alle Modifikationen einer Datenbankinstanz abgewickelt werden müssen.

In Abschnitt 4.4 führen wir mit den sog. *Links* weitere Datenobjekte in das Datenmodell ein, die Verweise auf Knoten in Baumrepräsentationen darstellen. Jedoch sollen Links im logi-

schen Datenmodell ESCHER$^+$, so wie es dem Anwender zur Verfügung steht, nicht in Erscheinung treten, da die „Adressen“ der Knoten von Baumrepräsentationen selbst nichts mit dem logischen Modell zu tun haben, sondern lediglich mit der Art, wie Datenobjekte „intern“ repräsentiert werden. So sollen Links auch nur „intern“ zur Verfügung stehen, um mit ihnen die Semantik von Iteratoren über Kollektionen anzugeben. Auf diese Weise wird weiterhin eine stringente Trennung der internen und der logischen Ebene gewährleistet. In Abschnitt 4.4 werden weitere Basisoperationen eingeführt, die Iteratoren erzeugen und manipulieren.

In Abschnitt 4.5 werden benutzerdefinierte Operationen eingeführt. Zu ihnen gehören die Methoden, die einem bestimmten Objekttyp zugeordnete, benutzerdefinierte Operationen sind. Somit vervollständigen wir die Spezifikation eines Objekttyps um eine operationale Komponente. Daneben gibt es noch „freie“ benutzerdefinierte Operationen, die unabhängig von einer Objekttypdefinition existieren. Dazu sollen auch von einem Endanwender aufrufbare „Hauptprogramme“ gehören.

Die benutzerdefinierten Operationen müssen natürlich in einer geeigneten Programmiersprache kodiert werden. In [Pau94] wurde mit SCRIPT eine persistente Programmiersprache für ein Datenmodell entworfen, das ebenfalls eine Erweiterung des eNF^2-Modells darstellt. Die Semantik von SCRIPT wurde in [Pau94] lediglich informell angegeben. In dieser Arbeit führen wir die Sprache BASESCRIPT ein, die unter syntaktischen Gesichtspunkten wesentlich „primitiver“ als SCRIPT ist, aber dennoch dieselbe Mächtigkeit besitzt. Die Syntax und Semantik von BASESCRIPT geben wir in Abschnitt 4.6 vollständig an. Während Anweisungen und Ausdrücke in SCRIPT sehr komplex aufgebaut sein können, so daß die Abarbeitung bzw. Auswertung zu einer ganzen Folge von Aufrufen von Basisoperation führen kann, ist dies in BASESCRIPT nicht der Fall: Dort wird pro Anweisung höchstens ein sehr einfacher Pfadausdruck ausgewertet oder genau eine Operation ausgeführt. Wir schlagen deshalb BASESCRIPT als Zielsprache einer Übersetzung von Scripten (d.h. von in SCRIPT geschriebenen, benutzerdefinierten Operationen) vor. Es entstehen sog. Basisscripten, die gemäß der Ausführungen in Abschnitt 4.6 in sehr einfacher Weise von einem Interpreter Schritt für Schritt abgearbeitet werden können, wodurch elementare Zustandsübergänge des Laufzeitzustandes ausgelöst werden.

In Abschnitt 4.7 kehren wir dann zum „höheren“ Sprachlevel zurück, dem die Sprache SCRIPT zuzurechnen ist. Wir modifizieren den Vorschlag aus [Pau94], um SCRIPT an die Gegebenheiten des Datenmodells ESCHER$^+$ anzupassen. Die Syntax von SCRIPT$^+$, wie wir unsere SCRIPT-Variante nennen, ist in Anhang B.3 zu finden. Ohne Anspruch auf Vollständigkeit zu erheben, geben wir in Abschnitt 4.7.3 einige Beispiele für die Übersetzung von SCRIPT$^+$ nach BASESCRIPT an. Schließlich gehen wir in Abschnitt 4.7.4 auf weitere Sprachelemente von SCRIPT$^+$ ein. Dabei geht es insbesondere um die Behandlung von Varianten und um die dynamische Typerweiterung (dynamische Spezialisierung).

In Abschnitt 4.8 weisen wir auf die Notwendigkeit hin, das Datenmodell um zusätzliche Basistypen und Basisoperationen erweitern zu können, und diskutieren existierende Ansätze, die Erweiterbarkeit des Datenmodells zu realisieren.

Die Zusammenfassung und Diskussion in Abschnitt 4.9 schließt die Behandlung des operationalen Teils von ESCHER$^+$ ab.

4.1 Laufzeitzustand

Für die formale Beschreibung der Semantik von Operationen benötigen wir den Begriff des *Laufzeitzustandes*. Der Laufzeitzustand muß zunächst natürlich einen Datenbankzustand INST(DS) eines Datenbankschemas DS enthalten. Daneben sind zur Laufzeit aber auch flüchtige (transiente) Datenobjekte zu berücksichtigen, die durch Ausführung von Operationen erzeugt werden. Ferner zählen dazu auch Ergebnisse von Anfragen. Wir führen daher den Begriff des Laufzeitzustandes ein, der neben den persistenten auch transiente Datenobjekte erfaßt.

Die Namen der Tabellen sind die *persistenten* Namen, die gerade auch die globalen Namen in unserem Datenmodell sein sollen, d.h. sie sind überall sichtbar. Wir setzen

$$\text{GLOBNAMES} := \{name\,(\tau) \mid \tau \in \text{TABLES}\}$$

Damit transiente Datenobjekte während ihrer Lebensdauer erreichbar sind, müssen sie an *transiente* Namen gebunden werden. In Abschnitt 4.5 werden benutzerdefinierte Operationen eingeführt. Während ihrer Abarbeitung sollen die transienten Namen gerade die Namen der lokal deklarierten Variablen sowie der formalen Parameter sein. Der Einfachheit halber verzichten wir also – genauso wie [Pau94] – auf die Möglichkeit, lokale Namen einer Operation in anderen Operation sichtbar zu machen, wie es in den meisten Programmiersprachen üblich ist.

Für die Abarbeitung des i-ten geschachtelten Operationsaufrufs wird eine Bindungsfunktion

$$\alpha_i: \mathcal{N} \cdots> \mathcal{V} \tag{4.1}$$

erzeugt, deren Definitionsbereich die Menge der Namen für lokale Variablen und formale Parameter sowie der Namen aller Tabellen ist. Die Namen für lokale Variablen und formale Parameter werden über α_i an Knoten von Baumrepräsentationen gebunden, die persistente oder transiente Datenobjekte darstellen. Es ist klar, daß diese Bindungen in Form eines Stacks verwaltet werden müssen.

Wir kommen nun zur formalen Definition des Laufzeitzustandes.

Definition 4.1 (Laufzeitzustand, elementarer Zustandsübergang)

Ein *Laufzeitzustand* Z über einem Datenbankschema DS wird zu jedem Zeitpunkt angegeben durch ein 4-Tupel (INST(DS), D_{trans}, L, R) mit

- INST(DS) ist ein Datenbankzustand zu DS. Insbesondere gehört dazu die Menge D_{pers} der *persistenten* Datenobjekte (siehe Abschnitt 3.9).
- $D_{trans} \in \mathcal{FSet}(\mathcal{ValTree}(\text{TYPES}))$, und es gilt:
  – $pref(r) = 0$ für $r \in \{ root(B_v) \mid (B_v, B_s) \in D_{trans} \}$
  – Die Baumrepräsentationen in $D_{pers} \cup D_{trans}$ sind paarweise disjunkt
- L ist eine Liste $L = <\alpha_1, \ldots, \alpha_k> \in \mathcal{PFun}(\mathcal{N}, \mathcal{V})^*$, $k \geq 0$, wobei für $1 \leq i \leq k$ gilt:
  – $\text{Def}(\alpha_i)$ ist endlich und $\text{GLOBNAMES} \subseteq \text{Def}(\alpha_i)$,
  – $\alpha_i(n) \in \{r \in V(B_v) \mid (B_v, B_s) \in D_{pers} \cup D_{trans}\}$
  – $\alpha_i(name(\tau)) = inst(\tau)$ für $\tau \in \text{TABLES}$
- $R \in \mathcal{V} \cup \{null\}$ ist der Rückgabewert der letzten beendeten Operation

D_{trans} ist die Menge der *transienten* Datenobjekte in Z.

Es sei $\mathcal{Z}$ die Menge aller möglichen Laufzeitzustände. Ein *elementarer Zustandsübergang* ist gegeben durch ein Paar $(Z^{alt}, Z^{neu}) \in \mathcal{Z}^2$.
In einem *Startzustand* Z_0 soll $L = <>$, $D_{trans} = \{\}$, $R = null$ gelten.

Für einen Laufzeitzustand Z sei schließlich

$$V(Z) := \{ r \in V(B_v) \mid (B_v, B_s) \in D_{pers} \cup D_{trans} \}$$

die Menge aller Knoten der Instanzenbäume des Laufzeitzustandes. ❒

Ein Laufzeitzustand ist also im wesentlichen eine Sammlung von Datenobjekten (bzw. ihren Baumrepräsentationen), die wir in persistente und transiente Datenobjekte partitionieren, und Namensbindungen an einige dieser Datenobjekte.

Beim Aufruf einer benutzerdefinierten Operation wird die Liste L um eine weitere Bindung α_{n+1} verlängert, wobei diese dann auch zur aktuellen Bindung wird, d.h. es kann nur auf die über Namen aus $Def(\alpha_{n+1})$ erreichbaren Datenobjekte zugegriffen werden. Darunter sind insbesondere alle Tabellen. Beim Rücksprung wird α_{n+1} wieder aus der Liste entfernt, und α_n wird wieder zur aktuellen Bindung.

In einer Implementation von ESCHER$^+$ soll eine automatische *garbage collection* erfolgen, die den Speicherplatz für nicht mehr erreichbare Datenobjekte freigibt. So müssen etwa beim Rücksprung aus einer Funktion alle durch den Aufruf und die Abarbeitung der Funktion erzeugten Datenobjekte (bzw. Baumrepräsentationen) wieder gelöscht werden, sofern sie nicht – als Folge der Ausführung von Operationen – von D_{trans} in die Menge D_{pers} wechselten. Dies kann z.B. die Folge eines Einfügens von (ω, t) in eine Kollektion sein, die die *pref*-Zähler der Objektzustände $I(t')(\omega)$ für alle t' mit t *isa** t' inkrementiert. Umgekehrt werden die *pref*-Zähler der Objektzustände $I(t')(\omega)$ durch ein Entfernen von (ω, t) aus einer Kollektion wieder dekrementiert. Hat ein solcher Zähler den Wert 0 und ist der Zustand nicht mehr erreichbar, so soll er von der *garbage collection* „entsorgt“ werden. Wir werden bei der Angabe der Semantik für die Basisoperationen die Manipulation der *pref*-Zähler nicht explizit beschreiben und gehen davon aus, daß alle Operationen die *pref*-Zähler in einem konsistenten Zustand halten, d.h. für alle Wurzelknoten eines Wertebaumes aus D_{pers} gilt die Bedingung (iv) aus Def. 3.22, für alle Wurzelknoten r eines Wertebaumes aus D_{trans} gilt $pref(r) = 0$.

Im Zusammenhang mit der Manipulation der persistenten Datenobjekte aus D_{pers} ist es wichtig, daß die Konsistenz dieser Daten, d.h. die Einhaltung von Integritätsbedingungen, langfristig gewährleistet bleibt. Die Bereitstellung von Mechanismen zur Konsistenzerhaltung ist eine der zentralen Eigenschaften von Datenbanksystemen. Von großer Bedeutung sind dabei Transaktionen, die als Folgen von Datenbank-Operationen definiert sind. Jegliche Manipulation von persistenten Datenobjekten darf nur innerhalb einer Transaktion stattfinden. Es wird davon ausgegangen, daß sich die Datenbank unmittelbar zu Beginn und nach Beendigung einer Transaktion in einem konsistenten Zustand befindet. Während der Ausführung einer Transaktion darf die Konsistenz allerdings verletzt werden. Transaktionen werden entweder komplett oder gar nicht ausgeführt (Atomizität einer Transaktion). Weder im ESCHER-Prototyp noch in der Sprache SCRIPT aus [Pau94] werden Transaktionen berücksichtigt. Dieses Manko soll für ESCHER$^+$ nicht gelten. In Kapitel 5.1 wird für ESCHER$^+$ ein einfaches Transaktionskonzept eingeführt. Neben den Verletzungen von Integritätsbedingungen, die bei „normalem“ Programmablauf erkannt werden (siehe ebenfalls Kapitel 5) und „erwartete“ Ereignisse darstellen, können zur Laufzeit natürlich auch „unerwartete“ Ereignisse auftreten, die ebenfalls eine

Reaktion erfordern. Dazu gehören insbesondere Ausnahmen, die bei der Ausführung einer Basisoperation entstehen, wie z.B. bei einer Division durch Null. Ferner betrifft dies Laufzeitfehler wie z.B. den Zugriff auf ein nicht existierendes Listen- oder Feldelement. Es wäre nicht zu rechtfertigen, in dieser Arbeit das Auftreten von Ausnahmen völlig zu ignorieren. Gleichwohl wollen wir hier die Diskussion einer Ausnahmebehandlung auf das Notwendigste beschränken. Bei der Beschreibung der Semantik der Basisoperationen werden wir deshalb angeben, in welchen Situationen eine Ausnahme auszulösen ist. Um die Konsistenz der Datenbankinstanz zu sichern, soll das Laufzeitsystem beim Auftreten einer Ausnahme wie folgt reagieren:

- Falls beim Auftreten einer Ausnahme eine Transaktion aktiv ist, dann wird diese zurückgesetzt (Abort).
- In Abhängigkeit von der „Schwere“ der aufgetretenen Ausnahme wird die Ausführung des Programms sofort beendet oder fortgesetzt. Soll nach dem Zurücksetzen einer Transaktion das Programm fortgesetzt werden, dann wird als nächstes der unmittelbar nach dem abgebrochenen Transaktionsblock folgende Schritt abgearbeitet.

Auf eine genauere Differenzierung der Ausnahmen hinsichtlich ihrer „Schwere“ wird an dieser Stelle verzichtet. Neben Ausnahmen, die besonders „schwerwiegende“ Ereignisse darstellen, sind auch andere Ereignisse denkbar, die eher Informationscharakter besitzen. Eine unserer Basisoperationen wird z.B. `insert` sein, die das Einfügen eines Elementes in eine Menge bewirkt. Ist das Element bereits in der Menge enthalten, so soll `insert` ein Ereignis auslösen, daß über den Versuch des Einfügens eines Duplikates informiert. Bei einer Erweiterung des dynamischen Teils des Datenmodells um eine differenziertere Ereignisbehandlung könnte dann abgefragt werden, ob dieses Ereignis ausgelöst wurde, was für den weiteren Programmablauf durchaus von Interesse sein könnte.

4.2 Signaturen für Operationen

Jeder Operation soll eine Signatur zugeordnet werden, die die Typen (Strukturen) der Eingabeparameter und des Rückgabewertes festlegt. Wegen der Generizität der Konstruktoren müssen auch Operationen definiert werden können, die für bestimmte formale Parameter aktuelle Parameter unterschiedlicher Struktur zulassen. Beispielsweise soll es eine Operation zum Einfügen in Mengen geben, die unabhängig von der Struktur der Elemente der Menge anwendbar ist. Solche Operationen bezeichnet man auch als *generische* Operationen.

Die erste zu lösende Aufgabe besteht darin, den generischen Operationen geeignete Signaturen zuordnen zu können. Wir führen daher einen verallgemeinerten Signaturbegriff ein, der *Strukturvariablen* in Signaturen zuläßt. Die verallgemeinerten Signaturen werden als syntaktische Konstrukte eingeführt.

Definition 4.2 (Signatur, generische Signatur, Strukturvariablen)

Eine *Signatur* ist ein Ausdruck aus $\mathcal{L}$(*sign*) mit folgenden Grammatikregeln:

sign ::= *argList* `->` *result*

result ::= *extStruct* | `void`

argList ::= [*arg* { ',' *arg* }]

arg ::= [*identifier* ':'] *extStruct*

Bei einem erweiterten Typausdruck $\in \mathcal{L}$(*extStruct*) können an die Stelle von Namen von Typen auch *Strukturvariablen* der Form '\' *identifier* treten (vgl. Anhang B.2).
Die Menge aller Signaturen werde mit $\mathcal{Sign}$ (TYPES) bezeichnet.
Mit $\mathcal{Sign}$ (BASETYPES, CONSTR) werde die Menge aller Signaturen bezeichnet, in denen keine Namen von Typen aus DEFTYPES oder VARTYPES vorkommen. Eine Signatur heißt *generisch*, wenn sie eine Strukturvariable enthält.

Ist $s_1, \ldots, s_n \texttt{ -> } s \in \mathcal{L}$ (*sign*) eine Signatur, dann seien

args $(s_1, \ldots, s_n \texttt{ -> } s) := \langle s_1, \ldots, s_n \rangle$

result $(s_1, \ldots, s_n \texttt{ -> } s) := s$

die *Parameterstrukturliste* bzw. die Rückgabestruktur der Signatur. ❒

Beispiel 4.1

Für eine Operation `insert` zum Einfügen in eine Menge ist <`{\e}`, `\e` > eine geeignete Parameterliste. Der erste Parameter ist eine Menge, deren Elementstruktur mit der Struktur des zweiten Parameters übereinstimmen muß, was durch die Verwendung derselben Strukturvariablen `\e` an zwei Stellen ausgedrückt wird. Findet zur Laufzeit beispielweise ein Aufruf mit der aktuellen Parameterliste $\langle r_1, r_2 \rangle$ mit *struct* $(r_1) = c_{set}$ (β_{int}) bzw. *struct* (r_2) = string statt, dann handelt es sich um einen Typfehler zur Laufzeit, da die Strukturvariable `\e` nicht gleichzeitig mit β_{int} und β_{string} belegt werden kann. ❒

Diese Syntax für Signaturen eignet sich sowohl für die noch einzuführende Sprache BASE-SCRIPT als für die Sprache SCRIPT$^+$ (siehe die Abschnitte 4.6 und 4.7).

Wird eine Operation mit der aktuellen Parameterliste $\langle r_1, \ldots, r_m \rangle \in V(Z^{alt})^*$ aufgerufen, dann muß zunächst sichergestellt sein, daß $\langle$*struct*$(r_1), \ldots,$ *struct*$(r_m)\rangle$ zu der Parameterstrukturliste $\langle s_1, \ldots, s_n \rangle$ der Signatur „paßt“. Eine notwendige Bedingung ist n = m. Ferner muß gelten:

Kommen in $\langle s_1, \ldots, s_n \rangle$ keine Strukturvariablen vor, so gilt *struct* $(r_i) = s_i$ für $1 \le i \le n$.
Werden in der Signatur Strukturvariablen $\backslash v_j$ verwendet, dann muß es eine Belegung der Strukturvariablen $\backslash v_j$ mit $\sigma_j \in \mathcal{S}$ (TYPES) geben, so daß *struct* $(r_i) = s_i'$ für $1 \le i \le n$, wobei die s'_i aus s_i durch Substitution aller $\backslash v_j$ durch σ_j hervorgehen.

Es ist wünschenswert, daß für die Kodierung der benutzerdefinierten Operationen eine Sprache verwendet wird, die *streng getypt* ist. Für einen gegebenen Quellcode kann dann zugesichert werden, daß zur Laufzeit keine Typfehler auftreten.

4.3 Basisoperationen

Zur „Grundausstattung“ des Datenmodells gehört neben den Basistypen und Konstruktoren eine *vordefinierte operationale Schnittstelle* zur Manipulation eines Laufzeitzustandes. Die Operationen dieser Schnittstelle werden *Basisoperationen* genannt. Der Aufruf einer Basis-

operation soll einen elementaren Zustandsübergang von Z^{alt} nach Z^{neu} auslösen.

Die Basisoperationen werden ähnlich wie die Typen und Tabellen aus TYPES bzw. TABLES als abstrakte Objekte o_i einer Menge BASEOPS = $\{o_1, \ldots, o_n\}$ eingeführt. Die Semantik einer Basisoperation o_i ist gegeben durch eine partielle Funktion

$$f_i \in \mathcal{PF}un\,(\mathcal{Z} \times \mathcal{V}^*, \mathcal{Z}), \tag{4.2}$$

die die möglichen elementaren Zustandsübergänge in Abhängigkeit von den Eingabeparametern beschreibt: Ist $Z^{alt} \in \mathcal{Z}$ der alte Zustand, die Liste $L = <r_1, \ldots, r_m> \in V(Z^{alt})^*$ die aktuelle, zur Signatur von o_i „passende“ Parameterliste und $f_i\,(Z^{alt}, L) = Z^{neu}$, dann ist Z^{neu} der Nachfolgezustand. Ist o_i eine Funktion, dann ist die Komponente R von Z^{neu} der Wurzelknoten des Baumes, der den zurückgegebenen Funktionswert repräsentiert.

Wir kommen nun zur formalen Definition von Basisoperationen.

Definition 4.3 (Basisoperationen)

Es sei eine endliche Menge BASEOPS = $\{o_1, \ldots, o_n\}$ gegeben, $n \in IN_0$, deren Elemente *Basisoperationen* heißen. Auf BASEOPS seien folgende Funktionen definiert:
Eine injektive Funktion

$$name: \text{BASEOPS} \rightarrow \mathcal{N},$$

die jeder Basisoperation einen Namen zuordnet. Ferner ordne

$$sign: \text{BASEOPS} \rightarrow \mathcal{S}ign\,(\text{BASETYPES, CONSTR}), \tag{4.3}$$

jeder Basisoperation eine einfache Signatur zu. Schließlich wird jeder Basisoperation über die Funktion

$$semantics: \text{BASEOPS} \rightarrow \mathcal{PF}un\,(\mathcal{Z} \times \mathcal{V}^*, \mathcal{Z}) \tag{4.4}$$

ihre Semantik zugeordnet.

Eine Operation mit *sign* (o) $\in \mathcal{S}ign$ (BASEOPS) wird *elementare Operation* genannt. Ihre Signatur nimmt also nur auf Basistypen bezug. Ist *sign* (o) eine generische Signatur, dann nennen wir o auch *generische Operation*. ❐

Die Operationen in BASEOPS gehören zu der Gruppe der „built-in“ Operationen, die *a priori* vorgegeben sind. Letztendlich soll die genaue Art der Implementation der Basisoperationen und insbesondere die dabei verwendete Sprache nicht interessieren. Deshalb geben wir ihre Semantik auch sprachunabhängig als eine Funktion aus $\mathcal{PF}un(\mathcal{Z} \times \mathcal{V}^*, \mathcal{Z})$ an, die elementare Zustandsübergänge beschreibt. Gleichwohl wählen wir in den nachfolgenden Unterabschnitten eine halbformale Pascal-artige Beschreibung zur algorithmischen Bestimmung des Nachfolgezustandes bei der Ausführung einer Basisoperation.

Zur Beschreibung der Semantik von Basisoperationen verwenden wir häufig eine „Generatorfunktion“

$$create: \mathcal{V}al^*(\text{TYPES}) \rightarrow \mathcal{V} \tag{4.5}$$

und eine „Kopierfunktion“

$$copy: \mathcal{V} \cdots> \mathcal{V} \tag{4.6}$$

mit folgender Semantik:

- *create* ((v, s)) erzeugt für (v, s) $\in \mathcal{V}al^*$(TYPES) eine Baumrepräsentation $(B_v, B_s) \in \varphi_{val}$ ((v, s)), die paarweise disjunkt zu allen bisher erzeugten Elementen aus

$D_{pers} \cup D_{trans}$ ist, fügt sie in D_{trans} ein und gibt *root*(B_v) zurück.

- Es sei r ∈ $\mathcal{V}$ der Wurzelknoten einer Baumrepräsentation aus $\mathcal{ValTree}$(TYPES). Dann ist *copy*(r) := *create*(*val*(r)), d.h. es wird eine „Kopie“ der Baumrepräsentation mit r als Wurzelknoten erzeugt.

4.3.1 Elementare Operationen

Zu der *a priori* gegebenen Menge BASETYPES gehört auch eine *a priori* gegebene Menge von elementaren Operationen. Es wäre nun mehr als müßig, an dieser Stelle alle elementaren Operationen aufzuführen, die für Basistypen aus BASETYPES relevant sind. Wir wollen daher annehmen, daß in BASEOPS u.a. die üblichen arithmetischen Operationen, Vergleichsprädikate und Stringoperationen enthalten sind. Wichtig ist für die Abgeschlossenheit des Datenmodells auf jeden Fall, daß die elementaren Operationen über die Menge BASEOPS „registriert“ sind. Zur Sicherung der Abgeschlossenheit des Datenmodells gehört auch, daß der Basistyp `boolean` zur „Minimalausstattung“ gehören muß, um Prädikate als Funktionen formulieren zu können.

Beispiel 4.2 Ein einfaches Beispiel für ein Element aus BASEOPS ist $o_{floatdiv}$ mit

- *name* ($o_{floatdiv}$) = "`floatdiv`"
- *sign* ($o_{floatdiv}$) = `float, float -> float`
- *semantics* ($o_{floatdiv}$) :

  Die aktuelle Parameterliste sei <x, y> mit x = (i, float), y = (j, float).

  if j = 0 then
  Ausnahme `div_by_zero` auslösen;
  else if (i = *null* or j = *null*)
  R^{neu} := *create*((*null*, float));
  else
  R^{neu} := *create*((i/j, float));

❒

Eine wesentliche Eigenschaft von elementaren Operationen soll sein, daß ihre Wirkung „nebeneffektfrei“ hinsichtlich ihrer Wirkung auf die persistenten Daten ist, d.h. für elementare Operationen soll $D_{pers}^{neu} = D_{pers}^{alt}$ gelten.

4.3.2 Generische Update-Operationen

In diesem Abschnitt werden die *generischen Update-Operationen* vorgestellt, die eine besonders wichtige Gruppe unter den Operationen in BASEOPS darstellen. Als Update-Operationen werden alle diejenigen Operationen bezeichnet, die die Bäume in $D_{pers} \cup D_{trans}$ aus Z^{alt} verändern.

a) Einfügen eines Elementes in eine Menge:

name: `insert`

sign: `{\s}, \s -> void`

semantics: Die aktuelle Parameterliste sei <S, e>.

if ($\exists$ r $\in$ *children* (S): *Tree*(r) $\cong$ *Tree*(e) [1] then
  Ausnahme `duplicate_insert` auslösen
else begin
  r := *copy* (e);
  Kante (S, r) einfügen
  (wobei *ord* (r) von der physischen Speicherung der Menge S abhängt);
end;

Wird also versucht, ein Duplikat in eine Menge einzufügen, dann wird eine Ausnahme erzeugt, die diese Information wiedergibt. In vielen Situationen kann so auf einen möglicherweise fehlerhaften Verlauf eines Bearbeitungsvorganges in einer Anwendung hingewiesen werden.

Es stellt sich die Frage, wieso zunächst eine Kopie erzeugt wird und nicht der Knoten e selbst in S eingefügt wird. Zu berücksichtigen ist jedoch, daß der Knoten e möglicherweise bereits einen Vaterknoten besitzt, so daß nach dem Einfügen ohne Kopieren der Knoten e zwei Väter hätte! Wäre e hingegen der Wurzelknoten eines Wertebaums, der eine Tabelle τ repräsentiert, dann ist ein Einfügen ohne Kopieren ebenfalls unzulässig, denn danach wäre $inst(\tau) \notin \{root(B_v) \mid (B_v, B_s) \in D_{pers}\}$, was nach Definition 3.20 nicht sein darf.

b) Einfügen eines Elementes in eine Multimenge:

name: `insertInBag`

sign: `{*\s*}, \s -> void`

semantics: Die aktuelle Parameterliste sei <B, e>.

r := *copy* (e);
Kante (B, r) einfügen
(wobei *ord* (r) von der physischen Speicherung der Multimenge B abhängt);

c) Einfügen eines Elementes in eine Liste:

name: `insertInList`

sign: `<\s>, \s, int -> void`

semantics: Die aktuelle Parameterliste sei <L, e, pos> mit pos = (n, int).

if n = *null* then
  Ausnahme `null_position` auslösen
else if (n ≤ 0 or n > *degree* (L) + 1) then
  Ausnahme `list_index_error` auslösen
else begin
  r := *copy*(e);
  Kante (L, r) einfügen mit ord^{neu} (r) = n
  (ord^{neu} wird dabei implizit gesetzt, d.h. für alle r' mit ord^{alt} (r') ≥ n
  wird ord^{neu} (r') := ord^{alt}(r') + 1 gesetzt);
end;

1. Diese Bedingung ist z.B. dann erfüllt, wenn *parent* (e) = S gilt!

d) Löschen eines Elementes aus einer Kollektion (Menge, Multimenge, Liste):

name: `remove`

sign: `\c, \s -> void`

semantics: Die aktuelle Parameterliste sei <C, e>.
Es gelte *type* (*struct* (C)) ∈ {c_{set}, c_{list}, c_{bag}}.
if *parent* (e) ≠ C then
Ausnahme `no_member` auslösen
else begin
Kante (C, e) entfernen
(*ord*^{neu} wird dabei implizit gesetzt, d.h. für alle r' mit *ord*^{alt} (r') > *ord*^{alt} (elem)
wird *ord*^{neu} (r') := *ord*^{alt}(r') – 1 gesetzt)
end;

e) Veränderung der Position eines Elementes in einer Liste:

name: `move`

sign: `<\s>, int, int -> void`

semantics: Die aktuelle Parameterliste sei <L, from, to> mit from = (n, int), to = (m, int).
if (n = *null* or m = *null*) then
Ausnahme `null_position` auslösen;
else if (n ≤ 0 or n > *degree* (L) or m ≤ 0 or m > *degree* (L)) then
Ausnahme `list_index_error` auslösen
else
ord^{neu} (r) := m für r ∈ *children* (L) mit *ord*^{alt} (r) = n
(Für alle anderen Knoten r' ∈ *children* (L) wird *ord*^{neu} entsprechend
angepaßt);

f) Update auf einfachen Knoten oder Objekt-Knoten:

name: `update`

sign: `\s, \s -> void`

semantics: Die aktuelle Parameterliste sei <r_1, r_2>. Es gelte:
isSimple (*struct* (r_1)) ∨ *type*(*struct* (r_1)) ∈ DEFTYPES.
value^{neu}(r_1) := *value*^{alt}(r_2)[2];

Die Update-Operation `update` ist also nur für Knoten geeignet, deren Struktur entweder ein einfacher Typ oder ein Objekttyp ist. Für das Update eines Knotens, dessen Struktur ein variabler Typ ist, ist die nachfolgende Operation `updateVar` vorgesehen. Für die Modifikation von Varianten wird eine eigene Update-Operation bereitgestellt, da dabei i.a. nicht allein eine Modifkation der Beschriftung *value* stattfindet, sondern vorhandene Teilbäume nicht mehr erreichbar gemacht bzw. neue Teilbäume erzeugt werden.

---

2. Zur Erinnerung: Die Funktion *value* ist die Beschriftung der Blattknoten einesWertebaumes (vgl. Abschnitt 3.5.2.2)

g) Update einer Variante:

name: `updateVar`

sign: `\s, \s -> void`

semantics: Die aktuelle Parameterliste sei <r_1, r_2>. Es gelte *type*(*struct* (r_1)) ∈ VARTYPES.

if *label*(r_1) ≠ *null* then
  Kante (r_1, *get_child*(r_1, 1)) entfernen;
label^{neu}(r_1) := *label*(r_2)*;*
if *label*(r_2) ≠ *null* then begin
  r := *copy* (*get_child*(r_2, 1));
  Kante (r_1, r) hinzufügen
end;

4.3.3 Weitere generische Operationen

Neben den generischen Update-Operationen definieren wir noch einige generische Operationen, die jedoch keine Update-Operationen sind.

a) Test auf Wertegleichheit zweier komplexer Werte:

name: `equal`

sign: `\s, \s -> boolean`

semantics: Die aktuelle Parameterliste sei <r_1, r_2>.

R^{neu} := *create* ((*Tree*(r_1)≅*Tree*(r_2), boolean));

b) Anzahl der Elemente einer Kollektion:

name: `card`

sign: `\c -> int`

semantics: Die aktuelle Parameterliste sei <C>. Es gelte *isCollection* (*struct* (C)).

R^{neu} := *create* ((*degree* (C), int));

c) Test, ob eine Kollektion leer ist:

name: `empty?`

sign: `\c -> boolean`

semantics: Die aktuelle Parameterliste sei <C>. Es gelte *type* (*struct* (C)) ∈ {c_{set}, c_{list}, c_{bag}}.

R^{neu} := *create* ((*degree* (C) = 0, boolean));

d) Position eines Elementes in einer Liste:

name: `pos`

sign: `\e -> int`

semantics: Die aktuelle Parameterliste sei <elem>.

Es gelte $\neg$ *isRoot* (elem) $\wedge$ *struct* (*parent* (elem)) = c_{list}.

R^{neu} := *create* ((*ord* (elem), int));

Die folgende Basisoperation wird benötigt, um einen „type cast“ eines varianten Wertes auf die für ihn zutreffende Alternative durchzuführen.

e) Zugriff auf den „eigentlichen“ Wert einer Instanz eines varianten Typs:

name: `strip`

sign: `\t -> \s`

semantics: Die aktuelle Parameterliste sei <r> mit r = (v, t), wobei t $\in$ VARTYPES.

if v = *null* then
  Ausnahme `null_variant` auslösen;
else
  R^{neu} := *get_child* (r, 1);

Falls v $\neq$ *null*, dann ist r = ((*l*, v), t) mit (*l*, s) $\in$ *variants*(t). Daraus ergibt sich die Belegung von `\s` mit der Struktur s.

4.3.4 Operationen für Objekttypen

Im Zusammenhang mit Objekttypen sind die bereits in den Abschnitten 3.6 und 3.7.2 erwähnten Operationen `initObject` und `addType` von zentraler Bedeutung. Sie benötigen als Eingabeparameter den Namen eines Objekttyps. Um zu erreichen, daß auch Typen als Parameter von Operationen zulässig sind, müssen diese ebenfalls getypte Datenobjekte sein. Dazu haben wir mit der Einführung eines Meta-Schemas in Abschnitt 3.10 bereits die notwendigen Voraussetzungen geschaffen. Im Meta-Schema ist ein Objekttyp $t_{deftype}$ mit dem Namen `deftype` eingeführt worden. In jedem Datenbankschema sind die Objekttypen Instanzen des Objekttyps $t_{deftype}$. In Signaturen der folgenden Operationen verwenden wir somit den Namen `deftype`, wenn ein Objekttyp, d.h. ein getyptes Datenobjekt (t, $t_{deftype}$), als Parameter übergeben werden soll.

a) Erzeugen einer neuen Instanz zu einem Objekttyp:

name: `initObject`

sign: `deftype -> \t`

semantics: Die aktuelle Parameterliste sei <r> mit *val* (r) = (t, $t_{deftype}$) mit t $\in$ DEFTYPES und t ist ein *isa*-Wurzeltyp.

R^{neu} := *create* ((ω, t));[3]

$I(t)^{neu}(\omega)$:= *create*(*initVal*(*struct*(t)));

Die Operation `initObject` wird mit solchen Typen als Parameter aufgerufen, die keine Supertypen besitzen. Die folgende Operation `addType` ermöglicht das Hinzufügen weiterer Typen, so daß für ein Objekt multiple Typzugehörigkeit erlangt werden kann.

b) Hinzufügen eines weiteren Typs zu einem Objekt:

3. Dabei ist ω ein noch nicht verwendetes Element aus Ω.

name: `addType`

sign: `\t, deftype -> \t2`

semantics: Die aktuelle Parameterliste sei $<r_1, r_2>$ mit $val(r_1) = (\omega, t)$ und $val(r_2) = (t', t_{deftype})$, wobei $t, t' \in$ DEFTYPES.

```
if ω = null then
    Ausnahme null_object auslösen
else if t' ∈ types(ω) then
    Ausnahme type_already_added auslösen
else if not (t' isa* t) then
    Ausnahme no_subtype auslösen
else begin
    I(t')^neu(ω) := create(initVal(struct(t')));
    R^neu := create ((ω, t'));
    add_type := (∃ t'' ∈ DEFTYPES: t'' isa t' ∧ t'' ∉ types(ω));
    while add_type do begin
        I(t'')^neu(ω) := create(initVal(struct(t'')));
        add_type := (∃ t'' ∈ DEFTYPES: t'' isa t' ∧ t'' ∉ types(ω))
    end
end;
```

Durch `addType` werden außer t' also auch alle Supertypen von t' hinzugefügt, sofern das Objekt ω diese vorher noch nicht besaß. Auf diese Weise werden „Typ-Lücken“ des Objektes ω hinsichtlich der *isa*-Hierarchie vermieden, und die Bedingung (3.30) wird eingehalten. Sowohl `initObject` als auch `addType` initialisieren die neuen *pref*-Zähler mit 0. Ferner ergibt sich die Belegung der Strukturvariable `\t2` mit t'.

Die folgenden Operationen sind notwendig für den Umgang mit multipler Typzugehörigkeit von Objekten.

c) Typ-Perspektive für ein Objekt ändern:

name: `cast`

sign: `\t, deftype -> \t2`

semantics: Die aktuelle Parameterliste sei sei $<r_1, r_2>$ mit $val(r_1) = (\omega, t)$ und $val(r_2) = (t', t_{deftype})$, wobei $t, t' \in$ DEFTYPES.

```
if t' ∈ types(ω) then
    R^neu := create ((ω, t'));
else begin
    Ausnahme incorrect_type_cast auslösen;
    R^neu := create ((null, t'));
end;
```

Mittels `cast` kann also ein „type cast“ durchgeführt werden. Konnte für das Objekt ω zunächst via r_1 nur auf $I(t)(\omega)$ zugegriffen werden, so ist via R^{neu} nun ein Zugriff auf $I(t')(\omega)$ möglich.

d) Test auf persistente Typzugehörigkeit eines Objekts:

name: `hasPersType?`

sign: `\t, deftype -> boolean`

semantics: Die aktuelle Parameterliste sei sei $<r_1, r_2>$ mit $val(r_1) = (\omega, t)$ und $val\ (r_2) = (t', t_{deftype})$, wobei t, t' ∈ DEFTYPES.

$R^{neu} := create\ ((\omega \in Def(I(t')) \wedge pref(I(t)(\omega) > 0, boolean))$;

Analog zu `hasPersType?` gibt es einen weiteren Test `hasTransType?` mit

$R^{neu} := create\ ((\omega \in Def(I(t')) \wedge pref(I(t)(\omega) = 0, boolean))$;

4.4 Links und Iteratoren

Wir haben komplexe Datenobjekte und die zugehörigen Operationen unter der intensionalen Semantik [Cat+94] betrachtet, nach der beispielsweise eine Menge ein „Behälter“ ist, der während seiner Lebensdauer wechselnden Inhalt hat, nämlich gerade die in der Menge enthaltenen Elemente, die sich auf „Plätzen“ innerhalb des „Behälters“ befinden. Dies führte uns zu den Baumrepräsentationen, bei denen Knoten die Rolle von „Behältern“ und „Plätzen“ spielen.

Es stellt sich nun die Frage, ob für das Datenmodell ESCHER$^+$ ein Zeigertyp benötigt wird, wie man ihn aus vielen Programmiersprachen kennt. Auf diese Weise werden jedoch die interne und logische Ebene vermischt, da Verweise als Instanzen von Zeigertypen implementationsabhängige Größen (Adressen) sind. Motivation für die Verwendung von Zeigern in Programmiersprachen ist i.d.R. der Wunsch, *physisches* object sharing zu ermöglichen, indem mehrere Verweise auf eine und dieselbe Speicherstelle existieren.

Für ESCHER$^+$ benötigen wir jedoch keinen weiteren Mechanismus für object sharing, da bereits *logisches* object sharing durch die Verwendung von Objekttypen unterstützt wird: Sind zwei unterschiedliche Knoten r_1 und r_2 mit $val\ (r_1) = (\omega, t)$ und $val\ (r_2) = (\omega, t)$ erreichbar, dann „teilen“ sich diese beiden Knoten die Interpretation $I\ (t)\ (\omega)$.

Es sollte unbedingt der Versuchung widerstanden werden, die „Identifikatoren“ der Knoten als Datentyp im logischen Datenmodell anzubieten, nur weil es aus implementatorischer Sicht keine Schwierigkeiten machen würde, von Bäumen zu allgemeineren Graphenstrukturen überzugehen. Die *Object-Manager*-Schnittstelle [Weg91b] des ESCHER-Prototyps operiert ebenfalls auf Baumdarstellungen komplexer Werte, die in etwa die von uns gewählte Form haben. Jeder Knoten hat einen physischen „Identifikator“, der durch einen sog. Record-Identifier (RID) gegeben ist. In einer früheren Version von ESCHER wurde im logischen Datenmodell ein Datentyp *Link* angeboten, der aus einem „Hochhiefen“ dieser physischen Identität in das logische Modell entstand und der dem Benutzer wie jeder andere Datentyp zur Verfügung stand.

Trotz der soeben gemachten Aussagen ist es in einigen Situationen wünschenswert, zur Laufzeit Zeiger bzw. Verweise auf Knoten eines Baumes zur Verfügung zu haben:

- Für einen Scan der Elemente einer Kollektion benötigt man – wie bei Embedded SQL – einen *Cursor*, der mittels navigatorischer Operationen sukzessive alle Elemente der Kollektion durchläuft. In der Sprache SCRIPT [Pau94] gibt es eine `for`-Schleife, in der ein *Iterator* über alle Elemente einer Kollektion läuft, die eine angegebene Bedingung erfül-

len. In beiden Fällen ist die aktuelle Iterator-Position durch einen Zeiger auf einen Knoten gegeben.

- Auf der graphischen Oberfläche des ESCHER-Prototyps bewegt der Benutzer sog. Finger (vgl. Abschnitt 1.1). Ein Finger ist nichts anderes als ein Zeiger auf einen Knoten eines Wertebaumes.
- Es sei `Zahlen` eine Menge von integer-Werten. Die Sprache SCRIPT [Pau94] erlaubt die Anweisung `Zahlen rem 42`, die das Element 42 aus `Zahlen` löschen soll. Zunächst ist jedoch der Knoten r ∈ *children* (α(`Zahlen`)) [4] mit *value* (r) = 42 zu finden (falls er überhaupt existiert!). Die „Adresse“ r muß an geeigneter Stelle zwischengespeichert werden, um im nächsten Schritt die `remove`-Operation mit der aktuellen Parameterliste <α(`Zahlen`), r> aufzurufen.
- In Abschnitt 5.4 des nachfolgenden Kapitels, in dem Integritätsbedingungen behandelt werden, wird gezeigt, daß gewisse Integritätsbedingungen nur dann effizient überprüft werden können, wenn die zur Überprüfung benötigten Knoten möglichst „schnell“ ermittelt werden können. Dazu werden zusätzliche Indexe gepflegt, die Verweise auf die jeweils relevanten Knoten liefern.

Es muß jedoch noch einmal nachdrücklich betont werden, daß der Gebrauch von Zeigern für die oben genannten Situationen reserviert sein soll. Ein Zeigertyp soll insbesondere nicht bei der Definition benutzerdefinierter Datenbankschemata zur Verfügung stehen, da sie auf logischer Ebene unerwünscht sind. Ein Zeigertyp soll eben kein zu allen anderen Datentypen gleichberechtigter Datentyp des logischen Modells sein, sondern auf die interne Ebene beschränkt bleiben.

Definition 4.4 (Links)

Wir erweitern $\mathcal{S}$(TYPES) zu $\mathcal{S}^+$(TYPES), indem wir zu CONSTR den *Link-Konstruktor* c_{link} hinzufügen und in Def. 3.9 folgende Zusatzregel einführen:

Ist $s \in S_k$ für $0 \le k \le i$, dann ist $c_{link}(s) \in S_{i+1}$.

Es ist dann $\mathcal{S}^+$(TYPES) wiederum die Vereinigung aller S_i, $i \ge 0$, die bei Hinzunahme dieser Zusatzregel gebildet werden können. Wir setzen *name* (c_{link} (s)) := `link(`*name*(s)`)`.
Der Wertebereich ist abhängig vom aktuellen Laufzeitzustand Z:

dom (c_{link} (s)) := {r ∈ V(Z) | *struct* (r) = s} ∪ {*null*}.

Das bedeutet, daß der Wertebereich nur die Knoten umfaßt, die die Struktur s besitzen, d.h. es gibt nur „getypte“ Verweise. Ein Element von *dom* (c_{link} (s)) wird *Link* vom Typ s genannt. ❒

Ein Wertebaum für einen getypten Link (r, c_{link}(s)) ist ein einzelner Knoten r' mit den Beschriftungen *struct* (r') = c_{link}(s) und *value* (r') = r (vgl. Abb. 4.1).

Abbildung 4.1: graphische Veranschaulichung eines Links

4. α ist die Funktion aus Def. 4.1, die Bezeichner an Knoten in Wertebäumen bindet.

Im folgenden werden die grundlegenden Operationen zur Manipulation von Links vorgestellt, wenn sie als Werte von Iteratoren verwendet werden. Sie gehören auch zur Menge BASEOPS von Basisoperationen.

a) Setzen eines Links auf das erste Element einer Kollektion

name: `firstIn`

sign: `\c -> link(\s)`

semantics: Es sei <C> die aktuelle Parameterliste. Es gelte *isCollection*(*struct* (C)).

s := *substruct* (*struct* (C));
if *children* (C) ≠ ∅ then
  R^{neu} := *create* ((*get_child* (C, 1), link(s)))[5]
else begin
  R^{neu} := *create* ((*null*, link(s)));
  Ausnahme `empty_collection` auslösen
end;

b) Test, ob ein Link einen definierten Wert hat:

name: `defined?`

sign: `link(\s)-> boolean`

semantics: Die aktuelle Parameterliste sei <r> mit r = (r', link(s)).

R^{neu} := *create* ((r' = *null*, link (s)));

c) Freigabe eines Links:

name: `free`

sign: `link(\s)-> void`

semantics: Die aktuelle Parameterliste sei <r>.

$value^{neu}$ (r) := *null*;

Die folgenden Operationen sind für die Verwendung eines Links als Iterator bestimmt. Sie führen eine „Bewegung“ des Iterators durch Modifikation der Beschriftung *value* aus bzw. testen, ob eine solche Bewegung möglich ist.

d) Vorwärtsbewegung eines Iterators:

name: `next`

sign: `link(\s) -> boolean`

semantics: Die aktuelle Parameterliste sei <it> mit it = (r, link(s)).

if r = *null* then
  Ausnahme `null_link` auslösen
else if (*isRoot* (r) or not *isCollection* (*struct* (*parent* (r)))) then
  Ausnahme `not_in_collection` auslösen

5. Daraus folgt auch die Belegung der Strukturvariablen `\s` mit s.

else begin
  if *ord*^{alt} (r) < *degree* (*parent* (r)) then begin
    value^{neu} (it) := *get_child* (*parent* (r), *ord* (r) + 1)
  else
    value^{neu} (it) := *value*^{alt} (it);
  R^{neu} := *create* ((*value*^{alt}(it) ≠ *value*^{neu} (it), boolean))
end;

e) Positionieren eines Iterators auf das erste Element:

name: `first`

sign: `link(\s) -> boolean`

semantics: Die aktuelle Parameterliste sei <it> mit it = (r, link(s)).

if r = *null* then
  Ausnahme `null_link` auslösen
else if (*isRoot* (r) or not *isCollection* (*struct* (*parent* (r)))) then
  Ausnahme `not_in_collection` auslösen
else begin
  value^{neu} (it) := *get_child* (*parent* (r), 1);
  R^{neu} := *create* ((*value*^{alt}(it) ≠ *value*^{neu} (it), boolean))
end;

f) Test, ob ein Iterator auf das erste Element einer Kollektion zeigt:

name: `onFirst?`

sign: `link(\s) -> boolean`

semantics: Die aktuelle Parameterliste sei <it> mit it = (r, link(s)).

if r = *null* then
  Ausnahme `null_link` auslösen
else if (*isRoot* (r) or not *isCollection* (*struct* (*parent* (r)))) then
  Ausnahme `not_in_collection` auslösen
else
  R^{neu} := *create* ((*ord* (r) = 1, boolean));

Neben den bisher genannten Operationen sind die Operationen `lastIn`, `prior`, `last`, `onLast?` auf analoge Weise definiert.

g) Erzeugen eines Link-Wertes, der auf einen gegebenen Knoten zeigt:

name: `getLink`

sign: `\s -> link(\s)`

semantics: Die aktuelle Parameterliste sei <r>.

R^{neu} := *create* ((r, link(*struct* (r)));

4.5 Benutzerdefinierte Operationen

4.5.1 Kodierte Operationen

Für Anwendungen, die auf einer Datenbank operieren, ist es nötig, anwendungsspezifische Operationen definieren zu können. Sie werden analog zu dem Vorgehen für die Basisoperationen in einer Menge DEFOPS zusammengefaßt. Damit es sich wirklich um *benutzerdefinierte* Operationen handelt, soll sich die Semantik einer Operation aus DEFOPS im Unterschied zu den Basisoperationen durch den Benutzer (i.d.R. der Anwendungsprogrammierer oder der Datenbankadministrator) in geeigneter Form angeben lassen. Dazu schreibt der Benutzer Code in einer geeigneten Sprache. Es wird also, in der Terminologie von [AKW92], zwischen *base methods* (Operationen in BASEOPS) und *coded methods* (Operationen in DEFOPS) unterschieden.

In [Pau94] wurde für ESCHER die Sprache SCRIPT zum Kodieren benutzerdefinierter Operationen eingeführt. Ein *Script* (d.h. eine in SCRIPT geschriebene Funktion) besteht jedoch nicht einfach aus Aufrufen von Basisfunktionen, sondern SCRIPT versteht sich als höhere Programmiersprache mit einer benutzerfreundlichen Syntax, die auch die Bildung komplexer Ausdrücke zuläßt. Ein Script muß vor der Ausführung deshalb zunächst in ein sog. *Basisscript* übersetzt werden, das quasi seine „Assembler-Fassung“ darstellt. Ein Basisscript ist ein Script einer *Basissprache*, die wir BASESCRIPT nennen wollen und die gegenüber SCRIPT eine stark vereinfachte Syntax hat. Der Anweisungsteil eines Basisscriptes läßt sich schrittweise durch einen Interpreter abarbeiten. Die Syntax ist so konzipiert, daß in jedem Schritt höchstens eine Basisoperation oder ein anderes Basisscript aufgerufen wird. Die Semantik eines in SCRIPT geschriebenen Scriptes ist dann durch die Ausführung des entsprechenden Basisscriptes gegeben.

Für die Definition benutzerdefinierter Operationen gehen wir daher gleich davon aus, daß ihre Kodierung in Form eines Basisscriptes vorliegt, das durch ein Wort aus $\mathcal{L}$(*baseScript*) gegeben ist, wobei *baseScript* ein Nichtterminal der Sprache BASESCRIPT ist. Es sei jedoch darauf hingewiesen, daß BASESCRIPT zu „primitiv“ ist, als daß sie das geeignete Sprachlevel für einen Anwendungsprogrammierer darstellt. Vielmehr soll BASESCRIPT Zielsprache einer Übersetzung aus einer höheren Programmiersprache sein. In Abschnitt 4.7.3 werden einige Beispiele zur Übersetzung von SCRIPT$^+$, unserer Variante von SCRIPT, nach BASESCRIPT angegeben.

Definition 4.5 (benutzerdefinierte Operationen)

Zu jedem ESCHER$^+$-Datenbankschema gehört eine Menge DEFOPS = $\{o_1, \ldots, o_n\}$, $n \in \mathbb{N}_0$, die zu BASEOPS disjunkt ist. Die Elemente von DEFOPS heißen *benutzerdefinierte Operationen*. Auf DEFOPS sind folgende Funktionen definiert:

name: DEFOPS $\rightarrow \mathcal{N}$

sign: DEFOPS $\rightarrow \mathcal{S}ign$ (TYPES)[6]

Ferner gibt es eine Funktion

6. Man beachte, daß in Def. 4.3 an dieser Stelle $\mathcal{S}ign$ (BASETYPES) steht, d.h. erst in DEFOPS ist die Verwendung von Objekttypen in Signaturen erlaubt.

code : DEFOPS → $\mathcal{L}$(*baseScript*),

die jedem o ∈ DEFOPS Programmcode in textueller Form zuordnet. ❐

An dieser Stelle soll ein erster Eindruck von der Syntax der Sprache BASESCRIPT gegeben werden. Der äußere Rahmen eines Basisscriptes orientiert sich im „Look & Feel“ an der Sprache SCRIPT aus [Pau94]:

baseScript ::= '`[`' *sign* '`|`' *deklTeil* '`|`' *anwTeil* '`]`' (4.7)

Die Syntax für *sign* entspricht der Syntax für *sign* aus Def. 4.2, allerdings müssen die Argumente Namen haben.

Der Deklarationsteil *deklTeil* besteht aus einer (möglicherweise leeren) Folge von Variablendeklarationen der Form

deklTeil ::= { *identifier* '`:`' *struct* '`;`' }

In Abschnitt 4.6 gehen wir detailliert auf die Syntax des Anweisungsteils sowie die Semantik von Basisscripten ein. Man beachte, daß wir in BASESCRIPT Variablen vom Typ `link(s)` zulassen!

4.5.2 Methoden für Objekttypen

Nach der Einführung der benutzerdefinierten Operationen können wir auch den operationalen Teil (Verhaltensteil) einer Objekttyp-Definition angeben.

Dazu ergänzen wir die Menge der auf DEFTYPES definierten Funktionen (vgl. Def. 3.17) um die Funktion

ops: DEFTYPES → $\mathcal{FSet}$(DEFOPS), (4.8)

die jedem Objekttyp eine Menge benutzerdefinierter Funktionen (*Methoden*) aus DEFOPS zuordnet. Für alle t ∈ DEFTYPES muß gelten:

o ∈ *ops* (t) ⇒ *args*(*sign*(o))[1] = t, (4.9)

d.h. der erste Parameter („Receiver“) ist vom Typ t selbst.

In der DDL-Anweisung `define type` werden Name und Signatur von Methoden über die `ops`-Klausel definiert:

ops ::= `-ops` *opList*

opList ::= *opDecl* {'`,`' *opDecl* }

opDecl ::= *identifier* '`(`' *sign* '`)`'

Dabei sollen die Namen der Operationen paarweise disjunkt sein. Mit der Abarbeitung einer `define type`-Anweisung wird für jedes Element der *opList* eine neue Methode o_{neu} ∈ DEFOPS generiert. Für

opname `(` p_1`:` s_1`, …,` p_k`:` s_k `->` s `)` ∈ $\mathcal{L}$(*opDecl*)

wird *name* (o_{neu}) := opname, *sign* (o_{neu}) := `self` : t, p_1`:`s_1`, …,` p_k`:`s_k `->` s [7] und

$$code(o_{neu}) := \begin{cases} \texttt{[self:}t_{neu}\texttt{,}\ p_1\texttt{:}\ s_1\texttt{, …,}\ p_k\texttt{:}\ s_k\ \texttt{|| return}\ initVal\ (s)\texttt{]} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{, falls } s \neq \texttt{void} \\ \texttt{[self:}t_{neu}\texttt{,}\ p_1\texttt{:}\ s_1\texttt{, …,}\ p_k\texttt{:}\ s_k\ \texttt{|| return]}, \text{ falls } s = \texttt{void} \end{cases}$$

Es wird also *code*(o_{neu}) zunächst mit einer „trivialen Implementation“ initialisiert.

Nach Auswertung der `ops`-Klausel ist *ops* (t) die Menge aller neu generierten Operationen.

Die Kodierung einer Methode o, d.h. *code*(o), läßt sich über folgende DDL-Anweisung ändern:

```
modify
  -op defType'.'opName
  -code script
end modify;
```
(4.10)

Dabei ist *defType* der Name eines Objekttyps. Das Script *script* sei bereits in die Basissprache übersetzt, so daß weiterhin *code*(o) $\in \mathcal{L}$(*baseScript*) gilt. In der `op`-Klausel muß der Name der Methode mit dem Typnamen qualifiziert werden, da in unterschiedlichen Typen gleichnamige Methoden definiert sein können.

Beispiel 4.3 Der Definition des Objekttyps `Buch` aus Beispiel 3.6 kann nach der `struct`-Klausel folgende `ops`-Klausel hinzugefügt werden:

```
-ops AnzAutoren(-> integer),
```

wobei `AnzAutoren` die Anzahl der Autoren eines Buches zurückgeben soll. Die intendiert Semantik wird ermöglicht, wenn die fehlende Implementation über die `modify`-Anweisung gesetzt wird, indem man für *script*

```
[self: Buch -> string
|c: integer|
c := card(self.Autoren);
return c;]
```

einsetzt[8]. ❒

Natürlich muß nach einer Modifikation das neue Script zur gegebenen Signatur konform sein, d.h. die Parameter müssen jeweils genau die in der Signatur angegebene Struktur haben. Ist dies der Fall, dann schreiben wir *conform*(*code*(o), *sign*(o)). Grundsätzlich muß also gelten:

$$\forall\, o \in \text{DEFOPS: } \mathit{conform}(\mathit{code}(o), \mathit{sign}(o)) \tag{4.11}$$

Gegenüber [Pau94] fällt auf, daß wir hier die Modifikation eines Scriptes zum Bestandteil der DDL machen, während in [Pau94] Funktionen selbst über ein Script modifiziert werden können. Dies liegt in den unterschiedlichen Ansätzen der Integration einer operationalen Komponente in das Datenmodell begründet. Im Datenmodell nach [Pau94] wird der operationale Teil durch scriptwertige Attribute modelliert, deren Typ gerade durch die Angabe einer Signatur festgelegt ist. Die Modifikation eines scriptwertigen Attributes erfolgt dann wie die Modifikation jedes anderen Attributes über die DML, sprich über ein Script. In unserem Datenmodell wird deutlicher zwischen der strukturellen und der operationalen Komponente eines Objekttyps unterschieden. Dies trägt der Tatsache Rechnung, daß die Spezifikation und Implementation des Verhaltens eines Objektes viel eher in den Bereich der konzeptuellen Modellierung

7. D.h. formaler Name des Receiver-Parameters ist `self`.
8. Wir verzichten an dieser Stelle auf nähere Angaben zur Sprache SCRIPT$^+$ und gehen davon aus, daß das angegebene Beispiel direkt verständlich ist.

gehört, also über die DDL erfaßt werden muß, und nicht als eine ad-hoc-Entscheidung „im laufenden Betrieb“ über die DML abgehandelt wird.

In Abschnitt 3.7 wurde die Relation *isa* als Subtypbeziehung eingeführt und die damit in Verbindung stehende Vererbung struktureller Eigenschaften erläutert. Bei objektorientierten Datenmodellen kommt die Vererbung von Operationen und ihren Implementierungen dazu. Dies kann auch als *code sharing* zwischen verschiedenen Typen bezeichnet werden. Dadurch wird eine unnötige Reimplementation von Operationen in den Subtypen vermieden, was eine sehr augenfällige Form der Wieder- oder auch Weiterverwendung darstellt. Andererseits soll auch eine Redefinition einer Operation möglich sein, für die es bereits eine gleichnamige Operation in einem direkten oder indirekten Supertyp gibt, um die Vererbung zu unterdrücken, falls dies notwendig ist („Overriding“). Dies ist die Parallele zur (wesentlich seltener vorkommenden) Redefinition von Attributen in Objekttypen. Eine Redefinition einer Methode findet für einen Objekttyp statt, wenn der Methodenname und eine Signatur in der `ops`-Klausel der Objekttypdefinition vorkommt.

Für ESCHER$^+$ soll es auch die für OOPLs übliche Vererbung von Methodenimplementationen geben. Für die Methodenresolution, d.h. die Suche nach der für einen Objekttyp und einen Methodennamen relevanten Implementation, wird nun ein Algorithmus angegeben.

Es sei METHODS := $\bigcup_{t \in \text{DEFTYPES}} \mathit{ops}(t)$

Für jeden Objekttyp t ∈ DEFTYPES sei die Funktion

$\mathit{OpAccess}\ (t) : \mathcal{N} \cdots > \text{METHODS}$

definiert, die für einen Objekttyp t jedem gültigen Methodennamen die korrekte Methode aus METHODS zuordnet. Der Algorithmus zur Bestimmung von *OpAccess* lautet:

1. Setze $f_t : \mathcal{N} \cdots > \text{METHODS}$ mit
   $\text{Def}(f_t) := \{\mathit{name}\ (o) \mid o \in \mathit{ops}\ (t)\}$ und $f_t\ (m) := \mathit{name}^{-1}(m)$
   (Man beachte, daß auf $\text{Def}(f_t)$ die Umkehrfunktion von *name* existiert!)
2. Falls t *isa* t' für ein t' ∈ DEFTYPES, dann führe durch:
   2.1. Bestimme *OpAccess* (t') und setze $f_{t'} := \mathit{OpAccess}\ (t')$
   2.2. Berücksichtige das „Overriding“ von Methoden:
   $f_{t'}\ (m) := \perp$ für alle $m \in D := \text{Def}(f_t) \cap \text{Def}(f_{t'})$
   (d.h. die Methodenimplementation des Subtyps hat Vorrang)
   2.3. Setze $f_t := f_t \cup f_{t'}$
3. f_t ist die gesuchte Funktion *OpAccess* (t).

Mit Hilfe der Funktionen $\mathit{OpAccess}(t)$ können – ähnlich wie bei den Funktionen $\mathit{AttrAccess}(t)$ aus (3.28) – die immer gleichen Suchvorgänge entlang der Subtyp-Hierarchie vermieden werden.

Gilt $\mathit{OpAccess}(t)(m) = o$, aber $o \notin \mathit{ops}(t)$, dann ist $o \in \mathit{ops}(t')$ für einen Supertyp t' von t, d.h. der Typ t erbt die Implementation $\mathit{code}(o)$ der Methode mit dem Namen m von einem Supertyp t'.

Eine nachträgliches Hinzufügen einer Methode bzw. die Redefinition der Implementierung einer Methode („Overriding“) namens m für den Typ t erfolgt über die DDL-Anweisung

```
define operation
  -name defType '.' identifier
  -sign sign
  [ -code script ]
end define;
```

Für *defType* wird der Name des Objekttyps t und für *identifier* der Name m der Methode eingesetzt. Voraussetzung ist, daß noch kein $o \in ops(t)$ mit $name(o) = m$ existiert. War m bereits in $Def(OpAccess(t))$, dann handelt es sich um eine Redefinition, ansonsten um das Hinzufügen einer neuen Methode. Es wird eine neue Operation o' in $ops(t)$ und in DEFOPS eingefügt und $name(o'') := m$, $sign(o'') := sign(o')$ gesetzt. Die „Implementation“ $code(o'')$ ergibt sich aus der `code`-Klausel. Fehlt sie, so wird $code(o')$ mit der „trivialen Implementation“ besetzt. Es muß dann auch $OpAccess^{neu}(t')(m) := o'$ für alle Objekttypen t' mit t' *isa** t gesetzt werden, für die $OpAccess^{alt}(t')(m) = OpAccess^{alt}(t)(m)$ galt.

Genauso wie bei der Redefinition von Attributen sind bei der Redefinition von Methoden zunächst keine Einschränkungen an die Gestalt der Signatur der redefinierten Methode erforderlich. In vielen OOPLs wird jedoch die Einhaltung der sog. *Kontravarianzregel* gefordert:

> Es sei $\preceq$ die in Abschnitt 3.11 erwähnte Subtyp-Relation auf $\mathcal{S}$(TYPES).
> Gilt t *isa** t' und gibt es $o \in ops(t)$ und $o' \in ops(t')$ mit $name(o) = name(o')$, dann gilt für die Signaturen $sign(o) = t \times s_1 \times \ldots \times s_n \rightarrow s$ und $sign(o') = t' \times s'_1 \times \ldots \times s'_m \rightarrow s'$:
> (1) $n = m$
> (2) $\forall\ 1 \le i \le n: s'_i \preceq s_i$
> (3) $s \preceq s'$
> Die Signaturen stehen dann in einer Kontravarianz-Beziehung zueinander,
> und wir schreiben $sign(o) \le_{contrav} sign(o')$.

Die Kontravarianzregel ist notwendig, um beim dynamischen Binden (*late binding*) Typsicherheit zu gewährleisten. Wir verzichten auf eine ausführliche Motivation der Kovarianzregel und verweisen auf die Literatur [KA90, Kem91, KM94]. In ESCHER$^+$ kommt spätes Binden nur im Zusammenhang mit varianten Typen vor. Dies wird in Abschnitt 4.7.4 behandelt, und dort kommen wir auch auf die Kovarianzregel zurück.

4.5.3 Weitere benutzerdefinierte Operationen

Um auch unabhängig von Typdefinitionen benutzerdefinierte Operationen definieren zu können, verwenden wir ebenfalls das DDL-Statement `define operation`, allerdings mit der `name`-Klausel

```
-name identifier
```

Das Qualifizieren des Namens mit einem Typnamen entfällt hier. Seine Abarbeitung erzeugt eine neue Operation o_{neu}, die Element einer Menge USEROPS $\subseteq$ DEFOPS wird, die zu METHODS disjunkt ist. Es gelte DEFOPS = METHODS $\cup$ USEROPS. Die Elemente von USEROPS werden *unabhängige* oder *freie Operationen* genannt.

Die Namen der Operationen soll innerhalb USEROPS eindeutig sein, d.h. *names* eingeschränkt auf USEROPS ist injektiv. Voraussetzung für eine gültige `define operation`-

Anweisung ist also, daß der gewählte Name noch für keine unabhängige Operation verwendet wurde.

Modifiziert werden kann *code* (o) für ein o ∈ USEROPS wiederum über eine `modify`-Anweisung, wobei die Qualifizierung des Operationennamens mit einem Typnamen entfällt.

Einige der Operationen aus USEROPS sollen auch „Hauptprogramme“ einer Applikation sein, d.h. sie haben keine Eingabeparameter und geben *void* zurück und dienen somit als operationale „Einstiegspunkte“ in eine Anwendung.

Das Löschen von Operationen aus DEFOPS geschieht über

`drop operation` [*defType* `'.'`] *identifier* `';'`

Das Löschen von Methoden ist eine Modifikation der Definition des im qualifizierten Namen genannten Objekttyps. Gleichzeitig hat das Löschen aber auch Auswirkungen auf alle Objekttypen, die die zu löschende Methode erben. Nach `drop operation` für eine Methode müssen deshalb die Funktionen *OpAccess*(t) aktualisiert werden.

Man beachte, daß in [Pau94] kein nachträgliches Hinzufügen oder Löschen von Methoden für einen Objekttyp möglich ist.

4.6 Die Sprache BASESCRIPT

In diesem Abschnitt wird die Sprache BASESCRIPT vorgestellt, die die Sprache der bereits erwähnten Basisscripten sein soll. Für Basisscripten gilt der in Abschnitt 4.5.1 in (4.7) angegebene syntaktische Rahmen. Die Syntax der Signatur und des Deklarationsteils ist ebenfalls bereits in Abschnitt 4.5.1 dargelegt worden. Wir können deshalb sofort die Syntax des Anweisungsteils angeben.

4.6.1 Die Syntax des Anweisungsteils eines Basisscriptes

Der Anweisungsteil besteht aus einer Folge von Anweisungen, die nach folgender Syntax gebildet werden können:

anwTeil ::= { *anw* }

anw ::= (*operation* | *zuweisung* | *ifAnw* | *whileAnw* | *returnAnw*) `';'`

operation ::= [*pathExpr* `'.'`] *opName* `'('` *argListe* `')'`

zuweisung ::= *identifier* `:=` (*operation* | *query* | *simpleExpr*)

simpleExpr ::= *pathExpr* | *atomic* | *set* | *list* | *bag* | *tuple* | *array*
| *variant* | *labelTest* | `null`

query ::= `query` `'('` *queryExpr* `')'`

queryExpr ::= Anfrageausdruck (siehe Abschnitt 4.6.5)

ifAnw ::= `if` *pathExpr* `then` *anw* { *anw* } [`else` *anw* { *anw* }] `end if`

whileAnw ::= `while` *pathExpr* `do` *anw* {*anw*} `end while`

returnAnw ::= `return` [*identifier*]

identifier	::= Name aus Def(α) oder Name eines Typs aus DEFTYPES ∪ VARTYPES
atomic	::= *int* | *float* | *char* | *string* | *bool* | // Literal eines atomaren Datenobjekts[9]
opName	::= Name einer Operation aus DEFOPS
argListe	::= [*pathExpr* {',' *pathExpr* }]
pathExpr	::= *identifier* | *pathExpr* '.' *attrName* // Zugriff auf Tupelkomponente | *pathExpr* '[' *pos* ']' // Zugriff auf Listenkomponente | *pathExpr* '[' *pos* { ',' *pos*} ']' // Zugriff auf Feldkomponente | *pathExpr* '^' // Dereferenzieren eines Links
attrName	::= Attributname eines Tupels
pos	::= *pathExpr* | *int* // Pfadausdruck oder integer-Literal
set	::= '{' { *simpleExpr* [',' *simpleExpr*]} '}'
list	::= '<' { *simpleExpr* [',' *simpleExpr*]} '>'
bag	::= '{*' { *simpleExpr* [',' *simpleExpr*]} '*}'
tuple	::= '[' { *attrName* ':' *simpleExpr* [',' *attrName* ':' *simpleExpr*]} ']'
array	::= '<' { *simpleExpr* [',' *simpleExpr*]} '|' *int* ':' *int* [',' *int* ':' *int*] '>'
variant	::= '(+' *identifier* '.' *label* ':' *simpleExpr* '+)'
labelTest	::= *simpleExpr* `is` *label*
label	::= Bezeichner für ein Label eines varianten Typs

Die Syntax von BASESCRIPT ist so angelegt, daß pro Anweisung höchstens eine Operation ausgeführt wird. In den folgenden Abschnitten geben wir die Semantik der SpracheBASE-SCRIPT an.

4.6.2 Semantik von einfachen Ausdrücken

Es soll nun die Semantik der Ausdrücke aus $\mathcal{L}$(*simpleExpr*) geklärt werden. Zunächst betrachten wir die Ausdrücke aus $\mathcal{L}$(*pathExpr*), die sog. *Pfadausdrücke*. Sie ermöglichen ein „Navigieren“ entlang eines Pfades in einem Wertebaum. Auf diese Weise wird der Zugriff auf eine Tupel-, Listen oder Feldkomponente sowie auf ein bestimmtes Attribut eines getypten Objektes (ω, t) möglich.

Es sei

$$\mu : \mathcal{L}(pathExpr) \dashrightarrow \mathcal{V}$$

eine *Zugriffsfunktion*, die einen Pfadausdruck in einen Knoten aus V(Z) „übersetzt“. Die Zugriffsfunktion μ ist gemäß der im folgenden genannten Regeln definiert.

Regel 1 – Direkter Zugriff über einen Namen:

$e \in \mathcal{L}$(*identifier*) und $e \in$ Def(α)
$\Rightarrow \mu(e) = \alpha(e)$

9. Vgl. die gleichnamigen Nichtterminale in Anhang B.3.

e ∈ $\mathcal{L}$(*identifier*) und e ∈ {*name*(t) | t ∈ DEFTYPES}
⇒ μ (e) = *create*((t, $t_{deftype}$)) mit *name*(t) = e

Regel 2 – Zugriff auf eine Tupelkomponente:

e = t . a, mit t ∈ $\mathcal{L}$(*pathExpr*), a ∈ $\mathcal{L}$(*attrName*), μ (t) ist definiert,
type (*struct* (μ (t))) = c_{tuple}, a ∈ Def(*attr_pos* (*struct* (t)))
⇒ μ (e) = *get_child* (μ (t), n) mit n = *attr_pos*(*struct* (t))(a)

Regel 3 – Zugriff auf ein Attribut eines Objektes:

e = p.a, mit p ∈ $\mathcal{L}$(*pathExpr*), a ∈ $\mathcal{L}$(*attrName*), μ (p) ist definiert,
val(μ (p)) = (ω, t) mit ω ∈ Ω und t ∈ DEFTYPES,
a ∈ Def(*AttrAccess* (t)) , *AttrAccess* (t) (a) = (t', n)
⇒ μ (e) = *get_child* (*I* (t') (ω), n).

Regel 4 – Zugriff auf eine Listenkomponente:

e = L [i] mit L ∈ $\mathcal{L}$(*pathExpr*), i ∈ $\mathcal{L}$(*pos*), μ(L) ist definiert,
type (*struct* (μ (L))) = c_{list},
Fall 1: i ∈ $\mathcal{L}$(*int*), 0 < i ≤ *degree* (μ (L))
⇒ μ (e) = *get_child* (μ (L), i)
Fall 2: i ∈ $\mathcal{L}$(*pathExpr*), *val*(μ (i)) = (k, int), 0 < k ≤ *degree* (μ (L))
⇒ μ (e) = *get_child* (μ (L), k)

Regel 5 – Zugriff auf eine Feldkomponente:

e = A [i_1, …, i_n] mit A ∈ $\mathcal{L}$(*pathExpr*), μ (A) ist definiert,
type (*struct* (μ (A))) = c_{array}.
Für ν = 1, …, n bestimme k_ν wie folgt:
Fall 1: i_ν ∈ $\mathcal{L}$(*int*) → setze $k_\nu = i_\nu$
Fall 2: i_ν ∈ $\mathcal{L}$(*pathExpr*) → k_ν ergibt sich aus *val*(μ (i_ν)) = (k_ν, int)
⇒ μ (e) = *get_child* (μ (A), n) wobei n = $lexord_{domain\ (struct\ (\mu\ (A)))}(k_1, \ldots, k_n)$

Regel 6 – Dereferenzieren eines Links:

e = p^ mit p ∈ $\mathcal{L}$(*pathExpr*),
μ (p) ist definiert, *type* (*struct* (μ (p))) = c_{link},
⇒ μ (e) = *value* (μ (p))

Falls keine der Regeln zutrifft, dann ist μ(e) undefiniert. In diesem Fall tritt ein Laufzeitfehler auf. Dies soll grundsätzlich zu einem Zurücksetzen (Abort) der aktuellen Transaktion, zum Zurücksetzen des Laufzeitstacks und zur Ausgabe einer Fehlermeldung führen.

Nun definieren wir die *Wertesemantik* beliebiger einfacher Ausdrücke e ∈ $\mathcal{L}$(*simpleExpr*). Sie ist gegeben durch die Funktion

$$\sigma : \mathcal{L}(simpleExpr) \cdots> \mathcal{V}al^*(\text{TYPES}), \tag{4.12}$$

die wie folgt definiert ist:

(i) Falls $e \in \mathcal{L}(pathExpr) \Rightarrow \sigma(e) = val(\mu(e))$

(ii) Falls $e = \{e_1, \ldots, e_n\} \in \mathcal{L}(set)$ und $\sigma(e_i) = (v_i, s)$ für $1 \le i \le n$
$\Rightarrow \sigma(e) = (\{v_1, \ldots, v_n\}, \{s\})$
Für $e \in \mathcal{L}(list, bag, array)$ analog.

(iii) Falls $e = [a_1 : e_1, \ldots, a_n : e_n] \in \mathcal{L}(tuple)$ und $\sigma(e_i) = (v_i, s_i)$ für $1 \le i \le n$
$\Rightarrow \sigma(e) = ([a_1: v_1, \ldots, a_n: v_n], [a_1: s_1, \ldots, a_n: s_n])$

(iv) Falls $e \in \mathcal{L}(atomic)$, dann ist e ein Literal für ein eindeutig bestimmtes getyptes Datenobjekt (v, t) für einen Basis- oder Aufzählungstyp t, d.h. σ (e) = (v, t).

(v) Falls $e = (+ n.l : e' +) \in \mathcal{L}(variant)$ mit $n = name(t)$ für $t \in$ VARTYPES,
$(l, s) \in variants(t)$, $\sigma(e') = (v, s)$
$\Rightarrow \sigma(e) = ((l, v), t)$

(vi) Falls $e = e'$ `is` $l \in \mathcal{L}(labelTest)$ und $\sigma(e') = (v, t)$ mit $t \in$ VARTYPES und $l \in labels(t)$
$\Rightarrow \sigma(e) = (\, l =$ `null`$, \beta_{boolean})$, falls $v = null$
$\sigma(e) = (\, l = l', \beta_{boolean})$, falls $v = (l', v')$

In allen anderen Fällen ist σ undefiniert. Die Wertesemantik einfacher Ausdrücke haben wir bereits für den Zugriff auf Listen- und Feldkomponenten benötigt. Sie spielt auch im Zusammenhang mit Zuweisungen eine Rolle (siehe Abschnitt 4.6.4).

4.6.3 Aufruf und Abarbeitung einer Operation

Es soll nun die Semantik des Aufrufs und der Abarbeitung einer beliebigen Operation – sei dies eine Basisoperation $o \in$ BASEOPS oder eine benutzerdefinierte Operation $o \in$ DEFOPS – angegeben werden.

Es sei $o \in$ BASEOPS gegeben. Aufruf und Abarbeitung von o besteht aus den folgenden Schritten:

(i) Ist $<arg_1, \ldots, arg_n>$ die aktuelle Parameterliste, dann wird zunächst $r_i := \mu(arg_i)$ bestimmt $(1 \le i \le n)$.

(ii) Übergang in den Nachfolgezustand $Z^{neu} = semantics(o)(Z^{alt}, <r_1, \ldots, r_n>)$, falls dieser exisitiert (Andernfalls wurde eine Ausnahme ausgelöst, auf die entsprechend reagiert werden muß).

Es sei nun $o \in$ DEFOPS. Die Semantik des Aufrufs und der Abarbeitung von o ist wie folgt gegeben:

(i) Ist $<arg_1, \ldots, arg_n>$ die aktuelle Parameterliste, dann wird zunächst $r_i := \mu(arg_i)$ bestimmt $(1 \le i \le n)$.

(ii) Ist L die Liste der Bindungsfunktionen des Laufzeitzustandes Z^{alt}, dann wird beim Aufruf eines Basisscriptes an L eine neue Bindungsfunktion $\alpha: \mathcal{N} \cdots> \mathcal{V}$ angehängt, die zur aktuellen Bindung wird. Zunächst ist $Def(\alpha) =$ GLOBNAMES und $\alpha(n) := inst(name^{-1}(n))$ für $n \in$ GLOBNAMES.

(iii) Parameterübergabe: Es sei p_i der Name des i-ten formalen Parameters von *code*(o). Es werde $\alpha(p_i) = r_i$ gesetzt. Das bedeutet, daß eine Parameterübergabe „call-by-reference“ stattfindet.

(iv) Schrittweise Abarbeitung des Deklarationsteils von *code*(o):
Für jede Variablendeklaration v : s ∈ $\mathcal{L}$(*varDekl*) wird
α(v) := *root* (*create*((*initVal*(s), s))) gesetzt.

(v) Schrittweise Abarbeitung des Anweisungsteils von *code*(o). Die Abarbeitung der einzelnen Anweisungen wird im folgenden Abschnitt erläutert.

Wir gehen davon aus, daß die aktuellen Parameter μ (arg_i) eine Struktur haben, die zu der in der Signatur gegebenen Struktur „paßt“ (gemäß den Ausführungen in Abschnitt 4.2), so daß Typprüfungen zur Laufzeit in Schritt (i) nicht notwendig sind. Diese Annahme ist gerechtfertigt, wenn die Sprache, die nach BASESCRIPT übersetzt wird, streng typisiert ist, wovon wir ausgehen wollen. Ist bei der Parameterauswertung μ (arg_i) für ein i undefiniert (z.B. ein Zugriff jenseits des gültigen Indexbereiches einer Liste oder eines Feldes), dann muß dies zur Auslösung einer Ausnahme führen.

4.6.4 Die Semantik des Anweisungsteils eines Basisscriptes

Wir betrachten nun die Abarbeitung der einzelnen Anweisungen. Auf die Semantik der if-Anweisung und der while-Schleife, die unmittelbar verständlich sein dürfte, wird allerdings verzichtet.

Abarbeitung einer return-Anweisung:

(i) Falls anw = `return` e ∈ $\mathcal{L}$(*returnAnw*) mit e ∈ $\mathcal{L}$(*identifier*), dann setze R^{neu}:=μ(e).
Für anw = `return` ∈ $\mathcal{L}$(*returnAnw*) ist R^{neu} := R^{alt}.

(ii) Das letzte Element der Liste L der Bindungsfunktionen wird aus L entfernt.

(iii) Für alle t ∈ DEFTYPES und alle ω ∈ Def(I(t)) mit *pref*(I(t)(ω)) = 0 wird
I(t)(ω) := ⊥ gesetzt (d.h. ω wird aus Def(I(t)) entfernt).

Auswertung eines Ausdrucks e ∈ $\mathcal{L}$(*operation*):

Wir betrachten Ausdrücke e ∈ $\mathcal{L}$(*operation*). Sie können für sich bereits eine Anweisung bilden oder aber die rechte Seite einer Zuweisung sein. Auf jeden Fall führt ihre Auswertung zum Aufruf und der Abarbeitung einer Operation.

(i) Falls e = op(`(`arg_1`, …,` arg_n`)` ∈ $\mathcal{L}$(*operation*)
⇒
– Bestimmung der eindeutig bestimmten Operation o ∈ BASEOPS ∪ USEROPS mit *name*(o) = op
– Aufruf und Abarbeitung von o mit der aktuellen Parameterliste <arg_1, …, arg_n> gemäß Abschnitt 4.6.3

(ii) Falls e = p`.`m`(`arg_1`, …,` arg_n`)` ∈ $\mathcal{L}$(*operation*)
⇒ (Aufruf einer Methode)
– Auswertung von μ(p) = r. Es muß gelten: *struct* (r) ∈ DEFTYPES.
– Bestimmung von o = *OpAccess* (*struct*(μ(p))) (op)
– Aufruf und Abarbeitung von o mit der aktuellen Parameterliste
<p, arg_1, …, arg_n> gemäß Abschnitt 4.6.3

Abarbeitung einer Zuweisung:

(i) Falls anw = id `:=` e $\in \mathcal{L}$(*zuweisung*) mit e $\in \mathcal{L}$(*simpleExpr*) und
id $\in$ Def(α) – GLOBNAMES $\Rightarrow$ α^{neu} (id) := *create*(σ(e)).
Bemerkung: Ist e $\in \mathcal{L}$(*pathExpr*), dann kann wegen σ(e) = *val*(μ(e)) und
create(*val*(r)) = *copy*(r) für r $\in \mathcal{V}$ auch α^{neu} (id) := *copy*(μ(e)) geschrieben werden.

(ii) Falls anw = id `:=` e $\in \mathcal{L}$(*zuweisung*) mit e $\in \mathcal{L}$(*operation*) und
id $\in$ Def(α) – GLOBNAMES
$\Rightarrow$
– Auswertung von e $\in \mathcal{L}$(*operation*) wie oben beschrieben
– α^{neu} (id) := R^{neu}

(iii) Falls anw = id `:= query(q)` $\in \mathcal{L}$(*zuweisung*) mit q $\in \mathcal{L}$(*queryExpr*) und
id $\in$ Def(α) – GLOBNAMES. Ferner sei σ(q) die Wertesemantik[10] der Anfrage q
$\Rightarrow$
– R^{neu} := *create*(σ(q))
– α^{neu} (id) := R^{neu}

Man beachte, daß bei Zuweisungen grundsätzlich id $\in$ Def(α) – GLOBNAMES gelten soll, d.h. Zuweisungen an globale Namen sind nicht möglich. Diese Entscheidung steht im Einklang mit der Forderung, daß persistente Datenobjekte nur über eine der generischen Update-Operationen manipuliert werden dürfen. Eine Zuweisung an einen lokalen Bezeichner führt immer zu einer Modifikation der Bindungsfunktion α und nicht zu einem Auswechseln von (Teil-)Bäumen. Diese Strategie muß gewählt werden, um unerwünschte Nebeneffekte zu vermeiden. Es sei beispielsweise T der Name einer Tabelle und der lokale Bezeichner x an einen inneren Knoten von α(T) gebunden (dies ist dann möglich, wenn x ein formaler Parameter einer benutzerdefinierten Operation ist, dem als aktueller Parameter ein von T ausgehender Pfadausdruck übergeben wird). Diese Situation ist in Abb. 4.2 veranschaulicht. Würde die

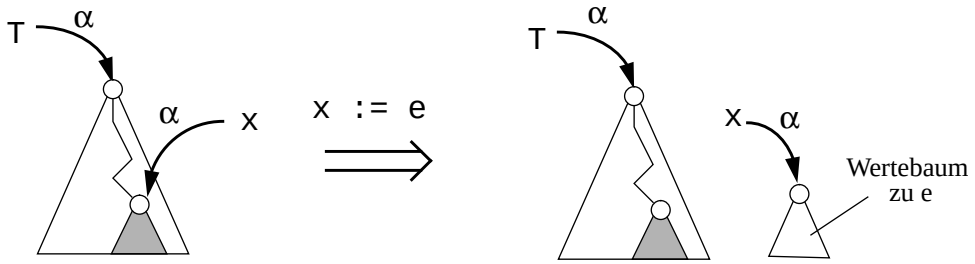


Abbildung 4.2: Veranschaulichung der Zuweisung an einen transienten Bezeichner

Zuweisung x := e ein Auswechseln des grau unterlegten Teilbaumes gegen einen anderen, den Ausdruck e repräsentierenden Baumes bewirken, dann würde eine Modifikation persistenter Datenobjekte stattfinden, ohne daß eine generische Update-Operation daran beteiligt wäre. Dies wollen wir jedoch grundsätzlich ausschließen. Angenommen x ist an einen atomaren Knoten gebunden, dann hat die Ausführung der Anweisung

10. Auf die Wertesemantik von Anfragen gehen wir im folgenden Abschnitt näher ein. Zunächst reicht es aus, darunter das Ergebnis der Anfrage zu verstehen.

```
update(x, e);
```

tatsächlich einen „Nebeneffekt“ auf die Tabelle T, wie man ihn von der Parameterübergabeart „call-by-reference“ aus anderen Programmiersprachen kennt.

4.6.5 Die Semantik von Anfragen in BASESCRIPT

Die Vorteile einer deklarativen gegenüber einer prozeduralen Formulierung einer Anfrage sind hinlänglich bekannt (vgl. auch die Diskussion in Abschnitt 2.2.1). Auch für ESCHER$^+$ soll es eine *deklarative* Anfragesprache geben. Beim Blick auf die Syntax von BASESCRIPT fällt auf, daß dort noch keine präzisen Aussagen über die Syntax der Anfragesprache gemacht werden. Wir erinnern daran, daß BASESCRIPT als Zielsprache der Übersetzung einer „höheren“ Sprache konzipiert ist. Im nächsten Abschnitt stellen wir die Sprache SCRIPT$^+$ vor, die in dieser Arbeit die Rolle dieser „höheren“ Sprache einnehmen soll. Eine Anfrage q$\in \mathcal{L}$(*queryExpr*) soll ein Ausdruck der Sprache SCRIPT$^+$ sein, d.h. $\mathcal{L}$(*queryExpr*) $\subseteq \mathcal{L}$(*expr*), wobei *expr* das Nichtterminal aus Anhang B.3 ist.

Es stellt sich nun die Frage, welche Teilmenge von $\mathcal{L}$(*expr*) genau mit $\mathcal{L}$(*queryExpr*) bezeichnet werden soll. Wir wollen hier noch nicht im Detail auf die Syntax von SCRIPT$^+$ eingehen. Jedoch können bereits die entscheidenden Kriterien für eine Mitgliedschaft in $\mathcal{L}$(*queryExpr*) genannt werden, da sie ohne weiteres verständlich sind. Zu $\mathcal{L}$(*queryExpr*) gehören alle Ausdrücke e $\in \mathcal{L}$(*expr*), für die einer der nachfolgenden Fälle zutrifft:

- e ist eine Vereinigungs-, Durchschnitts- oder Differenzbildung oder eine Listenkonkatenation, d.h. e = E_1 op E_2 mit op $\in$ {`union`, `minus`, `sect`, `append`}.
  Beispiel: `Personen minus {person1, person2}`
  (Dabei sei `Personen` eine Tabelle mit der Struktur `{Person}`, wobei `Person` der Objekttyp aus Beispiel 3.7 ist, ferner seien `person1` und `person2` lokale Variablen, die an Instanzen des Typs `Person` gebunden sind)
- e ist ein *Kollektionsformer*, d.h. e $\in \mathcal{L}$(*former*).
  Auf diese Ausdrücke gehen wir im Anschluß an diese Aufzählung näher ein.
- e ist ein quantifizierter boolescher Ausdruck, d.h. e $\in \mathcal{L}$(*quantifExpr*).
  Beispiel: `all p in Personen, exists h in p.Hobbies: h="Lesen"`
- e = E_1 `in` E_2 bzw. e = E_1 `is` [`proper`] `subset of` E_2,
  Dies Ausdrücke stellen eigentlich nur syntaktischen Zucker dar, da sie ohne Schwierigkeiten in quantifizierte boolesche Ausdrücke transformierbar sind.
  Beispiel: `{"Lesen", "Radfahren"} is subset of p.Hobbies`
  ist äquivalent zu
  `exists h in p.Hobbies: h="Lesen" or h="Radfahren"`

Ferner muß gelten, daß e keine Aufrufe von Scripten enthält, da für e sonst nicht zugesichert werden kann, daß es sich um eine *sichere*[11] Anfragen handelt. Auf diese Weise schließen wir auch aus, daß Anfragen bei ihrer Auswertung „Nebeneffekte“ auslösen, indem sie Scripten aufrufen, die ihrerseits Updates auf der Datenbankinstanz ausführen.

11. Eine Anfrage gilt als sicher, wenn die Terminierung ihrer Abarbeitung garantiert werden kann.

In [Pau94] wurden mit den Kollektionsformern, die auch als *Komprehensionsausdrücke* bezeichnet werden, spezielle Ausdrücke in die Sprache SCRIPT eingeführt, mit denen die Formulierung von `SELECT-FROM-WHERE`-Anfragen möglich ist. Wir geben dazu ein Beispiel an.

Beispiel 4.4 Es sei `Kunden` eine Tabelle mit der Struktur `{`s`}`, wobei s = `[Name:string, Adresse:[Ort:string, Str:string, Nr:int], ...]`. In SCRIPT ist

```
{k|k in Kunden: k.Adresse.Ort = "Kassel"}
```

ein *Mengenformer*, der der Anfrage nach der Menge aller Kunden aus Kassel entspricht. Dieselbe Anfrage läßt sich in einer `SFW`-Syntax in der Form

```
SELECT DISTINCT k
FROM k in Kunden
WHERE k.Adresse.Ort = "Kassel"
```

ausdrücken. Die zweite Fassung entspricht der Syntax von OQL, der Anfragesprache des ODMG'93-Standards [Cat+94]. ❒

Letztendlich ist die Wahl der konkreten Syntax vor allem eine „Geschmacksfrage". Viele Sprachvorschläge versuchen die Syntax ihrer Anfragesprache weiterhin in das `SFW`-„Korsett" von SQL einzupassen und erhoffen sich aufgrund der Verbreitung von SQL eine erhöhte Akzeptanz für die neue Anfragesprache. Diesen Weg verfolgt z.B. OQL, die Anfragesprache des ODMG'93-Standards [Cat+94]. Wir entscheiden uns für die von SCRIPT verwendete Syntax mit *Kollektionsformern*. Sie werden auch in anderen persistenten Programmiersprachen verwendet [Tri92, CT94].

Die Ausdrücke q ∈ $\mathcal{L}$(*queryExpr*) werden nicht in eine bestimmte Anweisungsfolge in BASESCRIPT übersetzt, sondern zunächst an eine spezielle Komponente des DBMS, den *Query Optimizer*, übergeben. Dieser ist für die Erstellung eines (möglichst) optimalen prozeduralen Ausführungsplans, den wir mit *plan*(q) bezeichnen, veranwortlich. Der Query Optimizer transformiert auf der Basis von algebraischen Umformungen und unter Ausnutzung von Gegebenheiten der internen Ebene (z.B. Sortierungen, Indexe) den Ausdruck q in einen Ausführungsplan *plan*(q). Die Details der Erstellung eines (optimalen) Ausführungsplans sollen nicht Gegenstand dieser Arbeit sein. Wir gehen im folgenden von der Existenz des Ausführungsplanes *plan*(q) aus. Die Auswertung des BASESCRIPT-Ausdrucks `query(q)` zur Laufzeit entspricht gerade der Ausführung von *plan*(q).

Zur Klärung der Frage, was unter der Semantik einer Anfrage zu verstehen ist, benötigen wir eine Abbildung

$$\sigma : \mathcal{L}(expr) \cdots> \mathcal{V}al^* \text{ (TYPES)}, \tag{4.13}$$

die analog zur Funktion σ aus (4.12) definiert ist und die jedem Ausdruck e ∈ $\mathcal{L}$(*expr*), also insbesondere auch jeder Anfrage in $\mathcal{L}$(*queryExpr*), eine *Wertesemantik* zuordnet. In vielen Fällen stimmt die Funktion σ mit der gleichnamigen Funktion aus (4.12) überein, da die BASESCRIPT-Ausdrücke aus $\mathcal{L}$(*simpleExpr*) in $\mathcal{L}$(*expr*) enthalten sind. Wir werden auf die Bestimmung der Wertesemantik σ(e) für e∈ $\mathcal{L}$(*queryExpr*) in Abschnitt 4.7.2 näher eingehen.

Ist q ∈ $\mathcal{L}$(*queryExpr*) eine syntaktisch korrekte Anfrage und ist σ(q) ihre Wertesemantik, dann ist die Auswertung des Ausdruckes `query(q)` bzw. die Abarbeitung von *plan*(q) äquivalent zu

$R^{neu} := create(\sigma(q))$

(vgl. Abschnitt 4.6.4, Abarbeitung einer Zuweisung, Fall (iii)).

Beispiel 4.5 (Fortsetzung von Beispiel 4.4)
Die Ausführung von

```
query({k|k in Kunden: k.Adresse.Ort = "Kassel"})
```

setzt die Abarbeitung des Ausführungsplans *plan*(`{k|k in … "Kassel"}`) in Gang. Dieser liefert einen neuen Knoten r mit *struct* (r) = {s}. Die Struktur des Anfrageergebnisses ist bereits durch die textuelle Anfrage bestimmt. Die Söhne von r sind *Kopien* aller Söhne r' von *inst* (`Kunden`), die das angegebene Prädikat erfüllen.
Es gilt dann *val*(r) = σ(`{k|k in Kunden: k.Adresse.Ort = "Kassel"}`). ❐

Es soll an dieser Stelle besonders darauf hingewiesen werden, daß das Anfrageergebnis nur einen „Snapshot“ des aktuellen Datenbankzustandes angibt. Ein nachträgliches Einfügen eines weiteren Tupels k mit k`.Adresse.Ort` = "Kassel" propagiert nicht in das Anfrageergebnis. Genauso führt die Modifikation einer Tupelkomponente (z.B. `Adresse.Str`) im Anfrageergebnis nicht zu einer Modifikation der Komponente im entsprechenden Tupel der Tabelle `Kunden`!

Wäre hingegen `Kunden` genauso wie die Tabelle `Personen` als Menge von Instanzen des Objekttyps `Person` aus Beispiel 3.7 definiert, dann ist es möglich, das Anfrageergebnis als Grundlage für die Modifikation der Attribute des „Original“-Objektes zu verwenden. Jeder Elementknoten r= (ω, t_{Person}) im Anfrageergebnis enthält den Objektidentifikator ω, der für den Zugriff auf die zugehörige Zustandsfunktion $I(t_{Person})(\omega)$ benötigt wird. Auf diese Weise kann ein Update im Anfrageergebnis an die „richtige“ Stelle im persistenten Datenbestand propagieren. Das bekannte „view update“-Problem wird also durch die Verwendung von Objekttypen entschärft.

In bestimmten Situationen ist es sinnvoll, über eine Anfrage Verweise auf die „Original“-Knoten zu finden, die ein gegebenes Selektionsprädikat erfüllen, z.B. um im nächsten Schritt diese Knoten zu löschen. Dies läßt sich zusammen mit der `getLink`-Operation verwirklichen, die nicht eine Kopie des Wertes für das Anfrageergebnis erzeugt, sondern einen Verweis auf die „Original“-Knoten. Intern – *nicht* durch den Benutzer[12]– wird dazu eine Anfrage

```
{getLink(k)|k in Kunden: k.Adresse.Ort="Kassel"}
```

erzeugt, deren Ergebnis eine Menge von Verweisen auf die Knoten ist, die Söhne von *inst*(`Kunden`) sind und das Selektionsprädikat erfüllen. Mittels Iteration über die Elemente im Anfrageergebnis könnten nun im nächsten Schritt alle in Kassel wohnenden Kunden aus der Tabelle `Kunden` gelöscht werden. ❐

12. Vor dem Benutzer sollen Link-Werte verborgen bleiben!

4.7 Eine Variante der Sprache SCRIPT

4.7.1 Grundlegende Vorbemerkungen

Es wurde bereits darauf hingewiesen, daß sich BASESCRIPT als „primitive“ Sprache versteht, in die benutzerdefinierte Operationen, die in einer „höheren“ Programmiersprache kodiert sind, übersetzt werden. Eine geeignete „höhere“ Sprache stellt die mehrfach erwähnte Sprache SCRIPT aus [Pau94] dar.

In diesem Abschnitt stellen wir mit SCRIPT$^+$ eine Variante der Sprache SCRIPT vor. Neben geringfügigen Modifikationen gegenüber SCRIPT enthält die Syntax von SCRIPT$^+$ auch einige Erweiterungen, auf die wir in Abschnitt 4.7.4 eingehen. Die Syntax von SCRIPT$^+$ ist in Anhang B.3 zu finden. Nach [Pau94] besteht SCRIPT aus einem DDL- und einem DML-Teil. Die in Anhang B.3 angegebene Syntax der DDL für ESCHER$^+$ ist gegenüber dem Vorschlag aus [Pau94] vollständig überarbeitet worden. Im weiteren beziehen wir uns ausschließlich auf die DML.

Die Sprachen SCRIPT$^+$ und BASESCRIPT haben einige Gemeinsamkeiten:

- Beide Sprachen verwenden denselben syntaktischen Rahmen (4.7) für benutzerdefinierte Operationen, d.h. die rechte Seite in (4.7) gibt auch den syntaktischen Aufbau jedes Scriptes an
- In [Pau94] werden keine expliziten Ausführungen zur Art der Parameterübergabe beim Aufruf von Scripten gemacht. Die für BASESCRIPT beschriebene Art der Parameterübergabe („call-by-reference“, vgl. Abschnitt 4.6.3) soll auch für SCRIPT$^+$ gelten.
- Sichtbar sind in einem Script die Namen der Tabellen (globale Namen), daneben nur noch die Namen der formalen Parameter und der lokal deklarierten Variablen

Ein wichtiger Unterschied zwischen SCRIPT aus [Pau94] und SCRIPT$^+$ ergibt sich aus der Verschiedenheit der zugrundeliegenden Datenmodelle: In [Pau94] gibt es einen weiteren Konstruktor FUN, mit dem funktionale Datentypen FUN($s_1, \ldots, s_n$ -> s) erzeugt werden können. Es ist

$$dom(\text{FUN}(s_1, \ldots, s_n \text{ -> } s)) = \{\ S \in \mathcal{L}(script) \mid S \text{ hat die Signatur } s_1, \ldots, s_n \text{ -> } s\ \}.$$

Der Konstruktor ist orthogonal zu allen anderen Konstruktoren anwendbar, so daß beispielsweise Attribute eines Tupels oder eines Objekttyps einen funktionalen Typ haben können. Der Wert eines funktionalen Attributs (gegeben durch ein Script) kann von Tupel zu Tupel bzw. von Objekt zu Objekt verschieden sein. Jedes Script ist unbenamt (anonym), kann sich aber selbst mit dem Bezeichner `me` aufrufen. Ist ein Script ein Attributwert eines Tupels oder Objektes, so kann im Script selbst über `owner` auf das Tupel bzw. das Objekt zugegriffen werden. Für ESCHER$^+$ verzichten wir auf diese Verschränkung struktureller und operationaler Eigenschaften. In ESCHER$^+$ ist jedes Script über die Funktion *code* aus Def. 4.5 einer Operation aus DEFOPS zugeordnet, die immer benamt ist, so daß `me` in der Syntax nicht berücksichtigt

werden muß[13]. Für Methoden wählen wir für den Zugriff auf das „Receiver“-Objekt statt `owner` den Bezeichner `self`.

4.7.2 Die Semantik von Anfragen in SCRIPT$^+$

In Abschnitt 4.6.5 wurden bereits die drei Klassen von Ausdrücken benannt, die wir als Anfragen bezeichnen und zusammen die Menge $\mathcal{L}$(*queryExpr*) bilden. Nun soll ausführlicher auf die Wertesemantik $\sigma(e)$ für $e \in \mathcal{L}$(*queryExpr*) eingegangen werden.

a) Vereinigung, Durchschnitt, Differenz, Konkatenation:

Für Mengen und Multimengen sind die Vereinigung, der Durchschnitt und die Differenz definiert. Es seien $E_1, E_2 \in \mathcal{L}$(*expr*) mit $\sigma(E_1) = (S_1, c_{set}(s))$, $\sigma(E_2) = (S_2, c_{set}(s))$.

- Für $q = E_1$ `union` $E_2 \in \mathcal{L}$(*queryExpr*) ist σ (E_1 `union` E_2) $= (S_1 \cup S_2, c_{set}(s))$.
- Für $q = E_1$ `sect` $E_2 \in \mathcal{L}$(*queryExpr*) ist σ (E_1 `sect` E_2) $= (S_1 \cap S_2, c_{set}(s))$.
- Für $q = E_1$ `minus` $E_2 \in \mathcal{L}$(*queryExpr*) ist σ (E_1 `minus` E_2) $= (S_1 \setminus S_2, c_{set}(s))$.

Analoges gilt für E_i mit $\sigma(E_i) = (B_i, c_{bag}(s))$, wobei die Vereinigung, der Durchschnitt und die Differenz wie in Abschnitt 3.2.1.2 definiert seien.

Für Listen tritt an die Stelle der Vereinigung die Konkatenation zweier Listen:
Es seien $E_1, E_2 \in \mathcal{L}$(*expr*) mit $\sigma(E_1) = (<a_1, \ldots, a_n>, c_{list}(s))$, $\sigma(E_2) = (<b_1, \ldots, b_m>, c_{list}(s))$.

- Für $q = E_1$ `append` $E_2 \in \mathcal{L}$(*queryExpr*) ist
  σ (E_1 `append` E_2) $= (<a_1, \ldots, a_n, b_1, \ldots, b_m>, c_{list}(s))$.

b) Komprehensionsausdrücke (Kollektionsformer)

Anfragen, die vergleichbar mit `SFW`-Ausdrücken anderer Anfragesprachen sind, werden in SCRIPT$^+$ durch Mengen-, Listen- oder Multimengenformer ausgedrückt. Sie sind Ausdrücke aus $\mathcal{L}$(*former*) (vgl. Anhang B.3).

Wir betrachten exemplarisch den Aufbau eines Mengenformers, d.h. eines Ausdrucks aus $\mathcal{L}$(*setFormer*). Ein Mengenformer $q \in \mathcal{L}$(*setFormer*) hat die Gestalt

$$q = \{E(i_1, \ldots, i_n) \mid i_1 \texttt{ in } C_1, i_2 \texttt{ in } C_2(i_1), \ldots, i_n \texttt{ in } C_n(i_1,\ldots,i_{n-1}): B(i_1,\ldots,i_n)\}, \quad (4.14)$$

Dabei ist i_1 `in` C_1, i_2 `in` $C_2(i_1)$, …, i_n `in` $C_n(i_1,\ldots,i_{n-1})$: $B(i_1,\ldots,i_n)$ eine sog. *range expression*. Die $C_k(i_1,\ldots,i_{k-1})$, $1 \le k \le n$, sind kollektionswertige Ausdrücke, also beispielsweise ein Pfadausdruck $p \in \mathcal{L}$(*pathExpr*) mit *isCollection*(*type*(*struct*(μ(p)))) oder selbst wieder ein Kollektionsformer. Die i_k, $1 \le k \le n$, sind Laufvariablen (oder auch Iteratoren), die über die Elemente der Kollektionen C_k iterieren. Dabei darf C_k von den Laufvariablen i_ν mit $\nu < k$ abhängen. $B(i_1,\ldots,i_n)$ ist ein boolescher Ausdruck, der von allen Laufvariablen abhängig sein darf. $E(i_1, \ldots, i_n)$ ist ein beliebiger Ausdruck, der ebenfalls von den Laufvariablen abhängen darf und *target expression* genannt wird. Die Struktur der Elemente der Resultatmenge ergibt sich aus der Struktur von $E(i_1, \ldots, i_n)$. Der Aufbau von Listen- und Multimengen-Formern ist

13. Bei einer Ergänzung des Meta-Schemas aus Abschnitt 3.10 um den operationalen Teil gibt es einen Objekttyp t_{defop}, und der Funktion *code* aus Def. 4.5 entspricht dann ein gleichnamiges Attribut in *struct*(t_{defop}).

entsprechend. Man erkennt unschwer die nicht unbeabsichtigte Verwandtschaft eines Mengenformers zu einem Ausdruck des Relationenkalküls.

Die Wertesemantik $\sigma(q)$ eines Kollektionsformers q ist gegeben durch das Ergebnis der Abarbeitung folgender n geschachtelter Schleifen:

```
q := {}; (bzw. {**} bzw. <> je nach Art des Kollektionsformers)
for i1 in C1 do
  for i2 in C2(i1) do
  …
    for in Cn(i1,…,in-1) do
      if B(i1,… ,in) then
        „Einfügen“ von E(i1,… ,in) in q;
      end if;
    end for;
  …
  end for;
end for; // q ist das Ergebnis der Anfrage
```

Wir wählen an dieser Stelle eine prozedurale Beschreibung der Semantik, da bei Listenformern das Ergebnis von der Reihenfolge abhängt, in der die Kollektionen C_i durchlaufen werden.

Die Semantik der `for`-Schleife sei wie folgt gegeben: Die Reihenfolge der Iteration über die Elemente von $C_k(i_1,\ldots,i_{k-1})$ ist durch die Beschriftung *ord* bestimmt. Ist s_k die Struktur der Elemente von $C_k(i_1,\ldots,i_{k-1})$ und gilt $\neg$*isCollection*(*type*(s_k)), dann wird der Rumpf der `for`-Schleife für den aktuellen Wert der Iterationsvariable i_k nicht ausgeführt, falls *null* der aktuelle Wert von i_k ist. Mit anderen Worten: Nullwerte werden übersprungen. Allerdings ist das Überspringen von Nullwerten nicht immer erwünscht, worauf wir weiter unten zurückkommen werden.

Hinsichtlich des „Einfügens“ gilt: Bei Mengenformern werden Duplikate nicht eingefügt, bei Listenformern entspricht das „Einfügen“ dem Anhängen des neuen Elementes an das Listenende.

c) Quantorisierte boolesche Ausdrücke:

Ein Quantorisierter boolescher Ausdruck ist syntaktisch mit einer Range Expression vergleichbar, wobei jeder Variablen i_k ein Existenz- oder Allquantor vorangestellt ist, d.h. e$\in \mathcal{L}$(*quantifExpr*) hat den Aufbau

$$e = Q_1\ i_1\ \texttt{in}\ C_1,\ Q_2\ i_2\ \texttt{in}\ C_2(i_1),\ \ldots,\ Q_n\ i_n\ \texttt{in}\ C_n(i_1,\ldots,i_{n-1})\texttt{:}\ B(i_1,\ldots,i_n) \qquad (4.15)$$

mit $Q_k \in$ {`exists`, `all`} für $1 \le k \le n$. Die Negation eines Quantors lassen wir nicht zu, da durch entsprechende Umformungen immer die Gestalt (4.15) erreicht werden kann.

Die Wertesemantik $\sigma(e)$ eines Ausdrucks e der Gestalt (4.15) sollte unmittelbar verständlich sein. Wir geben nun noch eine prozedurale Semantik an, nach der die Wertesemantik $\sigma(e)$ prinzipiell berechnet werden kann. Dazu wird eine `for`-Schleife mit Abbruchbedingung verwendet: Eine Schleife `for i in C and B` führt eine Iteration über alle Elemente der Kollektion `C` durch, bricht die Iteration jedoch ab, wenn die Bedingung `B` nicht erfüllt ist.

Es sei

$$e_k := Q_k\ i_k\ \texttt{in}\ C_k(i_1,\ldots,i_{k-1}),\ \ldots,\ Q_n\ i_n\ \texttt{in}\ C_n(i_1,\ldots,i_{n-1})\texttt{:}\ B(i_1,\ldots,i_n) \text{ für } 1 \le k \le n.$$

Es ist also $e = e_1$. Ferner sei $e_{n+1} := B(i_1, \ldots, i_n)$. Die Anweisungsfolge zur Berechnung von $\sigma(e) = \sigma(e_1)$ ist gegeben durch *eval*(e_1), wobei *eval*(e_k) wie folgt definiert ist:

eval(e_k) :=

```
ok_k := false;
for i_k in C_k(i_1, ..., i_k-1) and not ok_k do
  eval(e_k+1);
  ok_k := ok_k+1;
end for;                                  , falls k ≤ n und Q_k = exists

ok_k := true;
for i_k in C_k(i_1, ..., i_k-1) and ok_k do
  eval(e_k+1);
  ok_k := ok_k+1;
end for;                                  , falls k ≤ n und Q_k = all

ok_n+1 := B(i_1,...,i_n)                  , falls k = n+1
```

Nach Abarbeitung von *eval*(e_1) gibt ok_1 die Wertesemantik $\sigma(e) = \sigma(e_1)$ an.

Genauso wie bei der prozeduralen Semantik zur Auswertung von Kollektionsformern ist auch hier die Auswertung in Form von n ineinander geschachtelten `for`-Schleifen i.d.R. sehr ineffizient. Aufgabe der Anfrageoptimierung ist es, einen günstigeren Ausführungsplan zu finden.

Hinsichtlich der Behandlung von Nullwerten in Anfragen schlagen wir eine Strategie vor, nach der *null* wie jedes andere Datenobjekt behandelt: Ist t ein Typ mit *null* ∈ *dom*(t), dann gelte *null* ≠ v für alle v ∈ *dom*(t) − {*null*}. Ferner ergebe *null* = *null* den Wert *true* und *null* ≠ *null* konsequenterweise den Wert *false*. Ist auf *dom*(t) eine lineare Ordnung < definiert, dann gelte *null* < v für alle v ∈ *dom*(t) − {*null*}. Damit legen wir uns zwar darauf fest, daß *null* das kleinste Element innerhalb der linearen Ordnung ist, jedoch können wir dadurch einige Widersprücke vermeiden. Würden wir z.B. auf *dom*(t) die Prädikate θ ∈ {<, >, ≤ , ≥} so definieren, daß a θ b den Wert *false* ergibt, sobald a = *null* oder b = *null* ist, dann wäre u.a. die Gleichheit a ≥ b = ¬ (a < b) nicht länger gültig. Bei dreiwertiger Logik, wie sie im SQL/92-Standard Verwendung findet, werden Tautologien nicht richtig behandelt. Dazu betrachten wir folgende SQL-Anfrage:

```
SELECT * FROM Teile as t
WHERE t.Gewicht >= 50 or t.Gewicht < 50
```

Tupel `t` mit `t.Gewicht` = *null* sind im Anfrageergebnis nicht enthalten, da für diese die Bedingung der `WHERE`-Klausel zu *unknown* ausgewertet wird und sich nur solche Tupel für die Ergebnisrelation qualifizieren, für die die Selektionsbedingung zu *true* ausgewertet wird. Im Zusammenhang mit Nullwerten und der dreiwertigen Logik ließen sich noch eine Vielzahl von Widersprüchen und Paradoxien zu nennen, die z.B. in [DD93, AD93] nachgelesen werden können und die nicht zum Gegenstand dieser Arbeit gehören sollen. Hinsichtlich der Behandlung von Nullwerten verfolgen wir daher die oben beschriebene „Minimallösung“ und schließen uns der Empfehlung „Avoid nulls“ aus [DD93, S. 243] an!

Weiter oben wurde erläutert, daß Nullwerte bei der Anfrageauswertung ignoriert werden. Dadurch wird z.B. verhindert, daß es zu nicht zulässigen Zugriffen auf Attribute eines Objekttyps kommt.

Beispiel 4.6 Es sei `Kurse` eine Menge von Kursen, die (mindestens) durch die Attribute `Titel` und `Anmeldungen` beschrieben werden. Dabei sei `Anmeldungen` eine Liste von Personen, wobei die Reihenfolge in der Liste die zeitliche Reihenfolge der Anmeldungen wiedergebe. Die Anfrage

```
{ a | k in Kurse, a in k.Anmeldungen:
  k.Titel = "C++" and pos(a) > 15 and pos(a) <= 18 }
```

liefert die Personen auf den „Wartepositionen“ 16-18 des Kurses mit dem Titel "`C++`", d.h. `pos(a)` gibt die Position von `a` in der Liste an[14]. Der Zugriff auf `k.Anmeldungen` ist aber nur dann zulässig, wenn `k` ≠ *null* ist. Daher werden in der „äußeren Schleife“ `k in Kurse` alle Nullobjekte übersprungen. Dasselbe gilt für die innere Schleife: Ist eine der Personen auf den Wartepositionen 16-18 ein Nullobjekt, dann taucht dieses im Anfrageergebnis nicht auf, denn es wird bereits in der „inneren Schleife“ `a in k.Anmeldungen` übersprungen. ❒

Allerdings ist das Überspringen von Nullwerten nicht immer erwünscht. Möchte man z.B. herausfinden, an welchen Positionen der Anmeldungen des „C++“-Kurses anstelle einer „real existierenden“ Person der Wert *null* eingetragen ist, dann ist die Anfrage

```
{ pos(a) | k in Kurse, a in k.Anmeldungen:
  k.Titel = "C++" and a = null}
```

dazu ungeeignet, da die Nullwerte gerade übersprungen werden! Wir schlagen deshalb vor, daß für diese Situation die *Range Expressions* für die relevanten Iterationsvariablen um `with null`-Klauseln ergänzt werden können, so daß diese auch die Nullwerte in die Iteration einbeziehen. Die korrekte Anfrage lautet dann:

```
{ pos(a) | k in Kurse, a in k.Anmeldungen with null:
  k.Titel = "C++" and a = null}
```

4.7.3 Beispiele zur Übersetzung von SCRIPT+ nach BASESCRIPT

In diesem Abschnitt werden einige Konstrukte von SCRIPT+ anhand von Beispielen vorgestellt und ihre Übersetzung nach BASESCRIPT angegeben. Auf diese Weise ließe sich der Sprache SCRIPT+ insgesamt eine formale Semantik zuordnen. Wir beschränken uns jedoch auf einige Beispiele typischer SCRIPT+-Anweisungen und -Ausdrücke, da die lückenlose Präsentation der Syntax und Semantik einer persistenten Programmiersprache nicht Gegenstand dieser Arbeit sein soll.

Beispiel 4.7

SCRIPT+ erlaubt „benutzerfreundlichere“ Ausdrücke wie z.B.

```
m := 5+(f(n)*i);
```

Die Übersetzung nach BASESCRIPT ergibt

14. Genauer soll für `pos` die Semantik der gleichnamigen Basisoperation aus Abschnitt 4.3.3 zutreffen.

```
t1 := 5; t2 := f(n); t3 := mult(t2,i); m := add(t1,t3);
```

mit transienten Variablen `t1`, `t2` und `t3`. Hier wird besonders deutlich, daß BASESCRIPT wirklich nur als Zielsprache der Übersetzung eines Scriptes gedacht ist und nicht als Sprache für den Anwendungsentwickler. ❒

Beispiel 4.8 Es seien `S` bzw. `L` Namen, die an eine Menge bzw. Liste gebunden sind. Ferner sei `e` eine Variable, deren Struktur mit der Elementstruktur von `S` bzw. `L` übereinstimmt. In der nachfolgenden Tabelle werden einfache Update-Operationen nach BASESCRIPT übersetzt.

SCRIPT$^+$-Anweisung	Übersetzung nach BASESCRIPT
`S add e;`	`insert(S, e);`
`L add e at 5;`	`insertInList(L, e, 5);`
`L rem at 5;`	`remove(L, L[5]);`
`L move 4 to 5;`	`move(L, 4, 5);`

Tabelle 4.1: Übersetzung von Update-Operationen nach BASESCRIPT

❒

Das folgende Beispiel zeigt, daß die Übersetzung einer SCRIPT$^+$-Anweisung in die Basissprache nicht immer so einfach ist wie im letzten Beispiel.

Beispiel 4.9 Es sei `S` als eine Menge von integer-Werten definiert. Es soll die SCRIPT$^+$-Anweisung

```
S rem 42;
```

ausgeführt werden. Vor dem eigentlichen Löschen aus der Kollektion muß der zu löschende Knoten durch Auswertung einer Anfrage erst einmal gefunden werden! Dabei werden Link-Werte benötigt. Die SCRIPT$^+$-Anweisung wird übersetzt in

```
Q := query({getLink(i)| i in S: i = 42});
B := empty?(Q);
B := not(B);
if B then
  it := firstIn(Q);
  remove(S, it^^);
end if;
```

Es wird zunächst die (höchstens einelementige) Menge `Q` von Links ermittelt, die auf Knoten in der Menge der Söhne von `S` zeigen, die den Wert 42 repräsentieren. Auf das erste Element der Menge `Q` wird über einen Iterator `it` zugegriffen. Durch zweimaliges Dereferenzieren erhält man den zu löschenden Knoten. Dies ist also ein Beispiel für die interne Verwendung von Link-Werten, ohne daß der Benutzer davon Kenntnis hat. In Abb. 4.3 ist das soeben geschilderte Vorgehen graphisch veranschaulicht.

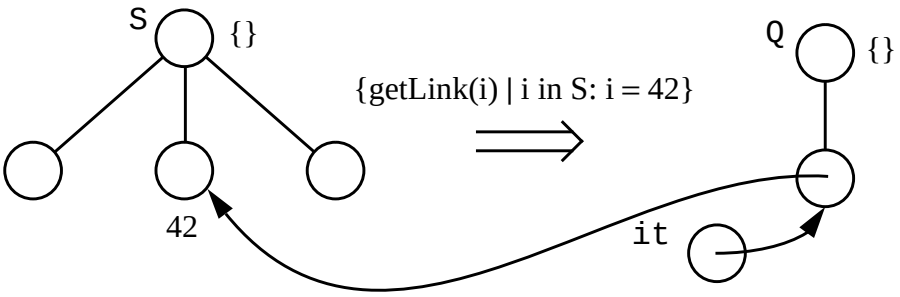


Abbildung 4.3: Abarbeitung der SCRIPT⁺-Anweisung `S rem 42;`

In der Anweisungsfolge der Basissprache werden Ergebnisse von Basisoperationen über Zuweisungungen an transiente Namen (hier: `Q`, `B` und `it`) gebunden. ❒

Beispiel 4.10 Link-Werte tauchen sowohl in SCRIPT als auch in SCRIPT⁺ nicht explizit auf. Es gibt aber eine `for`-Schleife zur Iteration über Kollektionen. Dies entspricht der von uns vertretenen Auffassung, nach der Links so weit wie möglich vor dem Benutzer verborgen werden (vgl. Abschnitt 4.4). Die Iteration über die Elemente einer Kollektion *C*, die eine Bedingung *B* erfüllen, läßt sich in SCRIPT⁺ wie folgt formulieren:

```
for i in C: B(i) do
  Anw(i)
end for;
```

Dabei ist `i in C: B(i)` eine einfache Range-Expression: `i` ist eine Iteratorvariable, *C* ein kollektionswertiger Ausdruck und *B*(`i`) ein von `i` abhängiges Prädikat. Es soll *Anw*(`i`) ausgeführt werden für alle i ∈ *C*, die *B*(i) erfüllen, d.h. es wird deklarativ spezifiziert, über welche Elemente einer Kollektion iteriert werden soll. Das Konstrukt wird in der Weise übersetzt, daß zunächst durch eine Anfrage alle relevanten Sohnknoten von `C` ermittelt werden. Danach erfolgt die eigentliche Iteration, die über einen Iterator `it` gesteuert wird und die sich über die Ergebnismenge der Anfrage (wiederum eine Menge von Links) erstreckt:

```
Q := query({getLink(i)| i in C: B(i)});
it := firstIn(Q);
B := defined?(it);
while B do
  i := it^^;
  B := next(it);
  Übersetzung von Anw(i)
end while;
```

Der Link `it` muß zweimal dereferenziert werden, um dem im Ausgangsscript erwarteten Wert von `i` zu entsprechen. Man beachte, daß der Iterator `it` bereits *vor* dem Auswerten von *Anw*(`i`) weitergesetzt wird! Dies ist notwendig, damit die Übersetzung auch für den Fall korrekt ist, daß im Rumpf der `for`-Schleife eine Löschoperation durchgeführt wird, wie in folgendem Beispiel:

```
for i in Personen: Personen.Adresse.Ort = "Kassel" do
  Personen rem i;
end for;
```

Die Übersetzung von *Anw* (`i`) = `Personen rem i;` ist `remove(Personen, i);` ❒

Beispiel 4.11 Ist `Person` der Objekttyp aus Beispiel 3.7, dann wird in SCRIPT$^+$ durch folgende Anweisung ein neues Objekt dieses Objekttyps erzeugt und an die Variable `neueP` gebunden:

```
neueP := Person(Name: [VName: "Sven", NName: "Thelemann"],
                GebDat: 28.05.63,
                Hobbies: {"Musik", "Kino"});
```

Die Anweisung wird übersetzt in folgende Anweisungsfolge:

```
neueP := initObject(Person);
vn := "Sven"; update(neueP.Name.VName, vn);
nn := "Thelemann"; update(neueP.Name.NName, nn);
dat := 28.05.63; update(neueP.GebDat, dat);
h := "Musik"; insert(neueP.Hobbies, h);
h := "Kino"; insert(neueP.Hobbies, h);
```

Nach der Initialisierung eines neuen `Person`-Objektes mit Default-Werten gemäß *initVal* aus (3.26) werden die Initialwerte aus der SCRIPT$^+$-Anweisung in entsprechende `update`- und `insert`-Operationen umgesetzt. ❒

Beispiel 4.12 Beim Initialisieren eines Objektes zu einem Typ `t`, der einen Supertyp hat, muß darauf geachtet werden, daß die Bedingung (3.30) eingehalten bleibt:

Zunächst wird `initObject`((t, $t_{deftype}$)) aufgerufen, zurückgegeben werde (ω, t). Ein Aufruf von `initObject` fügt das neu erzeugte Objekt ω lediglich in die Domäne *dom* (t), jedoch nicht automatisch in die Domänen der direkten oder indirekten Supertypen von t ein.
Dies erreicht man, indem die Funktion `addType`((ω, t), (t', $t_{deftype}$)) aufgerufen wird.

Die SCRIPT$^+$-Anweisung

```
einStudent := Student(
                Name:[VName: "Hans", NName: "Schmidt"],
                FB: 17);
```

wird also in folgende Anweisungen der Basissprache übersetzt:

```
p := initObject(Person);
einStudent := addType(p, Student);
vn := "Hans"; update(einStudent.Name.VName, vn);
nn := "Schmidt"; update(einStudent.Name.NName, nn);
fb := 17; update(einStudent.FB, fb);
```

Man beachte auch, daß die erste `update`-Operation auf das Attribut `Name` in *I* (t_{Person}) (ω) zugreift, da der Objekttyp `Student` dieses Attribut von `Person` erbt. Für das Update von `FB` wird dagegen auf *I* ($t_{Student}$) (ω) zugegriffen. ❒

Beispiel 4.13 In vielen Fällen ist das Ergebnis einer Anfrage ein eindeutig bestimmtes Datenobjekt. Zurückgegeben wird jedoch eine Kollektion. Um ein einzelnes Datenobjekt von

seiner umgebenden Kollektionsstruktur zu „befreien“ wird in [Pau94] das `the(...)`-Konstrukt bereitgestellt. Die SCRIPT$^+$-Zuweisung

```
p := the(p in Person: p.Name.NName = "Thelemann");
```

wird übersetzt in

```
q := query({p | p in Person: p.Name.NName = "Thelemann"});
c := card(q); eins := 1; b := equal(c,eins); b := not(b);
if b then   // card(q)≠1
  p := null;
else
  it := firstIn(q);
  p := it^;
end if;
```

Falls das Ergebnis der Anfrage kein eindeutiges Datenobjekt liefert, hat `p` also den Wert `null`. ❒

Beispiel 4.14 Bei Zuweisungen ist BASESCRIPT sehr restriktiv und läßt lediglich die Zuweisung an transiente Namen zu. In SCRIPT$^+$ werden Zuweisungen ähnlich restriktiv behandelt. Zuweisungen sind erlaubt, falls die linke Seite

- ein transienter Name ist
- ein Pfadausdruck p $\in \mathcal{L}$(*pathExpr*), wobei *pathExpr* das Nichtterminal aus Anhang B.3 ist, für den gilt: *type*(*struct*(μ(p))) $\in$ TYPES – CONSTR
  (Dann nämlich ist der Knoten μ(p) ein erlaubter Parameter für eine der beiden generischen Update-Operationen `update` und `updateVar`).

Im ersten Fall wird eine SCRIPT$^+$-Zuweisung `x :=` e wie folgt übersetzt: Die Auswertung des Ausdrucks e ergibt eine Folge von BASESCRIPT-Anweisungen, die mit der Zuweisung des Ergebnisses an eine temporäre Variable `tmp` beendet wird. Danach wird die BASESCRIPT-Zuweisung `x := tmp` ausgeführt.

Im zweiten Fall wird eine SCRIPT$^+$-Zuweisung p `:=` e mit p $\in \mathcal{L}$(*pathExpr*) wie folgt übersetzt: Zunächst erfolgt wieder die Auswertung des Ausdrucks e und die Zuweisung des Ergebnisses an eine temporäre Variable `tmp`. Danach folgt je nach Typ von p die BASESCRIPT-Anweisung `update(`p`, tmp)` oder `updateVar(`p`, tmp)`. So wird z.B.

```
neueP.Adresse.Ort := "Kassel";
```

übersetzt in

```
tmp := "Kassel"; update(neueP.Adresse.Ort, tmp);
```

Sind z.B. `S` und `T` Namen für Mengen mit gleicher Elementstruktur und ist `S` ein transienter Name, dann übersetzt sich `S := S minus T` einfach in `S := query(S minus T)`. Ist dagegen `S` ein Tabellenname, dann ist die Zuweisung `S := S minus T` nicht erlaubt. Jedoch kann mittels

```
for i in S: i in T do
  S rem i;
end for;
```

der gewünschte Effekt erzielt werden. Auf diese Weise bleibt auch in SCRIPT$^+$ gewährleistet, daß Modifikationen von persistenten Kollektionen nur über `add`, `rem` oder `move` durchge-

führt werden können.

In der Syntax von SCRIPT ist nach [Pau94] beispielsweise `S add e` gar keine Anweisung, sondern ein mengenwertiger Ausdruck. Der Wert des Ausdruckes ist gerade die um `e` erweiterte Menge `S`. Das bedeutet jedoch nicht, daß `e` tatsächlich in `S` eingefügt wird, d.h. die Auswertung des Ausdruckes ist ohne „Nebenwirkung“. In SCRIPT muß das Einfügen von `e` in `S` als Zuweisung `S := S add e` formuliert werden. Es ist jedoch auch die Zuweisung `T := S add e` erlaubt – mit ggf. recht dramatischen Auswirkungen: Ist `T` der Name einer Tabelle, dann wird die bisherige Instanz von `T` komplett „weggeworfen“, und `T` und `S` sind danach an isomorphe Kopien derselben Menge gebunden! Wir plädieren daher für einen eingeschränkteren Gebrauch von Zuweisungen, wie er oben beschrieben wurde und betrachten `S add e` in SCRIPT$^+$ nicht als Ausdruck, sondern als Anweisung. Somit darf `S add e` auch kein Bestandteil einer Anfrage sein. Dies bedeutet jedoch keine Einschränkung, da in SCRIPT$^+$ stattdessen der Ausdruck `S union {e}` verwendet werden kann. ❐

4.7.4 Weitere Aspekte der Sprache SCRIPT$^+$

Wir wollen nun auf weitere Konstrukte von SCRIPT$^+$ eingehen, die über den Sprachvorschlag aus [Pau94] hinausgehen. Dies betrifft den Umgang mit varianten Typen, insbesondere mit Objekttypvarianten. Daneben gehen wir auf die Sprachkonstrukte ein, die im Zusammenhang mit der dynamischen Spezialisierung stehen.

4.7.4.1 Variante Typen

Für den Umgang mit Instanzen varianter Typen orientieren wir uns an der Syntax aus [Pau94], die wiederum auf [AGO90] zurückgeht.

Es seien `Polyeder`, `Flaeche`, `Gerade` und `Punkt` Objekttypen zur Repräsentation 3-dimensionaler geometrischer Objekte. Ferner wird ein varianter Typ `3D` definiert, dessen Alternativen aus den genannten Objekttypen bestehen (die `ops needed`-Klausel soll zunächst außer acht gelassen werden):

```
define variant
  -name 3D
  -variants Polyeder, Flaeche, Gerade, Punkt, 3DGruppe
  -ops needed trans(vector(3) ->void),
      rot(winkel:float, achse: vector(3) ->void),
      scale(float -> void),
      draw(-> void)
end define;
```

Dabei ist die Alternative `3DGruppe` ein Objekttyp mit der Struktur

```
[Name: string, Gruppe: <3D>],
```

d.h. eine Instanz von `3DGruppe` stellt eine benamte Liste von einzelnen 3D-Objekten oder anderen 3D-Gruppen dar. Hier treffen wir also auf eine rekursive Varianten-Spezifikation, wie sie für „base part/composite part“-Situation – und um eine solche handelt es sich hier – typisch ist.

Es sei `geo` eine Variable, die an den Wert $((l, \omega), \nu_{3D})$ mit $\nu_{3D} \in$ VARTYPES gebunden ist. Der SCRIPT⁺-Ausdruck `geo is Polyeder` wird in den gleichlautenden Ausdruck in BASE-SCRIPT übersetzt (vgl. Abschnitt 4.6.2) und gibt also den Wert von `Polyeder` = *l* zurück.

Um von einem varianten Wert auf seinen „tatsächlichen“ Wert zu kommen, wird ein Ausdruck der Form *"expr* `as` *label*" ausgewertet. Falls nicht zugesichert werden kann, daß `geo is Polyeder` gilt, dann muß die SCRIPT⁺-Anweisung `poly := geo as Polyeder` wie folgt übersetzt werden:

```
b := geo is Polyeder;
if b then
  poly := strip(geo);
else
  poly := null;
end if;
```

Es sei nun `GeoGruppen` eine Tabelle mit der Struktur `{3DGruppe}`, die benamte Gruppen von 3D-Objekten verwaltet. Es soll nun die Gruppe mit dem Namen "`xyz`" ermittelt und für sie die Methode `draw` aufgerufen werden, die die Gruppe auf dem Bildschirm darstellt. In SCRIPT⁺ läßt sich dies wie folgt formulieren:

```
g := the(g in GeoGruppen: g.Name = "xyz");
g.draw();
```

Dabei wird die für den Objekttyp `3DGruppe` definierte Methode `draw` aufgerufen. Ihr läßt sich folgendes Script zuordnen:

```
[ self: 3DGruppe -> void | |
  for geo in self.Gruppe do geo.draw(); end for; return;]
```

Die Voraussetzung dafür, daß wir im Rumpf der `for`-Schleife tatsächlich `geo.draw()` schreiben dürfen, wurde mit der `ops needed`-Klausel in der Definition der Objekttypvarianten erfüllt. Sie gibt die Namen und Signaturen der Methoden an, die bei *allen* Alternativen von `3D` vorhanden sein müssen. Sie ist also als Konsistenzbedingung zu verstehen, damit gewisse „Gemeinsamkeiten“ bei den Alternativen zugesichert werden können.

Auf diese Weise wird in ESCHER⁺ für variante Typen ein *late binding* realisiert: Der Rumpf der `for`-Schleife wird übersetzt in

```
t := strip(geo); t.draw();
```

Der Typ von `t` steht erst zur Laufzeit fest, es kann jedoch zugesichert werden, daß auf jeden Fall die Methode `draw` ausführbar ist. Andernfalls wäre man im Umgang mit varianten Werten auf die Verwendung des `case`-Konstruktes angewiesen, das alle Alternativen einzeln abhandelt. Dies ist in [Pau94] der Fall. Ist man grundsätzlich auf das `case`-Konstrukt angewiesen, so erweist sich der praktische Umgang mit Varianten als eher unhandlich. Mit Hilfe der `ops needed`-Klausel erreichen wir eine wesentlich übersichtlichere Kodierung.

Analog zur `ops needed`-Klausel gibt es auch eine `struct needed`-Klausel, die gewisse Attribute in allen Alternativen einfordert. Außerdem ist der Inhalt dieser beiden Klauseln über eine `modify`-Anweisung änderbar (vgl. Anhang B.2).

Nachdem der Nutzen der `ops needed`- und `struct needed`-Klauseln für den Einsatz von Objekttypvarianten motiviert haben, bedarf es nun noch der Erweiterung der formalen

Definition varianter Typen gemäß Def. 3.13, damit auch die Information aus den …`needed`-Klauseln Bestandteil der Definition eines varianten Typs, genauer einer Objekttypvariante, ist.

Definition 4.6 (Ergänzung zu Def. 3.13 – variante Typen –)
Auf der Menge OBJVAR, die alle Objekttypvarianten aus VARTYPES enthält, seien zusätzlich die Funktionen

struct_needed: OBJVAR $\rightarrow \mathcal{S}$(TYPES),
ops_needed: OBJVAR $\rightarrow \mathcal{FS}et(\mathcal{N} \times \mathcal{S}ign$(TYPES))

definiert, deren Funktionswerte sich aus den …`needed`-Klauseln ergeben. Analog zur Bedingung (3.22) soll gelten: $\forall$ t $\in$ OBJVAR: *type*(*struct_needed*(t) = c_{tuple}).

Ein Typ t' $\in$ DEFTYPES ist genau dann eine gültige Alternative einer Objekttypvariante t$\in$ OBJVAR , wenn gilt:

- *struct*(t') $\leq_{tup}$ *struct_needed*(t) (vgl. Def. 3.12)
- $\forall$ (m, s) $\in$ *ops_needed*(t) $\exists$ o $\in$ *ops*(t'): *name*(o) = m $\wedge$ *sign*(o) $\leq_{contrav}$ s [15]

Tatsächlich muß die Signatur einer Operation für eine Alternative nicht genau die in der `ops needed`-Klausel angegebene Signatur s' haben, sondern es genügt, wenn beide in der Kontravarianz-Beziehung zueinander stehen. ❒

Das folgende Beispiel zeigt, daß bei der Verwendung von Objekttypvarianten als Elementtyp einer Kollektion ein explizites „Casting“ beim Einfügen in die Kollektion unnötig ist.

Beispiel 4.15 In der Tabelle `GeoGruppen` sei eine Gruppe von 3D-Objekten mit dem Namen "`xyz`" enthalten. In diese Gruppe soll ein neues Polyeder-Objekt eingefügt werden, an das die Variable `neuesPolyeder` (Typ `Polyeder`) gebunden sind. Ein *type cast* auf den Elementtyp `3D` ist in SCRIPT$^+$ nicht notwendig. Die Anweisungen

```
g := the(g in GeoGruppen: g.name = "xyz");
g add neuesPolyeder;
```

sind zulässig und leisten das Gewünschte. Die zweite Anweisung übersetzt sich in

```
geo := (+ 3D.Polyeder: neuesPolyeder +);
insert(g.Gruppe, geo);
```

❒

Zuletzt betrachten wir ein Beispiel zur Behandlung von varianten Typen in der Bedingung einer Range Expression. Es soll nach allen roten Flächen gefragt werden, die in einer Gruppe von 3D-Objekten der Tabelle `GeoGruppen` enthalten sind. Dazu stellen wir folgende Anfrage:

```
{f as Flaeche | g in GeoGruppen, f in g.Gruppe:
 f is Flaeche and (f as Flaeche).Farbe = "rot"}
```

Die Bedingung ist so formuliert, daß bei ihrer Auswertung von links nach rechts sichergestellt ist, daß auf das Attribut `Farbe` nur dann zugegriffen wird, wenn für `f` tatsächlich die Alternative `Flaeche` zutrifft. Falls nämlich `f is Flaeche` *false* ergibt, dann wird der zweite Teil der Konjunktion nicht mehr ausgewertet. Fordert man, daß vor jedem „type cast“ mittels `as`

15. $\leq_{contrav}$ ist die Kontravarianz-Beziehung aus Abschnitt 4.5.2

ein Test auf seine Zulässigkeit stehen muß, dann führt dies jedoch zu syntaktisch umständlichen Ausdrücken. Für Range Expressions wollen wir deshalb auch zunächst „unsicher“ erscheinende „type casts“ zulassen. So soll in obiger Anfrage die Bedingung

```
(f as Flaeche).Farbe = "rot"
```

ebenfalls zulässig sein. Damit es dann aber zu keinem Laufzeitfehler kommt, fordern wir für Bedingungen in einer Range Expression folgende Auswertungsstrategie:

> Bei der Auswertung einer Bedingung einer Range Expression, die einen Ausdruck der Form "e `as` *l*" enthält, wird unmittelbar vor dem „type cast“ getestet, ob "e `is` *l*" gilt. Falls dies nicht der Fall ist, dann gilt die gesamte Bedingung als *false*.

4.7.4.2 Dynamische Spezialisierung

In Abschnitt 3.7.2 wurde die Notwendigkeit dynamischer Spezialisierung, d.h. des nachträglichen Hinzufügens weiterer Subtypen zu einem existierenden Objekt, herausgestellt. An dieser Stelle soll nun auf die in dieser Hinsicht relevanten Sprachmittel von SCRIPT$^+$ eingegangen werden.

Wir beziehen uns im folgenden auf die Objekttypen `Person` und `Student` aus Beispiel 3.7. Es seien `aPerson` bzw. `aStudent` lokale Variablen (d.h. transiente Namen) vom Typ `Person` bzw. `Student`. Die Anweisung

```
aPerson := Person(initVals);                                    (4.16)
```

erzeugt ein neues Objekt ω und bindet (ω, t_{Person}) an den Bezeichner `aPerson`. Das Nichtterminal *initVals* steht dabei für Initialisierungen ausgewählter Objektattribute. Es gilt nun $trans\_types(\omega) = \{t_{Person}\}$.

Wir erweitern die ursprüngliche Syntax von SCRIPT, um dem Objekt ω mittels

```
aStudent := aPerson add type Student(initVals)                  (4.17)
```

dynamisch einen neuen Typ hinzuzufügen, wobei *initVals* wiederum für Initialisierungen der Attribute des Typs `Student` steht. Über `add type` können einem Objekt ω alle direkten oder indirekten Subtypen der Typen aus $types(\omega)$ hinzugefügt werden. Ist *initVals* in (4.17) leer, dann übersetzt sich die SCRIPT$^+$-Anweisung einfach in

```
aStudent := addType(aPerson, Student);
```

Es gilt nun $trans\_types(\omega) = \{t_{Person}, t_{Student}\}$. Wäre `Student` kein direkter Subtyp von `Person`, dann müßten für alle auf dem Pfad entlang der Subtyp-Hierarchie zwischen `Student` und `Person` liegende Typen ebenfalls `addType`-Aufrufe erzeugt werden. Auf diese Weise wird die Gültigkeit der Bedingung (3.30) eingehalten.

Es seien `Personen` bzw. `Studenten` zwei Tabellen mit den Strukturen `{Person}` bzw. `{Student}`. Nach der Ausführung von

```
Personen add aPerson;                                           (4.18)
```

gilt $pref(I(t)(\omega)) = 1$, und es ist $pers\_types(\omega) = \{t_{Person}\}$, $trans\_types(\omega) = \{t_{Student}\}$. Jenseits des Gültigkeitsbereiches der lokalen Variable `aStudent` ist `Student` auch kein transienter Typ von ω mehr.

Genauso wie bei neu erzeugten Objekten hat das Hinzufügen eines weiteren Typs zu einem Objekt also zunächst „vorläufigen“ Charakter, da das Objekt unter der neuen Rolle bzw. den neuen Rollen noch nicht erreichbar im Sinne von Def. 3.22 ist. Mittels

```
Studenten add aStudent;                                    (4.19)
```

wird jedoch auch der Typ `Student` zu einem persistenten Typ von ω. Hätten wir auf (4.18) verzichtet, dann wären durch die Ausführung von (4.19) gleich beide Typen `Person` und `Student` zu persistenten Typen geworden.

Typerweiterung und Einfügen können auch „in einem Zug“ ausgeführt werden. Dies erreicht man mittels der Anweisung

```
Studenten add (aPerson add type Student(initVals));
```

Eine Spiegelbild zu `add type`, etwa in der Form von `drop type`, gibt es in ESCHER$^+$ nicht. Würden wir mit `drop type` den expliziten Entzug eines Typs t von einem Objekt ω zulassen, dann hätte dies zur Folge, daß alle Knoten r = (ω, t') mit t' *isa** t in der Datenbankinstanz ungültig werden, da ein Zugriff auf ω relativ zu t oder einer seiner Subtypen nicht erlaubt ist. Es müßte also die gesamte Datenbankinstanz nach solchen Knoten r durchsucht werden. Danach stellt sich sofort die Frage, was mit den gefundenen, ungültigen Knoten r zu geschehen habe. Es bietet sich folgendes Vorgehen an: Ist r Element einer Kollektion, dann wird r aus der Kollektion entfernt. Ist r eine Tupelkomponente, dann wird diese zu *null* gesetzt. Auf jeden Fall ist nach einem expliziten `drop type` i.a. einen aufwendigen Suchvorgang in Gang zu setzen, der auch zu einer vom Benutzer nicht beabsichtigten „Kettenreaktion“ von Lösch- bzw. Update-Operationen führen kann. Eines der Datenmodelle, das trotzdem an einem expliziten Typentzug festhält, ist COCOON [SLR+92, LS92]. Dort wird dynamische Typzugehörigkeit über die Operationen `gain` und `lose` gesteuert. Wir wollen für ESCHER$^+$ das Entfernen von Typen aus *pers_types*(ω) durch das Prinzip der Erreichbarkeit steuern. Erst wenn die letzte persistente Referenz auf ein Objekt ω bezüglich eines Typs t entfernt wurde, gehören t und alle seine Subtypen nicht länger zu den persistenten Typen von ω.

Mit folgender `for`-Schleife werden alle Studenten „exmatrikuliert“, die länger als 20 Semester studiert haben, indem sie aus `Studenten` entfernt werden:

```
for s in Studenten: s.Semester > 20 do
  Studenten rem s;
end for;                                                   (4.20)
```

Falls die zu entfernenden Objekte ω_i nur über die Tabelle `Studenten` unter dem Typ `Student` erreichbar waren, dann gilt nach dem Entfernen $pref(I(t_{Student})(\omega_i)) = 0$, und die ω_i werden sofort aus Def(I(t)) entfernt.

Wenn aber `Student` einen weiteren Subtyp t' hätte und t' ∈ *pers_types*(ω_i) für eines der mittels (4.20) aus `Studenten` entfernten ω_i gälte, dann wäre auch nach dem Löschen noch $pref(I(t_{Student})(\omega_i)) > 0$. Also hätte ω_i nach wie vor den persistenten Typ `Student`, wäre jedoch nicht mehr via `Studenten` unter diesem Typ erreichbar.

Der Test, ob für ein Objekt ω ein Typ t in *types*(ω) liegt, läßt sich in SCRIPT$^+$ ähnlich wie der Test auf eine Alternative für Varianten mit dem `is`-Prädikat durchführen. Dabei können wir jedoch noch nach transienter und persistenter Typzugehörigkeit unterscheiden. In Tabelle 4.2 wird die Übersetzung von `is`-Prädikaten nach BASESCRIPT anhand von Beispielen erläutert.

SCRIPT$^+$-Prädikat	Übersetzung nach BASESCRIPT
`aPerson is transient Student`	`b := hasTransType?(aPerson, Student);`
`aPerson is persistent Student`	`b := hasPersType?(aPerson, Student);`

`aPerson is Student`	`b1 := hasTransType?(aPerson, Student);`
	`b2 := hasPersType?(aPerson, Student);`
	`b := or(b1, b2);`

Tabelle 4.2: `is`-Prädikate zur Prüfung der Typzugehörigkeit

Fehlen die Schlüsselwörter `transient` und `persistent` in einem `is`-Prädikat, dann handelt es sich um eine Oder-Verknüpfung der Basis-Prädikate `hasTransType?` und `hasPersType?`. Nach der Ausführung der Anweisungen (4.16), (4.17) und (4.18) gilt:

`aPerson is persistent Person` → *true*
`aPerson is transient Person` → *false*
`aPerson is persistent Student` → *false*
`aPerson is transient Student` → *true*

Mittels

```
aPerson is transient Person
```

läßt sich auch testen, ob ein neu initialisiertes und an den Bezeichner `aPerson` gebundenes Objekt bereits erreichbar und somit persistent gemacht wurde. Man beachte auch, daß in den Beispielen aus Tabelle 4.2 `aPerson` durch `aStudent` ersetzt werden kann, ohne daß sich an den Werten der Prädikate etwas ändert, denn beide Namen sind an dasselbe Objekt ω gebunden.

Das folgende Beispiel illustriert, daß durch Entfernen von Objekten aus Kollektionen eine Typzugehörigkeit zumindest noch temporär aufrechterhalten werden kann. Wir nehmen an, daß mittels

```
s := the(s in Studenten:s.Name.NName = "Mustermann");
Studenten rem s;
```

der eindeutig bestimmte Student ω mit dem Namen "Mustermann" aus `Studenten` entfernt wird. Der lokale Name `s` bleibt jedoch an $(\omega, t_{Student})$ gebunden. Falls nach dem Entfernen von $pref(I(t_{Student})(\omega)) = 0$ ist, dann gilt:

`s is transient Student` → *true*
`s is persistent Student` → *false*

Das bedeutet, daß `Student` zwar kein persistenter Typ von ω ist, jedoch $I(t_{Student})(\omega)$ vorerst gerettet wurde, indem die Studenten-Information für ω noch über den lokalen Namen `s` erreichbar ist. Erst mit der nächsten `return`-Anweisung wird ω aus $Def(I(t_{Student}))$ entfernt!

Um ein Objekt unter einer bestimmten *Typ-Perspektive* zu betrachten, muß ein „type cast“ durchgeführt werden. Dies geschieht mit SCRIPT$^+$-Ausdrücken der Form *"expr* `as` *defType"*, d.h. wir verwenden ein Analogon zum `as`-Konstrukt für variante Typen.

Es seien $e \in \mathcal{L}(expr)$, $tname \in \mathcal{L}(defType)$ mit $\sigma(e) = (\omega, t)$ und $name(t) = tname$ für $t \in$ DEFTYPES. Der Ausdruck "e `as` tname" liefert das getypte Objekt (ω, t), falls $t \in types(\omega)$.

Andernfalls wird *null* zurückgegeben und eine Ausnahme erzeugt. In vielen Fällen ist e ein Bezeichner, so daß "e `as` tname" einfach in einen Aufruf `cast(e, tname)` übersetzt wird. Der Mengenformer

```
{p as Student | p in Personen: (p as Student).FB = 17}
```

filtert aus der Menge `Personen` diejenigen Objekte heraus, die den Typ `Student` besitzen und präsentiert diese im Anfrageergebnis auch unter dieser „Sicht“. Genauso wie wir es bereits für `as`-Ausdrücke im Zusammenhang mit varianten Typen gefordert hatten, soll auch hier die Bedingung der Range Expression zu *false* ausgewertet werden, sobald ein unzulässiger „type cast“ zur Auswertung ansteht. Man beachte, daß die letztgenannte Anfrage nur dann äquivalent ist zu

```
{s | s in Studenten: s.FB = 17},
```

falls `Studenten` eine Teilmenge von `Personen` ist, was bisher jedoch an keiner Stelle explizit gefordert wurde.

4.8 Erweiterbarkeit des Datenmodells

Bereits in Abschnitt 2.4.2 wurde die *Erweiterbarkeit* eines DBMS als entscheidendes Kriterium für ihren Einsatz in unterschiedlichen Anwendungsgebieten herausgestellt. Im Datenmodell ESCHER$^+$ kann das „Grundmodell“, das aus den Basistypen und Basisoperationen besteht, um Objekttypen, Aufzählungstypen, variante Typen und benutzerdefinierte Operationen erweitert werden. Erweiterbarkeit bedeutet in diesem Sinne die Definition anwendungsspezifischer Typen und Operationen, die zu Komponenten eines konkreten Datenbankschemas werden. Erweiterbarkeit hat jedoch auch einen anderen Aspekt, nämlich die Möglichkeit der flexiblen Erweiterung des zur Verfügung stehenden Satzes an Basistypen und Basisoperationen, die die *fundamentalen* Bausteine des Datenmodells bilden.

Die zum „Standardrepertoire“ eines DBMS zählenden Basistypen spiegeln zumeist den „klassischen“ Anwendungsbereich kommerzieller Datenbanksysteme wider: Es handelt sich im wesentlichen um numerische und String-Datentypen, die auch in den verbreiteten Programmiersprachen zu finden sind, zuweilen auch um Basistypen wie `date` und `money`. Es wäre nicht adäquat, einen fundamentalen Datentyp, wie z.B. `date`, als Objekttyp definieren zu müssen. Die Lösung kann nur darin bestehen, `date` tatsächlich als einen neuen Basistyp inklusive der zugehörigen Basisoperationen einführen zu können, der dann für alle Datenbankschemata zur Verfügung steht.

Neben der Erweiterung um „einfache“ Datentypen wie `date` besteht auch Bedarf an der Erweiterung um neuere Datentypen, wie sie etwa in Multimedia-Anwendungen vorkommen. Dazu gehören z.B. unterschiedliche Formate für Pixel-Graphiken mit entsprechenden Operationen (Import, Export, Darstellung, Konversion zwischen Formaten, Manipulation, Extraktion von Teilinformation wie z.B. die Anzahl der verwendeten Farben), die vom Standpunkt des Datenmodells als atomar anzusehen sind. Wurde bisher nur von der Erweiterbarkeit hinsichtlich der Basistypen gesprochen, so ist auch eine Erweiterbarkeit der Menge der Konstruktoren denkbar. Theoretische Überlegungen wurden dazu in [MS91] angestellt. Allerdings ist der Bedarf an Erweiterbarkeit an diesem Punkt als eher gering einzustufen.

Das relationale DBMS Ingres bietet die Möglichkeit an, über die sog. „Object Management Extension“ [Ing91] dem System neue Basistypen in der Art von POSTGRES-ADTs (siehe Abschnitt 2.4.2) hinzuzufügen. Die Bezeichnung „Object Management Extension“ ist allerdings irreführend: Es werden keine Objekttypen, sondern ADTs im Sinne von *Werte*typen definiert. Während [RS87] für POSTGRES den Eindruck vermittelt, daß eine Erweiterung durch wenige Zeilen Code realisiert wird, ist in der Realität von Ingres ein „en passant“-Hinzufügen neuer Basistypen nicht möglich. Dies ist jedoch auch nicht anders zu erwarten, da die Erweiterungen an der Schnittstelle zwischen logischem Datenmodell und seiner Implementation angesiedelt sind. Eine Erweiterung in Ingres erfordert das Schreiben von umfangreichem C-Code, durch dessen Ausführung verschiedene Systemtabellen manipuliert werden. Für eine Erweiterung werden im wesentlichen folgende Informationen benötigt:

- eine C-Datenstruktur für die interne Repräsentation des ADTs
- C-Routinen zur Konversion zwischen interner Darstellung der ADT-Werte und verschiedenen Formaten ihrer textuellen Darstellung
  (für die Ausgabe auf dem Bildschirm, für die Verwendung in der Anfragesprache)
- C-Routinen zur Unterstützung der Query-Optimierung und der physischen Speicherung in einem der Ingres-Speichermodi ISAM, BTREE, HASH
- C-Routinen zum Check der Korrektheit eines Wertes und zur Erzeugung eines Default-Wertes
- C-Routinen zur Implementation der neuen Basisoperationen, die auf dem ADT definiert sind

Nach einer gründlichen, innerhalb einer Testinstallation ausgeführten Testphase können die Erweiterungen inkrementell zur Arbeitsinstallation hinzugebunden werden. In [Ing91] werden für das Beispiel der Erweiterung um einen Datentyp `ord_pair`, dessen Instanzen geordnete Paare von Zahlen sind, 77 (!) Seiten C-Code dokumentiert, was ein eindeutiges Indiz für die Komplexität bereits relativ simpler Erweiterungen ist. Allerdings muß berücksichtigt werden, daß die Erweiterungen u.a. auch Informationen für den Query Optimizer berücksichtigen und eine nahtlose Integration der neuen Operationen in die SQL-Syntax anstreben (ggf. durch weitere Überlagerung von Operatorzeichen wie `=`, `+`, `<=` usw.), so daß „ad-hoc“-Erweiterungen sowieso unrealistisch sein müssen.

Die Kommerzialisierung von POSTGRES, das „objekt-relationale“ Datenbanksystem Illustra [Ill96a], hat erkannt, daß der Aufwand einer Erweiterung um neue Basistypen eher abschreckend ist. Es werden deshalb für unterschiedliche Anwendungsbereiche vorgesehene *Data Blades* angeboten, die dem System modulartig hinzugefügt werden können. Data Blades sind Basistyp-Bibliotheken, die mit Klassenbibliotheken objektorientierter Systeme vergeichbar sind. Im Jahr 1993 standen vier solche Bibliotheken zur Verfügung:

- *Foundation Data Blade* mit ca. 40 aus den relationalen DBMS bekannten „Standard“-Datentypen
- *Text Data Blade*: Datentyp für Textdokumente unterschiedlicher Formate. Verschiedene Operationen zum Retrieval von Texten. Texte werden beim Speichern automatisch indiziert. In einer Tabelle können Texte in unterschiedlichen Formaten koexistieren.
- *Spatial Data Blade*: Verschiedene 2D-Datentypen mit Manipulationsoperationen. Im-plementiert ist die Zugriffsstruktur R-Tree [Gut84].

- *Image Data Blade*: Verschiedene Graphik-Formate für Bitmap-, Greyscale- und Farbgraphiken, Import/Export/Konvertierungs-Funktionen sowie Bildmanipulationsoperationen

Anfang 1996 gab es immerhin schon 12 verschiedene Data Blades, darunter ein Statistik-Paket, eine Kopplung an das relationale DBMS Sybase und ein Modul zur Unterstützung eines World Wide Web Servers [Ill96b].

Mit diesem Ansatz wird zweierlei erreicht: Zum einen wird der Entwickler von der offensichtlich langwierigen und fehleranfälligen Entwicklung neuer Basistypen befreit, indem er sich das für ihn geeignete Paket einfach dazukauft (über das „DataBlade Developer's Kit" besteht jedoch immer noch die Möglichkeit, ähnlich wie bei Ingres, selbst aktiv zu werden). Zum anderen werden Data Blades zusammen mit anderen Software-Herstellern entwickelt oder bei diesen eingekauft. Auf diese Weise wird eine Integration von bisher dateiorientierten Anwendungen (z.B. Desk Top Publishing) in eine Datenbankumgebung möglich. Während der Ingres-Ansatz keinen Vererbungsmechanismus kennt, bietet Illustra Vererbung an, und jeder Datentyp aus einem gegebenen Data Blade kann gemäß der individuellen Anforderungen verfeinert werden. Leider werden – wie bei kommerziellen Anbietern üblich – keine Einblicke in die Realisation des Vererbungsmechanismus preisgegeben, so daß keine genaueren Aussagen über die Leistungsfähigkeit dieses Ansatzes möglich sind.

Hinsichtlich der Erweiterbarkeit haben wir für ESCHER$^+$ in diesem und im vorangegangenen Kapitel die notwendigen Grundlagen geschaffen. Eine Implementation des ESCHER$^+$-Datenmodells gilt dann als erweiterbar, wenn dies für die Mengen BASETYPES, CONSTR und BASEOPS zutrifft. Diese Mengen bilden gewissermaßen die Schnittstelle zwischen dem logischen Datenmodell und der Implementation eines konkreten DBMS. Durch eine adäquate Metamodellierung kann die Erweiterbarkeit maßgeblich unterstützt werden. Für ESCHER$^+$ wurde in Abschnitt 3.10 ein Meta-Schema entwickelt, das jedoch zunächst nur den strukturellen Teil des Datenmodells abdeckt. In ihm sind für die Mengen BASETYPES und CONSTR gleichnamige Tabellen vorgesehen. In das Meta-Schema läßt sich aber auch die Informationen zum operationalen Teil des Datenmodells leicht integrieren. Dazu werden im Meta-Schema weitere Objekttypen t_{baseop} und t_{defop} definiert, deren Instanzen die Elemente der Mengen BASEOPS und DEFOPS repräsentieren. Die Attribute dieser Objekttypen ergeben sich aus den in Def. 4.3 und 4.5 angegebenen Funktionen. So hat t_{baseop} u.a. das Attribut `semantics`. In einer konkreten Realisierung können die Werte dieses Attributes Zeiger auf Funktionen sein, die die Basisoperationen implementieren. Die Objekttypen $t_{basetype}$ und t_{baseop} werden i.d.R. auch weitere, hier und in Abschnitt 3.10 nicht genannte Attribute besitzen, die implementationsspezifisch sind und Informationen für den zugrundeliegenden Speichermanager enthalten (z.B. über die Art der Speicherung eines Basistyps: feste Länge mit n Bytes, variable Länge usw.). Entscheidend ist, daß das Meta-Schema die geeignete Schnittstelle ist, um neue Basistypen und Basisoperationen hinzuzufügen oder bestehende zu modifizieren. Für das implementierte logische Datenmodell sind dann genau diejenigen Basistypen und Basisoperationen verfügbar, die in den Tabellen BASETYPES bzw. BASEOPS des Meta-Schemas „registriert" sind.

4.9 Zusammenfassung und Diskussion

In diesem Kapitel wurde die operationale Komponente des Datenmodells ESCHER$^+$ eingeführt und formal beschrieben. Es wird zwischen vordefinierten Basisoperationen und benutzerdefinierten Operationen unterschieden.

Grundlage für die Beschreibung der Semantik aller Operationen sind die Baumrepräsentationen für Datenobjekte, wie sie in Kapitel 3 eingeführt wurden. Ein Laufzeitzustand besteht im wesentlichen aus einer Menge von Baumrepräsentationen, die manipuliert werden. Zu ihm gehören eine Datenbankinstanz sowie weitere, temporäre Baumrepräsentationen. In einer realen Implementierung soll eine automatische *garbage collection* dafür sorgen, daß der Speicher für nicht mehr erreichbare Bäume freigegeben wird.

Die Semantik der Basisoperationen wurde durch die Manipulation von Knoten, Kanten und Beschriftungen von Bäumen sowie durch die Erzeugung neuer Bäume angegeben. Während die Ausführung einer Basisoperation einen elementaren Zustandsübergang des Laufzeitzustandes auslöst, führt der Aufruf und die Abarbeitung einer benutzerdefinierten Operation zu einer Folge von Zustandübergängen. Wir haben mit BASESCRIPT eine primitive Sprache angegeben, deren Semantik vollständig angegeben wurde. Die Beschreibung in Abschnitt 4.6 eignen sich für die Implementation eines Interpreters, der ein in BASESCRIPT geschriebenes Basisscript schrittweise abarbeitet.

BASESCRIPT versteht sich als „primitive“ Zielsprache der Übersetzung aus einer „höheren“ Sprache. Wir setzen die Arbeiten an dem Sprachvorschlag SCRIPT aus [Pau94] fort, indem wir die Syntax erweitern bzw. modifizieren. Der so entstehenden Sprache SCRIPT$^+$ ließe sich eine vollständige formale Syntax zuordnen, was in dieser Arbeit jedoch nur auf Beispiele beschränkt bleibt. Die Erstellung von Ausführungsplänen für Anfragen muß ebenfalls Gegenstand zukünftiger Arbeiten sein, da die Optimierung von Anfragen nicht Gegenstand dieser Arbeit sein soll.

Mit der Einführung benutzerdefinierter Operationen wurde die Schnittstelle zu Instanzen von Objekttypen um eine Verhaltenskomponente ergänzt. Jedem Objekttyp wird eine Menge von Methoden zugeordnet. In den Scripten, die die Implementation einer Methode darstellen, kann direkt auf *alle* Attribute des Receiver-Objektes zugegriffen werden. Neben den Methoden können weitere, von Objekttypen „unabhängige“ Operationen definiert werden, die zusammengenommen die Menge USEROPS bilden.

Es wird im allgemeinen nicht erwünscht sein, daß ein Benutzer *alle* benutzerdefinierten Operationen einer Anwendung aufrufen darf. Vielmehr ist eine Einteilung in öffentlichen (*public*) und private (*private*) Operationen anzustreben, und nur die öffentlichen Operationen bilden das operationale Interface, das dem Benutzer zur Verfügung steht. Es handelt sich dann um eine Form der selektiven Einkapselung für Operationen. Nur die öffentlichen Operationen sind direkt aufrufbar, genauso wie nur auf öffentliche Attribute eines Objekttyps direkt zugegriffen werden darf. Die Differenzierung von Operationen nach *public* und *private* wird Gegenstand von Abschnitt 5.5.2 sein.

Eine andere Interpretation des Prinzips der Einkapselung bei Operationen bezieht sich auf die Möglichkeit der Modifikation der Implementation, d.h. des einer Operation zugeordneten Scriptes. Es sollte klar sein, daß die Erstellung und Modifikation benutzerdefinierter Operationen die Aufgabe von Spezialisten sein muß. Aus diesem Grund weichen wir von dem Vor-

schlag aus [Pau94] ab, der Scripten als Werte funktionaler Attribute auffaßt und somit eine „Reimplementation“ von Operationen für einzelne Objekte auch durch den Endanwender zuläßt. Dies kann in [Pau94] durch das Entziehen einer Schreibberechtigung auf das jeweilige funktionale Attribut verhindert werden, wobei jedoch unklar bleibt, ob es immer noch einen „Super-User“ geben soll, für den die Einschränkungen nicht gelten. In ESCHER$^+$ ist die Modifikation eines Scriptes Bestandteil der DDL, die nur bestimmten Benutzergruppen, wie z.B. Systemadministratoren und Anwendungsprogrammierern, zugänglich ist.

Es ist allerdings wünschenswert, daß sich besonders ausgebildete „gelegentliche“ Benutzer [Zeh89], die Fachleute auf ihrem Sachgebiet sind, eine individuelle Sammlung von *Retrieval-Scripten* anlegen können. Darunter sind solche Scripten zu verstehen, die lediglich lesende Zugriffe auf eine Datenbankinstanz enthalten. Dies können deklarative Anfragen, aber auch prozedural formulierte Auswertungen sein. Es ist dagegen nicht angebracht, dem „gelegentlichen“ Benutzer die Erstellung eigener Scripten zu ermöglichen, die eine Datenbankinstanz modifizieren. Eine Ausdifferenzierung nach unterschiedlichen Benutzern oder Benutzergruppen soll zukünftigen Arbeiten vorbehalten bleiben und wird deshalb im folgenden nicht näher behandelt.

Wir halten insbesondere die Möglichkeit der dynamischen Spezialisierung für eine wichtige Eigenschaft von ESCHER$^+$, die einen wesentlichen Gewinn gegenüber anderen Vorschlägen zur Erweiterung des eNF^2-Modells darstellt. Aufgrund der Bedeutung dieser Erweiterung soll nun noch auf andere existierende Ansätze zur dynamischen Typerweiterung eingegangen werden.

Die verschiedenen Typen, die ein Objekt besitzt, werden in vielen Arbeiten auch als *Rollen* bezeichnet. Eine sehr frühe Arbeit, die den Begriff „Rolle“ in diesem Kontext benutzt, ist [BD77], in der ein Rollen-Konzept für das Netzwerk-Modell vorgeschlagen wird.

Das OODBMS Iris [FAC+89] war unseres Wissens das erste DBMS, das tatsächlich die Veränderbarkeit der Typzugehörigkeit während der Lebensdauer eines Objektes berücksichtigte. Mit

```
Add type Student to aPerson
```

wird ein weiterer Typ `Student` dem an den Bezeichner `aPerson` gebundenen Objekt hinzugefügt. Der zusätzliche Typ ist sofort persistent. Ein Typ muß einem Objekt explizit wieder entzogen werden. Dafür ist die Operation `Remove type` zuständig. Einen Nachteil des Ansatzes von Iris sehen wir darin, daß bei jedem Zugriff auf ein Objekt *alle* aktuellen Typen eines Objektes relevant sind. Es wird in jedem Fall versucht, zur Laufzeit den minimalen Typ eines Objektes zu ermitteln und ausgehend von diesem auf ein Attribut zuzugreifen oder eine Methode aufzurufen (spätes Binden). Falls ein minimaler Typ nicht existiert, so wird der Zugriff durch eine vom Anwender spezifizierte Regel festgelegt. Leider enthält [FAC+89] keine nähere Information darüber, in welcher Form diese Regeln konkret zu formulieren sind. In Iris gibt es also keinen Zugriff relativ zu einem bestimmten Typ und demzufolge auch keine „type cast“ im Stile von "… `as` …" .

Offensichtlich basiert spätes Binden auf der Voraussetzung, daß jedem Objekt zur Laufzeit immer ein minimaler Typ zugeordnet werden kann. Davon kann man jedoch nicht mehr ausgehen, wenn ein dynamisches Hinzufügen von Typen erlaubt ist. Es gibt in der Literatur einige Vorschläge, die trotzdem spätes Binden unterstützen wollen: In [AGO90] wird die DBPL Nuovo Galileo vorgestellt. Das Hinzufügen weiterer Typen wird dort durch eine `extend`-Operation erreicht. Das späte Binden zur Auflösung eines Methodenaufrufes ist in Nuovo Galileo abhängig von der zeitlichen Reihenfolge des Hinzufügens der Typen. Genauso wie bei

Iris ist kein „type cast“ möglich, so daß sich ein Objekt überall gleich verhält, egal unter welchem Typ man zugreift.

Der in [AGO90] gemachte Ansatz wurde in [ABG+93] für die DBPL Fibonacci weiterentwikkelt. Auch hier ist spätes Binden abhängig von der Reihenfolge der Hinzunahme von Typen. Dies soll an folgendem Beispiel illustriert werden. Wir gehen von den Rollen `Person`, `Student` und `Employee` aus, wobei die beiden letztgenannten Sub-Rollen von `Person` sind. Für jede Rolle gibt es eine Methode `Introduce`, die die Ausgabe eines Strings erzeugt (eine Ausgabe wird durch vorangestelltes `>>>` angezeigt).

```
let aPerson = role Person …
                  let Name = "Jack Daniels"; … end;
aPerson.Introduce;
>>> My name is Jack Daniels.
let aStudent = ext aPerson to Student …
                  let Faculty = "Math"; … end;
aPerson.Introduce;
>>> My name is Jack Daniels. I am a Math student.
(aPerson as Person)!Introduce;
>>> My name is Jack Daniels.
let anEmployee = ext aPerson to Employee;
aPerson.Introduce;
>>> My name is Jack Daniels. I am an employee.
```

Ein Aufruf `Introduce` für `aPerson` wird also immer an den zuletzt aquirierten Subtyp weitergegeben, falls dieser eine eigene Implementation der Methode besitzt. Dieser Mechanismus zur Auflösung von Methodenaufrufen kann mitunter überraschende Resultate ergeben. Ein statisches Binden erfolgt nur bei einem expliziten „type cast“ mittels `as`, wie z.B. `(aPerson as Person)!Introduce`. Es ist fraglich, ob spätes Binden um jeden Preis tatsächlich erwünscht ist. Bei einem Aufruf `aPerson.Introduce` wird wohl eher erwartet, daß sich die Methodenresolution nach dem Typ der Variable `aPerson` richtet. Falls kein eindeutiger minimaler Typ vorliegt, so halten wir Programme, in denen mittels expliziter „type casts“ auf einen Subtyp die gewünschte Methodenimplementation „angesteuert“ wird, für übersichtlicher, wenn dies auch mit längerem Code verbunden ist. In ESCHER$^+$ bleibt das späte Binden daher für den Methodenaufruf bei varianten Typen vorbehalten. Der Vollständigkeit halber sei erwähnt, daß in Fibonacci Rollen mittels `drop` … `from` … explizit abgegeben werden müssen.

Richardson und Schwarz schlagen in [RS91] ein Datenmodell vor, in dem ein Objekt mehrere *Aspekte* haben kann. Es wird zwischen abstrakten Typen (gegeben durch eine Methodenschnittstelle) und Implementationen (bestehend aus privaten Attributen und öffentlichen Methoden) unterschieden. Ein bezüglich einer Implementation erzeugtes Objekt gehört zu einem abstrakten Typ, wenn es mindestens über die Methoden des Typs verfügt (Typzugehörigkeit durch *Konformanz*). Ein Aspekt ist eine Implementation, die die Implementation eines Objektes eines bestimmten Typs erweitert, so daß dieses dann zu einem weiteren Typ gehört. Das Modell unterstützt jedoch keine Vererbung, und alle Methoden des „Supertyps“ müssen in einer Aspekt-Definition explizit genannt werden. In [RS91] gibt es keinen Test, ob ein Objekt einen bestimmten Aspekt besitzt oder nicht, auch fehlt eine Operation zum Aspekt-„cast“.

Stein und Zdonik stellen in [SZ89] ein Konzept namens *Clovers* vor. Wie in ESCHER+ wird ein Objekt zu einem bestimmten Typ t erzeugt, und kann später mit der Operation `become` zu jedem Subtyp t' von t erweitert werden, wobei mehrere Subtyperweiterungen unabhängig voneinander stattfinden können. Der Name Clover entstand aus der anschaulichen Vorstellung, daß sich die Subtypen eines Typs T wie ein Kleeblatt um den Typ T herum gruppieren (mehrere minimale Typen!). Ein expliziter Zugriff auf ein Objekt bezüglich eines bestimmten Typs im Sinne eines „type cast“ ist nicht möglich. Vielmehr muß mit Hilfe der Operationen `coerc-` und `coerc+` eine Navigation entlang der Typhierarchie durchgeführt werden, um zum gewünschten Typ zu gelangen. `coerc-` geht dabei zu einem Supertyp über, während `coerc+` zu einem Subtyp wechselt. Es gibt ein Prädikat `has`, das testet, ob ein Objekt einen Typ besitzt oder nicht, womit das Risiko von Laufzeitfehlern bei `coerce+` reduziert werden kann.

Von Pernici wird in [Per90] ein Rollenkonzept vorgestellt, das sich für die konzeptuelle Modellierung eignet. Jedem Typ (in [Per90] Klasse genannt) wird eine Menge von Rollentypen zugeordnet. Jeder Rollentyp wird beschrieben durch eine Menge von Attributen, Methoden, Zuständen und Regeln. Die Zustände in einem Rollentyp können als besonders ausgezeichnetes Attribut mit einem Aufzählungstyp aufgefaßt werden (z.B. wäre {*on*, *off*} eine denkbare Menge von Zuständen). Die Regeln spielen in diesem Datenmodell eine zentrale Rolle. Sie geben statische und dynamische Integritätsbedingungen an. Jeder Typ besitzt eine Basisrolle, die die globalen Charakteristika eines Objektes bestimmt. Die Rollentypen stehen in keiner *isa*-Beziehung zueinander, vielmehr wird über die Regeln die Gültigkeit der Hinzunahme oder Abgabe von Rollen kontrolliert. Besonders ist bei diesem Ansatz, daß ein Objekt ein und dieselbe Rolle mehrfach besitzen kann. Aus diesem Grund müssen einem Objekt interne Rollenidentifikatoren zugeordnet werden, damit bei Rollenmultiplizität auf eine bestimmte Rolleninstanz zugegriffen werden kann. Pernici läßt in [Per92] Beziehungen (im Sinne von Assoziationen) zu anderen Objekten vollständig außer acht. Es sei jedoch daran erinnert, daß der Begriff „Rolle“ auch im Zusammenhang mit Beziehungen verwendet wird. Dort bezeichnet Rolle den Namen einer Komponente einer Beziehung. Tatsächlich lassen sich die Beispiele für Rollenmultiplizität aus [Per90] auf mehrfaches Auftreten eines Objektes unter einer Rolle in einer Beziehung zurückführen.

Kapitel 5

Integritätsbedingungen und Konsistenzerhaltung

Zu Beginn der Übersicht in Kapitel 2 wurde neben der strukturellen und der operationalen Komponente eines Datenmodells die sog. Integritätskomponente genannt, die Bestandteil jedes Datenmodells sein sollte (vgl. auch [Bro84, Nav92, Dat95]). Nachdem für ESCHER$^+$ in den vorangegangenen Kapiteln der strukturelle und operationale Teil des Datenmodells ausführlich beschrieben wurden, wenden wir uns in diesem Kapitel der Spezifikation von Integritätsbedingungen in einem Datenbankschema, deren Überwachung sowie weiteren Fragen der Konsistenzerhaltung zu.

Gerade im Zusammenhang mit persistenten Daten hat die Erhaltung der Integrität bzw. Konsistenz[1] eine besondere Bedeutung, da diese Daten über einen längeren Zeitraum hinweg ein kongruentes Abbild eines Umweltausschnittes bieten sollen. Inkonsistenzen können nicht toleriert werden, da dies zum einen zu falschen Annahmen über den Zustands des „realen“ Umweltausschnittes führen kann, zum anderen sich Inkonsistenzen wechselseitig so „hochschaukeln“ können, daß der Datenbestand schlimmstenfalls unbrauchbar wird und dies somit seinem Verlust gleichkommt.

Eines unserer Ziele ist die deklarative Formulierung von Integritätsbedingungen in einem Datenbankschema und deren automatische Überprüfung zur Laufzeit sein. Das explizite „Ausprogrammieren“ von Konsistenzprüfungen in Anwendungsprogrammen ist unsicher, sofern nicht die Korrektheit des jeweiligen Programmstücks hinsichtlich der Erhaltung der Konsisten-

1. Wir verwenden die Begriffe „Integrität“ und „Konsistenz“ als Synonyme.

zerhaltung bei seiner Ausführung verifiziert werden kann, und bedeutet nicht zuletzt oft auch eine Überlastung des Anwendungsprogrammierers.

Eng verbunden mit der Erhaltung der Konsistenz ist das Konzept der Transaktion, das in den operationalen Teil eines jeden Datenmodells, das Integritätsbedingungen unterstützt, integriert sein sollte. Transaktionen erlauben die temporäre Verletzung von Integritätsbedingungen, sichern jedoch gleichzeitig zu, daß beim Abschluß (Commit) der Transaktion noch bestehende Integritätsverletzungen erkannt werden und in diesem Falle alle im Verlauf der Transaktion gemachten Modifikationen auf der Datenbank zurückgesetzt werden. In Abschnitt 5.1 werden wir das formale Datenmodell um flache Transaktionen erweitern.

In der Übersicht zu Integritätsbedingungen in Abschnitt 2.1.2 wurde angesprochen, daß prinzipiell ein breites Spektrum von Integritätsbedingungen in deklarativer Form angegeben werden kann, wozu neben statischen auch dynamische (transitionale oder temporale) Bedingungen gehören. Allerdings ist in vielen Fällen ihre automatische Überwachung in DBMS-Implementationen heute noch unrealistisch. Für ESCHER$^+$ beschränken wir uns daher auf wichtige Klassen statischer Integritätsbedingungen, die in Abschnitt 5.3 eingeführt werden.

In Abschnitt 5.4 skizzieren wir die Überwachung der Integritätsbedingungen auf der Basis interner ECA-Regeln (vgl. Abschnitt 2.4.3.2). Es wird aufgezeigt, daß im Rahmen von ESCHER$^+$ die automatische Überwachung selbst sehr einfach erscheinender Integritätsbedingungen eine nichttriviale Aufgabe sein kann. Wir diskutieren die dabei auftretenden Probleme, wobei deutlich wird, daß für die effiziente Überprüfung vieler Integritätsbedingungen auf interner Ebene weitere Hilfsstrukturen (Indexe) notwendig sind.

In Abschnitt 5.5 geht es um die in den vorangegangenen Kapiteln bereits angekündigte selektive Einkapselung von Attributen eines Objekttyps und von Operationen. Es besteht die Möglichkeit, Teile eines Objektzustandes vollständig einzukapseln oder aber das direkte Schreibrecht auf diese Teile zu entziehen. Auf diese Weise können konsistenzgefährdende direkte Modifikationen verhindert werden. Bei Operationen wird zwischen privaten und öffentlichen unterschieden. Nur solche Operationen sollten als öffentlich gelten, für die nachgewiesen werden kann, daß sie immer konsistenzerhaltend sind.

In Abschnitt 5.6 gehen wir auf Konsistenzbedingungen für Datenbankschemata ein. Da jedes Datenbankschema als Instanz des Meta-Schemas (siehe Abschnitt 3.10) aus „gewöhnlichen“ Datenobjekten besteht, darf es auch nur innerhalb einer Transaktion modifiziert werden. Im Kontext der Definition bzw. Modifikation von Datenbankschemata sprechen wir dann von DDL-Transaktionen. Für Objekttypdefinitionen geben wir zusätzliche Konsistenzbedingungen an, die alle denkbaren Objekttypdefinitionen auf die „sinnvollen“ einschränken.

5.1 Transaktionen

Bei der Sicherung der Konsistenz einer Datenbankinstanz spielen Transaktionen eine Schlüsselrolle. Auf die Notwendigkeit der Integration eines Transaktionskonzeptes in das Datenmodell wurde bereits in Abschnitt 4.1 hingewiesen. Die grundlegenden Prinzipien von Transaktionen sollen kurz genannt werden:

- Transaktionen sind Folgen von Anweisungen, die auf einer Datenbankinstanz operieren
- Jeder Zugriff auf ein persistentes Datenobjekt muß innerhalb einer Transaktion geschehen
- Unmittelbar zu Beginn und unmittelbar nach dem Abschluß einer Transaktion befindet sich die Datenbankinstanz in einem konsistenten Zustand
- Während der Ausführung einer Transaktion darf die Konsistenz der Datenbankinstanz verletzt werden
- Eine Transaktion wird entweder komplett oder gar nicht ausgeführt, d.h. eine aktive Transaktion kann abgebrochen und alle in ihr gemachten Änderungen zurückgesetzt werden (Atomizität einer Transaktion)

Von den für Transaktionen charakteristischen ACID-Eigenschaften interessieren wir uns in dieser Arbeit nur für das „A“ (*atomicity*) und das „C“ (*consistency*). Die Rolle von Transaktionen im Mehrbenutzerbetrieb („I“ wie *isolation*) und im Zusammenhang mit Recovery-Mechanismen („D“ wie *durability*) sind hier nicht weiter von Interesse[2].

Im ESCHER-Prototyp sowie in der für ESCHER entworfenen persistenten Programmiersprache SCRIPT [Pau94] ist kein Transaktionskonzept vorgesehen. Aus diesem Grunde ist es auch nicht verwunderlich, daß im Zusammenhang mit ESCHER die Definition und Überwachung deklarativer Integritätsbedingungen bisher nicht möglich war. Wollte man für ESCHER$^+$ weiterhin auf explizite Transaktionen verzichten, gleichzeitig jedoch die Einhaltung von Integritätsbedingungen fordern, dann müßte jede einzelne SCRIPT$^+$-Anweisung als Transaktion betrachtet werden, deren Wirkung rückgängig gemacht wird, sofern ihre Ausführung zu einer Konsistenzverletzung führt. Dann sind jedoch keine temporären Verletzungen über mehrere Anweisungen hinweg möglich. Zudem erhöht sich der Aufwand zur Integritätssicherung aufgrund ständig notwendiger Überprüfungen.

Für ESCHER$^+$ greifen wir auf das einfache Konzept der flachen Transaktion zurück, da dies für die Belange dieser Arbeit ausreichend ist. Beim flachen Transaktionsmodell gibt es höchstens eine aktive Transaktion, d.h. Transaktionen dürfen nicht ineinander geschachtelt werden.

Das flache Transaktionsmodell soll nun in den formalen Rahmen von ESCHER$^+$ eingefügt werden. Dazu erweitern wie die Definition des Laufzeitzustandes. Nach Def. 4.1 ist ein Laufzustand definiert als ein 4-Tupel (INST(DS), D_{trans}, L, R). Dabei ist INST(DS) eine Datenbankinstanz zum Datenbankschema DS, D_{trans} eine Menge transienter Datenobjekte, L der Laufzeitstack, und R enthält das Resultat des letzten Funktionsaufrufs. Wir erweitern nun den Laufzeitzustand um eine weitere Komponente BI („*b*efore *i*mage“). Sie wird unmittelbar zu Beginn einer Transaktion mit einer Kopie der aktuellen Datenbankinstanz INST(DS) besetzt. Alle Operationen und Anfragen beziehen sich weiterhin auf die erste Komponente des Laufzeitzustandes, also auf INST(DS). Soll eine Transaktion zurückgesetzt werden (d.h. es wird ein Rollback durchgeführt), dann wird die erste Komponente des Laufzeitzustandes einfach wieder durch BI ersetzt. Das Rollback bezieht sich also wie üblich nur auf die persistenten Datenobjekte. Natürlich ist für eine effiziente Implementation das Kopieren der gesamten Datenbankinstanz nach BI ein völlig unrealistisches Vorgehen. In einer Implementation muß ein Recovery-Mechanismus dafür sorgen, daß während einer Transaktion lediglich die „Delta-Information“ mitprotokolliert wird, die sich aus der Ausführung von Update-Operationen

2. Umfassendere Information zu Transaktionen sind etwa [GR93] oder auch dem Übersichtsartikel [Özs94] zu entnehmen.

ergibt und die notwendig ist, um beim Zurücksetzen einer Transaktion zum ursprünglichen Zustand der Datenbankinstanz zurückzukehren. Für das formale Modell von ESCHER$^+$ ist es jedoch ausreichend, die soeben geschilderte, vereinfachte Sichtweise zugrundezulegen. Falls sich zu einem Zeitpunkt keine Transaktion in der Abarbeitung befindet, dann setzen wir BI = *null*.

Für den Fall eines Aborts der aktuellen Transaktion benötigen wir für das korrekte Zurücksetzen des Laufzeitstacks L die Information über die Stacktiefe von L beim Start der Transaktion. Dies ist gerade die aktuelle Länge der Liste L beim Start einer Transaktion, und sie wird in einer weiteren zusätzlichen Komponente TSI (*t*ransaction *s*tart *i*ndex) des Laufzeitzustandes abgelegt. Ist keine Transaktion aktiv, soll TSI den Wert *null* haben.

In Tabelle 5.1 sind die drei zusätzlichen Basisoperationen aufgeführt, die im Zusammenhang mit Transaktionen in ESCHER$^+$ notwendig sind. Alle drei Basisoperationen haben die Signatur `"-> void"`.

name	*semantics*

`beginT`	if BI ≠ *null* then
	  Ausnahme `transaction_already_active` auslösen
	else begin
	  BI := Kopie von INST(DS);
	  TSI := | L |
	end;

`commit`	if BI = *null* then
	  Ausnahme `no_active_transaction` auslösen
	else if TSI ≠ | L | then
	  Ausnahme `commit_not_allowed` auslösen
	else begin
	  ok := Test auf Integrität;
	  if ok then begin
	  BI := *null*;
	  TSI := *null*
	  end
	end;

`abort`	if BI = *null* then
	  Ausnahme `no_transaction_pending` auslösen
	else begin
	  Zurücksetzen des Laufzeitstacks L;
	  TSI := *null*;
	  INST(DS) := BI;
	  BI := *null*;
	  Sprung zur Anweisung, die auf die
	  nächste `commit`-Anweisung folgt
	end;

Tabelle 5.1: Basisoperationen für Transaktionen

Bemerkungen zur Semantik von `beginT`, `abort` und `commit`:

- Mit `beginT()` wird eine neue Transaktion gestartet.
- Die Operation `commit()` darf nur innerhalb desjenigen Basisscriptes aufgerufen werden, in dem auch das zugehörige `beginT()` steht.
- Bei der Abarbeitung von `commit()` wird getestet, ob Integritätsverletzungen vorliegen. Ist dies nicht der Fall, dann wird die Transaktion abgeschlossen, was in unserem formalen Modell einfach durch Nullsetzen von BI und TSI erreicht wird. Andernfalls findet – noch als Bestandteil des Tests auf Integrität – ein Aufruf von `abort()` statt, was zum Rollback der aktuellen Transaktion führt.
- Über einen Aufruf von `abort()` kann jederzeit ein explizites Rollback durchgeführt und die gerade aktive Transaktion abgebrochen werden.
  Das in der Semantik von `abort` erwähnte Zurücksetzen des Laufzeitstacks L entspricht genau dem (|L| − TSI)-fachen Ausführen von `return`-Anweisungen (vgl. Abschnitt 4.6.4).

In der Syntax von SCRIPT$^+$ ist eine Transaktion eine Anweisung, die nach der Syntaxregel

transaction ::= `transaction begin` *statements* `end transaction ';'`

gebildet wird. Eine SCRIPT$^+$-Transaktion wird wie folgt nach BASESCRIPT übersetzt:

```
beginT();
Übersetzung von statements
commit();
```

Die Basisoperation `abort` hat in der Syntax von SCRIPT$^+$ ein gleichnamiges Gegenstück, während dies für `commit` nicht der Fall ist. Eine Transaktion soll tatsächlich als eine unzerlegbare operationale Einheit betrachtet werden. Es gibt daher kein „Teil“-Commit, d.h. die Ausführung eines `transaction begin … end transaction`-Blockes kann nicht in „goto“-Manier durch eine `commit`-Anweisung „abgekürzt“ werden.

5.2 Strukturpfadausdrücke

Auch in diesem Kapitel halten wir an Baumrepräsentationen als Beschreibungsgrundlage für alle Ausführungen zum ESCHER$^+$-Datenmodell fest. Es seien also insbesondere alle Strukturen durch Strukturbäume repräsentiert, d.h. die Werte der Funktionen *struct* aus (3.21) und (3.31) für Objekttypen und Tabellen seien die Wurzelknoten von Strukturbäumen $B_{struct(t)}$. Ferner sei für jeden varianten Typ t ∈ VARTYPES ein eindeutig bestimmter Variantenbaum $B_{variants(t)}$ gegeben.

Im Zusammenhang mit Integritätsbedingungen benötigen wir eine Syntax zur textuellen Bezeichnung aller Knoten in einem Struktur- oder Variantenbaum. Dazu verwenden wir sog. *Strukturpfadausdrücke*, die Wörter aus $\mathcal{L}$(*structPathExpr*) sind. Dabei gelten folgende Syntaxregeln:

structPathExpr ::= *identifier path*

path ::= {*step* }

step ::= (`'.#'` | `'.'` *attrName* | `':'` *label*)

Ein Strukturpfadausdruck beginnt mit dem Namen einer Tabelle, eines Objekttyps oder eines varianten Typs und wird mit einem *Strukturpfad* p ∈ $\mathcal{L}$(*path*) fortgesetzt, der den „Weg“ zu einem bestimmten Knoten in einem Struktur- oder Variantenbaum beschreibt. Der Strukturpfad kann auch leer sein. Je nach Art des „Startpunktes“ eines Strukturpfadausdruckes sprechen wir dann von *Tabellen-*, *Objekttyp-* bzw. *Variantenpfadausdrücken.*

Die Semantik eines Strukturpfadausdruckes s = np ∈ $\mathcal{L}$(*structPathExpr*) mit p ∈ $\mathcal{L}$(*path*) ist gegeben durch eine Funktion μ_{struct}: $\mathcal{L}$(*structPathExpr*) ···> $\mathcal{V}$, die einen Knoten in einem Strukturbaum liefert. Der Wert μ_{struct} (s) ergibt sich nach folgenden Regeln:

Regel 1 – Direkter Zugriff über einen Namen („Auswahl“ des Startpunktes):

Für s = n ∈ {*name*(t) | t ∈ TABLES} ∪ {*name*(t) | t ∈ DEFTYPES} sei
t ∈ TABLES ∪ DEFTYPES derart, daß *name*(t) = n
⇒ μ_{struct} (s) = *root*($B_{struct(t)}$),
wobei $B_{struct(t)}$ der eindeutig bestimmte Strukturbaum zu t ist.

Für s = n ∈ {*name*(t) | t ∈ VARTYPES} sei t ∈ VARTYPES derart, daß *name*(t) = n
⇒ μ_{struct} (s) = *root*($B_{variants(t)}$),
wobei $B_{variants(t)}$ der eindeutig bestimmte Variantenbaum zu t ist.

Regel 2 – Zugriff auf die Substruktur einer Kollektion:

Mit Hilfe eines **.#**-Schrittes wird von einem Kollektionsknoten zu seinem Elementknoten übergegangen.

Es sei s = s'**.#**, mit s' ∈ $\mathcal{L}$ (*structPathExpr*), μ_{struct} (s') sei definiert,
ferner gelte *isCollection*(*type*(μ_{struct} (s')))
⇒ μ_{struct} (s) := *get_child* (μ_{struct} (s'), 1).

Regel 3 – Zugriff auf eine Tupelkomponente:

Es sei s = s'**.** a, mit s' ∈ $\mathcal{L}$ (*structPathExpr*), a ∈ $\mathcal{L}$ (*attrName*), μ_{struct} (s') sei definiert,
ferner gelte *type* (μ_{struct} (s'))) = c_{tuple}, a ∈ Def(*attr_pos* (μ_{struct} (s')))
⇒ μ_{struct} (s) := *get_child* (μ_{struct} (s'), k) mit k = *attr_pos*(μ_{struct} (s'))(a)

Regel 4 – Zugriff auf eine Alternative eines varianten Typs:

Es sei s = n**:** *l*, mit n ∈ {*name*(t) | t ∈ VARTYPES},
l ∈ Def(*label_pos*(μ_{struct} (n)))
⇒ μ_{struct} (s) := *get_child* (μ_{struct} (n), k) mit k = *label_pos* (μ_{struct} (n))(*l*).

Kann für s ∈ $\mathcal{L}$(*structPathExpr*) die Bestimmung von μ_{struct}(s) mit Hilfe obiger Regeln nicht erfolgreich durchgeführt werden, dann handelt es sich um einen unzulässigen Strukturpfadausdruck.

Man beachte, daß keine eigene Regel für den Zugriff auf ein Attribut eines Objekttyps benötigt wird: Ist n der Name eines Objekttyps t ∈ DEFTYPES, dann ist bereits μ_{struct}(n) eine Tupelstruktur, und für den Übergang zu einem Attribut ist Regel 3 anwendbar.

Es sei `T` eine Tabelle mit der in Abb. 3.3 angegebenen Struktur. In Tabelle 5.2 sind alle Strukturpfadausdrücke s, die ausgehend von `T` gebildet werden können, und ihre Semantik $\mu_{struct}(s)$ angegeben (Die s_i sind die Knoten-IDs aus Abb. 3.3). ❒

s	$\mu_{struct}(s)$
`T`	s_0
`T.#`	s_1
`T.#.Name`	s_2
`T.#.Zeugnis`	s_3
`T.#.Zeugnis.#`	s_4
`T.#.Zeugnis.#.Fach`	s_5
`T.#.Zeugnis.#.Note`	s_6

Tabelle 5.2: Beispiel für Strukturpfadausdrücke

Bemerkung: Die Syntax für Strukturpfadausdrücke läßt sich ohne weiteres modifizieren, so daß die explizite Nennung von `.#`-Schritten im Inneren eines Strukturpfadausdruckes überflüssig wird. Beispielsweise kann man den Strukturausdruck `T.#.Zeugnis.#.Fach` zu `T.Zeugnis.Fach` vereinfachen. Bei der Auswertung eines solchen Ausdruckes kann ohne weiteres erkannt werden, daß man z.B. von `T` ausgehend nicht den Schritt `.Zeugnis` durchführen kann, ohne einen impliziten `.#`-Schritt einzuschieben. Ein `.#`-Schritt muß allerdings explizit genannt werden, wenn er der letzte Schritt in einem Strukturpfadausdruck ist (z.B. in `T.#`). Die explizite Aufführung aller `.#`-Schritte hat den Vorteil, daß die oben angegebenen, sehr einfachen Regeln ausreichen, um einen Strukturpfadausdruck auszuwerten. Im folgenden werden wir Strukturpfadausdrücke aber auch ohne innere `.#`-Schritte angeben.

5.3 Integritätsbedingungen

Für ESCHER$^+$ sollen eine Reihe von Integritätsbedingungen eingeführt werden, die zur Laufzeit automatisch überprüft werden. Dazu gehören:

- Schlüsselbedingungen
- Inklusionsbedingungen
- Disjunktheitsbedingungen
- lokale Wertebedingungen

Alle genannten Bedingungen sind statische Integritätsbedingungen. In ESCHER$^+$ können Integritätsbedingungen mit der Definition einer Tabelle, eines Objekttyps oder eines varianten Typs verknüpft werden. Dazu wird die jeweilige `define`-Anweisung der DDL (vgl. Anhang B.2) um eine `constraint`-Klausel ergänzt. Diese hat die Syntax

constraint ::= `-constraint` *constrDef* { `,` *constrDef* }

constrDef ::= [*identifier*] `(` *condition* `)` [*constrMode*]

constrMode ::= `hard` | `soft`

Dabei ist *identifier* der Name der Integritätsbedingung. Dieser wird benötigt, um zu einem späteren Zeitpunkt eine Integritätsbedingung wieder löschen zu können. Zum Löschen steht die `drop constraint`-Anweisung zur Verfügung (siehe ebenfalls Anhang B.2). Wird kein Name angegeben, dann erhält die Integritätsbedingung einen systemgenerierten Namen[3]. Das Nichtterminal *condition* steht für die „eigentliche“ Integritätsbedingung. Schließlich entscheidet der Wert des Nichtterminals *constrMode* (für *constraint mode*) über den Zeitpunkt der Überprüfung der Integritätsbedingung:

- Der Modus `hard` fordert die sofortige Überprüfung der Integritätsbedingung, sobald eine Operation ausgeführt werden soll oder ausgeführt wurde, die diese Bedingung möglicherweise verletzen wird oder verletzt hat.
- Der Modus `soft` verschiebt die Überprüfung einer möglicherweise verletzten Integritätsbedingung auf den Commit-Zeitpunkt der Transaktion.

Der Default-Wert soll `soft` sein. Eine grundsätzliche Sofortprüfung aller Integritätsbedingungen wäre in vielen Fällen zu „streng“, da es insbesondere bei komplexeren Transaktionen notwendig werden kann, temporäre Konsistenzverletzungen zuzulassen.

Auch der SQL/92-Standard unterscheidet zwischen sofortiger und zurückgestellter Prüfung. Dort werden Integritätsbedingungen nicht mit `hard` und `soft`, sondern mit `immediate` und `deferred` attributiert.

Analog zu den bisherigen Komponenten eines Datenbankschemas werden alle Integritätsbedingungen als abstrakte Objekte c_i in einer Menge CONSTRAINTS zusammengefaßt.

Definition 5.1 (Integritätsbedingungen)

Zu jedem ESCHER$^+$-Datenbankschema gehört eine Menge CONSTRAINTS = $\{c_1, \ldots, c_n\}$, $n \in IN_0$. Die Elemente von CONSTRAINTS heißen *Integritätsbedingungen* sind.
Auf CONSTRAINTS sind die folgende Funktionen definiert:

name : CONSTRAINTS $\rightarrow \mathcal{N}$

condition : CONSTRAINTS $\rightarrow \mathcal{L}$(*condition*)

mode : CONSTRAINTS $\rightarrow$ {*hard*, *soft*}

Die Funktion *name* soll injektiv sein. Die Werte dieser Funktionen ergeben sich aus den Angaben einer `constraint`-Klausel in einer DDL-Anweisung. ❐

Beispiel 5.1

In Abschnitt 5.3.1 führen wir als wohl wichtigste Klasse von Integritätsbedingungen die sog. Schlüsselbedingungen ein, die spezielle funktionale Abhängigkeiten darstellen. Wir betrachten nun eine Tabelle `T` mit der Struktur

3. In diesem Fall muß es natürlich möglich sein, den Namen einer Integritätsbedingung über einen Schema-Browser oder eine Schema-Anfragesprache zu ermitteln.

```
{[  A:string,
    B:  {[  C: integer,
            D: {[E: string, F: integer, G: integer]}
        ]}
]}
```

Für die Tabelle T wird z.B. mittels

```
-constraint IB1 (T.B.D has key E)
```
(5.1)

eine gültige Schlüsselbedingung definiert. Sie hat den Namen IB1, und ihr Modus ist soft, d.h. sie wird erst am Ende der Transaktion überprüft.

Der Strukturpfadausdruck T.B.D (bzw. T.#.B.#.D in seiner Langform) bezeichnet einen Knoten $s = \mu_{struct}(T.\#.B.\#.D)$ im Strukturbaum der Tabelle T, der für die Semantik der Schlüsselbedingung eine wichtige Rolle spielt. Die Semantik von (5.1) ist wie folgt gegeben:

Ist r ein beliebiger Knoten im Wertebaum der Tabelle T, für den *struct*(t) = s ist, dann muß die Bedingung (in SCRIPT$^+$-Syntax)

```
card(D) = card({ x.E | x in D: x.E <> null})
```
(5.2)

erfüllt sein, wenn man für D den Knoten r einsetzt. Die Prüfbedingung der Schlüsselbedingung ist hier als Vergleich zweier Kardinalitäten angegeben. Ist die Schlüsselbedingung verletzt, dann liefert der Ausdruck auf der rechten Seite eine kleinere Kardinalität als der Ausdruck auf der linken Seite. Mit der Schlüsselbedingung ist gleichzeitig eine *not-null*-Bedingung für das Schlüsselattribut E verknüpft, denn (5.2) ist auch dann nicht erfüllt,wenn für ein Elementtupel in r die Komponente E nullwertig ist. ❒

Es sei ausdrücklich darauf hingewiesen, daß die Bedingung (5.2), der aufwendige Vergleich zweier Kardinalitäten, nicht notwendigerweise die zur Laufzeit tatsächlich zu überprüfende Bedingung ist. Vielmehr sollte sich die zu überprüfende Bedingung nach der jeweiligen Update-Operation richten, die eine potentielle Integritätsverletzung hervorruft: Wird beispielsweise in D ein neues Element y eingefügt, dann ist bei sofortiger Prüfung die Suche eines Elementes x in D, das auf den Schlüsselkomponenten mit y übereinstimmt, vor dem Einfügen i.d.R. weniger aufwendig als der Kardinalitätsvergleich (5.2) nach dem Einfügen. Es muß deshalb unterschieden werden zwischen den Bedingungen, die für die formale Semantik von Integritätsbedingungen angegeben werden, und den tatsächlich zur Laufzeit eingesetzten Prüfbedingungen unterschieden werden. Zu letzteren werden wir in Abschnitt 5.4 eine Reihe von Beispielen angeben.

Notation: Ist B eine Prüfbedingung (in SCRIPT$^+$-Syntax), in der die freien Variablen $x_1, \ldots, x_n$ vorkommen[4], dann sei

$$\sigma(B; x_1 \leftarrow r_1, \ldots, x_n \leftarrow r_n) \in \{true, false\}$$

die Wertesemantik (bzw. der Wert) der Bedingung, wenn man zu ihrer Auswertung die Variablen x_i an die Knoten r_i bindet. Diese Notation ist vergleichbar mit der Funktion σ aus (4.12) bzw. (4.13), die die Wertesemantik für BASESCRIPT- bzw. SCRIPT$^+$-Ausdrücke angibt. Dort ist es nicht nötig, die Bindungen $x_1 \leftarrow r_1, \ldots, x_n \leftarrow r_n$ explizit anzugeben, da sie sich aus der aktuellen Bindungsfunktion α des Laufzeitzustandes ergeben. Streng genommen müßten wir jeder

4. Wir müßten statt B also genauer $B(x_1, \ldots, x_n)$ schreiben.

Variablen x_i eine Struktur s_i zuordnen und bei der Auswertung für gegebene Knoten r_i testen, ob $struct(r_i) = s_i$ gilt. Darauf wollen wir jedoch der Einfachheit halber verzichten.

In Beispiel 5.1 ist also die Integritätsbedingung `IB1` für r erfüllt, falls $\sigma(B; D \leftarrow r) = true$ ist, wobei B die Bedingung aus (5.2) sei. Man beachte, daß die Integritätsbedingung aus (5.1) direkt in einen quantorisierten Ausdruck in SCRIPT$^+$-Syntax umgesetzt werden kann. Wir erhalten:

```
all x1 in T, all x2 in x1.B:
card(x2.D) = card({ y.E | y in x2.D: y.E <> null})        (5.3)
```

Man erkennt, daß für jeden `.#`-Schritt im Strukturpfadausdruck `T.#.B.#.D` eine an einen Allquantor gebundene Variable x_i eingeführt wird. Abgesehen von der Tatsache, daß die Angabe der Integritätsbedingung aus (5.1) in der Form (5.3) sehr aufwendig ist, ist die SCRIPT$^+$-Syntax für quantorisierte Ausdrücke nicht ausdrucksstark genug, um einige wünschenswerte Integritätsbedingungen auszudrücken. So scheitert die Syntax der quantorisierten booleschen Ausdrücke bei der Formulierung einer Integritätsbedingung, die für alle Instanzen des Objekttyps `Person` fordert, daß der Nachname einer Person nicht der leere String sein darf:

```
all p in Person: p.Name.NName <> ""
```

ist kein gültiger SCRIPT$^+$-Ausdruck, da `Person` keine Tabelle ist! `Person` soll hier die Menge aller persistenten Instanzen des Typs (die Typextension) bezeichnen, auf die von SCRIPT$^+$ aus jedoch nicht zugegriffen werden kann. Würde man `Person` durch den Tabellennamen `Personen` ersetzen, dann gälte die angegebene Bedingung nur für Instanzen des Typs `Person`, die in dieser Tabelle enthalten sind, nicht jedoch für andere persistente Objekte des Typs `Person`.

Es bedarf also einer speziellen Syntax, um Integritätsbedingungen angeben zu können, die über die Ausdrucksmittel von SCRIPT$^+$-Bedingungen hinausgehen. In den folgenden Abschnitten werden die Syntax und Semantik der weiter oben genannten Integritätsbedingungen im einzelnen besprochen. Wie bereits in Beispiel 5.1 deutlich wurde, sind dabei die in Abschnitt 5.2 eingeführten Strukturpfadausdrücke äußerst hilfreich, um Integritätsbedingungen in kompakter Weise formulieren zu können.

5.3.1 Schlüsselbedingungen

Schlüsselbedingungen sind für die Verwaltung großer Bestände gleichartiger Daten und dabei insbesondere zur eindeutigen Identifikation von Anwendungsobjekten durch den Benutzer vorgegebene Attributwerte unerläßlich. In ESCHER$^+$ gibt es verschiedene syntaktische Varianten der Definition von Schlüsselbedingungen, die in unterschiedlichen Situationen relevant werden.

Variante 1: Schlüsselbedingungen für Kollektionen

Für jede Kollektion, deren Elemente Tupel, Instanzen eines Objekttyps oder einer Objekttypvariante sind, können Schlüsselbedingungen festgelegt werden, die für die Eindeutigkeit der Elemente hinsichtlich der in der Bedingung genannten Attribute sorgen.

Die in Beispiel 5.1 angegebene Integritätsbedingung gehört zu dieser ersten Variante. Für das Nichtterminal *condition*, das durch den Bedingungstext ersetzt wird, gilt folgende Syntaxregel:

condition ::= *structPathExpr* `has key` *key*

Ein Schlüssel *key* wird durch einen einzelnen *Schlüsselpfad* oder eine Menge von Schlüsselpfaden bestimmt, die im letzteren Fall zwischen '`[`' und '`]`' aufgezählt werden und zu den einzelnen *Schlüsselattributen* führen.

Für das Nichtterminal *key* gelten genauer folgende Syntaxregeln:

key ::= *keyPath* | '`[`' *keyPath* { '`,`' *keyPath* } '`]`'

keyPath ::= *attrName* { '`.`' *attrName* }

Indem wir nicht nur einfache Attributnamen, sondern allgemeiner sog. Schlüsselpfade zulassen, berücksichtigen wir die bereits im eNF^2-Datenmodell gegebene Möglichkeit zur Ineinanderschachtelung von Tupelstrukturen, wodurch Attribute zu semantisch zusammengehörenden Einheiten aggregiert werden können.

Es folgen weitere Beispiele zu Schlüsselbedingungen.

Beispiel 5.2

a) Für die Tabelle `T` aus Beispiel 5.1 sind neben der in (5.1) genannten Bedingung auch folgende Schlüsselbedingungen möglich:
`T has key A` und `T.B has key C` sowie `T.B.D has key[E,F]`.

b) Ist `Person` der Objekttyp aus Beispiel 3.7 sowie eine Tabelle `Personen` mit der Struktur `{Person}`gegeben, dann drückt die Schlüsselbedingung

```
Personen has key [Name.NName, Adresse.Ort]
```

aus, daß in ihr nicht zwei verschiedene´ Personen enthalten sind, die den gleichen Nachnamen haben und am gleichen Ort wohnen. Die Schlüsselbedingung wäre auch dann in derselben Form zulässig, wenn die Elemente von `Personen` Tupel mit entsprechendem Aufbau wären.

c) Es sei `Fahrzeuge` eine Tabelle, die als Menge von Instanzen der Objekttypvariante `Verkehrsmittel` aus Beispiel 3.1 definiert ist. `Verkehrsmittel` verlange in der `struct needed`-Klausel ein string-wertiges Attribut `Name`, d.h. alle Alternativen sollen dieses Attribut besitzen. Dann ist

```
Fahrzeuge has key Name
```

eine gültige Schlüsselbedingung für die Tabelle `Fahrzeuge`. Es gibt dann in `Fahrzeuge` zu einem gegebenen Namen höchstens ein Element, das diesen Namen hat, wobei sich die Eindeutigkeit über alle Alternativen erstreckt. Entscheidend ist, daß der Elementtyp von `Fahrzeuge` eine Objekttypvariante ist und die Schlüsselpfade unabhängig von der für ein Element zutreffenden Alternative Gültigkeit besitzen. Dies ist bei beliebigen varianten Typen i.a. nicht der Fall. ❐

Die **Semantik** einer Schlüsselbedingung "np `has key` k" mit k = [k_1, …, k_r] lautet[5]:

Eine Datenbankinstanz INST(DS) erfüllt die Schlüsselbedingung "np `has key` k" ⇔

$$\forall\, r \in V(D_{pers})\text{: } \mathit{struct}(r) = \mu_{struct}(np) \Rightarrow \sigma(B;\, C \leftarrow r) \quad [6] \qquad (5.4)$$

5. Den Fall "np `has key` k_1" mit $k_1 \in \mathcal{L}$(*keyPath*) führen wir auf "np `has key` k" mit k = [k_1] zurück.

mit der Bedingung B =

```
card(C) = card({[a1:x.k1,a2:x.k2, ...,ar:x.kr] | x in C:
x.k1 <> null and x.k2 <> null and ... and x.kr <> null})
```

Die Bedingung (5.4) liest sich wie folgt: Für alle persistenten Knoten r, deren zugeordneter Strukturknoten durch np bezeichnet wird, gilt die Bedingung B, wenn man den in B vorkommenden Bezeichner `C` an den Knoten r bindet. Aus der zu (5.4) gehörenden Bedingung B geht hervor, daß für die Schlüsselkomponenten grundsätzlich „*not null*“-Bedingungen gelten sollen.

Damit es sich bei "np `has key` k" um eine gültige Schlüsselbedingung handelt, müssen einige Voraussetzungen erfüllt sein, die sicherstellen, daß die oben angegebene Bedingung B korrekt ausgewertet werden kann.

Dazu gehen wir zunächst auf die für eine Schlüsselbedingung *zulässigen Schlüsselpfade* ein. Wie bereits in Beispiel 5.2 b) gezeigt, sind nicht nur Attributnamen als Schlüsselkomponenten zugelassen, sondern auch Schlüsselpfade der Form $k_i = a_{i1}.a_{i2}....a_{im}$. Auf diese Weise berücksichtigen wir die Möglichkeit der Aggregation zusammengehöriger Attribute durch Ineinanderschachtelung von Tupelstrukturen: Für den Objekttyp `Person` aus Beispiel 3.7 werden die Attribute `VName` und `NName` unter der Tupelstruktur `Name` zusammengefaßt. Analoges gilt für die Attribute `PLZ`, `Ort` und `Strasse`, die zusammen eine `Adresse` bilden. Als Schlüsselpfade sind alle diejenigen Pfade $a_1.a_2....a_m$ zugelassen, die – ausgehend von einem Tupelknoten – auf ihrem „Weg“ nur Tupelknoten besuchen und schließlich zu einem Knoten r mit *isSimple*(*type*(r)) führen, d.h. als Schlüssel lassen wir nur „druckbare“ Typen und Objekttypen zu. Diese Pfade nennen wir *einfache Pfade*.

Formal: Ist $s \in \mathcal{L}(structPathExpr)$ ein Strukturpfadausdruck mit $type(\mu_{struct}(s)) = c_{tuple}$, dann sei SIMPLEPATHS(s) die Menge aller Pfade $a_1.a_2....a_m$ mit der Eigenschaft

– $type(\mu_{struct}(s.a_1.a_2....a_j)) = c_{tuple}$ für $0 \le j < m$
(d.h. der Weg führt höchstens über Tupelknoten)

– $isSimple(type(\mu_{struct}(s.a_1.a_2....a_m)))$
(d.h. der Pfad führt zu einem „druckbaren“ Typ oder einem Objekttyp)

Es ist SIMPLEPATHS(`Person`) = {`Name.NName`, `Name.VName`, `Adresse.PLZ`, `Adresse.Ort`, `Adresse.Strasse`}

Nun kommen wir zu den angekündigten Bedingungen für die Gültigkeit einer Schlüsselbedingung "np `has key` k". Es muß gelten:

(i) μ_{struct}(np) ist definiert, und es gilt $isCollection(type(\mu_{struct}(np)))$,
d.h. np führt zu einem Kollektionsknoten.

(ii) $type(\mu_{struct}(np.\#)) \in \{c_{tuple}\} \cup$ DEFTYPES $\cup$ OBJVAR, d.h. der Elementtyp von μ_{struct}(s) ist entweder ein Tupel, ein Objekttyp oder eine Objekttypvariante. Komplexe Schlüsselkomponenten, wie z.B. Mengen, werden nicht zugelassen.

(iii) Falls $type(\mu_{struct}(np.\#)) = c_{tuple}$, dann gilt
$k_i \in$ SIMPLEPATHS(np.#) für $1 \le i \le r$

(iv) Falls $type(\mu_{struct}(np.\#)) = t \in$ DEFTYPES, dann muß k_i ein gültiger Pfad im Strukturbaum des Typs t oder eines seiner Supertypen t' sein (Wir müssen die Vererbung von Attributen berücksichtigen!). Genauer bedeutet dies:

---

6. $V(D_{pers})$ ist die Menge aller Knoten in Wertebäumen aus D_{pers} (vgl. Def. 3.22).

Ist $k_i = a_{i1}.a_{i2} \ldots .a_{in_i}$ und ist $AttrAccess(t)(a_{i1}) = (t', n')$, d.h. ist a_{i1} ein vom Typ t' geerbtes Attribut, dann ist $k_i \in$ SIMPLEPATHS($name(t')$).

(v) Falls $type(\mu_{struct}(np.\#)) = t \in$ OBJTYPES mit $variants(t) = \{(l_1, t_1), \ldots, (l_k, t_k)\}$, dann muß für $1 \le i \le r$ gelten:

– k_i ist ein gültiger Pfad in $struct\_needed(t)$,
d.h. für alle Alternativen t_i ist $\mu_{struct}(name(t_v).k_i)$ definiert

– Für $1 \le v \le k$ muß gelten: Ist $k_i = a_{i1}.a_{i2} \ldots .a_{in_i}$ und ist $AttrAccess(t_v)(a_{i1}) = (t_v', n)$, dann muß $k_i \in$ SIMPLEPATHS($name(t_v')$) sein.

Variante 2: Bedingungen zur Duplikatfreiheit für Listen und Felder

Während man für Mengen per definitionem die Existenz von Duplikaten ausschließt, möchte man für Listen und Felder in vielen Fällen ebenfalls Duplikatfreiheit erzwingen. Dazu eignen sich die Schlüsselbedingungen der Variante 1 nicht immer. Liegt z.B. eine Liste von Instanzen eines Objekttyps vor, dann gibt es bislang keine Möglichkeit, diese Objekte zu Schlüsseln der Liste zu machen. Analoges gilt z.B. auch für eine Liste von integer-Werten. Für diese Situationen führen wir eine weitere Variante für Schlüsselbedingungen ein. Zu ihr gehört die Syntaxregel

condition ::= *structPathExpr* `has unique elements`

Beispiel 5.3 Es sei `Kurs` ein Objekttyp mit dem listenwertigen Attribut `Teilnehmer`, dessen Elemente Objekte vom Typ `Person` sind. Um zu gewährleisten, daß sich ein Teilnehmer nicht mehrfach für einen Kurs anmeldet, wird für `Kurs` die Integritätsbedingung

```
Kurs.Teilnehmer has unique elements
```

formuliert. ❒

Für eine zulässige Schlüsselbedingung der Variante 2 muß gelten:

$type(\mu_{struct}(np)) \in \{c_{list}, c_{array}) \wedge isSimple(type(\mu_{struct}(np)))$.

Für Mengen und Multimengen ist eine solche `has unique elements`-Bedingung nicht zugelassen. Für Mengen wäre sie offensichtlich redundant, und für Multimengen ist ja gerade charakteristisch, daß sie Duplikate enthalten dürfen.

Die **Semantik** einer Schlüsselbedingung "np `has unique elements`" zum Namen n ist wie folgt gegeben:
Eine Datenbankinstanz INST(DS) erfüllt die Schlüsselbedingung
"np `has unique elements`" $\Leftrightarrow$

$\forall r \in V(D_{pers})$: $struct(r) = \mu_{struct}(np) \Rightarrow \sigma(B; C \leftarrow r)$

mit der Bedingung B =

```
card( {* x | x in C *}) = card({ x | x in C })
```

Bei dieser Semantik kann eine Liste bzw. ein Feld beliebig viele Nullwerte enthalten, ohne daß dadurch diese Variante einer Schlüsselbedingung verletzt wäre. So erfüllen z.B. `<1, 2, null>` und `<1, null, 2, null>` eine Schlüsselbedingung der Variante 2, während dies für `<1, 2, null, 1>` nicht zutrifft.

Variante 3: „globale“ Schlüssel für Objekttypen

Im Zusammenhang mit Objekttypen t ∈ DEFTYPES ist es in vielen Fällen wünschenswert, eine Schlüsselbedingung für die Typextension von t ∈ DEFTYPES anzugeben (d.h. die Menge aller Instanzen eines Objekttyps t), auch wenn diese Menge kein expliziter Bestandteil eines ESCHER$^+$-Datenbankschemas ist. Auf diese Weise kann man jedoch sicherstellen, daß Objekte auch über den „beschränkten Horizont“ einer Kollektion hinaus (vgl. Schlüsselbedingungen der Variante 1) durch Schlüsselattribute eindeutig bestimmt sind.

Eine Schlüsselbedingung, die sich über die Typextension eines Objekttyps t erstreckt, muß der folgenden Syntaxregel genügen:

condition ::= *defType* `has key` *key*

Dabei ist *defType* der Name eines Objekttyps. Mit einer solchen „globalen“ Schlüsselbedingung entfällt die Notwendigkeit der Angabe von Schlüsselbedingungen der Variante 1 für Kollektionen von Objekten des Typs t, da diese von der „globalen“ Schlüsselbedingung impliziert werden.

Beispiel 5.4 Ist `Buch` der Objekttyp aus Beispiel 3.6, dann ist

```
Buch has key Signatur
```

eine Schlüsselbedingung der Variante 3. Aus obiger Schlüsselbedingung für den Objekttyp `Buch` folgt, daß implizit auch die Schlüsselbedingung `Bestand has key Signatur` gilt, wobei `Bestand` die Tabelle aus Beispiel 3.9 ist. ❒

Eine Schlüsselbedingung "n `has key` [k_1, …, k_r]" ist zulässig, wenn gilt:

(i) $\mu_{struct}(n) = t \in$ DEFTYPES.

(ii) k_i ist für alle i mit $1 \le i \le r$ ein gültiger Pfad im Strukturbaum von t oder eines seiner Supertypen t'. Genauer:
Falls $k_i = a_{i1}.a_{i2} \ldots .a_{in_i}$ und $AttrAccess(t)(a_{i1}) = (t', n')$,
dann ist $k_i \in$ SIMPLEPATHS(*name*(t')).

Schlüsselbedingungen dieser Variante beziehen sich – wie erwähnt – auf die Typextension eines Objekttyps. Allgemein ist für t ∈ DEFTYPES die *Typextension* von t definiert als

$$Ext(t) := \{ \omega \in \Omega \mid \omega \in \mathrm{Def}(I(t)) \wedge t \in pers\_types(\omega) \} \qquad (5.5)$$

Zur Typextension eines Objekttyps gehören also nur die *persistenten* Objekte dieses Typs. Bislang kann auf eine Typextension *Ext*(t) nicht über einen Namen zugegriffen werden, da ESCHER$^+$ nur Namen benutzerdefinierter Kollektionen (Tabellen) kennt. Während bislang alle Prüfbedingungen als SCRIPT$^+$-Ausdrücke angegeben werden konnten, ist dies hier nicht mehr der Fall. Es stellt sich die Frage, auf welche Weise die Einhaltung dieser Integritätsbedingung überhaupt effizient überprüft werden kann. Ein Durchsuchen der gesamten Datenbankinstanz nach Objekten des Typs t ist völlig indiskutabel. Wir gehen deshalb davon aus, daß *Ext*(t) als interne Tabelle vorliegt, auf die nur das System selbst zugreifen kann. In SCRIPT$^+$-Ausdrücken werde die Typextension von t mit *name*(t) = n mit `EXT(n)` bezeichnet.

Die **Semantik** einer Schlüsselbedingung "n `has key` k" mit k = [k_1, …, k_r] ist nun wie folgt gegeben:

Eine Datenbankinstanz INST(DS) erfüllt die Schlüsselbedingung "n `has key` k" ⇔

$$\forall r \in V(D_{pers}): struct(r) = \mu_{struct}(n) \Rightarrow \sigma(B_n)$$

mit der Bedingung B_n =

```
card(EXT(n)) = card({[a1:x.k1,a2:x.k2, …,ar:x.kr]| x in EXT(n):
x.k1 <> null and x.k2 <> null and … and x.kr <> null})     (5.6)
```

Variante 4: „globale“ Schlüssel für Objekttypvarianten

Wir gehen nun noch einen Schritt weiter, indem wir globale Schlüsselbedingungen auch für Objekttypvarianten zulassen. Die Syntax lautet

condition ::= *varType* `has key` *key*

Dabei muß *varType* der Name einer Objekttypvariante sein.

Für die Objekttypvariante `Verkehrsmittel` aus Beispiel 3.1 ist

```
Verkehrsmittel has key Name
```

eine gültige Schlüsselbedingung, sofern `Name` ein Attribut ist, das in allen Alternativen von `Verkehrsmittel` enthalten ist.

Die Semantik dieser Form einer Schlüsselbedingung entspricht der für Schlüsselbedingungen der Variante 3, allerdings ist `EXT(n)` in der Bedingung B aus (5.6) durch `EXT(`n_1`) union` … `union EXT(`n_k`)` zu ersetzen, wobei n_1, …, n_k die Namen der Alternativen der Objekttypvariante t sind. Diese Form einer Schlüsselbedingung ist also noch „globaler“ als die der Variante 3, da sogar typübergreifende Eindeutigkeit gefordert wird.

5.3.2 Inklusionsbedingungen

Nach den Schlüsselbedingungen wird mit den Inklusionsbedingungen eine weitere, wichtige Gruppe von Integritätsbedingungen eingeführt. Mit ihnen lassen sich Teilmengenbeziehungen für Kollektionen angeben. Mit ihrer Hilfe läßt sich aber auch der Wertebereich einer einfachen Tupelkomponente oder Objektattributes auf die Elemente in einer bestimmten Kollektion einschränken.

Die Syntaxregel für Inklusionsbedingungen lautet:

condition ::=
(*structPathExpr* `in` *structPathExpr* |
structPathExpr `is included in` *structPathExpr*) [*cascade*]

Bis auf weiteres lassen wir das optionale Nichtterminal *cascade* außer Betracht.

Das nachfolgende Beispiel illustriert Situationen, in denen Inklusionsbedingungen von Nutzen sind.

Beispiel 5.5

a) Es seien `Studenten` bzw. `Personen` zwei Tabellen mit den Strukturen `{Student}` bzw. `{Person}`. Zwar ist `Student` ein Subtyp von `Person`, allerdings ist es ohne weiteres möglich, daß es Objekte in `Studenten` gibt, die keine Elemente von Personen sind. Dies ist nicht immer erwünscht. Mit Hilfe einer Inklusionsbedingung kann jedoch erzwungen werden, daß jederzeit `Studenten` eine Teilmenge von `Personen` ist. In ESCHER$^+$ läßt sich dies über die Inklusionsbedingung

```
Studenten.# in Personen
```

erreichen. Sie besagt, daß alle Elemente (deshalb der `.#`-Schritt!) von `Studenten` in `Personen` enthalten sind. Alternativ lassen wir auch folgende Syntax zu:

```
Studenten is included in Personen
```

b) Es sei `Projekt` ein Objekttyp mit dem Attribut `Leiter` vom Typ `Angestellter`. Als Werte dieses Attributes seien nur solche Angestellte zulässig, die in der Tabelle `Manager` (mit der Struktur `{Angestellter}`) enthalten sind. Dieser Sachverhalt wird durch die Inklusionsbedingung

```
Projekt.Leiter in Manager
```

erfaßt. Damit sind Angestellte, die nicht Elemente von `Manager` sind, als Projektleiter nicht zugelassen.

c) Es sei `Artikel` ein Objekttyp, der Zeitschriftenartikel beschreibt. Es gebe ein mengenwertiges Attribut `Stichwoerter`, das die für einen Artikel zutreffenden Schlüsselwörter in Form von Strings enthält. Jedoch sollen nur „genormte“ Schlüsselwörter zugelassen sein, die in einer Tabelle `NormStichwoerter` zusammengefaßt sind. Wir formulieren dazu die Inklusionsbedingung

```
Artikel.Stichwoerter is included in NormStichwoerter
```

d) Das folgende Beispiel zeigt die Verwendung einer Inklusionsbedingung im Zusammenhang mit Objekttypvarianten: Kaufhaus XYZ sei Anbieter unterschiedlicher Produkte, die als Instanzen verschiedener Objekttypen modelliert werden, die alle zusammen unter einer Objekttypvariante `Produkt` zusammengefaßt werden. Eine Alternative von `Produkt` sei z.B. der Objekttyp `Elektro`. Die Tabelle `AngebotXYZ` mit der Struktur `{Produkt}` enthalte das Angebot von XYZ. In der Tabelle `ElektroUmsatz` mit der Struktur

```
{[Produkt: Elektro, Jahr: integer, Umsatz: integer]}
```

soll der Umsatz von XYZ für Elektroartikel erfaßt werden. Die Inklusionsbedingung

```
ElektroUmsatz.#.Produkt in AngebotXYZ
```

sorgt dafür, daß nur solche Elektroartikel in der Tabelle vorkommen, die XYZ auch anbietet. ❒

Jede Inklusionsbedingung "np `is included in` s"[7] läßt sich auch in der Form "np`.# in` s" angeben, wie wir dies schon in Beispiel 5.5 a) gezeigt hatten. Die alternative Notation mit "`is included in`" hebt die einzuhaltende Inklusionsbeziehung für Kollektionen deutlicher hervor. Von nun an beschränken wir uns jedoch auf Inklusionsbedingungen der Form "np `in` s".

Für die Gültigkeit einer Inklusionsbedingung "np `in` s" müssen folgende Voraussetzungen erfüllt sein:

(i) $\mu_{struct}(np)$ und $\mu_{struct}(s)$ sind definiert, und es gilt $isCollection(type(\mu_{struct}(s)))$.

(ii) s ist ein Tabellenpfadausdruck (d.h. s beginnt mit dem Namen einer Tabelle) und enthält keinen `.#`-Pfad

(iii) Für $t_1 = type(\mu_{struct}(np))$ und $t_2 = type(\mu_{struct}(s.\#))$ trifft einer der folgenden Fälle zu:

7. Wir schreiben nicht `is subset of`, damit nicht der Eindruck entsteht, daß die an einer Inklusionsbeziehung beteiligten Kollektionen immer Mengen sind.

– $t_1, t_2 \in$ BASETYPES $\cup$ ENUMTYPES $\wedge\ t_1 = t_2$ (vgl. Beispiel 5.5 c))

– $t_1 \in$ DEFTYPES $\wedge\ t_2 \in$ DEFTYPES $\wedge\ t_1$ *isa** t_2 (vgl. Beispiel 5.5 a), b))

– $t_1 \in$ OBJVAR $\wedge\ t_2 \in$ OBJVAR $\wedge\ t_1 = t_2$

– $t_1 \in$ DEFTYPES $\wedge\ t_2 \in$ OBJVAR
$\wedge\ \exists\ (l, t') \in$ *variants*(t_2): t_1 *isa** t' (vgl. Beispiel 5.5 d))

Wir beschränken uns also auf Inklusionsbedingungen für einfache Datenobjekte bzw. für Instanzen von Objekttypvarianten, die ja – vereinfacht gesagt – lediglich um ein Label ergänzte Objekte sind.

Die **Semantik** einer Inklusionsbedingung "np `in` s" ist wie folgt definiert:

Eine Datenbankinstanz INST(DS) erfüllt "np `in` s" $\Leftrightarrow$

$$\forall\ r \in V(D_{pers})\text{: } \mathit{struct}(r) = \mu_{struct}(np) \Rightarrow \sigma(B;\ x \leftarrow r) \tag{5.7}$$

mit

$$B = \begin{cases} \texttt{x in \{y as } l \texttt{|y in s: y is } l\texttt{\}} & \text{, falls } t_1 \in \text{DEFTYPES} \wedge t_2 \in \text{OBJVAR} \\ & (t_1 \text{ und } t_2 \text{ wie in Voraussetzung (iii))} \\ \texttt{x in \{y|y in s\}} & \text{, sonst} \end{cases}$$

Inklusionsbedingungen stehen in Verwandtschaft zu Fremdschlüsselbedingungen im Relationenmodell. Letzere spielen bei rein wertebasierten Datenmodellen wie dem Relationenmodell eine wichtige Rolle bei der Gewährleistung der referentiellen Integrität. Dort ist man darauf angewiesen, Verweise von einem Tupel auf ein anderes implizit durch Wertegleichheit von Tupelkomponenten auszudrücken. Im Relationenmodell müssen im Schema `FOREIGN KEY`-Bedingungen angegeben werden, um sog. „dangling references“ zu verhindern, die durch Löschen eines Tupels unter Beibehaltung der in anderen Tupeln enthaltenen Verweise auf dieses Tupel entstehen können. Zur Gewährleistung der referentiellen Integrität benötigt man in ESCHER$^+$ bei der Modellierung mit Objekttypen keine expliziten Fremdschlüsselbedingungen: Die Identität $\omega \in \Omega$ eines Objektes ist gewissermaßen der einzige „Fremdschlüssel“ für Instanzen eines Objekttyps. Die referentielle Integrität wird für Objekte automatisch eingehalten, da zugesichert wird, daß für alle erreichbaren Knoten r= (ω, t) gilt: $\omega \in$ Def(I(t')) für alle t' $\in$ DEFTYPES mit t *isa** t'. Es kann also keine „dangling object references“ geben in dem Sinne, daß ein Knoten r= (ω, t) zum persistenten Datenbestand gehört, dieser jedoch nicht relativ zum Typ t oder einem Supertyp von t interpretiert werden kann. Die Gewährleistung der referentiellen Integrität ist in unserem Modell also eine *inhärente* Integritätsbedingung (vgl. Abschnitt 2.1.2). Eine Fremdschlüsselbedingung hat im Relationenmodell jedoch eine Doppelfunktion, denn sie drückt gleichzeitig eine Inklusionsbedingung aus, die besagt, daß die Fremdschlüsselverweise auf Tupel in einer bestimmten Relation „zeigen“. Tupel in einer anderen Relation, die für einen Verweis prinzipiell auch in Frage kämen, können nicht referenziert werden. Auch für ESCHER$^+$ gibt es Inklusionsbedingungen, die besagen, daß gewisse Datenobjekte in einer bestimmten Tabelle enthalten sein müssen. Es soll jedoch nochmals ausdrücklich darauf hingewiesen werden, daß Inklusionsbedingungen in ESCHER$^+$ nicht notwendig sind, um die referentielle Integrität für Objekte sicherzustellen.

Mit der Definition einer Integritätsbedingung haben wir bislang keine explizite Fehlerreaktion verknüpft, die beim Auftreten einer Integritätsverletzung dafür sorgen soll, die Integrität wie-

derherzustellen, und dabei „generisch“ genug ist, um eine adäquate Reaktion in allen Anwendungssituationen zu sein. Als einzig „vernünftige“ generische Reaktion bleibt etwa bei der Verletzung einer Schlüsselbedingung nur das Zurücksetzen der aktuellen Transaktion. Für Inklusionsbedingungen stellt sich hingegen die Situation günstiger dar. Hier können Einfüge- und Lösch-Regeln angegeben werden, die dafür sorgen, daß die Integritätsverletzung automatisch korrigiert wird.

Wir betrachten dazu die Fälle a) und b) aus Beispiel 5.5:

- Soll ein Objekt in die Tabelle `Studenten` eingefügt werden, das nicht Element von `Personen` ist, dann bleibt die Inklusionsbedingung gültig, falls das Einfügen in `Studenten` auch zu einem Einfügen desselben Objektes in `Personen` führt (*kaskadierendes Einfügen*).
- Wird aus `Personen` ein Objekt entfernt, das Element von `Studenten` ist, dann bleibt die Integritätsbedingung gültig, wenn das Objekt auch aus `Studenten` gelöscht wird (*kaskadierendes Löschen*)

Im Fall b) ist die Situation ähnlich:

- Das Update des Attributes `Leiter` eines Objektes vom Typ `Projekt` kann dazu führen, daß der neue Wert kein Element der Tabelle `Manager` ist, was ggf. durch kaskadierendes Einfügen in `Manager` korrigiert werden kann.
- Umgekehrt wird auch durch das Entfernen eines Angestellten aus `Manager`, der Leiter eines Projektes `p` ist, die Inklusionsbedingung verletzt, was aber korrigiert werden kann, indem `p.Leiter` zu *null* gesetzt wird. In dieser Situation sprechen wir ebenfalls von kaskadierendem Löschen.

Mit einer Inklusionsbedingung in ESCHER$^+$ soll also die Angabe einer Fehlerreaktion in Form von kaskadierendem Löschen und/oder kaskadierendem Einfügen verbunden werden. Für das bisher nicht besprochene, optionale Nichtterminal *cascade* führen wir folgende Syntaxregeln ein:

cascade ::= `cascade` *cascadeOp* [`','` *cascadeOp*]

cascadeOp ::= `add` | `rem`

"`cascade add`" steht für kaskadierendes Einfügen, "`cascade rem`" für kaskadierendes Löschen, "`cascade add, rem`" für sowohl kaskadierendes Einfügen als auch kaskadierendes Löschen.

5.3.3 Disjunktheitsbedingungen

Auch für die „Negation“ einer Inklusionsbedingung soll eine entsprechende Integritätsbedingung formuliert werden können: Gewisse Datenobjekte sollen gerade *nicht* in einer anderen Kollektion enthalten sein.

Beispiel 5.6 Es seien `GruppeA` und `GruppeB` zwei Tabellen, die jeweils die Struktur `{Person}` haben. Um sicherzustellen, daß beide Mengen disjunkt sind, läßt sich die Disjunktheitsbedingung

```
GruppeA is disjoint to GruppeB
```

oder – äquivalent –

```
GruppeA.# not in GruppeB
```

angeben. Alle Datenobjekte, denen der Strukturbaumknoten `GruppeA.#` zugeordnet ist, dürfen keine Elemente von `GruppeB` sein. ❐

Eine Disjunktheitsbedingung hat die Syntax

condition ::=
(*structPathExpr* `not in` *structPathExpr*
| *structPathExpr* `is disjoint to` *structPathExpr*) [*cascade*]

Eine Disjunktheitsbedingung "np `is disjoint to` s" läßt sich auch in der Form "np`.#` `not in` s" angeben. Für die in Beispiel 5.6 angegebene Bedingung kann alternativ auch "`GruppeA.# not in Gruppe B`" geschrieben werden. Wir können uns daher auf Disjunktheitsbedingungen der Form "np `not in` s" beschränken. Für die Gültigkeit von Disjunktheitsbedingungen gelten dieselben Voraussetzungen an np und s wie für Inklusionsbedingungen.

Die **Semantik** einer Disjunktheitsbedingung "np `not in` s" ist wie folgt definiert:

Eine Datenbankinstanz INST(DS) erfüllt "np `not in` s" $\Leftrightarrow$

$\forall$ r $\in$ V(D_{pers}): *struct*(r) = μ_{struct}(np) $\Rightarrow$ σ(`not` B; `x` $\leftarrow$ r)

mit B aus (5.7).

5.3.4 Lokale Wertebedingungen

Es wird nun mit den sog. *lokalen Wertebedingungen* eine Klasse von Integritätsbedingungen eingeführt, mit denen u.a. Einschränkungen für Werte von Tupelkomponenten und für Attribute eines Objektes angegeben werden können. Lokale Wertebedingungen in ESCHER$^+$ stellen eine Verallgemeinerung der *column constraints* aus SQL/92 dar.

Wir beginnen zunächst mit einigen Beispielen zu dieser Klasse von Integritätsbedingungen.

Beispiel 5.7
a) Für den Objekttyp `Person` lassen sich folgende lokale Wertebedingungen angeben:

- Alle Personen müssen 1960 oder später geboren sein:

  ```
  year(d) >= 1960 with d = Person.GebDat
  ```

  (Es wird angenommen, daß `date` ein Basistyp ist, für den auch eine Basisoperation `year` definiert ist)
  Durch "`with d = Person.GebDat`" wird ausgedrückt, daß die Bedingung für alle die Knoten r gelten soll, für die *struct*(r) = μ_{struct}(`Person.GebDat`) gilt.
- Der Nachname darf bei keiner Person nullwertig oder der leere String sein:

  ```
  n <> null and n <> "" with n = Person.Name.NName
  ```
- Vor- und Nachname zusammen dürfen höchstens 50 Zeichen lang sein:

  ```
  length(n.VName) + length(n.NName) <= 50
  with n = Person.Name
  ```

  (Es wird angenommen, daß auf Strings eine Basisoperation `length` definiert ist)
  Hier wird eine Bedingung gleich an zwei „Spalten“ (`VName` und `NName`) gestellt.

- Für die Menge der Hobbies einer Person soll eine *Kardinalitätsbedingung* angegeben werden. Die Menge sei nicht leer, enthalte aber höchstens 12 Elemente. Dies läßt sich als lokale Wertebedingung wie folgt angeben:

```
card(h) >= 1 and card(h) <= 12 with h = Person.Hobbies
```

b) Eine Integritätsbedingung, die für die Tabelle `Personen` (eine Menge von Personen) Nullobjekte als Elemente verbietet, lautet:

```
p <> null with p = Personen.#
```

Hier muß `.#` explizit genannt werden, da es sich um den letzten Schritt in einem Tabellenpfadausdruck handelt.

c) Es sei `Mitarbeiter` eine Tabelle, die eine Menge von Instanzen eines Objekttyps `Angestellter` verwaltet. Für `Angestellter` sei ein mengenwertiges Attribut `Kenntnisse` definiert. Die Bedingung

```
"SQL" in a.Kenntnisse with a = Mitarbeiter.#
```

drückt aus, daß für die Tabelle `Mitarbeiter` nur solche Angestellte in Frage kommen, die SQL-Kenntnisse besitzen.

d) Für den varianten Typ `AdrVar` aus Beispiel 3.4 sind folgende lokale Wertebedingungen sinnvoll:

```
p > 0 with p = AdrVar:postf
hnr > 0 with hnr = AdrVar:str.HNr
```

❒

Eine lokale Wertebedingung hat die Syntax

condition ::= *expr* `with` *identifier* '=' *structPathExpr*

Dabei ist *expr* ein boolescher Ausdruck.

Für die Zulässigkeit einer lokalen Wertebedingung "B `with` x = np" muß gelten:

- B ist ein boolescher Ausdruck, in dem ausschließlich die freie Variable x verwendet wird.
- Ist $r \in V(D_{pers})$ ein Knoten mit $struct(r) = \mu_{struct}(np)$, dann darf für die Auswertung von B ausschließlich auf solche Knoten zugegriffen werden, die sich im Wertebaum *Tree*(r) befinden. Das bedeutet: Ist ein Knoten in *Tree*(r) ein Objekt-Knoten $r = (\omega, t)$, dann darf in B auf kein Attribut des Objektes ω zugegriffen werden, da dazu der Wertebaum *Tree*(r) verlassen werden muß.

Aus diesen Voraussetzungen erklärt sich auch der Begriff „lokal“. Eine lokale Wertebedingung für Instanzen `a` eines Objekttyps `Angesteller` darf also nicht die Bedingung `a.Gehalt <= a.Boss.Gehalt` enthalten, da dazu nicht nur auf das Objekt `a` zugegriffen wird, sondern auch auf ein Attribut des Objektes `a.Boss`.

Bei genauer Betrachtung von Beispiel 5.7 fällt auf, daß zunächst noch die Möglichkeit besteht, ein und dieselbe lokale Wertebedingung auf unterschiedliche Weise auszudrücken. So hätte man für die erste Bedingung in Beispiel 5.7 a) auch

```
year(p.GebDat) >= 1960 with p = Person
```

schreiben können. Allerdings wollen wir vereinbaren, daß der Strukturpfadausdruck np in einer lokalen Wertebedingung immer die Struktur des kleinsten umgebenden Datenobjektes bezeichnet, das alle in der Bedingung vorkommenden Attribute enthält. Dies ist im obigen Bei-

spiel das Attribut `GebDat` selbst. In der dritten Bedingung in Beispiel 5.7 bezeichnet `Person.Name` die Struktur des kleinsten umgebenden Datenobjektes.

Es steht nun noch die formale Angabe der Semantik lokaler Wertebedingungen aus.

Die **Semantik** für eine Wertebedingung "B `with` x = np" lautet:
Eine Datenbankinstanz INST(DS) erfüllt die Wertebedingung "B `with` x = np" $\Leftrightarrow$

$$\forall\, r \in V(D_{pers}): \mathit{struct}(r) = \mu_{struct}(np) \Rightarrow \sigma(B;\, x \leftarrow r)$$

Damit haben wir die Behandlung aller zu Beginn des Abschnittes angekündigten Klassen von Integritätsbedingungen durchgeführt.

5.4 Überwachung der Integritätsbedingungen

Für die Überwachung von Integritätsbedingungen werden in der Literatur folgende Alternativen genannt (vgl. [Lip89, HS95]):

- Überwachung durch einen *Integritätsmonitor*: Dieser wird als spezielle Komponente eines DBMS implementiert, die alle Integritätsbedingungen zentral überwacht. Die in einem Datenbankschema angegebenen Integritätsbedingungen werden zur Entwurfszeit automatisch in eine für den Integritätsmonitor geeignete interne Form transformiert.
- Überwachung durch *sichere Transaktionen*: Es werden Transaktionen definiert, für die (im Idealfall) auf der Grundlage von Vor- und Nachbedingungen formal verifiziert werden kann, daß sie konsistenzerhaltend sind. Die Schnittstelle zur Datenbank besteht dann nur aus diesen sicheren Transaktionen, die auch *Standardtransaktionen* genannt werden.

Das explizite „Ausprogrammieren“ von Integritätsprüfungen durch den Anwendungsprogrammierer sollte nur der letzte Ausweg sein, wenn es keine andere Möglichkeit der Kontrolle einer anwendungsspezifischen Integritätsbedingung gibt. Nachteilig ist dabei insbesondere, daß dieselben Integritätsprüfungen oftmals an verschiedenen Stellen in Anwendungsprogrammen wiederholt werden müssen.

Für die im vorangegangenen Abschnitt eingeführten Integritätsbedingungen schlagen wir vor, zu ihrer Überwachung einen Integritätsmonitor einzusetzen, der auf einer Auswertung interner ECA-Regeln (vgl. Abschnitt 2.4.3.2) basiert: Durch die Ausführung einer generischen Update-Operation auf den persistenten Daten wird möglicherweise eine Integritätsbedingung verletzt. Durch die Update-Operation wird ein Ereignis E ausgelöst, das eine oder mehrere ECA-Regeln aktiviert. Gewißheit darüber, ob tatsächlich eine Integritätsverletzung vorliegt, verschafft die Auswertung der Prüfbedingung C („condition“) einer Regel. Ergibt diese Auswertung den Wert *true*, dann wird die Aktion A ausgeführt. Die Aktion besteht in den meisten Fällen aus dem Zurücksetzen der aktiven Transaktion, im Fall von Inklusionsbedingungen kann A auch aus korrigierenden Maßnahmen (kaskadierendes Einfügen bzw. Entfernen) bestehen, die die Konsistenz wiederherstellen.

Es soll nun auf einige Aspekte der Überwachung von Integritätsbedingungen in ESCHER$^+$ genauer eingegangen werden. Ein lückenloser Implementationsvorschlag würde jedoch den Rahmen dieser Arbeit sprengen. Vielmehr geht es um das prinzipielle Vorgehen bei der Über-

wachung der Integritätsbedingungen aus Abschnitt 5.4 und um Hinweise auf dabei auftretende Probleme.

Ereignisse

Integritätsverletzungen sind mögliche Konsequenzen der Ausführung von Update-Operationen auf persistenten Datenobjekten. Der Integritätsmonitor erfährt von der Ausführung einer Update-Operation über *Ereignisse*. Jede der generischen Update-Operationen aus Abschnitt 4.3.2 löst zwei Ereignisse aus:

- Unmittelbar nach ihrem Aufruf wird ein sog. `before`-Ereignis ausgelöst.
- Unmittelbar vor dem Rücksprung an die Aufrufstelle wird, falls während der Ausführung der Update-Operation keine Ausnahme ausgelöst wurde, ebenfalls ein Ereignis ausgelöst, das sog. `after`-Ereignis

Die Unterscheidung ist sinnvoll, um Bedingungen bereits vor bzw. erst nach der Ausführung einer Operation testen zu können.

Zu jedem auftretenden Ereignis E gehören aktuelle Ereignisparameter, die zur Auswertung des CA-Teils einer ECA-Regel benötigt werden und die an die formalen Namen der Ereignisparameter gebunden werden. Die aktuellen Parameter der `before`-Ereignisse stimmen immer mit den aktuellen Parametern der gleichnamigen generischen Update-Operation überein. Dies ist bei den `after`-Ereignissen nicht immer der Fall: So werden bei Aufruf und Abarbeitung von `insert` mit der aktuellen Parameterliste $<r_1, r_2>$ die Ereignisse

- `before insert(S, elem)` mit der Bindung (`S` $\leftarrow r_1$, `elem` $\leftarrow r_2$) und
- `after insert(S, elem)` mit der Bindung (`S` $\leftarrow r_1$, `elem` $\leftarrow r_3$)

ausgelöst. Dabei ist r_3 der Wurzelknoten der eingefügten Kopie von *Tree*(r_2) [8]. Bei der Durchführung einer `update`-Operation mit der aktuellen Parameterliste $<r_1, r_2>$ werden die Ereignisse

- `before update(x, x_neu)` mit der Bindung (`x` $\leftarrow r_1$, `x_neu` $\leftarrow r_2$) und
- `after update(x, x_alt)` mit der Bindung (`x` $\leftarrow r_1$, `x_alt` $\leftarrow r_3$)

ausgelöst[9]. Dabei ist r_3 ein neuer Knoten, der den alten Wert von `x` vor dem Update enthält.

Kopplungsmodi

Je nachdem, ob für eine Integritätsbedingung der Modus `hard` oder `soft` angegeben wurde, findet ihre Überprüfung sofort oder erst in der Commit-Phase der Transaktion statt. Diese beiden Situationen werden in Abb. 4.4 veranschaulicht. Wichtig ist, daß auf die Auswertung der Prüfbedingung („Condition“) unmittelbar die Reaktion („Action“) folgt. Im Zusammenhang mit ECA-Regeln wird üblicherweise von den beiden Kopplungsmodi `immediate` und `deferred` gesprochen, was wir in diesem Abschnitt ebenfalls tun wollen.

8. Es sei daran erinnert, daß `insert` immer eine Kopie einfügt.
9. Allerdings nur, falls `x` und `x_neu` verschieden sind.

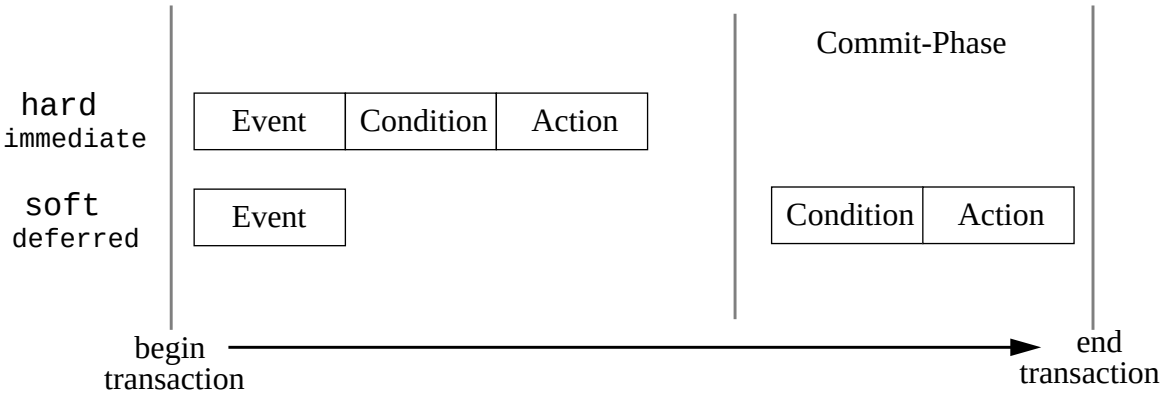


Abbildung 4.4: Überprüfung einer potentiellen Integritätsverletzung nach Eintreten eines Events für die beiden Modi `hard` und `soft`

Integritätsbedingungen als interne ECA-Regeln

Den Einsatz von ECA-Regeln zur Überwachung der ESCHER$^+$-spezifischen Integritätsbedingungen wollen wir nun anhand von Beispielen erläutern. Eine allgemeine und vollständige Beschreibung der Transformation von Integritätsbedingungen in ECA-Regeln würde den Rahmen dieser Arbeit sprengen.

Eine ECA-Regel wird durch verschiedene Regelkomponenten beschrieben, darunter natürlich die *event*-, *condition*- und *action*-Komponente. In der *coupling*-Komponente wird der Kopplungsmodus angegeben. Auf weitere Komponenten gehen wir ein, sobald wir sie benötigen. In Die Bedingungen der *condition*-Komponente sowie die Anweisungen der *action*-Komponente werden in SCRIPT$^+$-Syntax angegeben.

Besonders einfach sind die Regeln zur Überwachung lokaler Wertebedingungen. Man betrachte die Bedingung "`year(d) >= 1960 with d = Person.GebDat`" aus Beispiel 5.7 a). Ist der Modus für diese Bedingung `hard`, d.h. soll sie sofort überprüft werden, dann wird folgende ECA-Regel erzeugt:

event : `before update(d, d_neu) with d = Person.GebDat`
coupling : `immediate`
condition : `not(year(d_neu) >= 1960)`
action : `abort;`

Diese Regel muß genau dann ausgewertet werden, wenn durch eine update-Operation ein `before update`-Ereignis mit der aktuellen Parameterbindung (`d` $\leftarrow r_1$, `d_neu` $\leftarrow r_2$) ausgelöst wird und $struct(r_1) = \mu_{struct}$(`Person.GebDat`) gilt. Letzteres wird im *event*-Teil der Regel durch "`with d = Person.GebDat`" ausgedrückt. Das bedeutet, daß "`with` …" wie ein Filter wirkt, der dafür sorgt, daß die Regel nur dann aktiviert wird, wenn dem an `d` gebundenen Knoten ein bestimmter Knoten in einem Strukturbaum zugeordnet ist. Somit bindet "`with` …" die Regel an einen bestimmten Strukturbaumknoten. Die angegebene Regel hat den Kopplungsmodus `immediate`, d.h. wir gehen davon aus, daß für die Integritätsbedingung der Modus `hard` angegeben wurde. Für den Modus `soft` lautet die ECA-Regel:

event : `after update(d, d_alt) with d = Person.GebDat`
coupling : `deferred`

condition : `not(year(d) >= 1960)`
action : `abort;`

Schwieriger wird es, geeignete ECA-Regeln für andere Integritätsbedingungen als lokale Wertebedingungen anzugeben. Wir betrachten dazu ECA-Regeln für Schlüsselbedingungen.

Beispiel 5.8

Wir modifizieren die Struktur der Tabelle `T` aus Beispiel 5.1, indem wir auf der tiefsten Schachtelungsebene die Tupelstruktur

```
[E: string, F: integer, G: integer]
```
(5.8)

durch einen Objekttyp `t` ersetzen, dessen Zustandsstruktur durch (5.8) gegeben ist. Die Tabelle `T` hat also die Struktur

```
{[  A:string, B:{[C: integer, D: {t} ]} ]}
```

Es sollen die Schlüsselbedingungen
"`T has key A`", "`T.B has key C`" und "`T.B.D has key E`" gelten. ❒

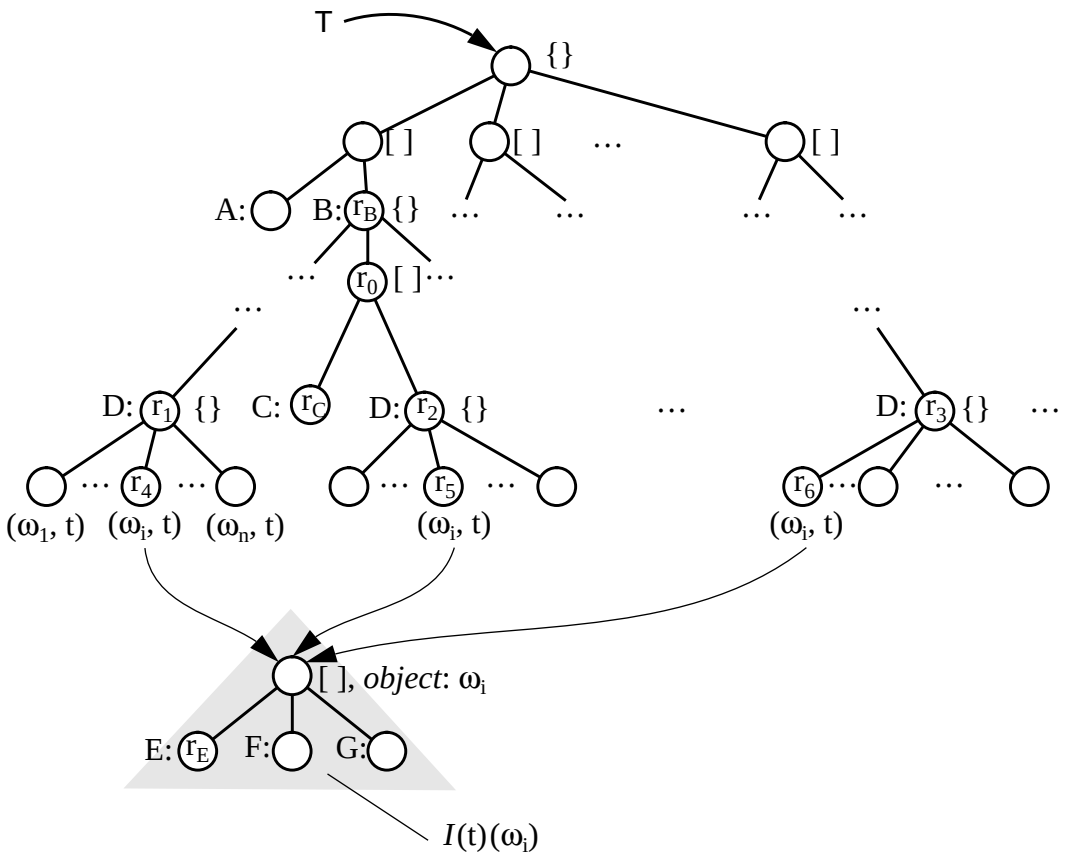


Abbildung 4.5: Der Wertebaum zur Tabelle `T` mit „object sharing“ auf tiefer Schachtelungsebene

Für die Schlüsselbedingung "`T.B has key C`" müssen mehrere Ereignisse berücksichtigt werden. Zunächst kann die Schlüsselbedingung durch das Einfügen in eine Menge `B` gefährdet werden. Für den Fall sofortiger Prüfung ergibt sich die ECA-Regel

```
event     : before insert(S, elem) with S = T.B
coupling  : immediate
condition : exists x in S: x.C = elem.C
action    : abort;
```

Aber auch die Modifikation eines Attributs `C` kann die Konsistenz gefährden. In diesem Fall muß ausgehend von dem zu modifizierenden Knoten im Wertebaum 2 Schritte in Richtung Wurzel gegangen werden, um die zugehörige Menge zu finden (vgl. Abb. 4.5: Zum Knoten r_C gehört der Mengenknoten r_B). Dies wird in der sog. *prepare*-Komponente durchgeführt. In dieser zusätzlichen Komponente einer ECA-Regel werden weitere Parameter bestimmt, die zur Auswertung der Bedingung und der eventuellen Abarbeitung der Anweisungen der *action*-Komponente benötigt werden. Die Anweisungen der *prepare*-Komponente werden immer unmittelbar vor Auswertung der Bedingung abgearbeitet.

In der *prepare*-Komponente der nachfolgenden Regel wird eine interne Basisoperation `getParent` verwendet, die folgende Semantik hat:

> Erfolgt ein Aufruf von `getParent(p, n)` mit $p \in \mathcal{L}(pathExpr)$, $n \in IN_0$
> und ist $\mu(p)=r$, dann wird ein neuer, link-wertiger Knoten r' mit
> $val(r') = (parent^n(r), c_{link}(parent^n(struct(r))))$ zurückgegeben.

Das bedeutet, daß wir innerhalb der intern generierten ECA-Regeln die in Abschnitt 4.4 eingeführten Verweise auf Knoten (Links) zulassen. Da ECA-Regeln automatisch generiert werden, steht dies steht nicht im Widerspruch zu der ebenfalls in Abschnitt 4.4 geäußerten Forderung, die Verwendung von Links auf der logischen Ebene nicht zuzulassen.

Damit können wir die ECA-Regel zur Überwachung der Integritätsbedingung "`T.B has key C`" bei einem Update des Attributes `C` in einem `t`-Objekt angeben:

```
event     : before update(c, c_neu) with c = T.B.C
coupling  : immediate
prepare   : S := getParent(c, 2);
condition : c_neu <> null and exists x in S^: x.C = c_neu
action    : abort;
```

Ist `c` an den Knoten r_C in Abb. 4.5 gebunden, dann liefert der Funktionsaufruf `getParent(c, 2)` einen Verweis auf den Knoten r_B. In der *condition* wird jedoch nicht der Verweis (`S`) benötigt, sondern der dereferenzierte Verweis (`S^`).

Man beachte auch, daß die in den letzten beiden ECA-Regeln angegebenen Bedingungen eine wesentlich effizientere Überprüfung ermöglichen, als dies für Bedingungen in Form von Kardinalitätsvergleichen der Fall wäre, die in Abschnitt 5.3.1 im Zusammenhang mit der formalen Semantik von Schlüsselbedingungen genannt wurden.

Betrachten wir nun die Schlüsselbedingung "`T.B.D has key E`". Für die sofortige Kontrolle der Schlüsselbedingung beim Einfügen in eine Menge `D` sorgt die ECA-Regel

```
event     : before insert(S, elem) with S = T.B.D
coupling  : immediate                                    (5.9)
condition : exists x in S: x.C = elem.C
action    : abort;
```

Wesentlich problematischer ist allerdings die Modifikation des Attributes `E` im Zustand eines Objektes vom Typ `t`. Ein Blick auf Abb. 4.5 zeigt uns, daß das Objekt ω_i Element mehrerer

Mengen `D` ist (*object sharing*!). Wird in $I(t)(\omega_i)$ das Attribut `E` geändert, dann müssen alle Mengen `D`, die das Objekt ω_i enthalten, ermittelt werden, da für diese Mengen eine potentielle Verletzung der Schlüsselbedingung "`T.B.D has key E`" vorliegt. Ein Durchsuchen des gesamten Wertebaumes zur Tabelle `T` nach allen diesen Mengen `D` wäre allerdings ein völlig ineffizientes Vorgehen. Günstiger ist es, wenn auf interner Ebene Indexe zur Verfügung stehen, die für Objekte ω „Rückwärtsverweise“ auf diejenigen Söhne r von Mengenknoten zum Attribut `D` liefern, die das Objekt ω repräsentieren, d.h. für die $value(r) = \omega$ gilt.

Wir schlagen deshalb vor, für eine Schlüsselbedingung "np `has key` k" mit $type(\mu_{struct}(np.\#)) = t \in$ DEFTYPES $\cup$ OBJVAR die Generierung und Pflege eines Indexes zu verbinden, der für jedes getypte Datenobjekt (ω, t) die Menge

`backRefs[np.#]`$((\omega, t))$

$:= (\ \{\ r \in V(D_{pers}) \mid struct(r) = \mu_{struct}(np.\#) \wedge value(r) = \omega\},\ c_{set}(c_{link}(t))\ )$

liefert. `backRefs[np.#]`$((\omega, t))$ ist also wiederum ein getyptes Datenobjekt, genauer die Menge aller Verweise auf Knoten in $V(D_{pers})$, denen der Strukturbaumknoten $\mu_{struct}(np.\#)$ zugeordnet ist und die das Objekt ω repräsentieren. Für ω_i aus Abb. 4.5 ist beispielsweise `backRefs[T.B.D.#]`$((\omega_i, t)) = (\{r_4, r_5, r_6\}, c_{set}(c_{link}(t)))$.

Ferner benötigen wir eine interne Basisoperation `getObject`, die Antwort gibt auf die von einem Knoten r gestellte Frage „Welches ist das Objekt ω, in dessen Zustandsbaum ich mich befinde?“. Die genaue Semantik lautet:

> Erfolgt ein Aufruf von `getObject(p)` mit $p \in \mathcal{L}(pathExpr)$, ist $\mu(p) = r$ und $r \in V(Tree(I(t)(\omega)))$ für ein $t \in$ DEFTYPES und $\omega \in Def(I(t))$, dann wird ein neuer, Knoten r' mit $val(r') = (\omega, t)$ zurückgegeben. Andernfalls wird eine Ausnahme ausgelöst.

Wird `getObject` mit dem aktuellen Parameter r_E (siehe Abb. 4.5) aufgerufen, dann wird ein Knoten r' mit $val(r')=(\omega_i, t)$ zurückgegeben. Die Funktion `getObject` kann beispielsweise durch Zurückgehen zur Wurzel des Zustandsbaumes und Lesen einer weiteren Beschriftung *object* (vgl. wiederum Abb. 4.5) realisiert werden.

Die nachfolgende ECA-Regel kontrolliert die Schlüsselbedingung "`T.B.D has key E`" bei Modifikationen des Attributs `E`. In der *prepare*-Komponente wird zunächst die Menge `S_Set` erzeugt. Sie besteht danach aus allen Knoten für Mengen zum Attribut `D`, die das über `getObject` ermittelte Objekt ω enthalten. Neu ist auch die *iterate*-Komponente: Es wird über die Elemente in `S_Set` iteriert, und für jedes Element wird die Bedingung der *condition*-Komponente überprüft wird. Bei einer Änderung des Attributs `E` in $I(t)(\omega_i)$ ist `S_Set` $= \{r_1, r_2, r_3\}$.

```
event     : before update(e, e_neu) with e = t.E
coupling  : immediate
prepare   : o := getObject(e);
            r_Set := backRefs[T.B.D.#](o);
            S_Set := {getParent(r^, 1) | r in r_set};
iterate   : S in S_Set
condition : exists x in S^: x.C = e_neu
action    : abort;
```

Die *iterate*-Komponente ließe sich auch „einzeilig“ als

```
S in {getParent(r^,1)|r in backRefs[T.B.D.#](getObject(e))};
```

schreiben, wodurch die *prepare*-Komponente überflüssig würde.

Ähnliche Situationen ergeben sich bei der Überwachung von Inklusionsbedingungen: Wir nehmen an, daß es neben `T` eine weitere Tabelle `T2` mit der Struktur `{t}` gibt. Die Inklusionsbedingung "`T.B.D.# in T2 cascade rem`" erfordert ebenfalls die Pflege eines Indexes für Rückwärtsverweise, auf den in der *prepare*-Komponente zugegriffen wird. Man beachte, daß kaskadierendes Löschen spezifiziert wurde, d.h. die Verletzung der Inklusionsbedingung durch Entfernen eines Objektes aus `T2` wird nicht durch Zurücksetzen der Transaktion „bestraft“, sondern durch Löschen des Objektes aus allen Mengen `D` kompensiert, die dieses Objekt enthalten. Die ECA-Regel lautet:

```
event     : before remove(S, elem) with S = T2
coupling  : immediate
prepare   : r_Set := backRefs[T.B.D.#](elem);
            i_Set := {[S:getParent(r^, 1),elem: r]| r in r_set};
iterate   : i in i_Set
condition : true
action    : i.S^ rem i.elem^;
```

Hätte man auf "`cascade rem`" verzichtet, dann lautete die ECA-Regel:

```
event     : before remove(S, elem) with S = T2
coupling  : immediate
prepare   : r_Set := backRefs[T.B.D.#](elem);
condition : card(r_Set) > 0
action    : abort;
```

Das Ereignis `check`

Die Möglichkeit beliebig tief geschachtelter Strukturen und der Verzicht auf die Angabe von Integritätsbedingungen für transiente Datenobjekte bringt es mit sich, daß weitere Konsistenzprüfungen notwendig werden, falls transiente Datenobjekte Persistenz erlangen.

Man betrachte dazu wieder die Tabelle `T` aus Abb. 4.4. Es sei `x` eine lokale Variable, deren Struktur mit der Elementstruktur von `T` übereinstimmt. Soll nun `x` in `T` eingefügt werden (über die SCRIPT$^+$-Anweisung `T add x`), dann können danach alle in Beispiel 5.8 genannten Schlüsselbedingungen verletzt sein. Es reicht nicht, beim Einfügen von `x` die Schlüsselbedingung "`T has key A`" zu überprüfen. Es gibt keine Möglichkeit, für transiente Datenobjekte wie `x` Integritätsbedingungen wie "`x.B has key C`" und "`x.B.D has key E`" anzugeben. Das bedeutet, daß für alle durch das Einfügen von `x` in `T` neu hinzukommenden mengenwertigen Attribute `B` und `D` die geforderten Schlüsselbedingungen "`T.B has key C`" und "`T.B.D has key E`" verletzt sein können.

Um die Konsistenz zu wahren, greifen wir auf das sog. *Finger*-Konzept zurück, das sich bereits bei der Implementierung des ESCHER-Prototyps [Weg91a] und auch als Interaktionspradigma auf der Oberfläche [Weg91b] bewährt hat. Ein Finger ist ein verallgemeinerter Cursor, mit dem das Navigieren in Bäumen möglich ist (vgl. auch Abschnitt 1.1). Wir schlagen nun folgendes Vorgehen vor: Erlangt ein bislang transientes Datenobjekt Persistenz, dann traversiert ein sog. *Check-Finger* den gesamten Wertebaum dieses Datenobjektes in depth-first-Strategie und löst bei jedem Knoten das Ereignis `check` aus, dessen einziger Parameter der aktuelle Knoten ist. Über ECA-Regeln, die auf das `check`-Ereignis reagieren, werden die notwendigen Konsistenzprüfungen erfaßt.

In unserem Beispiel werden folgende Regeln notwendig (es werden nur die *event*- und die *condition*-Komponente angegeben):

event : `check(S) with S = T.B`
condition : `card(S)= card({x.C | x in S: x.C <> null})`

und

event : `check(S) with S = T.B.D`
condition : `card(S)= card({x.E | x in S: x.E <> null})`

Trifft der Check-Finger auf einen Objektknoten r = (ω, t) und gehörte der Zustandsbaum *Tree*(*I*(t)(ω)) bislang zu den transienten Datenobjekten, dann muß der Check-Finger auch diesen und alle weiteren Zustandsbäume *Tree*(*I*(t')(ω)) mit t *isa** t', t' ≠ t, traversieren, die bislang transient waren: Wird beispielsweise in die Tabelle `Personen` mit der Struktur `{Person}` ein neues, bislang transientes `Person`-Objekt eingefügt und soll die letzte in Beispiel 5.7 a) angegebene lokale Wertebedingung gelten, so sorgt die ECA-Regel

event : `check(h) with h = Person.Hobbies`
condition : `card(h) >= 1 and card(h) <= 12`

für die Einhaltung der Wertebedingung für persistente `Person`-Objekte.

Zur Auswertung von Regeln bei zurückgestellter Prüfung

Integritätsbedingungen, für die eine zurückgestellte Prüfung vereinbart wurde (Modus `soft`), werden in ECA-Regeln mit dem Kopplungsmodus `deferred` transformiert. Wird eine Regel R_i mit diesem Kopplungsmodus aktiviert, dann wird (R_i, ($x_1 \leftarrow r_1$, …, $x_n \leftarrow r_n$)) in eine Liste aufgenommen, die wir *Commit-Checkliste* nennen. Dabei enthält ($x_1 \leftarrow r_1$, …, $x_n \leftarrow r_n$) Bindungen von Ereignisparametern r_j an in der Regel vorkommende Bezeichner x_j. Wird nun `commit()` aufgerufen, dann besteht der „Test auf Integrität“ (vgl. Tabelle 5.1) aus der Abarbeitung der in der Commit-Checkliste enthaltenen Regeln. Falls vorhanden, wird zunächst die *prepare*-Komponente ausgewertet, wodurch sich die Menge der Bindungen erweitert. Dann erfolgt die Auswertung der Bedingung(en)[10] und ggf. die Ausführung der Aktion.

Für die Überprüfung von Integritätsbedingungen zum Commit-Zeitpunkt ist nur der Netto-Effekt der Modifikationen, die während der Transaktion stattfanden, relevant. Es ist darauf zu achten, daß die Commit-Checkliste keine redundanten Regeln enthält. Diese Gefahr besteht beispielsweise bei mehrfachem Update eines einfachen Attributes. Es ist dann immer nur das letzte Update relevant, d.h. die Regel, die aufgrund eines vorangegangenen Updates in die Commit-Checkliste aufgenommen wurde, sollte wieder entfernt werden, um so Mehrfachprüfungen zu vermeiden. Wird ein Update durch ein nachfolgendes Update rückgängig gemacht, dann braucht gar keine Regel ausgewertet werden. Ähnliches gilt für das Einfügen eines Elementes in eine Kollektion und das nachträgliche Entfernen desselben Elementes, wodurch z.B. die nach dem Einfügen eventuell notwendige Überprüfung einer Schlüsselbedingung unnötig wird.

Ein Entfernen von Regeln aus der Commit-Checkliste ist auch dann notwendig, wenn Datenobjekte, für die Integritätsbedingungen überprüft werden sollen, während der Transaktion den Status der Persistenz verlieren. Für ein Beispiel betrachte man dazu wieder Abb. 4.5: Wird das

10. Enthält die Regel eine *iterate*-Komponente, dann wird eine Bedingung i.d.R. mehrfach (mit unterschiedlichen Parameterbelegungen) überprüft werden.

Element, dessen Wurzelknoten r_0 sei, aus einer Menge `B` entfernt, dann sind alle Knoten aus $V(\mathit{Tree}(r_0))$ nicht mehr persistent. Enthält die Commit-Checkliste eine Regel zur Überprüfung einer Schlüsselbedingung für eine nun nicht mehr persistente Menge `D`, die Nachkomme von r_0 ist, dann muß diese Regel aus der Commit-Checkliste entfernt werden. Möglicherweise verlieren auch Objekte, die Elemente einer nicht mehr persistenten Menge `D` sind, den Status der Persistenz, falls sie von nirgends sonst erreichbar sind. Integritätsbedingungen, die sich auf den Zustand dieser Objekte beziehen, sollen dann auch nicht überprüft werden, und die entsprechenden Regeln sind aus der Commit-Checkliste zu entfernen.

Zuletzt zeigen wir anhand eines Beispieles, daß bei zurückgestellter Prüfung in vielen Fällen eine andere Bedingung angegeben werden muß, als dies bei sofortiger Prüfung der Fall ist. Die ECA-Regel aus (5.9) ist für sofortige Prüfung vorgesehen. Die entsprechende Regel bei zurückgestellter Prüfung lautet:

event : `after insert(S, elem) with S = T.B.D`
coupling : `deferred`
condition : `card({ x.C | x in S: x.C = elem.C}) > 1`
action : `abort;`

Die in (5.9) angegebene Bedingung "`exists x in S: x.C = elem.C`" wäre für die zurückgestellte Prüfung nicht korrekt.

5.5 Konsistenzerhaltung durch Einkapselung

5.5.1 Einkapselung für Attribute von Objekttypen

Eine für ADTs bzw. AOTs[11] charakteristische Eigenschaft ist die Einkapselung ihrer Instanzen. Gemäß der „strengen Lehre“ bedeutet Einkapselung, daß kein direkter lesender oder schreibender Zugriff auf die Attribute von Objekttypen möglich ist, d.h. der Zustand von Objekten ist vollständig „verborgen“ (*information hiding*). Alle Zugriffe finden über ein Methoden-Interface statt (vgl. auch die Diskussionen der Abschnitte 2.5.1 und 3.11).

Die Forderung nach strenger Einkapselung ist begründet in dem Wunsch nach strenger Trennung zwischen Interface und Implementation eines ADTs. Sie ermöglicht die Änderung der unterlegten Implementation unter Beibehaltung des Interfaces. Für einen Objekttyp $t \in$ DEF-TYPES ist die Zustandsfunktion $I(t)$ jedoch nicht dem Implementationsteil (interne Ebene) zuzuordnen, sondern sie gehört zur logischen Ebene! Dennoch kann es notwendig sein, für die gesamte Zustandsfunktion $I(t)$ oder zumindest für einige Attribute eine Einkapselung einzufordern, um die Erhaltung der Konsistenz eines Objektes zu unterstützen. Bei einem Objekttyp `Quadrat` sollte z.B. die direkte Modifikation der Koordinaten seiner Eckpunkte nicht erlaubt sein, da dadurch sehr leicht die charakteristische Eigenschaft eines Quadrats (4 rechte Winkel, 4 gleichlange Seiten) zerstört werden kann. Hier ist es sinnvoll, die Manipulation von Instanzen des Objekttyps `Quadrat` nur über Methoden (z.B. `scale`) zuzulassen. Gibt es einen Subtyp `FarbigesQuadrat`, dann wäre es allerdings übertrieben, für die Modifikation des

11. In ESCHER$^+$ sind die AOTs gerade die Objekttypen.

zusätzlichen Attributes `Farbe` eine Methode definieren zu müssen. Selbst wenn die erlaubten Farben auf „rot“ und „grün“ beschränkt sein sollten, ließe sich dies als Integritätsbedingung formulieren (genauer als lokale Wertebedingung), die automatisch überwacht wird.

Es kann daher von der Forderung nach strenger Einkapselung abgerückt werden, wenn die Konsistenz auch auf andere Weise zugesichert werden kann. In ESCHER$^+$ sind Integritätsbedingungen im Zusammenspiel mit Transaktionen wichtige Garanten für die Konsistenz der Daten. Es ist aber unrealistisch, alle Nebenbedingungen des zu modellierenden Umweltausschnittes in Form von automatisch überwachten Integritätsbedingungen ausdrücken zu wollen, auch wenn wir für ESCHER$^+$ das Spektrum der unterstützten Integritätsbedingungen über die in Abschnitt 5.3 eingeführten, vergleichsweise elementaren Integritätsbedingungen hinaus erweitern würden. Deshalb ist es notwendig, zumindest Teile eines Objekttyps einkapseln zu können und diese dann nur über Methoden modifizierbar zu machen, für die vorausgesetzt wird, daß sie konsistenzerhaltende Operationen sind.

Für Objekttypen in ESCHER$^+$ soll nun eine selektive Einkapselung eingeführt werden, d.h. die einzelnen Knoten des Strukturbaums eines Objekttyps werden als entweder öffentlich oder als privat (d.h. als eingekapselt) klassifiziert. Die öffentlichen Knoten werden wiederum hinsichtlich des ihnen zugeordneten Zugriffsmodus unterschieden, wobei dann zwischen „readonly“-Knoten und „read & write“-Knoten unterschieden wird. Genauer gehen wir wie folgt vor:

Es sei $t \in$ DEFTYPES gegeben. Für einen zu *struct*(t) gehörenden Strukturbaum $B_{struct(t)}$ führen wir zusätzlich die Beschriftung

$$enc: V(B_{struct(t)}) \rightarrow \{readwrite, read, private\}$$

ein. Dabei steht *enc* natürlich für „encapsulation“. Die Werte *readwrite*, *read* und *private* bezeichnen wir als *Zugriffsmodi*.

Als Beispiel betrachten wir die Objekttypdefinition des Typs `Person` aus Beispiel 3.7. In Abb. 5.1 ist der zugehörige Strukturbaum skizziert. Die fett und schwarz umrandeten Knoten r haben die Beschriftung *enc*(r) = *readwrite*, die fett und grau umrandeten die Beschriftung *enc*(r) = *read*, während die normal umrandeten Knoten r die Beschriftung *enc*(r) = *private* besitzen.

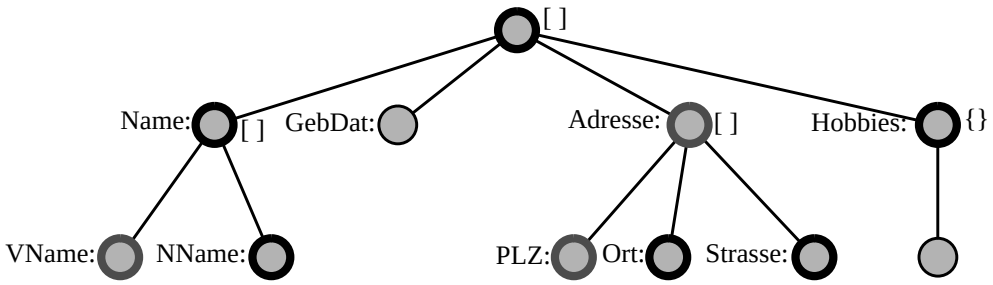


Abbildung 5.1: Der Strukturbaum zum Typ `Person` mit der Information über Verkapselung

In der textuellen Syntax wird die Art der Einkapselung durch die Schlüsselworte `read-write`, `read` oder `private` angezeigt, die in einen Typausdrück an den entsprechenden Positionen nachgestellt werden. Dabei ist `readwrite` der Default-Wert für den Fall, daß keines dieser Schlüsselwörter angegeben wird. Für das Beispiel in Abb. 5.1 lautet die um die Einkapselungsinformation erweiterte Strukturbeschreibung also:

```
[Name:[VName:string read, NName: string],
 GebDat: date private,
 Adresse:[PLZ:integer read, Ort:string,
          Strasse: string] read,
 Hobbies: { string private }]
```

Die Einteilung in *readwrite*, *read* und *private* für die Knoten eines Strukturbaumes hat für den Zugriff auf Instanzen des Objekttyps t folgende Konsequenzen:

- In der Ordnung *readwrite* > *read* > *private* drückt sich von links nach rechts eine immer stärkere Einschränkung des Zugriffs aus. Für jeden Knoten r' eines Wertebaums ist nicht der Wert *enc*(r) des zugeordneten Strukturbaumknotens r = *struct*(r') relevant, sondern der sog. *effektive* Zugriffsmodus. Dieser ist gegeben durch die stärkste Einschränkung des Zugriffs auf dem Weg vom Wurzelknoten des Strukturbaumes zum Knoten r.

Ist ein Knoten r in einem Strukturbaum z.B. mit *enc*(r) = *private* beschriftet, dann gelten auch alle Nachkommen dieses Knotens unabhängig von ihrer *enc*-Beschriftung als privat. In Abb. 5.1 ist der Knoten `Adresse` mit *read* beschriftet, d.h. auch für die Knoten `Ort` und `Strasse`, die selbst mit *readwrite* beschriftet sind, gilt der effektive Zugriffsmodus *read*. Wird durch eine Änderungsoperation (Beispiel folgt weiter unten) die Beschriftung für den Knoten `Adresse` auf *readwrite* gesetzt, dann kann sofort auf die Knoten `Ort` und `Strasse` zugegriffen werden. Ist der Wurzelknoten des Strukturbaums von t und aller seiner Supertypen mit *private* markiert, dann handelt es sich bei t um einen vollständig eingekapselten Objekttyp.

- Operationen aus USEROPS und Methoden anderer Objekttypen t'≠t dürfen lediglich auf solche Knoten r in den Zuständen von Objekten des Typs t zugreifen, deren zugehöriger Strukturknoten *struct*(r) effektiv öffentlich ist (d.h. für die der effektive Zugriffsmodus *read* oder *readwrite* ist). Die oben genannten Operationen bzw. Methoden dürfen nur dann den Knoten r direkt modifizieren, wenn für *struct*(r) der effektive Zugriffsmodus *readwrite* gilt. Dabei sprechen wir von einer direkten Modifikation von r, wenn auf r eine generische Update-Operation angewandt wird.
- Nur Methoden aus *ops*(t) dürfen auf *alle* Knoten in den Zuständen $I(t)(\omega)$ von Objekten ω lesend *und* schreibend zugreifen.
- Ist in einem Strukturbaum ein Kollektionsknoten öffentlich, der Sohnknoten dagegen privat, dann sind die Elemente der Kollektion nicht zugänglich. Es kann jedoch weiterhin die Kardinalität der Kollektion erfragt werden. In Abb. 5.1 trifft diese Situation auf den Mengenknoten `Hobbies` und seinen Sohnknoten zu.

Für den oben erwähnten Objekttyp `Quadrat` bietet sich an, den Zugriffsmodus für den gesamten Objektzustand auf *read* zu setzen, so daß die Koordinaten der Eckpunkte wenigstens gelesen werden können. Im Subtyp `FarbigesQuadrat` braucht für das Attribut `Farbe` keine Einschränkung des Zugriffsmodus angegeben werden, es gilt dann automatisch *readwrite*.

Die Modifikation der Einkapselung für einen Knoten ist über eine `modify`-Anweisung möglich, deren Syntax in Anhang B.2 angegeben ist (Nichtterminal *modEnc*). Die Anweisung

```
modify
  -structpath Person.Adresse
  -enc  readwrite
end modify;
```

setzt *enc*(r) := *readwrite* für den Knoten zum Attribut `Adresse` aus Abb. 5.1.

Im Zusammenhang mit Objekttypvarianten muß bei der Einkapselung von Attributen folgendes beachtet werden:

> Auch für die Attribute der `struct needed`-Klausel muß der Modus der Einkapselung angegeben werden, wobei jedoch *private* nicht in Frage kommt. Die Attribute, für die in der `struct needed`-Klausel *readwrite* angegeben wurde, müssen in allen Alternativen ebenfalls mit *readwrite* deklariert sein.

Ist diese Bedingung verletzt, dann kommt es zur Laufzeit möglicherweise zu einer unerlaubten direkten Modifikation eines Attributs in einer Alternative, bei der dieses Attribut als *read* oder gar als *private* deklariert wurde.

5.5.2 Einkapselung für Operationen

Ähnlich wie für Attribute von Objekttypen soll in ESCHER$^+$ auch für die Methoden eines Objekttyps eine Form von Einkapselung möglich sein. Wir schlagen vor, gleich bei der Definition einer Operation eine Klassifikation als private (*private*) oder öffentliche (*public*) Operation vorzunehmen.

Die Einteilung in *private* und *public* hat für Methoden folgende Bedeutung: Private Methoden sind nur von anderen Methoden desselben Objekttyps aufrufbar, nicht aber von anderen Operationen aus DEFOPS. Die Unterscheidung nach *private* und *public* bezieht sich also auf die Ausführungsrechte.

Die Klassifikation von Operationen nach *private* und *public* kann für die Belange der Konsistenzsicherung ausgenutzt werden: Operationen, für die (im Idealfall durch formale Verifikation) nachgewiesen wurde, daß sie konsistenzerhaltend sind, werden als *public* deklariert, während alle sonstigen Operationen *private* sind. Auf diese Weise läßt sich beispielsweise die topologische Konsistenz von geometrischen Objekten zusichern: Man modelliere sie als Instanzen eines Objekttyps und lasse ihre Modifikation nur über öffentliche Methoden zu, für die feststeht, daß sie die topologische Konsistenz erhalten.

Die formale Definition benutzerdefinierter Operationen (siehe Def. 4.5) wird um die Funktion

> *enc*: DEFOPS $\rightarrow$ { *public*, *private* }

ergänzt. In der `ops`-Klausel einer Objekttypdefinition kann für dort deklarierte Operationen spezifiziert werden, welches Ausführungsrecht für sie gelten soll. Es gelten die Syntaxregeln

> *opDecl* ::= *identifier* '`(`' *sign* '`)`' [*opEnc*]
>
> *opEnc* ::= `public` | `private`

In einer `define operation`-Anweisung wird das Ausführungsrecht über die `enc`-Klausel angegeben (vgl. die Syntax in Anhang B.2). Der Funktionswert *enc*(o) für eine neue Operation o wird dann entsprechend gesetzt. Der Default-Wert soll *public* sein.

Eine Änderung der Ausführungsrechte *public* oder *private* läßt sich auch für Operationen mittels einer `modify`-Anweisung durchführen. Hinsichtlich der Syntax sei wiederum auf Anhang B.2 verwiesen.

Im Zusammenhang mit Objekttypvarianten muß bei der Einkapselung von Methoden beachtet werden, daß die Methoden, die in der `ops needed`-Klausel genannt werden, in allen Alter-

nativen als *public* deklariert sind. Zur Laufzeit wird sonst ggf. versucht, eine Methode aufzurufen, die für die aktuell zutreffende Alternative als privat deklariert wurde.

5.6 Konsistenz von Datenbankschemata

Nach der Einführung des Meta-Schemas in Abschnitt 3.10 sind wir in der Lage, alle Datenbankschemata und die in ihnen enthaltene Information wie gewöhnliche Datenobjekte zu behandeln, da es sich um Instanzen von Objekttypen des Meta-Schemas handelt. Alle DDL-Anweisungen lassen sich auf Manipulationen dieser Instanzen zurückführen. Somit müssen auch alle DDL-Anweisungen im Rahmen einer Transaktion stattfinden.

Die Behandlung von Schemainformation wie gewöhnliche Daten bedeutet jedoch nicht, daß DDL-Anweisungen im Anweisungsteil von benutzerdefinierten Operation enthalten sein dürfen. Dies soll gerade nicht erlaubt sein! Jede benutzerdefinierte Operation muß sich darauf verlassen können, daß während ihrer Ausführung das Datenbankschema unverändert bleibt. Es soll also – wie üblich – deutlich zwischen der DDL und der DML unterschieden werden.

DDL-Anweisungen werden über eine besondere Schnittstelle zur Schemadefinition eingegeben. Im ESCHER-Prototyp erfolgt die Schemadefinition interaktiv über unterschiedliche Dialogfenster. Denkbar ist auch eine Kommandozeilen-Schnittstelle oder die Zusammenstellung aller DDL-Anweisungen in einer Textdatei. In jedem Fall müssen die DDL-Anweisungen in einer besonderen Transaktion, der *DDL-Transaktion*, zusammengefaßt werden. Syntaktisch wird eine DDL-Transaktion durch das Konstrukt

```
DDL_transaction begin
  ...
end DDL_transaction;
```

umschlossen. Während einer DDL-Transaktion kann die Konsistenz des aktuellen Datenbank-Schemas verletzt werden. Die Konsistenz wird erst am Ende der DDL-Transaktion überprüft. Nur so lassen sich z.B. zwei Objekttypen definieren, die sich in ihrem Strukturteil wechselseitig referenzieren.

Es ist denkbar, in einer Kommandozeilenschnittstelle eine einzelne DDL-Anweisung außerhalb eines `DDL-transaction`-Blocks zuzulassen und diese dann für sich als eine DDL-Transaktion zu betrachten, d.h. unmittelbar nach ihrer Abarbeitung wird die Konsistenz geprüft.

Hinsichtlich der Konsistenz eines Datenbankschemas sind im Verlauf dieser Arbeit bereits eine Vielzahl von Bedingungen genannt worden, die es einzuhalten gilt. Wir fassen sie an dieser Stelle noch einmal zusammen:

(i) *name*: TYPES $\cup$ TABLES $\rightarrow \mathcal{N}$ ist injektiv

(ii) Die Mengen BASETYPES, ENUMTYPES, VARTYPES, DEFTYPES und CONSTR sind paarweise disjunkt

(iii) $\forall$ t $\in$ DEFTYPES: $type(struct(t)) = c_{tuple}$

(iv) $\forall$ t $\in$ DEFTYPES $\forall$ o $\in$ ops(t): $args(sign(o))[1] = t$

(v) $\forall$ t $\in$ TABLES*: struct* (t) $\neq$ [ ]

(vi) *isa** ist eine Halbordung auf DEFTYPES

(vii) $\forall$ t $\in$ DEFTYPES: t *isa* t' $\wedge$ t *isa* t'' $\Rightarrow$ t' = t'' (einfache Vererbung)

(viii) $\forall$ t $\in$ VARTYPES: {s | (*l*, s) $\in$ *variants*(t)} – DEFTYPES $\neq \emptyset \Rightarrow$
struct_needed(t) = [ ] $\wedge$ *ops_needed*(t) = $\emptyset$

(ix) $\forall$ t $\in$ OBJVAR $\forall$ t' $\in$ {t' | (*l*, t') $\in$ *variants*(t)}:
– *struct*(t') $\leq_{tup}$ *struct_needed*(t)
– $\forall$ (m, s) $\in$ *ops_needed*(t) $\exists$ o $\in$ *ops*(t'): *name* (o) = m $\wedge$ *sign*(o) $\leq_{contrav}$ s

(x) $\forall$ o $\in$ DEFOPS: *conform* (*code*(o), *sign*(o))

Es besteht jedoch Bedarf an weiteren Bedingungen: Die Definition 3.17, in der es um die Strukturbeschreibung von Objekttypen geht, ist insofern unvollständig, als sie gewisse „unsinnige“ Strukturbeschreibungen noch nicht ausschließt. Im folgenden Beispiel werden die Fälle genannt, die hinsichtlich der Zulässigkeit von Objekttypdefinitionen berücksichtigt werden sollen.

Beispiel 5.9

(i) Die Definition des Objekttyps `Buch` aus Beispiel 3.6 ist nur dann konsistent, wenn alle in der `struct`-Klausel verwendeten Typnamen tatsächlich auch in TYPES vorkommen. Der Typ `Autor` muß daher auch als Objekttyp definiert sein, d.h. es ist für die *Abgeschlossenheit* der Typdefinitionen zu sorgen.

(ii) Sicher nicht erwünscht sind Typdefinitionen mit *struct* (t_{Person}) = [P : t_{Person}] oder *struct*(t_{Person}) = [P : {t_{Person}}]. In beiden Fällen wird versucht, den Typ t_{Person} allein durch sich selbst zu erklären. Nicht sinnvoll sind auch zirkuläre Definitionen von Objekttypen, wie z.B. t_{Beruf}, $t_{Job} \in$ DEFTYPES mit *struct* (t_{Beruf}) = [B : t_{Job}] und *struct* (t_{Job}) = [J : t_{Beruf}]. Es muß eine Bedingung gefunden werden, die sicherstellt, daß ein „abstrakter“ Typ t $\in$ DEFTYPES auf eine „konkrete“ Beschreibung zurückgeführt werden kann. Für „konkrete“ Beschreibungen stehen die Basis- und Aufzählungstypen (*printable types*) zur Verfügung. Es ist daher zu fordern, daß jeder Objekttyp auf zumindest einen *printable type* zurückgeführt werden kann.

(iii) Ebenfalls nicht sinnvoll ist t_{Beruf}, $t_{Job} \in$ DEFTYPES mit *struct* (t_{Beruf}) = [B : t_{Job}] und *struct* (t_{Job}) = [Name: β_{string} , Beschreibung: β_{string} , Gehalt: $t_{integer}$], da die „Indirektion“ über t_{Job} überflüssig ist. Ist jedoch t $\in$ DEFTYPES hinsichtlich der *isa*-Beziehung ein Subtyp eines anderen Typs t', dann soll es möglich sein, den „Differenztyp“ t durch ein einzelnes (Attribut, Objekttyp)-Paar zu beschreiben.

(iv) Für die Strukturbeschreibung von Objekttypen wurde das leere Tupel [ ] zugelassen. Damit ist es möglich, daß für einen Subtyp keine weiteren Attribute definiert werden müssen, falls lediglich eine Methode redefiniert oder neu hinzugefügt werden soll. Für Wurzeltypen der *isa*-Hierarchien soll eine Strukturbeschreibung durch das leere Tupel jedoch verboten sein, denn jeder Objekttyp soll mindestens ein Attribut besitzen. Gleichzeitig ist dafür zu sorgen, daß für Objekttypen t mit *struct*(t) = [ ] die Menge *ops*(t) nicht leer ist, d.h. es muß mindestens eine Methode neu hinzugefügt oder eine Methode redefiniert werden.

❒

Es sind also zusätzliche Konsistenzbedingungen notwendig, damit DEFTYPES eine zulässige Menge von Objekttypen ist. Bevor diese Konsistenzbedingungen angegeben werden können, werden einige Hilfsmittel bereitgestellt. Zunächst sei *usedTypes*(s) die Menge aller in s verwendeten Typen aus TYPES, d.h. für $s \in \mathcal{S}(\text{TYPES})$ sei

$$\textit{usedTypes}(s) := \begin{cases} s & \text{, falls } s \in \text{BASETYPES} \cup \text{ENUMTYPES} \cup \text{DEFTYPES} \\ \textit{usedTypes}(s') \cup \{c\} & \text{, falls } s = c(s') \text{ mit } c \in \{c_{set}, c_{bag}, c_{list}\} \\ (\bigcup_{i=1}^{n} \textit{usedTypes}(s_i)) \cup \{c_{tuple}\} & \text{, falls } s = [\, a_1 : s_1, \ldots, a_n : s_n \,] \\ \textit{usedTypes}(s') \cup \{c_{array}\} & \text{, falls } s = (s' \mid n_1 : m_1, \ldots, n_k : m_k) \\ (\bigcup_{i=1}^{n} \textit{usedTypes}(s_i)) \cup \{s\} & \text{, falls } s \in \text{VARTYPES mit} \\ & \quad \textit{variants}(s) = \{(l_1 : s_1), \ldots, (l_n : s_n)\} \end{cases}$$

Wir definieren nun die Relation $\textit{compoType} \subseteq \text{TYPES} \times \text{TYPES}$ durch

$(t, t') \in \textit{compoType} \Leftrightarrow$
$t \in \text{DEFTYPES} \wedge t' \in \bigcup \textit{usedTypes}(\textit{struct}(t''))$,
wobei die Vereinigung über alle Typen t'' mit t *isa** t'' läuft.

Ein Objekttyp t steht demnach zu allen Typen t' in der Relation *compoType*, die bei der Konstruktion der Struktur von t oder der Struktur eines Supertyps t'' verwendet wird, d.h. es wird die Vererbung von Eigenschaften berücksichtigt.

Mit $\textit{compoType}^+$ werde die transitive Hülle der Relation *compoType* bezeichnet.

Nach diesen Vorbereitungen können nun die Konsistenzbedingungen für die Definition von Objekttypen in ESCHER$^+$ angegeben werden.

Definition 5.2 (Konsistenz der Objekttypdefinitionen)

Es sei die Menge DEFTYPES eines Datenbankschemas gemäß Def. 3.20 gegeben.
DEFTYPES wird *konsistent* (oder *zulässig*) genannt, wenn die folgenden Bedingungen erfüllt sind:

(i) $\forall\, t \in \text{DEFTYPES}: \textit{usedTypes}(\textit{struct}(t)) \subseteq \text{TYPES}$

(ii) $\forall\, t \in \text{DEFTYPES}:$
$\{t' \in \text{TYPES} \mid (t, t') \in \textit{compoType}^+\} \cap (\text{BASETYPES} \cup \text{ENUMTYPES}) \neq \emptyset$

(iii) $\forall\, t \in \text{DEFTYPES}: \textit{usedTypes}(\textit{struct}(t)) = \{t'\} \Rightarrow t' \notin \text{DEFTYPES}$

(iv) $\forall\, t \in \text{DEFTYPES}: \textit{struct}(t) = [\,] \Rightarrow \textit{ops}(t) \neq \emptyset$

Die Bedingungen (i) – (iv) verhindern gerade die unerwünschten Situationen (i) – (iv) aus Beispiel 5.9. ❒

In [The93] wurde für ESCHER eine erste Erweiterung des eNF^2-Datenmodells um Objekttypen vorgeschlagen, wobei diese jedoch auf strukturelle Objektorientierung ohne Vererbung beschränkt war. Es wurde der Begriff *„zulässiges Typsystem“* für die Gesamtheit der Objekttypdefinitionen eingeführt. Nach [The93] muß ein zulässiges Typsystem Bedingungen erfüllen, die mit (i) und (ii) aus Def. 5.2 vergleichbar sind. Von Vossen und Witt wurden in [VW91] ebenfalls mit (i) und (ii) vergleichbare Konsistenzbedingungen für eine Menge von Typdefinitionen, die dort *Formate* genannt werden, angegeben. Sie berücksichtigen jedoch keine *isa*-Beziehung zwischen Objekttypen. Ihre Bedingungen können zudem nicht die Unzulässigkeit von $\textit{struct}(t_{Person}) = [P : \{t_{Person}\}]$ (siehe Fall (ii) in Beispiel 5.9) erkennen. Die Bedingungen

aus Definition 5.2 sind daher stärker und berücksichtigen zudem noch eine *isa*-Beziehung zwischen Objekttypen.

Die Forderung nach Abgeschlossenheit der Menge der Objekttypdefinitionen (Bedingung (i) in Def. 5.2) ist insbesondere im Zusammenhang mit dem Entfernen von Typdefinitionen über die DDL-Anweisung `drop type` von Bedeutung. Bedingungen für die Abgeschlossenheit eines Datenbankschemas müssen auch für andere Elemente eines Datenbankschemas angegeben werden: Das Löschen einer Typdefinition aus einem Datenbankschema kann beispielsweise eine benutzerdefinierte Operation $o \in$ DEFOPS invalidieren, die den Typ in ihrer Signatur oder in ihrer „Implementation“ *code*(o) verwendet. Auch das Löschen von Tabellen oder das Löschen einer Alternative eines varianten Typs – um nur einige der kritischen Schema-Modifikationen zu nennen – können die Abgeschlossenheit und somit einen wichtigen Aspekt der Konsistenz eines Datenbankschemas gefährden. Auf diese im Zusammenhang mit Schema-Evolution stehenden Probleme soll in dieser Arbeit jedoch nicht im einzelnen eingangen werden. Es sei auf [TS92] sowie auf die Arbeiten zum GOODSTEP-Projekt [GOO95] hingewiesen, die sich intensiv mit dem Thema Schema-Evolution in vergleichbaren Datenmodellen auseinandersetzen.

5.7 Zusammenfassung und Diskussion

In diesem Kapitel wurden Konzepte zur Konsistenzsicherung im ESCHER$^+$-Datenmodell vorgeschlagen.

Zunächst wurden flache Transaktionen in das formale Datenmodell eingeführt. Bei einer Implementation des Datenmodells ist zu erwägen, ob anstelle des flachen Transaktionsmodells nicht gleich geschachtelte Transaktionen berücksichtigt werden sollten. Letztere gehen auf Moss [Mos85] zurück und besitzen gegenüber flachen Transaktionen eine Reihe von Vorteilen (siehe auch [GR93]): Besteht eine Transaktion aus mehreren Arbeitsschritten, so ist es in vielen Situationen erwünscht, gewisse Teilschritte rückgängig zu machen, ohne gleich die gesamte Transaktion, d.h. alle bis dahin getätigten Arbeitsschritte, komplett zurückzusetzen. Bei der Zusammenstellung eines Reisepakets kann beispielsweise die Umbuchung eines Hotels durchgeführt werden, wobei jedoch bereits reservierte Flüge davon unberührt bleiben sollen. Dies wird von geschachtelten Transaktionen unterstützt, indem die bereits abgeschlossene Subtransaktion „Hotelbuchung“ wieder zurückgesetzt wird. In diesem Transaktionsmodell sind Transaktionen benamt, damit eine bereits durch Commit abgeschlossene Subtransaktion zu einem späteren Zeitpunkt mit Hilfe des Namens gezielt wieder rückgängig gemacht werden kann. Nach einem Commit einer Subtransaktion sind die von ihr gemachten Änderungen in der unmittelbar umgebenden Transaktion desselben Benutzers sichtbar, in allen anderen Transaktionen – insbesondere denen anderer Benutzer – nicht. Ein Abbruch einer umgebenden Transaktion führt zu einem Abbruch der in ihr enthaltenen noch aktiven Transaktionen bzw. zu einem Zurücksetzen (Undo) der gemachten Änderungen von in ihr enthaltenen Subtransaktionen, für die bereits ein Commit durchgeführt wurde.

In Abschnitt 5.3 wurden mit Schlüssel-, Inklusions-, Disjunktheits- und lokalen Wertebedingungen wichtige Klassen deklarativer Integritätsbedingungen eingeführt, die zur Laufzeit automatisch überwacht werden. In den Arbeiten zum eNF^2-Datenmodell und seinen Erweite-

rungen (vgl. [Pau94]) wurden Integritätsbedingungen bislang stark vernachlässigt. Für das Relationenmodell werden im Rahmen des SQL/92-Standards [DD93] bereits eine Vielzahl von Integritätsbedingungen unterstützt. Es stellte sich daher die Aufgabe, für unser wesentlich komplexeres Datenmodell geeignete Integritätsbedingungen anzubieten, die automatisch überwacht werden. Wir haben uns in dieser Arbeit auf vergleichsweise elementare Integritätsbedingungen beschränkt. Das Spektrum denkbarer Integritätsbedingungen ist natürlich viel größer. Unser Sprachvorschlag zur Spezifikation von Integritätsbedingungen auf Schemaebene ließe sich jedoch leicht erweitern. Als allgemeine Integritätsbedingung könnte – wie im SQL/92-Standard – jeder boolesche Ausdruck der Anfragesprache gelten. Dabei sind insbesondere quantorisierte Ausdrücke der Gestalt (4.15) relevant (siehe Abschnitt 4.7.4). Als Beispiel sei die Integritätsbedingung

```
all a in Abteilungen, exists m in a.Mitarbeiter:
"C++" in m.Kenntnisse
```

genannt. Sie fordert, daß es in jeder Abteilung einen Mitarbeiter mit C++-Kenntnissen geben soll. Daneben wäre die Syntax auch einfach zu erweitern, um transitionale Integritätsbedingungen, wie z.B. die Bedingung „Das Gehalt eines Angestellten darf nicht sinken“, zu erfassen. So ist

```
new(g) >= old(g) with g = Angestellter.Gehalt
```

eine transitionale lokale Wertebedingung, wobei `old(g)` für das Gehalt eines Angestellten zu Beginn der Transaktion steht.

In Abschnitt 5.4 wurde vorgeschlagen, einen Integritätsmonitor zu implementieren, der auf der Grundlage interner ECA-Regeln in der Lage ist, die Integritätsbedingungen aus Abschnitt 5.3 zur Laufzeit automatisch zu überwachen. Dazu werden die Integritätsbedingungen zur Entwurfszeit, d.h. bei der Abarbeitung einer DDL-Transaktion, in interne ECA-Regeln transformiert. Bei diesem Transformationsprozeß muß für jede Integritätsbedingung zunächst ermittelt werden, welche Ereignisse ihre Überprüfung notwendig machen. Ereignisse werden durch die Ausführung generischer Update-Operationen ausgelöst. Das spezielle Ereignis `check` ist im Zusammenhang mit der Konsistenzprüfung für ein komplexes Datenobjekt relevant, das bisher transient war und durch die Ausführung einer Update-Operation Persistenz erlangt. Da für das bislang transiente Datenobjekt nun möglicherweise weitere Konsistenzbedingungen gelten müssen, die sich aus der neuen „Position“ des Datenobjektes innerhalb eines persistenten Datenobjektes ergeben, müssen die Prüfungen dieser Bedingungen nun nachgeholt werden. Dazu besucht ein sog. Check-Finger alle Knoten des Wertebaumes des betreffenden Datenobjektes und aktiviert dabei über das Ereignis `check` überall dort eine ECA-Regel, wo eine potentielle Integritätsverletzung vorliegt. Bei der Transformation von Integritätsbedingungen in ECA-Regeln müssen für die *condition*-Komponenten Prüfbedingungen generiert werden, die effizient ausgewertet werden können. Dabei ist die Prüfbedingung abhängig von dem auslösenden Ereignis und vom Zeitpunkt der Überprüfung (sofortige oder zurückgestellte Prüfung). Es wurde gezeigt, daß zur effizienten Überprüfung einiger Integritätsbedingungen interne Hilfsstrukturen notwendig sind, die – vereinfacht gesagt – Rückwärtsverweise von Objekten ω auf Knoten in Wertebäumen liefern. Der Zugriff auf diese Hilfsstrukturen erfolgt in der sog. *prepare*-Komponente einer ECA-Regel.

Wie bereits in Abschnitt 2.4.3.2 diskutiert, werden ECA-Regel-Subsysteme zur Kontrolle von Integritätsbedingungen bereits in mehreren Prototypen eingesetzt [CW90, JQ92, GJ91, FPT93,

CFP+94, GD95]. In [Esw76] wurde für das Relationenmodell ein früher Vorschlag in dieser Richtung unterbreitet. Dort wird von einem Trigger-Subsystem gesprochen. In den 70er und frühen 80er Jahren ging es in zahlreichen Arbeiten zur Integritätssicherung im Relationenmodell um die Fragestellung, wie aus deklarativen Integritätsbedingungen oder Integritätsregeln Code erzeugt werden kann, der zur Übersetzungszeit in ein Anwendungsprogramm „eingebaut“ wird. Das bedeutet, daß die Integritätskontrolle vollständig durch das kompilierte Anwendungsprogramm selbst durchgeführt wird. Es ist dann kein eigenständiger Integritätsmonitor vorgesehen, der sich zur Laufzeit aufgrund eines Ereignisses „einschaltet“ und für die Dauer der Integritätsprüfung das laufende Programm „anhält“. Zu nennen sind zu diesem Ansatz u.a. die Arbeiten [Sto75] und [Web81]. Auch in [Web81] wird von einem „Monitor“ gesprochen, allerdings bezieht sich der Begriff dort auf das Überwachen des Einfügens von Prüfroutinen in ein Anwendungsprogramm zur Übersetzungszeit. Der Ansatz aus [Sto75], der für das relationale DBMS Ingres auch implementiert wurde, sieht vor, Update-Operationen so zu transformieren, daß sie den Test auf mögliche Integritätsverletzungen selbst enthalten und somit nur dann eine Wirkung haben, wenn sie keine Integritätsbedingung verletzen. Eine neuere Arbeit, die sich mit dem „Hineinkompilieren“ von Integritätstests in Transaktionen auseinandersetzt, ist [BD95], in der Integritätsbedingungen für das das objektorientierte Datenmodell O_2 untersucht werden.

Für den ESCHER-Prototyp ist ein separates, ereignisorientieres Regel-Subsystem besonders interessant, da die Arbeit mit der generischen Browser-Oberfläche selbst ereignisorientiert ist. Es steht nicht ein fester Programmablauf im Vordergrund, wie er traditionell durch ein Anwendungsprogramm gegeben ist, in das alle Integritätsprüfungen „hineinkompiliert“ werden können. Vielmehr löst der Benutzer auf der Oberfläche (durch Tastendrücke, Mausklicks usw.) bestimmte Ereignisse aus, auf die das DBMS reagieren soll. Es wäre eine ESCHER$^+$-spezifische Regelsprache denkbar, über die sich beispielsweise benutzerdefinierte Operationen an Ereignisse der Benutzeroberfläche binden lassen, ähnlich wie beispielsweise unter Motif [Hel93] sog. „callback“-Routinen an Ereignisse gebunden werden. Wie bereits in Abschnitt 2.4.3.2 erwähnt, stellen Regeln ein vielseitig einsetzbares Konzept dar. Eine zunächst nur für die Integritätssicherung vorgesehene Komponente zur Überwachung interner ECA-Regeln ist die geeignete Grundlage für künftige Erweiterungen.

Bisher sind die Ausdrucksmittel der Sprache SCRIPT$^+$ zu gering, um innerhalb eines Scriptes auf das Zurücksetzen einer Transaktion infolge einer Integritätsverletzung geeignet reagieren zu können. Es gibt nicht einmal die Möglichkeit zu einer Abfrage, ob die letzte Transaktion erfolgreich beendet oder zurückgesetzt wurde (etwa in der Form "`if aborted then` …"). Wir wollen an dieser Stelle kurz skizzieren, wie eine Behandlung von Integritätsverletzungen im Anschluß an eine Transaktion in die Syntax von SCRIPT$^+$ integriert werden kann. Dazu erweitern wir die Syntaxregel für das Nichtterminal *transaction* aus Anhang B.3 zu

transaction ::= `transaction begin`
statements
`end transaction`
[`handle violation of`
identifier `':'` *statements* { *identifier* `':'` *statements* } [`else` *statements*]
`end handle`]

In dem optionalen `handle violation`-Konstrukt wird jeder *identifier* durch den Namen einer Integritätsbedingung ersetzt. Wird eine Transaktion nicht erfolgreich abgeschlossen,

dann wird der `handle violation`-Teil ausgewertet. Erfolgte das Zurücksetzen der Transaktion aufgrund der Verletzung der Integritätsbedingung mit dem Namen n, dann wird – sofern vorhanden – die dem Namen n zugeordnete Anweisungsfolge abgearbeitet, ansonsten – falls vorhanden – die Anweisungsfolge des `else`-Zweiges.

Um über die automatische Überwachung elementarer Integritätsbedingungen hinaus der Forderung nach Konsistenz gerechtzuwerden, benötigen wir für ESCHER$^+$ ein Pendant zu den in Abschnitt 5.4 erwähnten sicheren Transaktionen. In ESCHER$^+$ sind Transaktionen zwar die atomaren Einheiten hinsichtlich der Konsistenzerhaltung, nicht jedoch die „Aufrufeinheit“: Unser Modell sieht nicht den Aufruf von Transaktionen vor, sondern die „Aufrufeinheit“ ist eine benutzerdefinierte Operation aus DEFOPS. Die Rolle von „sicheren Transaktionen“ können in unserem Modell jedoch besonders ausgezeichnete Operationen aus DEFOPS übernehmen. Nach Abschnitt 5.5.2 werden benutzerdefinierte Operationen in die Gruppen *private* und *public* eingeteilt. Als *public* sollten nur solche Operationen gelten, die nachweisbar konsistenzerhaltend sind und somit als „sichere Transaktionen“ gelten können. Ergänzt wird die Integritätssicherung durch die Möglichkeit zur Einkapselung von Objekttypen. Nach Abschnitt 5.5.1 ist eine selektive Einkapselung bzw. der Entzug des direkten Schreibrechts für ausgewählte Attribute eines Objekttyps möglich.

Kapitel 6

Beziehungen

Unter den in Abschnitt 2.1.1 genannten Abstraktionsmechanismen der semantischen Datenmodellierung wurde u.a. die *Assoziation* genannt. Darunter versteht man eine vergleichsweise „lockere“ Beziehung zwischen Anwendungsobjekten, die vielfach mit der Aggregation gleichgesetzt wird. Die Aggregation drückt jedoch eine wesentlich stärkere Bindung zwischen Anwendungsobjekten aus. Typischerweise gilt dies für besonders „starke“ Beziehungen, wie die „is part of“-Beziehung, bei der ein Objekt aus mehreren Komponentenobjekten besteht, die oftmals von ihrem umgebenden Objekt existenzabhängig sind. Wie bereits in Abschnitt 2.1.1 erwähnt, ist die Grenze zwischen „lockeren“ Beziehungen, die als Assoziationen gelten sollten, und besonders „starken“ Bindungen zwischen Objekten, die eine Aggregation darstellen, fließend.

In Abschnitt 6.1 diskutieren wir die verschiedenen, in der Datenmodellierung verbreiteten Möglichkeiten zur Erfassung von Beziehungen (im Sinne von Assoziationen) innerhalb eines Schemas. Dabei wird sich herausstellen, daß es durchaus problematisch ist, alle Arten von Beziehungen zwischen Anwendungsobjekten allein über die Aggregation zu modellieren, wie dies in vielen Datenmodellen üblich ist.

In Abschnitt 6.2 wird der in [The95] vorgestellte und in [The96] weiterentwickelte Ansatz zur Behandlung von Beziehungen im ESCHER$^+$-Datenmodell präsentiert. Neben Typen, Tabellen und Operationen tritt dabei die *Beziehung* als ein weiteres fundamentales Konstrukt des Datenmodells hinzu. Damit ist eine direkte Modellierung von Assoziationen in einem ESCHER$^+$-Datenbankschema möglich. Mit jeder Beziehung ist eine Menge von *Beziehungssichten* verknüpft, in denen für die Komponenten einer Beziehung festgelegt wird, auf welche Weise der Zugriff auf die Beziehungsinstanzen aus der Perspektive der jeweiligen Komponente erfolgen soll. Damit schaffen wir den Ausgleich zwischen einer „neutralen“ Definition einer Beziehung,

wie sie etwa auch beim ER-Modell vorzufinden ist, und „gerichteten Sichten“ auf eine Beziehung, die jeweils eine Komponente der Beziehung als ihren „Ausgangspunkt“ nehmen. Jede Sicht steht dem einer Komponente zugeordneten Objekttyp in Form von zusätzlichen Attributen zur Verfügung. Die formalen Definitionen zu den vorgenommenen Modellerweiterungen folgen in Abschnitt 6.3.

In Abschnitt 6.4 werden Integritätsbedingungen für Beziehungen eingeführt. Neben den besonders wichtigen Schlüsselbedingungen werden Kardinalitätsbedingungen, Wertebedingungen für die Komponenten und Attribute einer Beziehung, sowie Inklusionsbedingungen behandelt.

In Abschnitt 6.5 geben wir formale Bedingungen für die Konsistenz einer Datenbankinstanz hinsichtlich der im Datenbankschema definierten Beziehungen an. Da sich verschiedene Beziehungssichten zu einer Beziehung i.d.R. „überlappen“ und auf der Instanzenebene folglich Redundanz vorliegt, müssen besondere Konsistenzbedingungen dafür sorgen, daß man beim Zugriff über unterschiedliche Beziehungssichten keine zueinander widersprüchliche Information über die Beziehung erhält. Zur Sicherung der Konsistenz beim Vorliegen von sich überlappenden Beziehungssichten schlagen wir in Abschnitt 6.6 eine Überwachung auf der Basis intern generierter ECA-Regeln vor, wie dies bereits im vorangegangenen Kapitel für die Überwachung sonstiger Integritätsbedingungen der Fall war. In Abschnitt 6.7 werden die Ergebnisse dieses Kapitels zusammengefaßt und offene Fragen diskutiert.

Es sei darauf hingewiesen, daß von nun an die Begriffe „Beziehung“ und „Assoziation“ synonym verwendet werden.

6.1 Die Behandlung von Beziehungen in Datenmodellen

6.1.1 Ein einführendes Beispiel

Im Rahmen der semantischen Datenmodellierung wird vielfach noch zwischen der Aggregation und der Assoziation differenziert. Beim ER-Modell [Che76], dem klassischen Vertreter der semantischen Datenmodellierung, tritt die direkte Modellierung von Assoziationen in Form der *relationship types* am deutlichsten in Erscheinung. Beim Übergang von einem konzeptuellen zu einem logischen Schema (d.h. beim Datenbankentwurf) verschwindet üblicherweise die Unterscheidung zwischen Assoziation und Aggregation. Die logischen Datenmodelle stellen i.d.R. kein eigenes Konstrukt für die Assoziation zur Verfügung und behandeln Assoziationen wie Aggregationen. Dies gilt auch für das NF^2- und das eNF^2-Datenmodell, deren Vorzüge in erster Linie bei der Modellierung streng hierarchischer Anwendungsstrukturen liegen, für die die Aggregation ja auch das adäquate Modellierungskonstrukt ist.

Es folgt ein erstes konkretes Beispiel einer anwendungsspezifischen Beziehung, das im weiteren immer wieder aufgegriffen wird.

Beispiel 6.1

Zwischen Objekten der Typen `Person` und `Kurs` bestehe eine Beziehung `Teilnahme`. Über sie wird ausgedrückt, daß eine bestimmte Person an einem bestimmten Kurs teilnimmt

oder teilgenommen hat. Es handelt sich bei diesem Beispiel um eine binäre Beziehung mit den Komponenten `P` und `K` und zwei Attributen, nämlich `Anm` (für das Datum der Anmeldung für den Kurs) sowie `Erg` (für das Ergebnis eines Kurs-Tests , z.B. die in einer Klausur zum Kurs erreichte Punktzahl).

In Abb. 6.1 ist die Beziehung `Teilnahme` anhand eines ER-Diagramms veranschaulicht. Die Attribute der Objekttypen `Person` und `Kurs` werden nicht gezeigt.

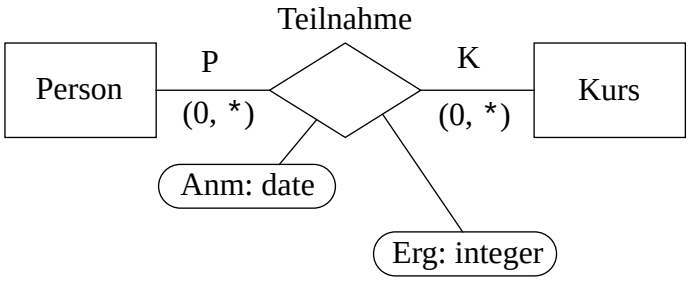


Abbildung 6.1: Die binäre Beziehung `Teilnahme` mit den Attributen `Anmeldung` und `Ergebnis`

Bei `Teilnahme` handelt es sich um eine Assoziation im Sinne einer "lockeren" Beziehung zwischen Instanzen der Typen `Person` und `Kurs`. Im Sinne des ER-Modells ist eine Instanz der Beziehung `Teilnahme` gegeben durch ein Tupel mit der Struktur

`[P: Person, K: Kurs, Anm: date, Erg: integer]`.

Die Kardinalitäten in (min, max)-Notation zeigen an, daß es sich um eine *m:n*-Beziehung handelt. Aufgrund der angegebenen Minimalkardinalitäten können Personen und Kurse existieren, ohne daß sie Komponenten einer Beziehungsinstanz sein müssen, d.h. die Teilnahme von `Person`- bzw. `Kurs`-Objekten an der Beziehung ist optional. ❒

Für Beziehungen ist charakteristisch, daß sie den beteiligten Komponententypen nachgeordnet sind in dem Sinne, daß für die Komponenten einer neuen Beziehungsinstanz immer auf bereits existierende Objekte zurückgegriffen werden muß und jede Beziehungsinstanz von ihren Komponenten existenzabhängig ist. Umgekehrt jedoch ist es – wie Beispiel 6.1 gezeigt hat – sehr wohl möglich, daß Objekte existieren, ohne Komponenten einer Beziehungsinstanz zu sein. Im Falle der Beziehung `Teilnahme` wäre z.B. die Existenzabhängigkeit einer Person von der Teilnahme an mindestens einem Kurs keine vernünftige Forderung.

Eine allgemeine Beziehung der Kardinalität n mit m Attributen ist in Abb. 6.2 in der graphischen Notation der ER-Modellierung abgebildet. Genauer genommen müßten wir von einem Beziehungs*typ* sprechen, der durch die Tupelstruktur

$$[c_1: t_1, \ldots, c_n: t_n, a_1: s_1, \ldots, a_m: s_m] \qquad (6.1)$$

beschrieben ist. Die Entity-Typen t_i sind die „eigentlichen“ Komponenten einer Beziehung, denen jeweils ein Komponentenname[1] c_i zugeordnet wird und deren Anzahl n die Kardinalität der Beziehung bestimmt. Die Attribute a_j dienen zur näheren Beschreibung einer Beziehungs-

1. In der Literatur werden die c_i vielfach auch als *Rollennamen* bezeichnet.

instanz. Ihnen ist im ER-Modell jeweils ein Basistyp zugeordnet. Eine *Beziehungsinstanz* ist durch ein Tupel der Struktur (6.1) gegeben.

Wir sagen, daß ein Objekt ω an der Beziehung R in der Rolle c_i teilnimmt, wenn es eine Beziehungsinstanz t der Struktur (6.1) und $t.c_i = \omega$ gibt.

Soll im Rahmen des Datenbankentwurfs ein ER-Schema in ein ESCHER$^+$-Datenbankschema transformiert werden, so bietet sich zunächst eine triviale Lösung an, bei der für jede Beziehung `R` einfach eine ESCHER$^+$-Tabelle mit der Struktur

$$\{[c_1: t_1, \ldots, c_n: t_n, a_1: s_1, \ldots, a_m: s_m]\} \tag{6.2}$$

erzeugt wird, in der alle Beziehungsinstanzen zusammengefaßt werden. Die so entstehende *Beziehungstabelle* ist also eine „flache“ Relation.

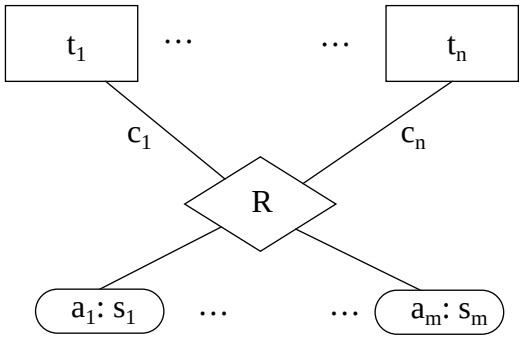


Abbildung 6.2: Graphische Repräsentation einer allgemeinen Beziehung im ER-Modell

Für die Beziehung `Teilnahme` aus Beispiel 6.1 könnten daher alle Beziehungsinstanzen in einer ESCHER$^+$-Tabelle `Teilnahme` mit der Struktur

```
{[P: Person, K: Kurs, Anmeldung: date, Ergebnis: integer]}
```

zusammengefaßt werden. Für die Tabelle `Teilnahme` muß außerdem die Schlüsselbedingung "`Teilnahme has key [P, K]`" angegeben werden, wenn man ausschließen möchte, daß sich eine Person mehrfach für denselben Kurs anmeldet.

Ist man an den Kursen interessiert, die eine Person namens „Müller“ besucht hat, so lautet die dazugehörige Anfrage

```
{t.K.Titel | p in Personen, t in Teilnahme:
 p.Name.NName = "Müller" and p = t.P}
```

Diese Lösung ist unbefriedigend, da in der Anfrage die explizite Nennung der „Verknüpfungstabelle“ `Teilnahme` erforderlich ist. Die Auswertung der Anfrage erfordert einen Join der beiden Tabellen `Personen` und `Teilnahme` (vgl. die Bedingung `p = t.P`).

Man erkennt, daß der gerade geschilderte Ansatz genau der Modellierung einer *m:n*-Beziehung entspricht, wie man sie im (flachen) Relationenmodell durchführen würde. Allerdings ist bekannt, daß bei der Transformation eines ER-Schemas in ein relationales Datenbankschema nach Möglichkeit auf die Einführung von separaten Beziehungstabellen als „Verknüpfungsrelationen“ verzichtet wird, wenn es sich um eine *1:1*- oder *1:n*-Beziehung handelt. In diesen Fällen wird die Tupelstruktur derjenigen Relationen, die Entity-Typen repräsentieren, um entsprechende Attribute und Fremdschlüsselbedingungen ergänzt.

In Datenmodellen für komplexe Objekte bieten geschachtelte Strukturen und insbesondere mengenwertige Attribute weitere Möglichkeiten zur Abbildung von Beziehungen innerhalb eines Datenbankschemas an. In Modellen, die die Definition von Objekttypen zulassen, ist es üblich, eine Beziehung über zusätzliche Attribute in die Struktur derjenigen Objekttypen zu integrieren, die Komponenten der Beziehung sind. Auch für ESCHER$^+$ bietet sich diese Möglichkeit an, und für unser Beispiel könnte dies z.B. in der folgenden Form geschehen:

Der Objekttyp `Person` habe die Struktur

```
[Name: ..., Adresse: ..., ..., Kurse: {Kurs}],
```
(6.3)

Das Attribut `Kurse` gibt die Menge der Kurse an, die eine Person besucht hat. Es wird angenommen, daß die Attribute `Anm` und `Erg` für diese „Sicht“ nicht von Interesse sind und deshalb nicht auftauchen.

Der Objekttyp `Kurs` habe die Struktur

```
[Titel: ..., Termin: ..., ..., T_Liste:
 <[Teilnehmer: Person, Anmeldung: date,
   Ergebnis: integer]>]
```
(6.4)

Das Attribut `T_Liste` gibt für einen Kurs die Liste aller Teilnehmer an.

Die Modellierung der Beziehung `Teilnahme` mittels (6.3) und (6.4) bezeichnen wir als *verteilte Aggregation*:

- Der Zustand der Objekttypen, die Komponenten einer Beziehung sind, ist eine Aggregation aus Attributen, die den Objekttyp „an sich“ beschreiben (wie z.B. Name und Adresse einer Person) und Attributen, die die Beziehungen zu anderen Objekten angeben.
- Die Aggregation ist *verteilt*, da es Attribute in verschiedenen Objekttypdefinitionen gibt, die zu ein und derselben Beziehung gehören. In unserem Beispiel existiert keine explizite Definition der Beziehung `Teilnahme`, vielmehr ist sie in verschiedenen Typdefinitionen „versteckt“.

Auf der Basis der Objekttypdefinitionen (6.3) und (6.4) vereinfacht sich die oben genannte Anfrage zu

```
{ k.Titel | p in Personen, k in p.Kurse:
  p.Name.NName = "Müller"}
```

Der „Umweg“ über eine Beziehungstabelle als „Verknüpfungstabelle“ ist nicht mehr erforderlich. Vielmehr kann jetzt *direkt* über ein Attribut des Objekttyps `Person` auf alle mit `p` in Beziehung stehenden Kurse zugegriffen werden, und dieses Attribut steht auch für Methoden des Objekttyps zur Verfügung. Dies ist ein wichtiger Vorteil der verteilten Aggregation gegenüber der Modellierung über eine separate Beziehungstabelle.

Bei den Attributen `Kurse` und `T_Liste` in (6.3) bzw. (6.4) handelt es sich um spezielle *Sichten* auf die „symmetrische“ Beziehung `Teilnahme`. Die „Symmetrie“ einer Beziehung besteht darin, daß alle Richtungen ihrer Traversierung ausgehend von einer beliebigen Komponente gleichberechtigt sind. Wir sagen auch, daß wir die Beziehung aus der *Perspektive* einer bestimmten Komponente betrachten. In unserem Beispiel ist die Frage nach allen Kursen, die eine Person besucht hat, genauso sinnvoll wie die Frage nach den Teilnehmern eines bestimmten Kurses. Die verteilte Aggregation unterstützt die „Symmetrie“ des Zugriffs in besonders günstiger Weise.

Jede Komponente einer Beziehung kann als Ausgangspunkt für eine Sicht auf diese Beziehung genommen werden. Dazu werden Attribute in die Struktur des mit der Komponente verbundenen Objekttyps integriert. Die Struktur dieser Attribute ergibt sich aus einer „Projektion“ der Beziehungstabelle auf die für die jeweilige Perspektive relevanten Komponenten und Attribute der Beziehung.

Zum Abschluß dieses Abschnittes sei auf eine weitere Besonderheit der Modellierung der Beziehung `Teilnahme` über verteilte Aggregation gemäß (6.3) und (6.4) hingewiesen: Die „Sicht“ auf die Beziehung aus der Perspektive des Objekttyps `Kurs` sieht vor, die an einem Kurs teilnehmenden Personen mit den Anmeldedaten und dem Klausurergebnis in einer *Liste* anzuordnen. Dies ist bei der „neutralen“ Modellierung über eine Beziehungstabelle gemäß (6.2), in der alle Beziehungsinstanzen ungeordnet in einer Menge zusammengefaßt werden, nicht vorgesehen. Beim Einfügen eines Kurses `k` in `p.Kurse` für eine Person `p` ist zunächst unklar, wie sich das zur Wahrung der Konsistenz in `k.T_Liste` einzufügende neue Tupel `t` mit `t.Teilnehmer` = `p` in die Ordnung der Liste einzuordnen hat. Wir schlagen vor, in diesem Fall ein Anfügen von `t` an das Ende der Liste vorzusehen.

6.1.2 Beziehungsredundanz

Die Technik der verteilten Aggregation schafft jedoch besondere Probleme, da auf diese Weise Redundanz in ein Datenbankschema eingeführt wird. Während beim Datenbankentwurf traditionell darauf geachtet wird, daß Redundanz vermieden wird (man denke an die Normalisierungsverfahren für relationale Schemata), rückt man von der Forderung der Redundanzfreiheit ab, wenn Beziehungen durch verteilte Aggregation modelliert werden.

Bei der Modellierung der Beziehung `Teilnahme` durch Integration entsprechender Attribute in die Struktur der jeweiligen Komponententypen, wie wir dies in (6.3) und (6.4) gemacht haben, findet zwar keine Replizierung der Beziehungsattribute `Anm` und `Erg` statt (auf diese kann nur über die Teilnehmerliste von `Kurs`-Objekten zugegriffen werden), jedoch wird der „Kern“ aller Beziehungsinstanzen repliziert, der aus den an einer Beziehungsinstanz beteiligten Komponenten besteht. In unserem Beispiel besteht der Kern einer Beziehungsinstanz vereinfacht gesagt aus (Person, Kurs)-Paaren, und diese werden sowohl über das Attribut `Kurse` in `Person`-Objekten als auch über das Attribut `Teilnehmer` der `Kurs`-Objekte erfaßt. Es liegt somit eine spezielle Form der Redundanz vor, die wir *Beziehungsredundanz* nennen.

Eine Modifikation der Teilnehmerliste `k.T_Liste` für ein `Kurs`-Objekt `k` (Einfügen oder Löschen eines Tupels `t` für einen Kursteilnehmer) muß in ein entsprechendes Update auf dem Attribut `Kurse` des eingefügten bzw. gelöschten Teilnehmers `t.Teilnehmer` umgesetzt werden, um die über die Attribute `Kurse` und `T_Liste` ausgedrückten Sichten auf die Beziehung `Teilnahme` zueinander konsistent zu halten. Entsprechendes gilt auch für die Modifikation von `p.Kurse` für `Person`-Objekte `p`. Eine Datenbankinstanz befindet sich genau dann in einem konsistenten Zustand bzgl. der Beziehung `Teilnahme`, wenn gilt:

```
{[P: p, K: k] | p in EXT(Person), k in p.Kurse}
= {[P: t.Teilnehmer, K: k] | k in EXT(Kurs), t in k.T_Liste}[2]
```

2. In den Mengenformern verwenden wir wieder die intern verfügbaren Extensionen von Objekttypen (vgl. Abschnitt 5.3.1).

In (6.3) und (6.4) kommt jedoch bislang nicht zum Ausdruck, daß diese Konsistenzbedingung, die eine „Synchronisation“ der Attribute `Kurse` bzw. `T_Liste` erforderlich macht, erfüllt sein muß!

Wollte man für ESCHER$^+$ weiterhin auf Redundanzfreiheit beharren, dann gibt es neben der immer bestehenden Möglichkeit, für eine Beziehung einfach eine Beziehungstabelle im Datenbankschema vorzusehen, nur noch die Option der *einseitigen* Aggregation, d.h. die Beziehung wird in die Struktur genau eines Objekttyps integriert.

Verzichten wir z.B. auf das Attribut `Kurse` in der Struktur des Objekttyps `Person`, dann wird die Traversierung der Beziehung aus der Perspektive der Kurse eindeutig bevorzugt, und die oben angesprochene „Symmetrie“ hinsichtlich der Traversierung wird nicht ausreichend unterstützt. Auf dieses Problem haben wir bereits in der Einleitung zu dieser Arbeit hingewiesen (vgl. Abschnitt 1.3). Für das Beispiel der Beziehung `Teilnahme` bedeutet dies, daß bei Verzicht auf das Attribut `Kurse` in (6.3) für das Traversieren der Beziehung aus der Perspektive einer Person eine Anfrage gestellt werden muß, während in umgekehrter Richtung lediglich ein einfacher Zugriff auf ein Attribut notwendig wird:

- Ist `p` ein Bezeichner für ein Objekt des Typs `Person`, dann liefert die Anfrage

  ```
  {[Kurs: k, Anm: t.Anmeldung, Erg: t.Ergebnis]
   | k in Kurse, t in k.T_Liste: t.Teilnehmer = p}
  ```

  die Menge der Kurse, an denen `p` teilnimmt, zusammen mit den jeweiligen Beziehungsattributen.
- Ist umgekehrt `k` ein Bezeichner für ein Objekt des Typs `Kurs`, dann liefert

  ```
  k.T_Liste
  ```

  die teilnehmenden Personen zusammen mit den jeweiligen Beziehungsattributen.

Wir sind der Auffassung, daß sich eine solche Ungleichbehandlung unterschiedlicher Traversierung auf logischer Ebene nicht manifestieren sollte. Es zeigt sich also, daß im Zusammenhang mit Beziehungen Redundanz sogar sinnvoll ist, sofern sie kontrolliert werden kann, d.h. wenn Sichten, die sich semantisch überlappen, keine einander widersprechende Information liefern.

Gezielt eingesetzte Redundanz zur adäquaten Modellierung binärer Beziehungen ohne Attribute ist bereits Bestandteil verschiedener Datenmodelle für komplexe Objekte, die mit ESCHER$^+$ vergleichbar sind. Um redundant vorhandene Information über Beziehungen bei verteilter Aggregation synchron halten zu können und auf diese Weise die Konsistenz auf der Instanzenebene zu gewährleisten, verwenden verschiedene Datenmodelle in ihrer DDL sog. `inverse`-Klauseln. Diese geben an, welche Attribute in den Typdefinitionen miteinander korrespondieren. Die Object Definition Language (ODL) des ODMG’93-Standards [Cat+94] erlaubt z.B. folgende Typdefinitionen (alle anderen Attribute der Typen sind weggelassen):

```
interface Person
(
  ...
  relationship Set<Kurs> Kurse
        inverse Kurs::T_Liste;
  ...
);
```

```
interface Kurs
(
  ...
  relationship List<Person> T_Liste
        inverse Person::Kurse;
  ...
);
```

Attribute, deren Typ ein Objekttyp oder eine Kollektion von Objekten ist, werden durch das Schlüsselwort `relationship` gekennzeichnet (gegenüber `attribute` für sonstige Attribute), wodurch zumindest darauf hingewiesen wird, daß diese Attribute Beziehungen zwischen Objekten wiedergeben. Allerdings ist mit der Verwendung des Wortes `relationship` keine besondere Semantik verbunden. Die Konsistenz ist allein von der korrekten Verwendung der `inverse`-Klauseln abhängig. Die Semantik der beiden `inverse`-Klauseln in obigem Beispiel ist gegeben durch die Forderung nach Einhaltung der Integritätsbedingung

$$\forall\, \omega \in \mathit{Ext}(t_{Person})\ \forall\, \omega' \in \mathit{Ext}(t_{Kurs}) : \omega' \in \omega.\text{Kurse} \Leftrightarrow \omega \in \omega'.\text{T\_Liste}$$

Beim Einfügen in oder Entfernen aus ω.Kurse bzw. ω'.T_Liste wird die jeweils andere Kollektion entsprechend angepaßt, damit die genannte Integritätsbedingung erfüllt bleibt. Einen analogen Mechanismus mit vergleichbarer Syntax bei der Schemadefinition findet man u.a. auch in den Datenmodellen MAD [Mit88] (siehe Abschnitt 2.4.3.1) und COCOON [SLR+92]. Im semantischen Datenmodell IFO können binäre Beziehungen zwischen abstrakten Typen durch zwei zueinander inversen Funktionen modelliert werden (vgl. Abb. 2.4 in Abschnitt 2.1.3).

Es fällt auf, daß in den oben aufgeführten ODL-Typdefinitionen die Beziehungsattribute `Anm` und `Erg` nicht auftauchen. Tatsächlich können `invers`-Klauseln in den uns bekannten Vorschlägen nur für binäre Beziehungen ohne Attribute verwendet werden. Somit kann Redundanz in einem Datenbankschema auch nur für diese Klasse von Beziehungen kontrolliert werden. Binäre Beziehungen ohne Attribute kommen zwar in der Praxis häufig vor, jedoch nicht ausschließlich! So ist man für Beziehungen mit einer Kardinalität n > 2 oder für Beziehungen mit Attributen nach wie vor auf andere Formen der Modellierung angewiesen, bei denen dann jedoch keine Redundanz zugelassen werden darf. Es bieten sich wiederum nur die beiden bereits oben genannten Möglichkeiten an: Entweder die Integration einer Beziehungssicht in die Struktur genau eines Komponententyps der Beziehung (einseitige Aggregation) oder die Definition einer separaten Beziehungstabelle[3]. Es wurde bereits gezeigt, daß die erste Option zu einer einseitigen Bevorzugung der Traversierung der Beziehung ausgehend von einem speziellen Komponententyp führt. Entscheidet man sich für die Definition einer Beziehungstabelle, dann bringt dies den Nachteil mit sich, daß für ein und dasselbe semantische Konzept, nämlich die Beziehung, verschiedene Formen der Modellierung verwendet werden: Für binäre Beziehungen ohne Attribute wird die Technik der verteilten Aggregation gewählt, für alle sonstigen Beziehungen wird eine separate Beziehungstabelle definiert.

3. Für objektorientierte Datenmodelle wird üblicherweise vorgeschlagen (vgl. [KM94]), Beziehungen mit einer Kardinalität > 2 und Beziehungen mit Attributen über einen eigenen Objekttyp zu definieren, dessen Attribute gerade die Komponenten und Attribute der Beziehung sind. Für Modelle, in denen auf die Extension der definierten Objekttypen zugegriffen werden kann, entspricht dann die Extension des für die Beziehung vorgesehenen Objekttyps der erwähnten Beziehungstabelle.

6.1.3 Schema-Modifikation und Beziehungen

Problematisch ist im Rahmen der bisher vorgestellten Möglichkeiten zur Modellierung von Beziehungen auch das nachträgliche Hinzufügen weiterer Beziehungen. Für das Hinzufügen neuer Beziehungen im Zuge von Schema-Evolution stehen prinzipiell wieder die bereits genannten Alternativen zur Verfügung:

- Modellierung über einseitige oder verteilte Aggregation: Die Strukturen aller an der neuen Beziehung beteiligten Objekttypen werden um neue Attribute erweitert, die jeweils eine Sicht auf die neue Beziehung darstellen.
  Es tritt dann das Problem auf, daß die neue Struktur eines Objekttyps nach der Erweiterung nicht mehr mit der Struktur der bereits existierenden Instanzen übereinstimmt. Diese müßten automatisch an die neue Struktur angepaßt werden. Eine ähnliche Situation findet man vor, wenn eine Beziehung, die mittels verteilter Aggregation modelliert wurde, nicht länger relevant ist und die entsprechenden Attribute aus den Objekttypdefinitionen entfernt werden soll. Dies hat auch die Veränderung von Objekttypdefinitionen mit der Notwendigkeit der Anpassung bestehender Instanzen zur Folge. Wir wollen jedoch nach Lösungen suchen, bei denen auf eine Modifikation bereits bestehender Instanzen verzichtet werden kann.
- Definition einer weiteren Beziehungstabelle: Diese Lösung erfordert keine Veränderungen an bereits existierenden Instanzen und bietet sich daher als „Standardlösung“ für das Hinzufügen neuer Beziehungen an. Wurde jedoch für Beziehungen, die bereits bei der Erststellung eines Datenbankschemas bekannt waren, die Methode der verteilten Aggregation gewählt und später hinzukommende Beziehungen dagegen grundsätzlich über neue Beziehungstabellen in das Datenbankschema integriert, dann werden Beziehungen insgesamt auf sehr inhomogene Weise behandelt. Der Benutzer muß dann genaue Kenntnis darüber besitzen, welche Beziehung auf welche Weise modelliert wurde, da dies für die Art der Anfrageformulierung relevant ist.

Das Problem der Anpassung bereits existierender Instanzen tritt auch dann auf, wenn man sich bei einer Beziehung zunächst für einseitige Aggregation entschieden hat und später eine weitere Sicht auf dieselbe Beziehung hinzufügen will. Schließt man die Modifikation der bestehenden Struktur eines Objekttyps aus (eben weil dies mit der Anpassung vorhandener Objekte verbunden wäre), dann bleibt als Alternative wiederum nur die Definition einer weiteren Tabelle. Wir wollen annehmen, daß man sich für die Beziehung `Teilnahme` zunächst mit der Integration des Attributs `T_Liste` in den Zustand des Objekttyp `Kurs` begnügt und das Attribut `Kurse` in (6.3) weggelassen hätte. Stellt man später fest, daß auch die alternative Sicht auf die Beziehung aus der Perspektive einer Person in das Datenbankschema integriert werden soll, dann kann eine „Beziehungstabelle“ `T` mit folgender Struktur definiert werden:

```
{[P: Person, Kurse: {Kurs}]}
```

Bisher wurde davon ausgegangen, daß Beziehungstabellen flache Relationen sind (und somit hinreichend „neutral“, um keinen Zugriff einseitig zu bevorzugen). Wir wählen für `T` jetzt jedoch die angegebene Schachtelungsstruktur, da sie sich besonders für den hier interessierenden Zugriff aus der Perspektive einer Person eignet. Somit werden sogar zur Modellierung *einer* Beziehung zwei verschiedene Konzepte eingesetzt! Die Traversierung der Beziehung ausgehend von `Kurs`-Objekten wird durch die einseitige Aggregation unterstützt, die Traver-

sierung aus der Sicht von `Person`-Objekten durch den Zugriff auf die Tabelle `T`. Wiederum muß der Anwender genau wissen, welches Konzept für welche Art des Zugriffs relevant ist.

6.2 Beziehungen in ESCHER$^+$

Die Ausführungen im vorangegangenen Abschnitt machten deutlich, daß auf logischer Ebene Bedarf an einem Konzept zur einheitlichen Modellierung von Beziehungen besteht. Eine „Mischmodellierung“ von Beziehungen einerseits über verteilte Aggregation, andererseits über Beziehungstabellen erzeugt ein inhomogenes Datenbankschema, in dem der Benutzer sich nur schwer zurechtfindet. Zudem ist die Frage der Redundanzkontrolle nach wie vor ungelöst.

In diesem Abschnitt wird für ESCHER$^+$ ein eigenständiges Modellierungskonstrukt für Beziehungen eingeführt, das zu einem weiteren fundamentalen Baustein des Datenmodells wird. Das ESCHER$^+$-spezifische Beziehungskonstrukt soll folgende Eigenschaften haben:

- Das Konstrukt unterstützt nicht nur binäre Beziehungen ohne Attribute, sondern Beziehungen mit beliebiger Kardinalität, für die auch Attribute definiert sein dürfen.
- Für jede Beziehung werden sog. *Beziehungssichten* definiert, die jeweils einer Komponente der Beziehung zugeordnet sind. Der Zugriff auf die Beziehungsinstanzen erfolgt immer über eine Beziehungssicht.
- Kontrolle von Beziehungsredundanz: Die in den verschiedenen Beziehungssichten replizierte Information wird zueinander konsistent gehalten.
- Einfaches Hinzufügen neuer Beziehungen in ein Datenbankschema: Beim Hinzufügen neuer Beziehungen ist keine Instanzenkonversion notwendig. Entsprechendes gilt auch für das Entfernen nicht mehr relevanter Beziehungen.

Wir erläutern das Konzept in diesem Abschnitt anhand einiger Beispiele, bevor in Abschnitt 6.3 die formalen Definitionen folgen.

Beispiel 6.2 Zur Definition einer Beziehung steht die DDL-Anweisung `define relship` zur Verfügung. Die Beziehung `Teilnahme` aus Beispiel 6.1 wird einem ESCHER$^+$-Datenbankschema durch folgende DDL-Anweisung hinzugefügt:

```
define relship
  -name Teilnahme
  -comp [P: Person, K: Kurs]
  -attr [Anm: date, Erg: integer]
  -key  [P, K]
  -view P -> [Kurse:{@K}],
        K -> [T_Liste: <[Teilnehmer:@P,
              Anmeldung:@Anm, Ergebnis: @Erg]>]
end define;
```

Ist die Beziehung `Teilnahme` zu einem späteren Zeitpunkt nicht mehr relevant, kann sie mittels `drop relship Teilnahme` wieder aus dem Datenbankschema entfernt werden. ❐

Durch Konkatenation der Tupelstrukturen der `comp`- und `attr`-Klausel erhält man die Struktur für die Beziehungsinstanzen in der Form von (6.1), die alle in einer Beziehungstabelle mit der Struktur (6.2) zusammengefaßt werden könnten. Mit einer `define relship`-Anweisung soll jedoch keine Instanziierung einer Beziehungstabelle verbunden sein! Vielmehr existiert diese nur „virtuell“. Die `key`-Klausel gibt einen Schlüssel für die „virtuelle“ Beziehungstabelle an. So übersetzt sich die `key`-Klausel in die Integritätsbedingung "`Teilnahme has key [P, K]`", wenn `Teilnahme` auch als Name der „virtuellen“ Beziehungstabelle angenommen wird.

Die Information über die somit ebenfalls „virtuellen“ Beziehungsinstanzen wird ausschließlich über die sog. Beziehungssichten instanziiert (d.h. materialisiert).

Während man üblicherweise gerade Sichten mit dem Begriff „virtuell“ verbindet (Sichten sind gewöhnlich selbst nicht materialisiert, sondern als Anfragen über materialisierten Basistabellen oder anderen Sichten definiert), soll im Zusammenhang mit Beziehungen und Beziehungssichten gerade die umgekehrte Situation gelten: Nicht die Beziehungsinstanzen liegen in materialisierter Form vor, sondern die Beziehungssichten. Damit werden wir der Tatsache gerecht, daß auf Beziehungen immer aus der Sicht einer Komponente zugegriffen wird.

Beziehungssichten werden in der `view`-Klausel definiert. Jeder Komponente einer Beziehung kann eine Beziehungssicht zugeordnet werden. Im Beispiel der Beziehung `Teilnahme` werden für beide Komponenten `P` und `K` Beziehungssichten angegeben, die wir mit `Teilnahme:P` und `Teilnahme:K` bezeichnen.

In der `view`-Klausel wird für jede Beziehungssicht eine Struktur angegeben, in der auch die Zusammenhänge mit den „Original“-Komponenten und -Attributen der `comp`- und `attr`-Klausel erfaßt werden. Dies geschieht durch Ersetzen der Typangaben in den Strukturen durch die Namen c_i (bzw. a_j) von Komponenten (bzw. Attributen), denen das Zeichen `@` vorangestellt wird. Die „Referenzen“ $@c_i$ (bzw. $@a_j$) sind von entscheidender Bedeutung, wenn es darum geht, die aufgrund der Existenz verschiedener Beziehungssichten entstehende Redundanz zu kontrollieren, d.h. die verschiedenen Beziehungssichten zueinander synchron zu halten. Man erhält daher die einer Beziehungssicht zugeordnete Struktur als Element von $\mathcal{S}$(TYPES), indem man die $@c_i$ (bzw. $@a_j$) durch den Typ t_i (bzw. durch s_j) substituiert:

- Der Beziehungssicht `Teilnahme:P` wird also die Struktur `[Kurse:{Kurs}]` zugeordnet.
- Zur Beziehungssicht `Teilnahme:K` gehört die Struktur `[T_Liste: <[Teilnehmer: Person, Anmeldung: date, Ergebnis: integer]>]`

Die Instanziierung einer Beziehungssicht erfolgt über eine Interpretationsfunktion, die der Beziehungssicht zugeordnet wird.

Zur Beziehungssicht `Teilnahme:K` gehört eine Interpretationsfunktion I(`Teilnahme:K`). Sie liefert für jedes beliebige `Kurs`-Objekt k ein getyptes Datenobjekt mit der Strukur `[T_Liste:<…>]`. Der Funktionswert I(`Teilnahme:K`)(k) gibt somit die Sicht auf die Beziehungsinstanzen wieder, an denen k in der Rolle `K` teilnimmt.

In Tabelle 6.1 ist für die Beziehung `Teilnahme` eine Beispielmenge von Beziehungsinstanzen zusammengestellt.

P	K	Anm	Erg
p_1	k_1	23.5.95	7
p_2	k_1	26.5.95	10
p_2	k_2	4.2.96	4
p_3	k_2	10.3.96	8
p_4	k_1	20.5.95	9
p_4	k_3	28.10.95	5

Tabelle 6.1: Beispielinstanzen zur Beziehung `Teilnahme`

Für das `Kurs`-Objekt k_1 liefert

I(`Teilnahme:K`)(k_1) =

```
[T_Liste:
 <[Teilnehmer: p1, Anmeldung: 23.05.95, Ergebnis: 7],
  [Teilnehmer: p2, Anmeldung: 26.05.95, Ergebnis: 10],
  [Teilnehmer: p4, Anmeldung: 20.05.95, Ergebnis: 9]>]
```

die Liste aller Teilnehmer des Kurses k_1.

Entsprechend gibt es für die Beziehungssicht `Teilnahme:P` eine Interpretationsfunktion I(`Teilnahme:P`), die für jedes `Person`-Objekt p die Sicht auf die Beziehung aus der Perspektive von p in der Rolle `P` angibt. Auf der Grundlage der Daten aus Tabelle 6.1 ist z.B.

I(`Teilnahme:P`)(p_2) = `[Kurse: {`k_1`,`k_2`}]`

die Menge aller Kurse, an denen p_2 teilnimmt oder teilgenommen hat.

Gegenüber dem Vorgehen der verteilten Aggregation, das uns zur Integration der Attribute `Kurse` und `T_Liste` in die Strukturen der Objekttypen t_{Person} und t_{Kurs} führte (siehe (6.3) und (6.4)), haben wir diese Attribute zunächst in die Beziehungssichten „ausgelagert“. Unser Ziel ist es jedoch, sie wie jedes „gewöhnliche“ Attribut der Objekttypen t_{Person} und t_{Kurs} verwenden zu können:

Ist `p` eine Variable, die an ein `Person`-Objekt gebunden ist, dann soll `p.Kurse` ein gültiger SCRIPT$^+$-Ausdruck sein, obwohl `Kurse` bislang nicht in Def(*AttrAccess*(t_{Person})) enthalten ist[4]. Analog soll `k.T_Liste` ein gültiger SCRIPT$^+$-Ausdruck sein, falls `k` eine Variable vom Typ `Kurs` ist. Damit erklärt sich auch, wieso die Struktur der Beziehungssicht `Teilnahme:P` die Tupelstruktur `[Kurse:{Kurs}]` und nicht einfach `{Kurs}` ist: In letzterem Fall würde ganz einfach ein Attributname fehlen, über den auf die Menge von Kursen zugegriffen wird!

Neben den Attributen, die für einen Objekttyp t neu definiert werden bzw. die t von seinen Supertypen erbt und die wir als die *Kern-Attribute* von t bezeichnen, kommen zu Def(*AttrAccess*(t)) nun auch die Attribute aus allen Beziehungssichten dazu, deren Ausgangspunkt eine Beziehungskomponente vom Typ t ist. Das bedeutet, daß der Objekttyp weitere

4. Nach Abschnitt 3.7.1 ist der Definitionsbereich von *AttrAccess*(t) die Menge aller für t direkt definierten oder ererbten Attribute ist.

Attribute „erbt“, nämlich die der Beziehungssichten. Während die Vererbung von Attributen aufgrund der *isa*-Beziehung als *vertikale* Vererbung bezeichnet werden kann, da ein Objekttyp Attribute (und Methoden) von den „Vorfahren“ in der *isa*-Hierarchie erbt, sprechen wir bei der Erweiterung eines Objektzustandes um die Attribute aus Beziehungssichten von *horizontaler* Vererbung. Durch die Hinzunahme der Attribute von Beziehungssichten erweitert sich der „Horizont“ eines Objektes, indem nun neben den Kern-Attributen auch Beziehungen zu anderen Objekten zum Zustand eines Objektes gerechnet werden. Wir unterscheiden daher auch einen *engeren* Objektzustand, der ausschließlich aus den Kern-Attributen besteht, und einem *erweiterten* Objektzustand.

In Abb. 6.3 wird mit Hilfe eines ER-Diagramms graphisch veranschaulicht, was unter dem erweiterten Zustand eines Objekttyps zu verstehen ist: Zum erweiterten Zustand des Objekttyps t_1 gehören neben den Kern-Attributen von t_1 auch die Attribute der Beziehungssichten der Beziehungen R_1, R_2 und R_3 (wir nehmen an, daß für diese Beziehungen jeweils eine Beziehungssicht definiert wird, die die jeweilige Komponente vom Typ t_1 als Ausgangspunkt hat). Damit ist für t_1-Objekte ein Traversieren der Beziehungen R_1, R_2 und R_3 durch einfachen Attributzugriff möglich. Für t_3 besteht der erweiterte Zustand nur aus den Kern-Attributen von t_3 sowie den Attributen der entsprechenden Beziehungssicht von R_3, da wir annehmen wollen, daß t_3-Objekte zwar an der Beziehung R_4 teilnehmen können, jedoch keine Beziehungssicht definiert wurde, die t_3-Objekte als Ausgangspunkt hat. Es besteht kein Zwang, für alle Komponenten eine Beziehungssicht zu definieren, wie eines der nachfolgenden Beispiele zeigt.

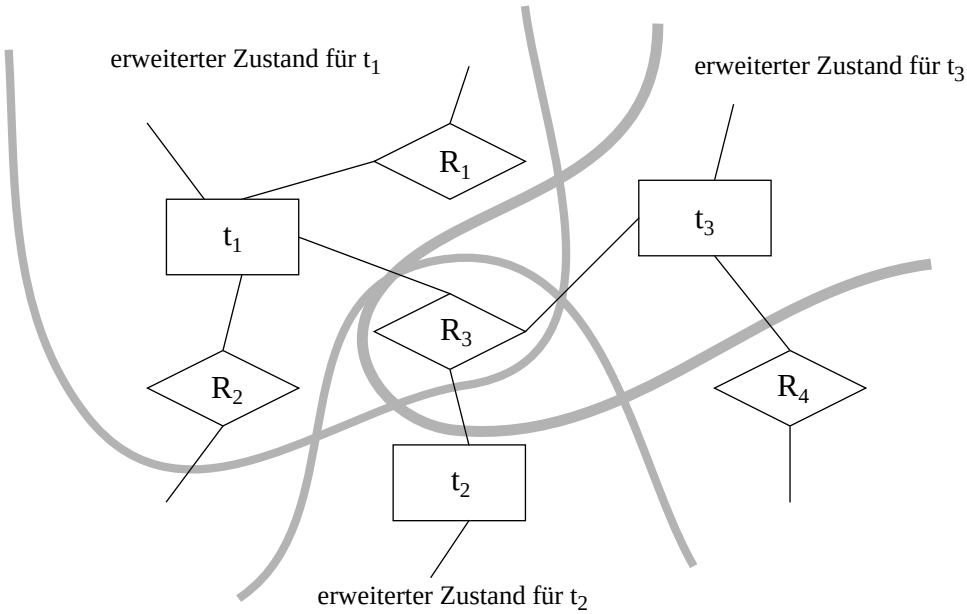


Abbildung 6.3: Veranschaulichung der erweiterten Zustände für Objekttypen

Es folgen nun weitere Beispiele zur Definition von Beziehungen in ESCHER$^+$.

Beispiel 6.3 Durch die Zugehörigkeit von Angestellten zu den Abteilungen einer Firma sei eine *1:n*-Beziehung gegeben. Attribut der Beziehung sei das Eintrittsdatum eines Mitarbeiters in eine Abteilung.

Wollten wir diese Beziehung über zusätzliche Kern-Attribute der Objekttypen `Angestellter` oder `Abteilung` erfassen (einseitige Aggregation), dann kann dazu eine der folgenden Möglichkeiten gewählt werden:

(i) Integration eines Attributes `Mitarbeiter` mit der Struktur `{[M: Angestellter, seit: date]}` in die Struktur des Objekttyps `Abteilung`.

(ii) Integration der Attribute `Abt`: `Abteilung` und `seit: date` in die Struktur des Objekttyps `Angestellter`.

Tatsächlich entfällt aber die Möglichkeit (i), da das Datenmodell für diesen Fall nicht über die Ausdrucksmittel verfügt, mit denen die Beschränkung der Mitarbeit eines Angestellten in höchstens einer Abteilung (d.h. die Tatsache, daß es sich nicht um eine *m:n*- sondern um eine *1:n*-Beziehung handelt) festgehalten werden kann. Die automatische Einhaltung der Bedingung

```
all ang in EXT(Angestellter):
card({ abt | abt in EXT(Abteilungen),
t in abt.Mitarbeiter: t.M = ang}) <= 1
```

kann im Rahmen der in Kapitel 5 eingeführten Integritätsbedingungen nicht erreicht werden. Aus diesem Grund muß Alternative (ii) gewählt werden[5]. Gerade für *1:n*-Beziehungen wäre jedoch eine Schachtelung, wie sie durch die Möglichkeit (i) gegeben ist, wünschenswert.

In einem ESCHER$^+$-Datenbankschema läßt sich die Beziehung `Mitarbeit` wie folgt zu definieren:

```
define relship
  -name Mitarbeit
  -comp [A: Abteilung, M: Angestellter]
  -attr [seit: date]
  -key  M
  -view A -> [Mitarbeiter:{[M: @M, seit: @seit]}],
        M -> [Abteilung: @A, Eintritt: @seit]
end define;
```

Die Tatsache, daß es sich um eine 1:n-Beziehung handelt, wird durch die Angabe des Schlüssel `M` in der `key`-Klausel ausgedrückt. ❒

Es folgt nun noch ein Beispiel für eine ternäre Beziehung, die – wie in den vorangegangenen Beispielen – außerdem ein Attribut besitzt.

Beispiel 6.4 Man betrachte die ternäre Beziehung `Unterricht`, die wie folgt definiert sei:

```
define relship
  -name Unterricht
  -comp [L: Lehrer, K: Klasse, F: Fach]
  -attr [Std: integer]
  -key  [K, F]
```

5. Dies ist im übrigen auch der Weg, den man bei der Erstellung eines (flachen) relationalen Datenbankschemas einschlagen würde.

```
    -view L ->
           [Dienstplan:{[Fach:@F, Klasse:@K, Stunden:@Std]}],
          K ->
           [Stundenplan:{[Fach:@F, Lehrer:@L, Anz_Std:@Std]}],
  end define;
```

Dabei seien `Lehrer`, `Klasse` und `Fach` geeignet definierte Objekttypen. Die Existenz einer Beziehungsinstanz `[L:` ω_1`, K:` ω_2`, F:` ω_3`, Std:` n`]` hat die Semantik:
„Lehrer ω_1 unterrichtet die Klasse ω_2 im Fach ω_3 wöchentlich n Stunden."

Es gilt die Schlüsselbedingung `[K, F]`, d.h. eine Klasse soll in einem Fach von genau einem Lehrer unterrichtet werden.

Es wird keine Beziehungssicht aus der Perspektive der Komponente `F` definiert. Dies ist für den Fall sinnvoll, daß die Traversierung der Beziehung ausgehend von einem `Fach`-Objekt so selten erforderlich ist, daß sich der Aufwand zur Pflege auch dieser Beziehungssicht nicht lohnen würde. ❒

6.3 Formale Definitionen

6.3.1 Beziehungen und Beziehungssichten

Im Datenmodell ESCHER$^+$ werden Beziehungen als weitere Grundelemente des strukturellen Teils eines Datenbankschemas eingeführt.

Definition 6.1 (Beziehungen)

Zu jedem Datenbankschema DS gehört eine endliche Menge RELSHIPS $= \{b_1, \ldots, b_n\}$, $n \geq 0$, deren Elemente *Beziehungen* genannt werden. Auf RELSHIPS seien folgende Funktionen definiert:

$$\textit{name}: \text{RELSHIPS} \rightarrow \mathcal{N},$$

die jeder Beziehung $b \in$ RELSHIPS einen eindeutigen Namen zuordnet.

$$\textit{struct}: \text{RELSHIPS} \rightarrow \mathcal{S}(\text{TYPES}), \tag{6.5}$$

für die gelten soll:

$\forall\, b \in$ RELSHIPS: *struct* $(b) = [c_1: t_1, \ldots, c_n: t_n, a_1: s_1, \ldots, a_m: s_m]$
mit $n \geq 2$ und $m \geq 0$,
$c_1, \ldots, c_n, a_1, \ldots, a_m \in \mathcal{A}$,
$t_i \in$ DEFTYPES für $1 \leq i \leq n$,
$s_j \in$ BASETYPES $\cup$ ENUMTYPES $(1 \leq j \leq m)$

Als Strukturen für Beziehungen sind demnach nur solche Tupelstrukturen zugelassen, deren Komponenten in zwei Gruppen partitioniert werden können:

- Die erste Gruppe besteht aus allen Paaren (c_i, t_i) der Tupelstruktur *struct*(b), bei denen einem Namen c_i ein Objekttyp t_i zugeordnet wird. Die Namen c_i werden als *Komponenten* (bzw. *Rollen*) der Beziehung bezeichnet.

- Die zweite Gruppe besteht aus allen sonstigen Paaren (a_j, s_j) der Tupelstruktur *struct*(b). Die Namen a_j werden als *Attribute* der Beziehung bezeichnet. Attribute unterscheiden sich von Komponenten also dadurch, daß ersteren *kein* Objekttyp zugeordnet wird.

Die Kardinalität einer Beziehung ist gegeben durch die Anzahl n ihrer Komponenten, d.h. Attribute tragen nicht zur Kardinalität bei.

Ferner sei

$$keys: \text{RELSHIPS} \rightarrow \mathcal{FS}et(\mathcal{FS}et(\{c_1, \ldots, c_n, a_1, \ldots, a_m\}))$$

eine Funktion, die jeder Beziehung b eine endliche Menge von (minimalen) *Schlüsseln* zuordnet. Dabei ist jeder Schlüssel eine Teilmenge der Menge der Komponenten und Attribute der Beziehung. ❒

Prinzipiell könnten Beziehungsattribute in ESCHER$^+$ einen beliebigen komplexen Typ $s_j \in \mathcal{S}$(TYPES) – DEFTYPES besitzen[6]. In dieser Arbeit beschränken wir uns auf Beziehungen mit Attributen, deren Struktur ein Basis- oder Aufzählungstyp ist. Die Erfahrung zeigt, daß damit die in der Realität anzutreffenden Beziehungen bereits gut erfaßt werden können.

Wir führen einige zusätzliche Funktionen ein, die im weiteren Verlauf verwendet werden:
Für b ∈ RELSHIPS seien

$$compDef(b) := \{(c_1, t_1), \ldots, (c_n, t_n)\},$$
$$comps(b) := \{c_1, \ldots, c_n\} \text{ und}$$
$$compType(b, c) := t, \text{ falls } (c, t) \in compDef(b).$$

Analog setzen wir für Beziehungsattribute

$$attrDef(b) := \{(a_1, s_1), \ldots, (a_m, s_m)\},$$
$$attrs(b) := \{a_1, \ldots, a_m\} \text{ und}$$
$$attrStruct(b, a) := s, \text{ falls } (a, s) \in attrDef(b).$$

Die `define relship`-Anweisungen der Beispiele 6.2, 6.3 und 6.4 erzeugen neue Beziehungen $b_{\text{Teilnahme}}$, $b_{\text{Mitarbeit}}$ und $b_{\text{Unterricht}}$ mit

- *name*($b_{\text{Teilnahme}}$) = "`Teilnahme`",
  struct($b_{\text{Teilnahme}}$) = `[P: Person, K: Kurs, Anm: date, Erg: integer]`
  keys($b_{\text{Teilnahme}}$) = {{`P`, `K`}}
- *name*($b_{\text{Mitarbeit}}$) = "`Mitarbeit`",
  struct($b_{\text{Mitarbeit}}$) = `[A: Abteilung, M: Angestellter, seit: date]`
  keys($b_{\text{Mitarbeit}}$) = {{`M`}}
- *name*($b_{\text{Unterricht}}$) = "`Unterricht`",
  struct($b_{\text{Unterricht}}$) = `[L: Lehrer, K: Klasse, F: Fach, Std: integer]`
  keys($b_{\text{Unterricht}}$) = {{`K`, `F`}}

In den aufgeführten Beispielen ist für jede Beziehung genau ein Schlüssel definiert. Für eine *1:1*-Beziehung b zwischen zwei Objekttypen `A` und `B` ist *key*(b) = {{`A`}, {`B`}}, d.h. Beziehungen können tatsächlich mehr als einen Schlüssel besitzen.

6. Wir schließen $s_j \in$ DEFTYPES aus, da in diesem Fall a_j kein Attribut der Beziehung, sondern eine Komponente wäre!

Bis jetzt erinnern Beziehungen in ESCHER$^+$ sehr stark an die *relationship types* des ER-Modells. Wir könnten nun die Definition einer Datenbankinstanz zu einem Datenbankschema (vgl. Def. 3.20) einfach um eine Menge

$$\{inst : \text{RELSHIPS} \cdots> \mathcal{V}al^*(\text{TYPES})\}$$

von Instanzenfunktionen ergänzen, wobei dann gelten soll:

$$\forall\, b \in \text{RELSHIPS}: inst\,(b) = (v, \{struct\,(b)\}) \text{ für ein } v \in \mathcal{FS}et(dom(struct(b)))$$

Das würde aber bedeuten, daß *inst*(b) den Wert einer Beziehungstabelle angibt, die alle Beziehungsinstanzen – Tupel aus *dom*(*struct*(b)) – in einer Menge zusammenfaßt. In den einführenden Beispielen haben wir jedoch deutlich gemacht, daß für ESCHER$^+$ ein anderer Weg gewählt werden soll, bei dem Beziehungen gerade nicht in Gestalt von Beziehungstabellen instanziiert werden sollen. Stattdessen werden in ESCHER$^+$ zu einer Beziehung $b \in$ RELSHIPS verschiedene Beziehungssichten definiert. In Abschnitt 6.3.2 wird die bisherige Definition einer Datenbankinstanz erweitert, indem alle Beziehungssichten über eigene Interpretationsfunktionen „instanziiert“ werden. Die Beziehungen $b \in$ RELSHIPS liefern lediglich den gemeinsamen „Rahmen“ für die unterschiedlichen Beziehungssichten.

Definition 6.2 (Beziehungssichten)

Neben einer Menge RELSHIPS von Beziehungen gehört zu jedem Datenbankschema DS eine Menge

$$\text{RELVIEWS} \subseteq \{(b, c) \mid b \in \text{RELSHIPS}, c \in comps(b)\},$$

deren Elemente wir *Beziehungssichten* nennen. Genauer nennen wir $(b, c) \in$ RELVIEWS die Beziehungssicht der Beziehung b aus der Perspektive der Komponente c oder einfach die Beziehungssicht von b mit Ausgangspunkt c. Die Menge RELVIEWS habe die Eigenschaft

$$\forall\, b \in \text{RELSHIPS}\ \exists\, c \in comps(b): (b, c) \in \text{RELVIEWS}$$

Das bedeutet, daß zu jeder Beziehung $b \in$ RELSHIPS mindestens eine Beziehungssicht in RELVIEWS existieren muß. Auf RELVIEWS seien folgende Funktionen definiert:

$$struct : \text{RELVIEWS} \to \mathcal{S}(\text{TYPES}), \tag{6.6}$$

$$corresp : \text{RELVIEWS} \to \mathcal{PF}un(\mathcal{L}(path), \mathcal{A})$$

Dabei legt *struct*(b, c) die Struktur der Beziehungssicht (b, c) fest. Analog zur Bedingung (3.22) für die Struktur von Objekttypen soll für alle $(b, c) \in$ RELVIEWS gelten:

$$type(struct(b, c)) = c_{tuple} \tag{6.7}$$

Für $(b, c) \in$ RELVIEWS ist der Wert von *corresp*(b, c) eine partielle Funktion, die angibt, welche „Stellen“ in *struct*(b, c) mit welchen Komponenten c_i bzw. Attributen a_j der Beziehung b korrespondieren (daher der Name *corresp*). Dabei wird eine „Stelle“ in *struct*(b, c) durch einen Pfad $p \in \mathcal{L}(path)$ angegeben. Weitere Erläuterungen zur Funktion *corresp* folgen weiter unten. Die Funktionen *corresp*(b, c) müssen injektiv sein, da in einer Beziehungssicht eine Komponente c_i oder ein Attribut a_j nicht mehrfach referenziert werden darf.

Ist $(b, c) \in$ RELVIEWS und $struct(b, c) = [a_1: s_1, \ldots, a_k: s_k]$, dann sei $Attrs(b, c) := \{a_1, \ldots, a_k\}$. *Attrs*(b, c) ist also die Menge aller Attribute der Beziehungssicht (b, c).

Für $b \in$ RELSHIPS sei RELVIEWS(b) := $\{(b, c) \mid (b, c) \in \text{RELVIEWS}\}$ die Menge der für die Beziehung b definierten Beziehungssichten. ❒

Die Funktionswerte der Funktionen *struct* aus (6.5) und (6.6) lassen sich – wie dies bereits für Typen und Tabellen der Fall war – als Wurzelknoten von Strukturbäumen $B_{struct(b)}$ bzw. $B_{struct(b,c)}$ interpretieren, die die jeweilige Struktur aus $\mathcal{S}$(TYPES) repräsentieren. Die Syntax und Semantik von Strukturpfadausdrücken $s \in \mathcal{L}$(*structPathExpr*) aus Abschnitt 5.2 wird nun erweitert, um auch alle Knoten in diesen Strukturbäumen textuell bezeichnen zu können. Als Ergänzung zu den Regeln zur Auswertung von Strukturpfadausdrücken, die in Abschnitt 5.2 aufgeführt wurden, definieren wir:

- Es sei $s = n \in \mathcal{L}$(*structPathExpr*) mit $n =$ *name*(b) für ein $b \in$ RELSHIPS
  $\Rightarrow \mu_{struct}(s) =$ *root*$(B_{struct(b)})$,
  wobei $B_{struct(b)}$ der eindeutig bestimmte Strukturbaum zur Beziehung b ist.
- Es sei $s = n$`:`$c \in \mathcal{L}$(*structPathExpr*) mit $n =$ *name*(b) für ein $b \in$ RELSHIPS
  und $(b, c) \in$ RELVIEWS
  $\Rightarrow \mu_{struct}(s) :=$ *root*$(B_{struct(b,c)})$,
  wobei $B_{struct(b,c)}$ der eindeutig bestimmte Strukturbaum zur Beziehungssicht (b, c) ist.

Die Definition einer Beziehungssicht (b, c) geschieht in der `view`-Klausel einer `define relship`-Anweisung und wird dort in der Form "`c -> s`" angegeben. Dabei ist s eine sog. *Referenzstruktur*. Referenzstrukturen werden analog zu Strukturen aus $\mathcal{S}$(TYPES) gemäß Def. 3.9 konstruiert, wobei jedoch $S_0 = \{@c_1, \ldots, @c_n, @a_1, \ldots, @a_m\}$ ist. An die Stelle von Typangaben treten also „Referenzen“ $@c_i$ bzw. $@a_j$ auf die Komponenten bzw. Attribute der Beziehung b. Die Menge aller für eine Beziehung $b \in$ RELSHIPS konstruierbaren Referenzstrukturen nennen wir $\mathcal{RefS}$(b). Die in einer Referenzstruktur in kompakter Form enthaltene Information wird nun auf die beiden Funktionen *struct* und *corresp* „verteilt“:

- *struct*(b, c) ergibt sich durch Substitution von $@c_i$ durch *compType*$(b, c_i) = t_i$ bzw. von $@a_j$ durch *attrType*$(b, a_j) = s_j$.
- Der Zusammenhang zwischen *struct*(b, c) und den Komponenten c_i bzw. Attributen a_i der Beziehung wird über die Funktion *corresp* festgehalten: Bezeichnet μ_{struct}(*name*(b)`:`cp_i) denjenigen Knoten in $B_{struct(b,c)}$, welcher der „Stelle“ in *struct*(b, c) entspricht, an der vor der Substitution $@c_i$ bzw. $@a_i$ stand, dann ist *corresp*$(b, c)(p_i) := c_i$ bzw. a_i.
  Ferner setzen wir *corresp*$(b, c)(\varepsilon) :=$ c, d.h. dem leeren Pfad ε wird der Ausgangspunkt der Beziehungssicht zugeordnet.

Sprechweise: Ist $x \in$ *comps*$(b) \cup$ *attrs*(b) und gilt $x \in$ Bild(*corresp*(b, c)), dann sagen wir, daß die Komponente bzw. das Attribut x von der Beziehungssicht *erfaßt* wird.

Wir illustrieren die formale Definition für Beziehungssichten anhand des Beispiels 6.2. Dort wurden zwei Beziehungssichten definiert:

- `Teilnahme:P`[7] mit

  struct(`Teilnahme:P`) = `[Kurse:{Kurs}]`
  corresp(`Teilnahme:P`) = {(ε, `P`), (`.Kurse.#`, `K`)}
- `Teilnahme:K` mit

  struct(`Teilnahme:K`) = `[T_Liste: <[Teilnehmer: Person, Anmeldung: date, Ergebnis: integer]>]`

7. Statt (b, c) schreiben wir für eine Beziehungssicht im folgenden auch *name*(b)`:`c, wie wir es bei der informellen Einführung von Beziehungssichten im vorangegangenen Abschnitt gemacht hatten.

corresp(`Teilnahme:K`) = {(ε, `K`), (`.T_Liste.#.Teilnehmer`, `P`),
(`.T_Liste.#.Anmeldung`, `Anm`), (`.T_Liste.#.Ergebnis`, `Erg`)}

Die Funktionswerte *corresp* lassen sich als gleichnamige Beschriftung

corresp : $\mathcal{V}$ ···> $\mathcal{A}$

in die Strukturbäume $B_{struct(b,\ c)}$ integrieren. In Abb. 6.4 sind die Strukturbäume mit der Beschriftung *corresp* für die Beziehung `Teilnahme` und den Beziehungssichten aus Beispiel 6.2 dargestellt.

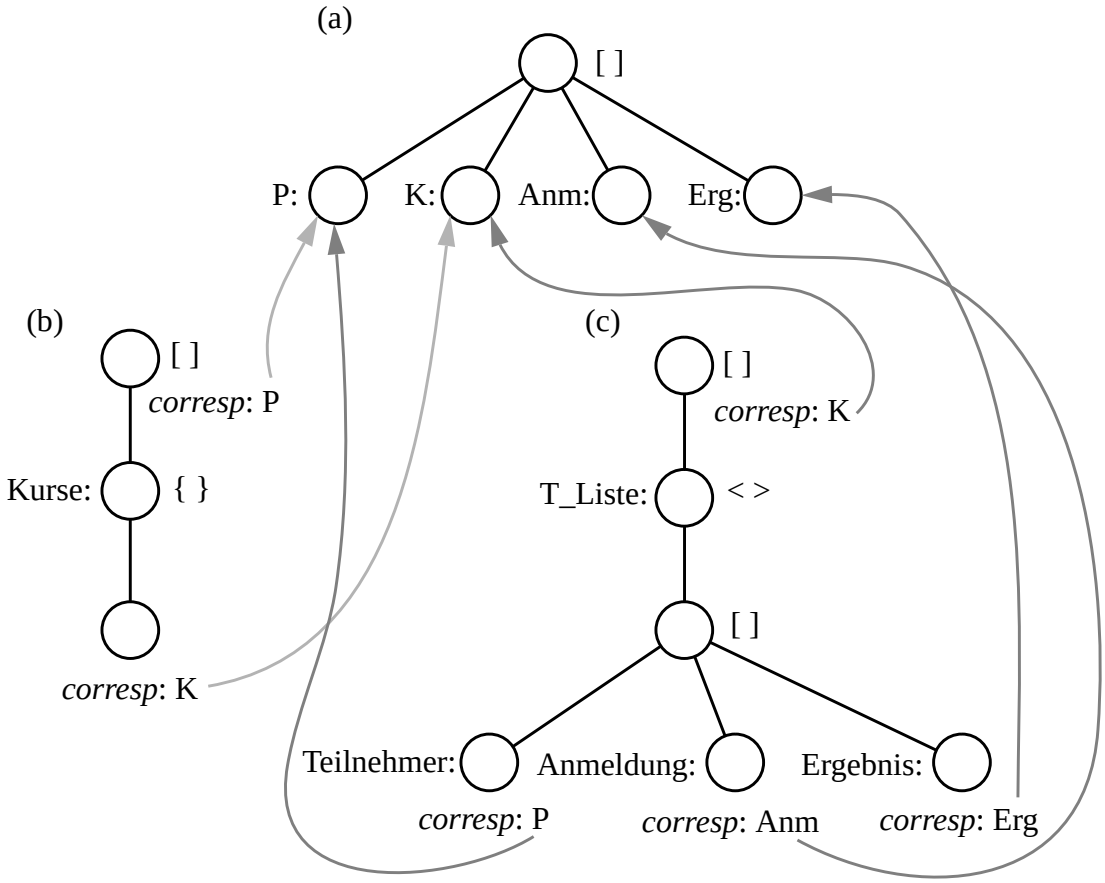


Abbildung 6.4: (a) Strukturbaum der Beziehung `Teilnahme`,
(b) Strukturbaum der Beziehungssicht `Teilnahme`: `P`,
(c) Strukturbaum der Beziehungssicht `Teilnahme`: `K`

6.3.2 Beziehungssichten auf der Instanzenebene

Nachdem im vorangegangenen Abschnitt die Definition eines Datenbankschemas um Beziehungen und Beziehungssichten erweitert wurde, wenden wir uns nun der Instanzenebene zu. Wir erweitern dazu die Definition einer Datenbankinstanz. Wir „instanziieren“ eine Beziehung b über die für sie definierten Beziehungssichten (b, c).

Definition 6.3 (Erweiterung der Definition einer Datenbankinstanz)
Die Definition einer Datenbankinstanz INST(DS) zu einem Datenbankschema DS (vgl. Def. 3.20) wird um eine Menge

$$\{I(b, c) : \Omega \cdots> \mathcal{V}al^*(\text{TYPES}) \mid (b, c) \in \text{RELVIEWS}\}$$

von *Interpretationsfunktionen* für Beziehungssichten erweitert. Für diese Interpretationsfunktionen muß gelten:

- $\forall$ (b, c) $\in$ RELVIEWS: Def(I(b, c)) $\subseteq$ Def(I(*compType*(b, c))),
  d.h. nur solche Objekte ω können unter der Beziehungssicht (b, c) interpretiert werden, die auch den der Komponente c zugeordneten Komponententyp besitzen.
- $\forall$ (b, c) $\in$ RELVIEWS $\forall$ $\omega \in$ Def(I (b, c)): I (b, c) (ω) = (v, *struct*(b, c))
  für ein v $\in$ *dom*(*struct*(b, c)).

❒

Analog zu der gerade durchgeführten Erweiterung der Definition 3.20 erweitert sich auch die alternative Definition einer Datenbankinstanz auf der Grundlage von Baumrepräsentationen gemäß Def. 3.21:

Ein Funktionswert I(b, c)(ω) läßt sich im Sinne von Def. 3.21 auch als Wurzelknoten eines Wertebaumes B_v einer Baumrepräsentation (B_v, $B_{struct(b, c)}$) auffassen.

Für t $\in$ DEFTYPES sei

$$relviews(t) := \{(b, c) \in \text{RELVIEWS} \mid compType(b, c) = t\},$$

d.h. *relviews*(t) enthält alle Beziehungssichten, die eine Instanz des Objekttyps t als ihren Ausgangspunkt haben.

Durch die Einführung der Interpretationsfunktionen I(b, c) haben wir die Menge der für ein getyptes Datenobjekt (ω, t) relevanten Interpretationen erweitert. Bislang setzte sich der strukturelle Zustand von (ω, t) aus der Menge $\{I(t')(\omega) \mid t' \in \text{DEFTYPES} \wedge t\ isa^*\ t'\}$ zusammen[8]. Nun tragen auch alle Funktionswerte I(b, c)(ω) mit (b, c) $\in$ *relviews*(t) zum „erweiterten Zustand“ von (ω, t) bei.

Bei der Erzeugung eines neuen Objektes ω zum Typ t $\in$ DEFTYPES mittels `initObject` wird zunächst nur I(t)(ω) initialisiert. Alle Interpretationsfunktionen I(b, c) mit (b, c) $\in$ *relviews*(t) bleiben an der Stelle ω undefiniert, d.h. I(b, c)(ω) = $\perp$. Entsprechendes gilt auch für das Hinzufügen eines weiteren Typs zu einem bereits existierenden Objektes mittels `addType`.

6.3.3 Zulässige Beziehungssichten

In diesem Abschnitt wenden wir uns der Frage zu, was eigentlich „sinnvolle“ Referenzstrukturen für Beziehungssichten einer Beziehung sind. Die in der `view`-Klausel angegebenen Referenzstrukturen müssen zu den Komponenten, Attributen und Schlüsseln der Beziehung „passen“. Die Notwendigkeit der Einschränkung vollständiger Freiheit bei der Angabe der Struktur für eine Beziehungssicht zeigt folgendes Beispiel.

8. Man beachte die Berücksichtigung der Vererbung.

Beispiel 6.5 Wir betrachten wieder die Beziehung `Teilnahme` und die `define relship`-Anweisung aus Beispiel 6.2. Hätten wir dort die `view`-Klausel mit

```
-view P -> [Kurs:@K],
      K -> [T_Liste: < @Anm >]
```

angegeben, so wären zwei „unsinnige“ Beziehungssichten entstanden. Jede Person könnte danach höchstens einen Kurs besuchen, d.h. {`P`} wäre bereits ein Schlüssel der Beziehung. Über die Beziehungssicht `Teilnahme:K` wäre jedem Kurs eine Liste von Anmeldedaten zugeordnet. Dies ergibt aber selbst dann keinen Sinn, wenn für die Beziehungssicht `Teilnahme:P` die Referenzstruktur `[Kurse: {@K}]` gewählt worden wäre, da die Zuordnung von Personen zu Anmeldedaten nicht ersichtlich wäre. Schließlich läßt sich auch keine vollständige Beziehungsinstanz rekonstruieren, da das Beziehungsattribut `Erg` in keiner Beziehungssicht auftaucht! ❒

Da für eine Beziehungssicht (b, c) die Namen der Tupelkomponenten in *struct*(b, c) hinsichtlich der Zulässigkeit der Beziehungssicht irrelevant sind, reduzieren wir die Notation für Referenzstrukturen weiter, indem wir die Namen für Tupelkomponenten weglassen. Für die Referenzstrukturen aus Beispiel 6.2 erhalten wir also die reduzierten Referenzstrukturen `[{@K}]` bzw. `[<[@P,@Anm,@Erg]>]`.

Die Frage, welches nun die „sinnvollen“ Referenzstrukturen für eine Beziehungssicht (b, c) sein sollen, wird durch die nachfolgende Definition beantwortet.

Definition 6.4 (zulässige Referenzstruktur)

Es sei nun eine Beziehung b ∈ RELSHIPS mit *struct*(b) = [c_1: t_1, …, c_n: t_n, a_1: s_1, …, a_m: s_m] gegeben.

Wir nennen s ∈ $\mathcal{RefS}$(b) eine *zulässige* Referenzstruktur für die Beziehungssicht (b, c_i), wenn s aus *struct*(b) durch Anwendung folgender Umformungsschritte entsteht:

1. Schritt: Erzeugung einer „Default“-Referenzstruktur

Es sind zwei Fälle zu unterscheiden.

Fall 1: {c_i} ∉ *keys*(b) [9]

Setze s := `[{[@`c_1`, …, @`c_{i-1}`, @`c_{i+1}`, …, @`c_n`, @`a_1`, …, @`a_m`]}]`

Fall 2: {c_i} ∈ *keys*(b)

Setze s := `[@`c_1`, …, @`c_{i-1}`, @`c_{i+1}`, …, @`c_n`, @`a_1`, …, @`a_m`]`

In Fall 1 sind durch die Vorgabe eines Objektes ω zur Komponente c_i die anderen Komponenten und Attribute der Beziehung nicht eindeutig bestimmt, und deshalb muß die Referenzstruktur eine *Menge* von Tupeln beschreiben. Die Tupelstruktur auf der äußeren Ebene ist aufgrund von Bedingung (6.7) erforderlich. Für Fall 2 könnte theoretisch auch die Referenzstruktur aus Fall 1 gewählt werden, allerdings ist dies nicht sinnvoll: Jedem Objekt ω, das in der Rolle c_i an der Beziehung teilnimmt, würde in diesem Fall über die Beziehungssicht (b, c_i) immer eine einelementige Menge zugeordnet werden.

9. c_i ist kein Schlüssel der Beziehung, kann aber zu einem zusammengesetzten Schlüssel gehören.

2. Schritt: Projektion auf die relevanten Komponenten und Attribute

Sollen gewisse Komponenten bzw. Attribute aus $\{c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n, a_1, \ldots, a_m\}$ durch die Beziehungssicht (b, c_i) nicht erfaßt werden, dann werden die entsprechenden Referenzen $@c_v$ bzw. $@a_\mu$ aus der nach Schritt 1 erhaltenen Referenzstruktur entfernt, d.h. es findet eine Projektion auf die für die Beziehungssicht benötigten Komponenten und Attribute statt.

Die in der Projektion enthaltenen Referenzen bezeichnen wir fortan mit $@x_1, @x_2, \ldots, @x_q$, da im folgenden nicht mehr zwischen Komponenten und Attributen unterschieden werden muß.

Hinsichtlich des Streichens von Referenzen sind folgende Regeln zu beachten:

(i) Es muß $q \geq 1$ sein, denn selbstverständlich dürfen nicht alle Referenzen gestrichen werden

(ii) Die Projektion muß alle Schlüsselattribute enthalten,
d.h. $\bigcup_{K \in \mathit{keys}(b)} K \subseteq \{c_i, x_1, \ldots, x_q\}$. Dies ist notwendig, damit Modifikationen in der Beziehungssicht (b, c_i) immer in eindeutig bestimmte Modifikationen in den anderen Beziehungssichten (b, c') umgesetzt werden können.

(iii) Für jedes $x \in \{c_1, \ldots, c_n, a_1, \ldots, a_m\}$ muß x entweder der Ausgangspunkt einer Beziehungssicht (b, x) sein, oder @x ist in der Referenzstruktur mindestens einer Beziehungssicht aus RELVIEWS(b) enthalten sein. Andernfalls würden die Beziehungssichten aus RELVIEWS(b) die Beziehung b nur unvollständig wiedergeben.

3. Schritt: Entfernen überflüssiger Tupelebenen

Ist in einer Referenzstruktur nach Anwendung der Schritte 1 und 2 eine Teilstruktur der Gestalt `{[@x]}` enthalten, so kann sie zu `{@x}` vereinfacht werden.

Dieser Schritt ist beispielsweise für binäre Beziehungen ohne Attribute relevant. Ist $\mathit{struct}(b) = [c_1: t_1, c_2: t_2]$ und ist $\{c_1\} \notin \mathit{keys}(b)$, dann erhalten wir für die Beziehungssicht (b, c_1) nach Schritt 1 die Referenzstruktur $s = [\{[@c_2]\}]$. In diesem Schritt kann die Referenzstruktur zu $[\{@c_2\}]$ vereinfacht werden.

4. Schritt: Zusammenfassung von Tupelkomponenten in Subtupeln

Innerhalb eines Tupels können zusammengehörige Attribute zu einem neuen Subtupel aggregiert werden[10]. Für Strukturen von Beziehungssichten besteht zwar im Regelfall wenig Bedarf an dieser Form der zusätzlichen Strukturierung, sie soll jedoch nicht ausgeschlossen werden.

In einer Referenzstruktur s sei nach Anwendung der Schritte 1 – 3 eine Teilstruktur der Gestalt

$$[@x_1, \ldots, @x_k] \tag{6.8}$$

enthalten. Ist $\{i_1, \ldots, i_p\} \subseteq \{1, \ldots, k\}$ sowie $\{j_1, \ldots, j_{k-p}\} := \{1, \ldots, k\} - \{i_1, \ldots, i_p\}$, dann ist auch die Referenzstruktur s' zulässig, die aus s entsteht, wenn die Teilstruktur (6.8) ersetzt wird durch

$$[[@x_{i_1}, \ldots, @x_{i_p}], @x_{j_1}, \ldots, @x_{j_{k-p}}]$$

Zur Illustration dieses Schrittes betrachte man die Beziehung `Mitarbeit` aus Beispiel 6.3. Für die Beziehungssicht `Mitarbeit:M` wird dort die Referenzstruktur

10. Man erinnere sich, daß beispielsweise für den Objekttyp `Person` die Attribute `PLZ`, `Ort` und `Strasse` zu einem tupelwertigen Attribut `Adresse` aggregiert wurden.

```
[Abteilung: @A, Eintritt: @seit]
```

angegeben. Um die Zusammengehörigkeit der Attribute `Abteilung` und `Eintritt` zu unterstreichen, kann eine weitere Tupelebene eingeführt und für die Beziehungssicht `Mitarbeit:M` die Referenzstruktur

```
[Mitarbeit:[Abteilung: @A, Eintritt: @seit]]          (6.9)
```

gewählt werden. Dazu muß in diesem Schritt die reduzierte Referenzstruktur `[@A, @seit]` in `[[@A, @seit]]`umgeformt werden.

5. Schritt: Ersetzen des Mengen- durch den Listenkonstruktor

Ist der Ausgangspunkt c_i der Sicht kein Schlüssel der Beziehung, dann enthält die nach Schritt 1 entstehende Struktur eine *Menge* von Tupeln. Es wurde bereits am Ende von Abschnitt 6.1.1 darauf hingewiesen, daß in einigen Fällen eine benutzerdefinierte Ordnung auf den Elementen gewünscht sein kann, d.h. an die Stelle der Menge soll eine *Liste* treten (vgl. die Struktur der Beziehungssicht `Teilnahme:K` aus Beispiel 6.2).

Daher erlauben wir, in diesem Schritt den Mengen- durch den Listenkonstruktor zu ersetzen.

6. Schritt: Vergabe von Namen für die Tupelkomponenten

Nachdem in den Schritten 1 – 5 Referenzstrukturen ausschließlich in der reduzierten Form (d.h. ohne Namen für die Tupelkomponenten) vorkamen, werden in diesem letzten Schritt Namen $\alpha \in \mathcal{A}$ für die Tupelkomponenten vergeben. Wir erhalten somit schließlich Referenzstrukturen aus $\mathcal{RefS}$(b), wie sie in der `view`-Klausel einer ESCHER$^+$-Beziehungsdefinition vorkommen.

Durch Vergabe von Attributnamen wird die unter Schritt 5 genannte reduzierte Referenzstruktur `[[@A, @seit]]`in die Form (6.9) gebracht. ❒

Man überzeugt sich leicht, daß sich alle in den bisherigen Beispielen angegebenen Referenzstrukturen für Beziehungssichten gemäß der in Def. 6.4 angegebenen Schritte konstruiert werden können.

In einer möglichen Erweiterung der Menge der zulässigen Referenzstrukturen ließen sich im Anschluß an Schritt 2 weitere Schachtelungsebenen in die bis dahin vorliegende Referenzstruktur einführen. Allerdings bleiben wir im Rahmen dieser Arbeit bei den oben angegebenen Umformungsschritten und beschränken uns auf die Angabe eines Beispiels zur weiteren Schachtelung von Referenzstrukturen.

Beispiel 6.6 Für die Beziehungssicht `Unterricht:K` aus Beispiel 6.4 ist durch weitere Nestung folgende Referenzstruktur möglich:

<table>
<tr><td colspan="3">Stundenplan:{}</td></tr>
<tr><td rowspan="2">Lehrer:@L</td><td colspan="2">Unterricht:{}</td></tr>
<tr><td>Fach:@F</td><td>Anz_Std:@Std</td></tr>
</table>

Vertauscht man in dieser Referenzstruktur die Attribute `Lehrer` und `Fach`, so ist die so entstehende Referenzstruktur nicht sinnvoll, da dann jede Menge `Unterricht` grundsätzlich

einelementig sein wird: Ist k ein Objekt des Typs `Klasse` und f ein Wert des Attributes `Fach` in I(`Unterricht:K`)(k), dann bilden k und f zusammen eine Wertekombination des Schlüssels `[K, F]`, so daß k und f zusammen auch Lehrer und Stundenzahl eindeutig bestimmen.

6.3.4 Der Zugriff auf die Attribute einer Beziehungssicht

Unser Ziel ist es, auf die Attribute aus *Attrs*(b, c), d.h. auf die in *struct*(b, c) definierten Attribute der Beziehungssicht (b, c), über Pfadausdrücke so zugreifen zu können, als ob diese Attribute zur Definition des Objekttyps t = *compType*(b, c) selbst gehören. Wir geben dazu ein Beispiel an:

Ist `p` ein Bezeichner vom Typ `Person`, dann soll `p.Kurse` ein ebenso gültiger Pfadausdruck sein wie `p.Name.VName`.

Die „Auslagerung“ von Attributen aus Objekttypdefinitionen in Beziehungssichten soll also vollkommen transparent sein. Um das angestrebte Ziel zu erreichen, müssen folgende Punkte berücksichtigt werden:

- Es müssen zusätzliche Bedingungen für die Eindeutigkeit von Attributnamen angegeben werden.

  Für alle t ∈ DEFTYPES muß gelten:

  $\forall$ (b, c) ∈ *relviews*(t): *Attrs*(b, c) ∩ *Attrs*(t) = ∅.

  $\forall$ (b, c), (b', c') ∈ *relviews*(t) : (b, c) ≠ (b', c') ⇒ *Attrs*(b, c) ∩ *Attrs*(b', c') = ∅.

  Das bedeutet, daß alle Attribute, die durch eine Beziehungssicht definiert werden, weder in der Struktur von t noch in einer anderen Beziehungssicht vorkommen dürfen. Nur so können Mehrdeutigkeiten beim Zugriff auf ein Attribut vermieden werden.

- Die Definition der Funktionen *AttrAccess*(t) für t ∈ DEFTYPES muß revidiert werden, um dem erweiterten Zustandsbegriff gerecht zu werden:

  Für t ∈ DEFTYPES ersetzen wir die Funktion *AttrAccess*(t) aus (3.28) durch

  AttrAccess (t) : $\mathcal{N}$ ···> (DEFTYPES ∪ RELVIEWS) × IN

  Ferner wird Schritt 1 im Algorithmus aus Abschnitt 3.7.1 zur Berechnung von *AttrAccess*(t) ersetzt durch:

  1.Setze f_t : $\mathcal{N}$ ···> (DEFTYPES ∪ RELVIEWS) × IN mit
  Def(f_t) := *Attrs*(t) und f_t (a) := (t, *attr_pos* (*struct* (t)) (a)) für a ∈ Def(f_t).
  Für alle (b, c) ∈ *relviews*(t) führe durch:
  Setze Def((f_t) := Def((f_t) ∪ *Attrs*(b, c) und
  f_t (a) := (t, *attr_pos* (*struct* (b, c)) (a)) für a ∈ *Attrs*(b, c).

  Die Tatsache, daß nun auch die Attribute der Beziehungssichten aus *relviews*(t) im Definitionsbereich von *AttrAccess*(t) liegen, ist ein erster Schritt in Richtung „Gleichbehandlung“ der Kern-Attribute und der Attribute der Beziehungssichten.

- Hinsichtlich der Semantik von Nullwerten wurden in ESCHER$^+$ bislang keine konkreten Festlegungen getroffen. Je nach Fall kann *null* die Semantik „Wert existiert, ist aber unbekannt“ („unknown“, *unk*-Semantik) oder aber die Semantik „Wert existiert nicht“ („does not exist“, *dne*-Semantik) haben.

Im Zusammenhang mit Beziehungssichten für Komponenten, die Schlüssel der Beziehung sind[11], ist jedoch eine genauere Unterscheidung notwendig. Wir betrachten wieder die Beziehung `Mitarbeit` aus Beispiel 6.3: Es sei `ang` ein Bezeichner für eine Instanz α des Objekttyps `Angestellter`. Zwei Fälle sind zu unterscheiden:

Fall 1: I(`Mitarbeit:M`)(α) = `[Abteilung: null, Eintritt: null]`.

Damit soll ausgedrückt werden, daß α zwar zu einer Abteilung gehört, die jedoch ebensowenig bekannt ist wie das Eintrittsdatum (*unk*-Semantik).

Für den Pfadausdruck `ang.Abteilung` ist σ(`ang.Abteilung`) = (*null*, $t_{Abteilung}$) [12].

Fall 2: I(`Mitarbeit:M`)(α) = $\perp$.

Damit wird ausgedrückt, daß α zu gar keiner Abteilung gehört. Auch in diesem Fall ist σ(`ang.Abteilung`) = (*null*, $t_{Abteilung}$) bislang die einzig „vernünftige“ Antwort auf die Frage nach der Wertesemantik von `ang.Abteilung`. Dann wären aber die Fälle 1 und 2 beim Zugriff auf `ang.Abteilung` nicht unterscheidbar!

Es zeigt sich an diesem Beispiel der Bedarf an einer genaueren Differenzierung hinsichtlich der Semantik von Nullwerten. Wir schlagen deshalb vor, im Fall von Beziehungssichten für Komponenten, die Schlüssel der Beziehung sind, zwischen $null_{unk}$ und $null_{dne}$ zu unterscheiden. Im Fall 1 gibt dann $null_{unk}$, im Fall 2 hingegen $null_{dne}$ die adäquate Semantik der Ausdruckes `ang.Abteilung` an.

Es sollen $null_{unk}$ und $null_{dne}$ jedoch nicht zwei verschiedene Nullwerte sein, die den bisherigen Nullwert *null* ersetzen. Vielmehr sind die Semantiken *unk* und *dne* als optionale Attributierungen des bisherigen Nullwertes *null* zu verstehen. In BASESCRIPT bzw. SCRIPT$^+$ sollen $null_{unk}$ bzw. $null_{dne}$ durch `null` bzw. `dne` bezeichnet werden.

Nun folgt der entscheidende Schritt, der zur angestrebten „Gleichbehandlung“ aller Attribute in Pfadausdrücken führt: Wir erweitern die Definition der Zugriffsfunktion μ aus Abschnitt 4.6.2. Davon ist Regel 3 betroffen, die die Semantik für den Zugriff auf die Attribute eines Objektes bestimmt. Die modifizierte Regel 3 lautet nun:

Es sei e = p.a $\mathcal{L}$(*pathExpr*), mit p $\in$ $\mathcal{L}$(*pathExpr*), a $\in$ $\mathcal{L}$(*attrName*), μ (p) sei definiert,
val(μ (p)) = (ω, t) mit $\omega \in \Omega$ und t $\in$ DEFTYPES,
a $\in$ Def(*AttrAccess* (t))
Fall 1: *AttrAccess* (t) (a) = (t', n) $\Rightarrow$ μ (e) = *get_child* (*I* (t') (ω), n)

Fall 2: *AttrAccess* (t) (a) = ((b, c), n)

Fall 2.1: $\omega \in$ Def(I(b, c)) $\Rightarrow$ μ (e) = *get_child* (*I* (b, c) (ω), n).

Fall 2.2: $\omega \notin$ Def(I(b, c)) $\Rightarrow$ I(b, c)(ω) := *create*(*initRelView*(*struct*(b, c)));
μ (e) = *get_child* (*I* (b, c) (ω), n).

Dabei ist *initRelView*: $\mathcal{S}$ (TYPES) $\rightarrow$ $\mathcal{V}al^*$(TYPES)
die Funktion zur Initialisierung einer Beziehungssicht, die wie folgt definiert ist:

$$initRelView(s) := \begin{cases} (null_{dne}, s) & \text{, falls } s \in \text{TYPES} - \text{CONSTR} \\ initVal(s) & \text{, sonst} \end{cases}$$

11. Vgl. Def. 6.4, Fall 2 in Schritt 1.
12. Dabei ist σ die Wertesemantik für Ausdrücke, vgl. (4.12), (4.13).

Damit ist gewährleistet, daß der Zugriff auf Attribute aus *Attrs*(b, c) über Pfadausdrücke jederzeit korrekt ist, auch wenn dazu auf eine bislang nicht definierte Interpretation I(b, c)(ω) zugegriffen werden muß. In diesem Fall ist die Auswertung eines Pfadausdrucks p.a ∈ $\mathcal{L}$(*pathExpr*) mit einem Nebeneffekt verbunden. Dieser besteht darin, daß das Initialisieren von I(b, c)(ω) beim Zugriff „nachgeholt“ wird, indem I(b,c)(ω) := *create*(*initRelView*(*struct*(b, c))) gesetzt wird.

Wir erläutern dieses Vorgehen anhand eines weiteren Beispieles.

Beispiel 6.7 Wir betrachten die Beziehung `Ausleihe` aus dem ER-Schema aus Abb. 1.8. Wir können sie über

```
define relship
  -name Ausleihe
  -comp [L: Leser, E: Exemplar]
  -attr [LDat: date]
  -key  E
  -view L -> [Ausleihen:{[Exemplar: @E, Leihdatum: @LDat]}],
       E -> [entliehen_von: @L, entliehen_am: @LDat]
end define;
```

einem ESCHER$^+$-Datenbankschema hinzufügen.

Es sei nun `leser` eine Variable, die an ein `Leser`-Objekt λ gebunden ist. Es gelte I(`Ausleihe:L`)(λ) = ⊥, d.h. Leser λ hat momentan keine Bücher entliehen. Es sei `A` eine lokale Variable mit der Struktur des Attributes `Ausleihen` der Beziehungssicht `Ausleihe:L`. Die SCRIPT$^+$-Anweisung

```
A := leser.Ausleihen;                                    (6.10)
```

wird in eine gleichlautende BASESCRIPT-Anweisung übersetzt, die gemäß den Ausführungen aus Abschnitt 4.6.4 (Abarbeitung einer Zuweisung, Fall (i)) abgearbeitet wird. Dabei muß insbesondere μ(`leser.Ausleihen`) bestimmt werden. Da I(`Ausleihe:L`)(λ) undefiniert ist, wird I(`Ausleihe:L`)(λ) zunächst mit `[Ausleihen:{}]` initialisiert. Nach Ausführung der Anweisung (6.10) ist die lokale Variable `A` folglich an die leere Menge gebunden. ❒

Bei der Abarbeitung der Anweisung (6.10) findet jedoch keine weitere Modifikation von I(`Ausleihe:L`)(λ) statt. Leser λ hat nach wie vor keine Bücher ausgeliehen. Um die Nichtteilnahme von λ an der Beziehung `Ausleihe` zu dokumentieren, wäre es plausibel, die im Zuge der Bestimmung von μ(`leser.Ausleihen`) durchgeführte Initialisierung von I(`Ausleihe:L`)(λ) mit `[Ausleihen:{}]` wieder rückgängig zu machen, d.h. I(`Ausleihe:L`)(λ) := ⊥ zu setzen.

Noch plausibler ist die Zurücknahme einer Initialisierung für die komplementäre Sicht auf die Beziehung `Ausleihe` aus der Perspektive eines `Exemplar`-Objektes ε. Gilt I(`Ausleihe:E`)(ε) = ⊥, bezeichnet `e` das (Buch-)Exemplar ε und ist die Bestimmung von μ(`e.entliehen_von`) oder μ(`e.entliehen_am`) erforderlich, dann soll I(`Ausleihe:E`)(ε) nur temporär mit *initRelView*(*struct*(`Ausleihe:E`)) initialisiert werden. Es ist günstig, wenn für nicht entliehene Exemplare ε grundsätzlich I(`Ausleihe:E`)(ε) = ⊥ gilt. Die Alternative wäre I(`Ausleihe:E`)(ε) = `[entliehen_von: dne, entliehen_am: dne]`. Da aber im Regelfall nur ein kleiner Teil des gesamten Bibliotheksbestan-

des entliehen ist, würde dies ein unnötige Überfrachtung der Datenbankinstanz mit Nullwerten bedeuten.

Wir schlagen deshalb folgendes Vorgehen vor:

Wird zur Bestimmung von $\mu(e)$ eines Pfadausdruckes e eine Initialisierung

$I(b, c)(\omega) := create(initRelView(struct(b, c)))$;

notwendig, dann wird am Ende der aktuellen Transaktion wieder

$I(b, c)(\omega) := \perp$

gesetzt, sofern zu diesem Zeitpunkt

$$val(I(b, c)(\omega)) = initRelView(struct(b, c)) \tag{6.11}$$

gilt.

Natürlich darf eine Initialisierung nicht rückgängig gemacht werden, falls die Bedingung (6.11) nicht gilt. Dies ist z.B. nach der Ausleihe des Exemplares `e` durch `leser` der Fall, was der Ausführung von

```
leser.Ausleihen add [Exemplar: e, Leihdatum: d];
```

bzw. von

```
e.entliehen_von := leser; e.entliehen_am := d;
```

entspricht.

Es soll nun genau spezifiziert werden, in welchen Situationen die Verwendung von `dne` in SCRIPT$^+$-Ausdrücken und -Anweisungen erlaubt und welche Semantik damit jeweils verbunden sein soll:

Es sei (b, c) eine Beziehungssicht einer Beziehung b mit $\{c\} \in keys(b)$.
Es sei $e = p_1p_2 \in \mathcal{L}(pathExpr)$ ein Pfadausdruck mit

- $\mu(p_1) = r \in \mathcal{V}$ und $val(r) \in (\omega, compType(b, c))$
- $p_2 \in Def(corresp(b, c)) - \{c\}$, d.h. $e = p_1p_2$ führt zu einem Blatt von $Tree(I(b, c)(\omega))$, das einer Komponente bzw. einem Attribut der Beziehung b repräsentiert.

Dann soll gelten:

- Die Vergleiche e `=` `dne` bzw. e `<>` `dne` sind erlaubt.
- Die Zuweisung e `:=` `dne` ist erlaubt und hat den Nebeneffekt, daß $I(b, c)(\omega) := \perp$ gesetzt wird, d.h. nach einer Zuweisung von `dne` an eine beliebige Tupelkomponente in $I(b,c)(\omega)$ nimmt ω nicht mehr in der Rolle c an der Beziehung b teil.
- Eine Zuweisung eines Datenobjektes ungleich $null_{dne}$ an e führt dazu, daß alle anderen Tupelkomponenten in $I(b,c)(\omega)$ auf $null_{unk}$ gesetzt werden. Damit ist erreicht, daß entweder alle Komponenten den Wert $null_{dne}$ besitzen oder aber keine diesen Wert hat.

In allen anderen Fällen ist die Verwendung von `dne` in SCRIPT$^+$-Ausdrücken und -Anweisungen nicht zulässig. Somit läßt sich beispielsweise der obige Ausleihvorgang nicht nur durch Löschen des entsprechenden Tupels aus `leser.Ausleihen` rückgängig machen, sondern auch durch eine der Zuweisungen `e.entliehen_von := dne` oder `e.entliehen_am := dne`.

Zuletzt sei in diesem Abschnitt darauf hingewiesen, daß sich das in Abschnitt 5.5.1 vorgestellte Konzept zur Einkapselung von Attributen eines Objekttyps ohne weiteres auch auf

Beziehungssichten übertragen läßt. Dies halten wir sogar für unbedingt erforderlich, da die Attribute von Beziehungssichten als Attribute des erweiterten Zustandes von Objekten in jeder Hinsicht den Kern-Attributen der Objekttypen gleichgestellt sein sollen, und dies muß folglich auch die Möglichkeit ihrer Einkapselung einschließen. Eine Einschränkung der erlaubten Updateoperationen auf Beziehungssichten kann für *unvollständige* Beziehungssichten erforderlich werden. Als unvollständig wird eine Beziehungssicht bezeichnet, die nicht alle Komponenten und Attribute einer Beziehung erfaßt, wie z.B. `Teilnahme:P` aus Beispiel 6.2: Beim Hinzufügen eines Kurses `k` zu `p.Kurse` fehlt die Information für die Beziehungsattribute. Es bietet sich an, die Komponenten `Anmeldung` und `Ergebnis` des in `k.T_Liste` einzufügenden Tupels mit `null` zu belegen. Falls jedoch eine Integritätsbedingung (siehe folgenden Abschnitt) für die Beziehung verlangt, daß das Anmeldedatum nicht nullwertig sein darf, dann darf das Hinzufügen eines weiteren Kurses `k` zu `p.Kurse` nicht erlaubt sein. Dazu kann `Teilnahme:P` zu einer „readonly“-Sicht gemacht werden, indem der Wurzelknoten r des zugehörigen Strukturbaumes mit *enc*(r) = *read* beschriftet wird.

6.4 Integritätsbedingungen für Beziehungen

In diesem Abschnitt diskutieren wir verschiedene Integritätsbedingungen für Beziehungen, die innerhalb einer `define relship`-Anweisung angegeben werden können. Für alle diese Integritätsbedingungen werden bei der Abarbeitung der DDL-Anweisung Integritätsbedingungen auf den Beziehungssichten (b, c) ∈ RELVIEWS(b) generiert. Mit ihnen liegen dann Integritätsbedingungen in der Form vor, wie sie im vorangegangenen Kapitel behandelt wurden. Zu ihrer Überwachung können diese schließlich in interne ECA-Regeln transformiert werden.

Wir gehen im folgenden davon aus, daß RELVIEWS(b) = *comps*(b) gilt, d.h. für alle Komponenten c der Beziehung soll eine Beziehungssicht (b, c) definiert sein. Andernfalls ließen sich einige der für eine Beziehung b angegebenen Integritätsbedingungen nicht direkt in Integritätsbedingungen auf den Beziehungssichten umsetzen.

Alle für die Beziehungssichten generierten Integritätsbedingungen sollen den Modus `hard` haben, d.h. sie werden sofort überprüft, wenn eine potentielle Integritätsvorletzung vorliegt. Von dieser Regel ausgenommen sind Integritätsbedingungen, die aufgrund von Kardinalitätsbedingungen (vgl. Abschnitt 6.4.2) generiert werden. Für sie soll der Modus `soft` gelten, d.h. sie werden erst an Ende der Transaktion überprüft. Damit wird ein temporäres Unter- bzw. Überschreiten der Minimal- bzw. Maximalkardinalität im Verlauf der Transaktion ermöglicht.

6.4.1 Schlüsselbedingungen

Bereits mit der Definition von Beziehungen wurden Schlüssel für Beziehungen eingeführt, die in der Menge *keys*(b) für b ∈ RELSHIPS zusammengefaßt werden. Schlüssel werden direkt in der `key`-Klausel[13] einer `define relship`-Anweisung definiert oder ergeben sich implizit aus (min, max)-Kardinalitätsbedingungen (siehe Abschnitt 6.4.2).

13. Hinsichtlich der vollständigen Syntax vgl. Anhang B.2 (Nichtterminal *relKeys*).

Im folgenden verzichten wir auf eine allgemeine Beschreibung des Verfahrens zur Generierung von Integritätsbedingungen auf den Beziehungssichten aus der Menge *keys*(b). Wir beschränken uns auf die Angabe repräsentativer Beispiele.

Beispiel 6.8

a) Für die Beziehung `Teilnahme` aus Beispiel 6.2 wurde der Schlüssel `[P, K]` angegeben. Daraus ergibt sich die Schlüsselbedingung

"`Teilnahme:K.T_Liste has key Teilnehmer`",

für die Beziehungssicht `Teilnahme:K`. Da `K` bereits der Ausgangspunkt der Beziehungssicht ist, „verkürzt“ sich der Schlüssel `[P, K]` zu `P`, was in `Teilnahme:K` der Tupelkomponente `Teilnehmer` entspricht.

Für das mengenwertige Attribut `Kurse` der Beziehungssicht `Teilnahme:P` muß hingegen keine Schlüsselbedingung generiert werden, da Duplikate für Mengen per definitionem ausgeschlossen sind. Da jedoch Schlüsselkomponenten grundsätzlich nicht nullwertig sein dürfen, muß für die Elemente der Menge `Kurse` die lokale Wertebedingung

"`k <> null with k = Teilnahme:P.Kurse.#`"

generiert werden.

b) Man betrachte die Beziehung `Ausleihe` aus Beispiel 6.7. Da {`E`} ∈ *keys*(`Ausleihe`), erscheint auf den ersten Blick die Generierung der Schlüsselbedingung

"`Ausleihe:L.Ausleihen has key Exemplar`"

für die Beziehungssicht `Ausleihe:L` erforderlich. Allerdings ist diese Schlüsselbedingung überflüssig, denn ein Exemplar kann nicht mehrfach entliehen werden (und von ein und demselben Leser schon gar nicht!): Bezeichnet `leser` ein `Leser`-Objekt und soll in `leser.Ausleihen` ein neues Tupel `t` mit `t.Exemplar <> null` eingefügt werden, dann ist dies nur erlaubt, falls `t.Exemplar.entliehen_von = dne` gilt.
So muß aufgrund von {`E`} ∈ *keys*(`Ausleihe`) lediglich die lokale Wertebedingung

"`e <> null with e = Ausleihe:L.Ausleihen.#.Exemplar`",

generiert werden, denn Schlüsselkomponenten dürfen nicht nullwertig sein. ❒

6.4.2 Kardinalitätsbedingungen

Kardinalitätsbedingungen in Form von (min, max)-Kardinalitäten können über die `card`-Klausel[14] einer Beziehungsdefinition angegeben werden.

Für die Beziehung `Mitarbeit` aus Beispiel 6.3 ist beispielsweise

```
-card A (0, *), M (1, 1)
```

eine sinnvolle `card`-Klausel. Die Angabe "`M (1, 1)`" fordert, daß `M` ein Schlüssel der Beziehung ist und daß jeder Angestellte in einer Abteilung mitarbeiten *muß*. Somit ist die Angabe "`-key M`" redundant, da sich {`M`} ∈ *keys*(b) aus der Maximalkardinalität für `M` ableiten läßt.

14. Hinsichtlich der genauen Syntax sei auf Anhang B.2 (Nichtterminal *relCard*) verwiesen.

Wir gehen davon aus, daß `(0, *)` die „Default“-Kardinalität ist, so daß im obigen Beispiel "`A (0, *)`" auch weggelassen werden könnte.

Ferner sollen die Angaben der `key`- und der `card`-Klausel zueinander „passen“:

So ist z.B. "`-key M`" zusammen mit "`-card M (0, 5)`" nicht sinnvoll, da aufgrund der Schlüsselbedingung die maximale Kardinalität nie größer als 1 sein kann.

In der `card`-Klausel einer Beziehung b sei die Angabe "c_i `(`n_{min}`,` n_{max}`)`" enthalten. Dann wird für die Beziehungssicht (b, c_i) eine lokale Wertebedingung generiert:

Fall 1: $\{c_i\} \notin$ *keys*(b).

Ist `(`n_{min}`,` n_{max}`)` $\neq$ `(0, *)`, dann wird die lokale Wertebedingung

"`card(x) >=` n_{min} `and card(x) <=` n_{max} `with x =` *name*(b)`:`c_i`.`α_0"

generiert. Falls $n_{min} = 0$ oder $n_{max} =$ `*`, dann wird der entsprechende Vergleich in der Bedingung weggelassen.

Fall 2: $\{c_i\} \in$ *keys*(b).

Ist `(`n_{min}`,` n_{max}`)` = `(1, 1)`, dann wird die lokale Wertebedingung

"`x.k <> dne with x =` *name*(b)`:`c_i"

generiert, wobei k $\in$ *Attrs*(b, c_i) beliebig ist.

Im Fall 2 kann die Prüfung auf eine beliebige Tupelkomponente beschränkt bleiben, denn entweder liefert der Zugriff auf alle Tupelkomponenten den Wert `dne`, oder aber kein Zugriff auf eine Tupelkomponente liefert `dne`.

Für das obige Beispiel ergibt sich demnach die lokale Wertebedingung

"`x.Abteilung <> dne with x = Mitarbeit:M`"

Wie bereits erwähnt, sollen die aufgrund der `card`-Klausel generierten Integritätsbedingungen erst an Ende der Transaktion überprüft werden, d.h. sie haben den Modus `soft`.

6.4.3 Wertebedingungen für Komponenten und Attribute

In der `val`-Klausel einer `define relship`-Anweisung können weitere elementare Bedingungen für Komponenten und Attribute einer Beziehung angegeben werden, die eine Einschränkung der für sie zulässigen Werte darstellen und vergleichbar mit den *column constraints* in SQL/92 sind. Die Syntax der `val`-Klausel ist gegeben durch

relVal ::= `-val` *expr* { `','` *expr* }

Dabei soll *expr* ein boolescher Ausdruck sein, in dem als einzige Variable eine Komponente oder ein Attribut x $\in$ *comps*(b) $\cup$ *attrs*(b) vorkommt.

Beispiel 6.9

a) Für die Beziehung `Teilnahme` aus Beispiel 6.2 werde der Wert des Attributes `Erg` (Anzahl der in einem Kurs-Test erreichten Punkte) auf das Intervall [0, 10] eingeschränkt:

```
-val 0 <= Erg and Erg <= 10
```

b) Für die Beziehung `Unterricht` aus Beispiel 6.4 werde

```
-val L <> null, Std > 0
```

angegeben, d.h. die Angabe eines Lehrers zu einer Fächer-Klasse-Kombination ist zwingend, und die Anzahl der Unterrichtsstunden muß größer 0 sein. ❒

Es sei e ∈ $\mathcal{L}$(*expr*) eine Bedingung der `val`-Klausel für ein x ∈ *comps*(b) ∪ *attrs*(b). Die Bedingung e wird in lokale Wertebedingungen für alle diejenigen Beziehungssichten (b, c) transformiert, für die x ∈ Bild(*corresp*(b, c)) gilt, d.h. die x erfassen.

Ist (b, c) eine Beziehungssicht mit *corresp*(b, c)(p) = x, dann wird die lokale Wertebedingung

"e `with` x = *name*(b)`:cp`"

generiert.

Für die Bedingungen aus Beispiel 6.9 werden u.a. die Integritätsbedingungen

```
"0 <= Erg and Erg <= 100 with Erg = Teilnahme:K.Ergebnis"
"L <> null with L = Unterricht:K.Stundenplan.#.Lehrer"
"Std > 0 with Std = Unterricht:L.Dienstplan.#.Stunden"
```

generiert. Es ist aber darauf zu achten, daß zur Laufzeit keine redundanten Mehrfachprüfungen vorgenommen werden. So wird neben der zuletzt genannten Integritätsbedingung auch noch die Bedingung

```
"Std > 0 with Std = Unterricht:K.Stundenplan.#.Anz_Std"
```

generiert. Nach einem Einfügen eines neuen Tupels in die Sicht `Teilnahme:L` braucht bei der Propagierung des Updates in die Sicht `Unterricht:K` die Bedingung an die Anzahl der Stunden nicht erneut geprüft werden.

6.4.4 Inklusionsbedingungen

Zuletzt sollen Inklusionsbedingungen für Beziehungen eingeführt werden. Das nachfolgende Beispiel macht deutlich, daß auch im Zusammenhang mit Beziehungen die Angabe von Inklusionsbedingungen wünschenswert sein kann.

Beispiel 6.10 Es sei `DV_Kurse` eine ESCHER$^+$-Tabelle mit der Struktur `{Kurs}`. Für die Beziehung `Teilnahme` soll die Einschränkung gelten, daß Personen nur an solchen Kursen teilnehmen dürfen, die in `DV_Kurse` enthalten sind. Dies kann über die `incl`-Klausel einer `define relship`-Anweisung ausgedrückt werden. In diesem Beispiel wird dazu

```
-incl K in DV_Kurse
```

angegeben. Die Bedingung besagt, daß alle Objekte ω, die in der Rolle `K` an der Beziehung teilnehmen, ein Element von `DV_Kurse` sein müssen. ❒

Die generelle Syntax der `incl`-Klausel ist wie folgt gegeben:

relInclConstr ::= `-incl` *inclCondition* { `,` *inclCondition* }
inclCondition ::= *identifier* `in` *structPathExpr* [*cascade*] (6.12)

Bis auf weiteres lassen wir die optionale *cascade*-Klausel außer Betracht.

Eine Inklusionsbedingung der Form "c `in` s" ∈ $\mathcal{L}$(*inclCondition*) für die Beziehung b ist zulässig, falls gilt:

- $c \in comps(b)$
- $s \in \mathcal{L}(structPathExr)$ beginnt mit dem Namen einer Tabelle und enthält keinen `.#`-Schritt
- $isCollection(type(\mu_{struct}(s)) \wedge type(\mu_{struct}(s\texttt{.\#})) = compType(b, c)$, d.h. s bezeichnet einen Kollektionsknoten, und die Elemente der Kollektion gehören zum Typ der Komponente c.

Es liege eine zulässige Inklusionsbedingung "c `in` s" für die Beziehung b vor. Die formale Semantik dieser Integritätsbedingung unterscheidet sich geringfügig von der Semantik der Inklusionsbedingungen aus Abschnitt 5.3.2. Sie ist wie folgt gegeben:

> Eine Datenbankinstanz INST(DS) erfüllt die Inklusionsbedingung
> "c `in` s" für die Beziehung b $\Leftrightarrow$
> $\forall$ r $\in$ $V(D_{pers})$: $struct(r) = \mu_{struct}(name(b)\texttt{:}c) \Rightarrow \sigma(B; \texttt{t} \leftarrow r)$
> mit B = `getObject(t) in` s

Das bedeutet: Nimmt ein Objekt ω an der Beziehung b in der Rolle c teil, dann muß ω auch in s enthalten sein.

Wie bei den Inklusionsbedingungen aus Abschnitt 5.3.2 lassen sich auch für den Fall der Verletzung von Inklusionsbedingungen für Beziehungen Korrekturoperationen angeben, die die Gültigkeit der Bedingung wiederherstellen, wodurch ein Abbruch der aktuellen Transaktion vermieden werden kann:

- *Kaskadierendes Einfügen*: Nimmt ein Objekt ω erstmalig in der Rolle c an der Beziehung b teil (d.h. es findet ein Übergang von $I(b, c)(\omega) = \perp$ nach $I(b, c)(\omega) \neq \perp$ statt) und gilt bislang $\omega \notin s$, dann wird ω in die durch s bezeichnete Kollektion eingefügt.
- *Kaskadierendes Löschen*: Wird ein Objekt ω aus der Kollektion s entfernt, gilt jedoch weiterhin $\omega \in Def(I(b, c))$, dann wird $I(b, c)(\omega) := \perp$ gesetzt, wobei natürlich die anderen Beziehungssichten entsprechend angepaßt werden müssen.

In der optionalen *cascade*-Klausel in (6.12) kann angegeben werden, ob für eine Inklusionsbedingung eine der genannten Einfüge- bzw. Löschregeln gelten sollen: Wie in Abschnitt 5.3.2 fordert "`cascade add`" kaskadierendes Einfügen und "`cascade rem`" kaskadierendes Löschen. Wird "`cascade add, rem`" angegeben, dann sollen beide Korrekturregeln gelten.

6.5 Konsistenz bei Vorliegen von Beziehungsredundanz

Es soll nun eine formale Bedingung entwickelt werden, anhand der festgestellt werden kann, ob eine Datenbankinstanz hinsichtlich einer Beziehung $b \in$ RELSHIPS konsistent genannt werden kann.

Zunächst müssen alle Beziehungssichten einer Beziehung b die für sie generierten Integritätsbedingungen erfüllen. Dies reicht jedoch noch nicht aus. Aufgrund sich überlappender Beziehungssichten in einem Datenbankschema haben wir es auf der Instanzenebene mit Redundanz zu tun. Die in zwei verschiedenen, sich überlappenden Beziehungssichten enthaltene Informationen über eine Beziehung dürfen zueinander nicht im Widerspruch stehen. Dies ist durch eine geeignete Bedingung auszudrücken.

Um die in verschiedenen Beziehungssichten (b, c_i) enthaltene Information miteinander vergleichbar zu machen, führen wir sie jeweils auf Mengen von Tupeln $t \in dom(struct(b))$ zurück. Diese Tupel sollen gerade die „virtuellen" Beziehungsinstanzen sein, die aus der jeweiligen Beziehungssicht ableitbar sind. Wir nennen sie die von der Beziehungssicht (b, c_i) *induzierten Beziehungsinstanzen*.

Wir fassen alle von (b, c_i) induzierten Beziehungsinstanzen in der Menge *relInst*(b, c_i) zusammen.

Bestimmung von *relInst*(b, c_i)

Die Mengen *relInst*(b, c_i) ergeben sich durch die Auswertung von SCRIPT$^+$-Anfragen. In ihnen wird auf die Extensionen von Objekttypen zugegriffen, von denen wir wie in Abschnitt 5.3.1 annehmen, daß sie intern zur Verfügung stehen.

Wir setzen:

- $\{x_1, \ldots, x_q\} := \text{Bild}(corresp(b, c_i))$, d.h. $x_1, \ldots, x_q$ sind die von der Beziehungssicht (b, c_i) erfaßten Komponenten und Attribute.
- $\{x_{q+1}, \ldots, x_{m+n}\} := (comps(b) \cup attrs(b)) - \{x_1, \ldots, x_q\}$, d.h. $x_{q+1}, \ldots, x_{m+n}$ sind die von (b, c_i) nicht erfaßten Komponenten und Attribute.
- $x_1 := c_i$
- $n := name(compType(b, c_i))$

Es sind die beiden Fälle „$\{c_i\} \notin keys(b)$" und „$\{c_i\} \in keys(b)$" zu unterscheiden.

Fall 1: $\{c_i\} \notin keys(b)$.

Fall 1.1: Es gelte $struct(b, c_i) = [\alpha_0 : \{[\alpha_1 : s_1, \ldots, \alpha_{n_i} : s_{n_i}]\}]$ oder
$struct(b, c_i) = [\alpha_0 : <[\alpha_1 : s_1, \ldots, \alpha_{n_i} : s_{n_i}]>]$ für ein $n_i \in$ IN.

Für $2 \le \nu \le q$ sei $p_\nu \in \mathcal{L}(path)$ durch $corresp^{-1}(x_\nu) = .\alpha_0 p_\nu$ bestimmt.

relInst(b, c_i) :=

$\{[x_1 : y, x_2 : tp_2, \ldots, x_q : tp_q, x_{q+1} : \text{null}, \ldots, x_{m+n} : \text{null}]$
$\mid y \text{ in Ext}(n), t \text{ in } y.\alpha_0\}$

Die von der Beziehungssicht nicht erfaßten Attribute und Komponenten werden also in allen Tupeln mit Nullwerten belegt.

Fall 1.2: Es gelte $struct(b, c_i) = [\alpha_0 : \{ s \}]$ oder $struct(b, c_i) = [\alpha_0 : < s >]$,
d.h. es ist $\text{Bild}(corresp(b, c_i)) = \{x_1, x_2\}$ mit $x_1 = c_i$ und $corresp(.\alpha_0.\#) = x_2$.

relInst(b, c_i) :=

$\{[x_1 : y, x_2 : t, x_3 : \text{null}, \ldots, x_{m+n} : \text{null}]$
$\mid y \text{ in Ext}(n), t \text{ in } y.\alpha_0\}$

Fall 2: $\{c_i\} \in keys(b)$.

Für $2 \le \nu \le q$ sei $p_\nu := corresp^{-1}(x_\nu) \in \mathcal{L}(path)$.

relInst(b, c_i) :=

$\{[x_1 : y, x_2 : yp_2, \ldots, x_q : yp_q, x_{q+1} : \text{null}, \ldots, x_{m+n} : \text{null}]$
$\mid y \text{ in Ext}(n) : yp_2 <> \text{dne}\}$

Wir geben nun einige Beispiele zur Berechnung von *relInst*(b, c_i) an.

Beispiel 6.11

a) Wir betrachten die Beziehungssichten zur Beziehung `Teilnahme` aus Beispiel 6.2: Es ist

relInst(`Teilnahme:P`) =
```
{[P: p, K: k, Anm: null, Erg: null]
 | p in EXT(Person), k in p.Kurse},
```
Die in der Beziehungssicht `Teilnahme:P` nicht erfaßten Attribute `Anm` und `Erg` werden in jedem Tupel mit Nullwerten besetzt. Für die Beziehungssicht `Teilnahme:K` ist

relInst(`Teilnahme:K`) =
```
{[P: t.Teilnehmer, K: k, Anm: t.Anmeldung, Erg: t.Ergebnis]
  | k in EXT(Kurs), t in k.T_Liste}.
```
b) Für die Beziehungssicht `Ausleihe:E` zur Beziehung `Ausleihe` aus Beispiel 6.7 ist

relInst(`Ausleihe:E`) =
```
{[L: e.entliehen_von, E: e, LDat: e.entliehen_am]
 | e in EXT(Exemplar): e.entliehen_von <> dne}.
```
❒

Sind (b, c) und (b, c') zwei verschiedene Beziehungssichten, dann ist

$\{x_1, \ldots, x_k\} := \text{Bild}(\mathit{corresp}(b, c)) \cap \text{Bild}(\mathit{corresp}(b, c'))$

die Menge der Beziehungskomponenten bzw. -attribute, in denen sich beide Beziehungssichten überlappen. Gilt $\{c, c'\} \subseteq \{x_1, \ldots, x_k\}$, dann muß es zu jedem Tupel $t \in$ *relInst*(b, c), für das t.c' $\neq$ *null* ist, ein „Gegenstück“ $t' \in$ *relInst*(b, c') geben, so daß beide Tupel auf den gemeinsamen Komponenten und Attributen $x_1, \ldots, x_k$ übereinstimmen. Andernfalls wären beide Beziehungssichten zueinander widersprüchlich.

Die Bedingung t.c' $\neq$ *null* ist wesentlich, denn wäre t.c' = *null*, dann kann es in *relInst*(b, c') gar kein „Gegenstück“ zu t geben! Man betrachte dazu beispielsweise die Beziehung `Ausleihe`: Es sei `e` ein Buch-Exemplar ε, das entliehen wird, jedoch sei der Leser unbekannt[15], d.h. *I*(`Ausleihe:E`)(ε) = `[entliehen_von: null, entliehen_am: 23.6.95]`. Da `entliehen_von` nullwertig ist, bleibt die „Gegensicht“ `Ausleihe:L` unverändert. Dann gibt es aber auch kein $t \in$ *relInst*(`Ausleihe:L`) mit `t.E` = ε.

Wir kommen nun zu den angekündigten Bedingungen für die Konsistenz einer Datenbankinstanz hinsichtlich einer gegebenen Beziehung b.

Definition 6.5 (Konsistenz einer Datenbankinstanz bzgl. einer Beziehung)

Eine Datenbankinstanz INST(DS) heißt *konsistent* bezüglich der Beziehung $b \in$ RELSHIPS, wenn gilt:

(i) Alle für die Beziehungssichten $(b, c) \in$ RELVIEWS(b) generierten Integritätsbedingungen (vgl. Abschnitt 6.4) sind erfüllt.

15. Diese Situation ist in der Realität natürlich zu vermeiden!

(ii) $\forall$ (b, c), (b, c') $\in$ RELVIEWS(b) :
$c \neq c' \wedge \{c, c'\} \subseteq \{x_1, \ldots, x_k\} := \text{Bild}(\textit{corresp}(b, c)) \cap \text{Bild}(\textit{corresp}(b, c'))$
$\Rightarrow$
$\{ [x_1: t.x_1, \ldots, x_k: t.x_k] \mid t \in \textit{relInst}(b, c) \wedge t.c' \neq \textit{null} \}$
$= \{ [x_1: t.x_1, \ldots, x_k: t.x_k] \mid t \in \textit{relInst}(b, c') \wedge t.c \neq \textit{null} \}$

❒

Bedingung (ii) fordert die Widerspruchsfreiheit der in zwei verschiedenen, sich überlappenden Beziehungssichten enthaltenen Information über die Beziehung b.

Beispiel 6.12 Man betrachte die Beziehung `Teilnahme` mit den Beispielinstanzen aus Tabelle 6.1 aus Abschnitt 6.2. Für I(`Teilnahme:K`)(k_1) soll der ebenfalls in Abschnitt 6.2 angegebene Funktionswert gelten, jedoch setzen wir willkürlich I(`Teilnahme:P`)(p_2) = `[Kurse: {`k_2`}]`. Offensichtlich liegt nun eine Konsistenzverletzung vor, denn p_2 nimmt nach Tabelle 6.1 auch am Kurs k_1 teil!

Es ist

`[P:` p_1`, K:` k_1`, Anm: 23.5.95, Erg: 7]` $\in$ *relInst*(`Teilnahme:K`), aber
`[P:` p_1`, K:` k_2`, Anm: null, Erg: null]` $\notin$ *relInst*(`Teilnahme:P`) .

Demzufolge ist

`[P:` p_1`, K:` k_1`]` $\in$ { `[P:` t.P`, K:` t.K`]` | t $\in$ *relInst*(`Teilnahme:K`) $\wedge$ t.P $\neq$ *null* }, aber
`[P:` p_1`, K:` k_1`]` $\notin$ { `[P:` t.P`, K:` t.K`]` | t $\in$ *relInst*(`Teilnahme:P`) $\wedge$ t.K $\neq$ *null* }.

Das bedeutet, daß die Bedingung (ii) aus Def 6.5 tatsächlich verletzt ist und somit die Konsistenzverletzung erkannt wird. ❒

6.6 Kontrolle der Beziehungsredundanz

Um die Einhaltung der Konsistenzbedingungen aus Def. 6.5 zu gewährleisten, müssen zur Laufzeit alle Update-Operationen auf Beziehungssichten überwacht werden. Führt eine Update-Operation auf einer Beziehungssicht zu einer Verletzung der Bedingung (ii) aus Def. 6.5, dann sollen automatische Korrekturen in den anderen Beziehungssichten vorgenommen werden, um die Gültigkeit der Bedingung (ii) wiederherzustellen.

Wie in Abschnitt 5.4 schlagen wir vor, die im Zusammenhang mit Beziehungen stehenden Konsistenzbedingungen durch ECA-Regeln überwachen zu lassen. Im Rahmen dieser Arbeit soll jedoch keine allgemeine Beschreibung der Transformation aller Konsistenzbedingungen für Beziehungen in ECA-Regeln angegeben werden. Vielmehr beschränken wir uns wie in Abschnitt 5.4 auf einige Beispiele, die einen Eindruck von Gestalt und Umfang der zu generierenden Regeln geben.

In Anhang C werden für die Beziehungen `Teilnahme` und `Ausleihe` alle ECA-Regeln angegeben, die zur Einhaltung der Bedingungen (i) und (ii) aus Def. 6.5 notwendig sind.

Es folgen einige Bemerkungen zu den Beispielen in Anhang C:

- Die Definition der Beziehung `Teilnahme` enthält im Unterschied zu Abschnitt 6.2 zusätzliche Bedingungen an das Beziehungsattribut `Erg` sowie die Inklusionsbedingung `K in DV_Kurse`.
- Die Regeln sind im Unterschied zu den Regeln aus Abschnitt 5.4, wo die Identifikation einer einzelnen Regel keine Rolle spielte, benamt. Diese Namen werden verwendet, um gewisse Regeln für die Dauer der Abarbeitung der Anweisungen einer *action*-Komponente außer Kraft zu setzen. Dies geschieht über eine interne Operation `suspend`, der als Parameter die Namen der außer Kraft zu setzenden Regeln übergeben werden.
- Die Notwendigkeit, einzelne Regeln temporär außer Kraft setzen zu können, ergibt sich aus folgendem Beispiel:
  Es sollen `k` bzw. `p` einen Kurs bzw. eine Person bezeichnen. Wir nehmen nun an, daß in den Regeln die `suspend`-Anweisungen fehlten. Wird ein neues Tupel `t` mit `t.Teilnehmer = p` in `k.T_Liste` eingefügt, dann wird die Regel `T4` aktiviert. In der daraufhin auszuführenden Aktion wird der Kurs `k` in die Menge `p.Kurse` eingefügt. Dies führt zur Aktivierung von Regel `T2`, und als Aktion wird – obgleich unnötig – versucht, in `k.T_Liste` ein weiteres Tupel `t` mit `t.Teilnehmer = p` einzufügen. Dies führt jedoch aufgrund der Regel `T8` zu einem Zurücksetzen der Transaktion, da das erneute Einfügen die Schlüsselbedingung für `k.T_Liste` verletzen würde.
  Daher ist es nötig, für die Dauer der Abarbeitung der *action*-Komponente der Regel `T4` die Regel `T2` außer Kraft zu setzen. Auf die Regel `T6` kann ebenfalls verzichtet werden, da die Überprüfung der Bedingung der Regel `T6` in diesem Fall immer *false* ergibt.
- Für die in Anhang C aufgeführten Regeln, in denen mittels `suspend` gezielt andere Regeln außer Kraft gesetzt werden, hätte man für die Dauer der Abarbeitung der *action*-Komponente auch einfach alle Regeln außer Kraft setzen können (durch `suspend()`, d.h. mit leerer Parameterliste). Jedoch ist in Anhang C das selektive Außerkraftsetzen gewählt worden, um einen Eindruck davon zu geben, in welcher Weise die Regeln voneinander „abhängen“.
- Die Regeln zur Beziehung `Teilnahme` lassen sich jeweils folgenden Konsistenzbedingungen zuordnen:
  Die Regeln `T1` bis `T5` dienen der Wiederherstellung der Gültigkeit von Bedingung (ii) aus Def 6.5. Die *condition*-Komponente ist in diesen Regeln grundsätzlich `true`, denn sobald das jeweilige Ereignis ausgelöst wird, liegt tatsächlich eine Verletzung von Bedingung (ii) vor, auf die *unbedingt* reagiert werden muß.
  Die Regeln `T6` bis `T8` ergeben sich aus der Schlüsselbedingung der Beziehung.
  Regel `T9` überprüft die lokale Wertebedingung, die sich aus der `val`-Klausel ergibt.
  Schließlich sind `T10` und `T11` die Regeln zum kaskadierenden Einfügen bzw. Löschen für die angegebene Inklusionsbedingung. Man beachte, daß in `T10` die Teilbedingung `card(T) = 1` dafür sorgt, daß die Aktion höchstens dann ausgeführt wird, wenn ein Kurs-Objekt `k` *erstmals* in der Rolle `K` an der Beziehung teilnimmt.
- Für die Regeln zur Beziehung `Ausleihe` gilt:
  Die Regeln `A1` bis `A6` dienen der Wiederherstellung der Gültigkeit von Bedingung (ii) aus Def 6.5. Die Regeln `A7` und `A8` sorgen dafür, daß in der Beziehungssicht `Ausleihe:L` ein Exemplar nie nullwertig ist (Exemplar ist Schlüssel!). Schließlich überprüfen die Regeln `A9` und `A10`, ob versucht wird, ein Exemplar mehrfach zu entleihen.

6.7 Zusammenfassung und Diskussion

In diesem Kapitel wurde das ESCHER$^+$-Datenmodell um *Beziehungen* erweitert, die einen weiteren fundamentalen Baustein des Datenmodells bilden. Beziehungen werden einem Datenbankschema über die DDL-Anweisung `define relship` hinzugefügt.

Während in den meisten logischen Datenmodellen für Beziehungen kein eigenständiges Modellierungskonstrukt zur Verfügung steht, wird für ESCHER$^+$ also ein anderer Weg gewählt. Es seien noch einmal die Charakteristika unseres Ansatzes genannt:

- Das ESCHER$^+$-spezifische Beziehungskonstrukt erlaubt die *zentrale* Definition beliebiger Beziehungen. Es besteht keine Einschränkung hinsichtlich der Kardinalität der Beziehung. Ferner dürfen Beziehungen eine beliebige Anzahl von Attributen besitzen.
- Mit jeder Definition einer Beziehung ist die Angabe einer Menge von *Beziehungssichten* verknüpft. Jede Beziehungssicht ist einer Komponente der Beziehung zugeordnet. Jede Beziehungssicht legt fest, welche Struktur die Datenobjekte haben, auf die aus der Perspektive der jeweiligen Komponente zugegriffen werden kann. Der Zugriff auf die Beziehungsinstanzen erfolgt immer über die Attribute, die in der Struktur einer Beziehungssicht enthalten sind. Die Attribute einer Beziehungssicht zu einer Komponente c tragen zum erweiterten Zustand der Instanzen des Objekttyps bei, der in der Beziehungsdefinition der Komponente c zugeordnet ist.
- Die Instanziierung einer Beziehung erfolgt nicht über eine „neutrale“ Beziehungstabelle, in der alle Beziehungsinstanzen zusammengefaßt werden. Vielmehr ist jeder Beziehungssicht eine Interpretationsfunktion zugeordnet, die die Beziehungssicht instanziiert.
- Die aufgrund sich überlappender Beziehungssichten entstehende Beziehungsredundanz wird zur Laufzeit kontrolliert, indem ECA-Regeln dafür sorgen, daß die in verschiedenen Beziehungssichten replizierte Information zueinander konsistent gehalten wird.
- Für jede Beziehung können Integritätsbedingungen angegeben werden: Neben Schlüsselbedingungen lassen sich Kardinalitätsbedingungen in Form von (min, max)-Kardinalitäten angeben. Es können auch Einschränkungen der zulässigen Werte für Komponenten und Attribute formuliert werden. Ferner bieten Inklusionsbedingungen die Möglichkeit, als Voraussetzung für die Teilnahme von Objekten an einer Beziehung zu verlangen, daß sie in bestimmten Kollektionen enthalten sind.

Die Vorteile unseres Ansatzes können wie folgt zusammengefaßt werden:

- In vielen mit ESCHER$^+$ vergleichbaren Datenmodellen für komplexe Objekte werden Beziehungen über die Methode der verteilten Aggregation in ein Datenbankschema abgebildet. Dabei findet eine „Zerstückelung“ der Definition einer Beziehung durch Verteilung auf verschiedene Objekttypdefinitionen statt. Typischerweise taucht der Name einer Beziehung gar nicht mehr auf, d.h. eine Beziehung wird durch die Verteilung auf unterschiedliche Attribute „versteckt“. Die zentrale Definition einer Beziehung in ESCHER$^+$ hebt diesen Nachteil auf.
- Durch die Definition verschiedener Beziehungssichten bleibt andererseits ein wichtiger Vorteil der verteilten Aggregation für ESCHER$^+$ erhalten: Die Definition verschiedener Beziehungssichten zu einer Beziehung entspricht prinzipiell der verteilten Aggregation, ohne daß jedoch die Strukturen von Objekttypen verändert werden müssen. Die für eine

Beziehungssicht definierten Attribute können in Pfadausdrücken wie jedes Kern-Attribut des jeweiligen Objekttyps verwendet werden.

- Beziehungen in ESCHER$^+$ unterstützen die geforderte „Symmetrie“ im Zugriff auf Assoziationen: Sind für alle Komponenten einer Beziehung Beziehungssichten definiert, dann wird die Traversierung der Beziehung in allen Fällen gleichartig behandelt, unabhängig davon, welche Komponente als Ausgangspunkt gewählt wird: Die Traversierung geschieht immer über den Zugriff auf Attribute der Beziehungssichten (Pfadausdruck in „Dot-Notation“). Eine Ungleichbehandlung (einfacher Zugriff über Pfadausdruck in „Dot-Notation“ in einigen Fällen, mehr oder weniger aufwendig zu formulierende Anfrage in anderen Fällen) findet nicht statt.
- Die Unterstützung verschiedener, sich überschneidender Beziehungssichten führt zu Redundanz auf der Instanzenebene. Während andere Ansätze Redundanzkontrolle nur für binäre Beziehungen ohne Attribute unterstützen (vgl. die `invers`-Klauseln etwa im Datenmodell des ODMG’93-Standards [Cat+94]), kann in ESCHER$^+$ die sog. Beziehungsredundanz für Beziehungen beliebiger Kardinalität und mit beliebig vielen Attributen kontrolliert werden.
- In ESCHER$^+$ werden über das Beziehungskonstrukt alle in einer Anwendung auftretenden Assoziationen auf homogene Weise in ein Datenbankschema aufgenommen. Es muß nicht, wie in anderen Datenmodellen, zur Erfassung von Beziehungen, die nicht zu der Klasse der binären Beziehungen ohne Attribute gehören, auf andere Formen der Modellierung (Definition einer Beziehungstabelle bzw. eines eigenen Objekttyps für die Beziehungsinstanzen) ausgewichen werden.
- Durch die Instanziierung der Beziehungssichten ist ein schneller Zugriff auf die Beziehungsinstanzen in der jeweils benötigten Form gewährleistet. Eine denkbare Alternative wäre die Materialisierung einer „neutralen“ Beziehungstabelle und die Berechnung der Interpretationsfunktionen I(b, c) zur Laufzeit durch die Ausführung von Anfragen auf der Beziehungstabelle. Dadurch ließe sich zwar Redundanz in den Daten vermeiden, jedoch halten wir den Aufwand zur Rendundanzkontrolle als Preis für einen schnelleren Zugriff für vertretbar.
- Die Entscheidung, eine Beziehung zusammen mit ihren Beziehungssichten an zentraler Stelle zu definieren, hat den Vorteil, daß ein einfaches Hinzufügen neuer Beziehungen zu einem bestehenden Datenbankschema möglich ist. Dabei sind dann keine Instanzenkonversionen notwendig, da alle Objekttypdefinitionen unverändert bleiben. Entsprechendes gilt auch für das Entfernen nicht mehr relevanter Beziehungen über die DDL-Anweisung `drop relship`.

Beziehungen sind ein typisches Konstrukt konzeptueller (bzw. semantischer) Datenmodelle. Nachdem sich die Beziehung als eines der Grundkonstrukte des Entity-Relationship-Modells seit langem bewährt hat, sind Beziehungen auch für die objektorientierte konzeptuelle Modellierung „wiederentdeckt“ worden. Es sei stellvertretend für viele weitere Modelle auf das OMT-Modell [RBP+94] von Rumbaugh et al. hingewiesen, das *associations* für die konzeptuelle Modellierung einführt.

Es gibt dagegen nur wenige logische Datenmodelle, die Beziehungen als ein explizites Modellierungskonstrukt zur Verfügung stellen.

In [Rum87] wird das Datenmodell einer objektorientierten Programmiersprache um flache Relationen ergänzt mit dem Ziel, in ihnen die Instanzen von Beziehungen zu sammeln. Relationen sind in [Rum87] also genau das, was wir in diesem Kapitel als Beziehungstabellen bezeichnet haben.

Auch in [Bee90] werden flache Relationen als weitere fundamentalen Elemente eines objektorientierten Datenmodells genannt, wobei auf denselben Zweck wie bei [Rum87] abgezielt wird.

In [DG90] wird ein Datenmodell für eine objektorientierte Erweiterung der Sprache Prolog beschrieben, das ein explizites Konstrukt zur Definition binärer Beziehungen anbietet. Die Beziehungen können Attribute besitzen. Daneben kann eine Vielzahl von Integritätsbedingungen (*assertions*) für Beziehungen und die an ihnen teilnehmenden Objekte formuliert werden.

Besonders erwähnt werden soll die Arbeit [AGO91]. In ihr wird das Datenmodell der objektorientierten DBPL Fibonacci (siehe auch [AGO95]) um Beziehungen erweitert. Dabei stellen sich Beziehungen als Verallgemeinerungen von Klassen heraus: Während eine Beziehung (*association*) eine Menge von Tupeln[16] mit beliebig vielen und beliebig typisierten Komponenten ist, wird eine Klasse (*class*) einfach als Menge von Tupeln mit genau einer objektwertigen Komponente aufgefaßt. Ähnlich wie in [DG90], aber ohne die Einschränkung auf binäre Beziehungen, wird ein breites Spektrum an beziehungsspezifischen Integritätsbedingungen eingeführt. Dazu gehören auch Einfüge- und Löschregeln, wie sie in ESCHER$^+$-Beziehungen in der `incl`-Klausel durch die „`cascade add, rem`“-Option ausgedrückt werden können. Für die Modifikation von Beziehungen sowie den assoziativen Zugriff auf einzelne Beziehungsinstanzen werden in [AGO91] folgende Operationen bereitgestellt:

assoc.**insert**(binding), assoc.**remove**(binding), assoc.**has**(binding), assoc.**get**(binding)

Daneben stehen die Operationen der Relationenalgebra (Vereinigung, Durchschnitt, Projektion, Selektion usw.) zur Verfügung, die auf Beziehungen „als Ganzes“ angewandt werden.

Es fällt auf, daß alle uns bekannten Vorschläge, die Beziehungen als explizites Konstrukt in das jeweilige Datenmodell integrieren, dies über die Einführung flacher Relationen bewerkstelligen. Die zur Verfügung stehenden Operationen entsprechen dann auch denen zur Manipulation flacher Relationen. Das bedeutet aber, daß für das Traversieren einer Beziehung ausgehend von einer bestimmten Beziehungskomponente immer eine Anfrage, die einen Join mit der jeweiligen Relation (der „Beziehungstabelle“) enthält, gestellt werden muß.

Unser Vorgehen unterscheidet sich hinsichtlich der Integration expliziter Beziehungen in das Datenmodell erheblich von den soeben erwähnten Arbeiten. In ESCHER$^+$ sollen Beziehungen nicht einfach über flache Relationen erfaßt werden. Statt dessen bilden Beziehungsdefinitionen den „Rahmen“ für die Definition der Beziehungssichten, die für alle Zugriffe von entscheidender Bedeutung sind. In einer Datenbankinstanz wird eine Beziehung über Interpretationen für die Beziehungssichten erfaßt und nicht über eine flache „Beziehungstabelle“.

In Form der Beziehungssichten machen wir die verteilte Aggregation auf beliebige Beziehungen anwendbar, wobei die sich dabei ergebende Redundanz kontrolliert wird. Damit verallgemeinern wir den bereits von vielen Datenmodellen praktizierten Ansatz, binäre Beziehungen

16. Tupel werden in [AGO91] als *binding* bezeichnet, da sie (Label,Wert)-Paare zu einer neuen „Einheit“ bündeln.

ohne Attribute durch Integration zusätzlicher Attribute in die beteiligten Komponententypen und durch Angabe entsprechender `invers`-Klauseln zu erfassen.

Das in diesem Kapitel vorgestellte Konzept stellt eine Weiterentwicklung unseres in [The95] vorgestellten Ansatzes dar. In [The95] werden keine Beziehungssichten definiert, sondern es wird noch davon ausgegangen, daß für die Abbildung einer Beziehung auf ein ESCHER$^+$-Datenbankschema folgende Alternativen zur Verfügung stehen:

- Aggregation zusätzlicher Attribute *direkt* in die Struktur der beteiligten Objekttypen
- Definition zusätzlicher Tabellen

Für die Beziehung `Teilnahme` aus Beispiel 6.2 kann gemäß [The95] folgendes Vorgehen gewählt werden:

- In der Struktur des Objekttyps `Kurs` sei das listenwertige Attribut `T_Liste` enthalten. Es liegt somit die in (6.4) angegebene Struktur vor.
- Der Objekttyp `Person` bleibt unverändert, d.h. er soll weiterhin nur Kern-Attribute wie z.B. `Name` oder `Adresse` enthalten.
  Zusätzlich wird aber eine Tabelle `Kursteilnehmer` mit der Struktur

  ```
  {[ P: Person, Kurse: {[K: Kurs, Ergebnis: integer]} ]}
  ```

  definiert. Für die Beziehung `Teilnahme` wird also ein „Mix“ der oben genannten Alternativen gewählt.

Nach [The95] sorgt die DDL-Anweisung

```
define relship
  -name Teilnahme
  -comp Kursteilnehmer.P = Kurs.T_Liste.Teilnehmer  as P,
        Kursteilnehmer.Kurse.K = Kurs               as K
  -attr Kurs.T_Liste.Anmeldung                      as Anm,
        Kursteilnehmer.Kurse.Ergebnis
                    = Kurs.T_Liste.Ergebnis         as Erg
  -key  [P, K]
end define;
```

dafür, daß der Inhalt der Tabelle `Kursteilnehmer` und die Zustände der `Kurs`-Objekte zueinander konsistent gehalten werden. Eine `define relship`-Anweisung gemäß [The95] definiert keine Beziehungssichten, sondern nimmt Bezug auf bereits im Schema existierende „Sichten“. Sie hat deshalb ausschließlich die Funktion einer Integritätsbedingung. In der `comp`- und `attr`-Klausel wird mit Hilfe von Strukturpfadausdrücken angegeben, welche „Stellen“ im Datenbankschema miteinander korrespondieren. Es geht also um eine Integration bestehender Sichten (vgl. auch [SP94]). Eine zentrale Definition von Beziehungen wird auf diese Weise insofern nicht unterstützt, als in einer Beziehungsdefinition immer auf bereits im Schema vorhandene Beziehungssichten Bezug genommen wird. Ein Modellierungs-„Mix“ für Beziehungen wie im obigen Beispiel erzeugt zudem ein inhomogenes Datenbankschema. Ferner gestaltet sich das Hinzufügen oder Entfernen von Beziehungen nicht einfach, weil man dazu ggf. bestehende Typdefinitionen verändern muß. Für das in dieser Arbeit und auch in [The96] dargelegte Konzept gelten diese Nachteile nicht.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Ausgangspunkt dieser Arbeit bildete das eNF^2-Datenmodell, das bislang als Datenmodell des Datenbank-Editors ESCHER verwendet wurde. Der ESCHER-Prototyp visualisiert komplexe Datenobjekte in Form von geschachtelten Tabellen auf einer graphischen Benutzeroberfläche. In ESCHER bewegt der Benutzer spezielle Cursor, die sog. Finger, durch die hierarchisch strukturierten Daten und führt Update-Operationen auf den Datenobjekten aus, auf die der aktive Finger gerade „zeigt“.

Die Erfahrungen im Umgang mit dem ESCHER-Prototyp haben ergeben, daß die von ESCHER unterstützten Interaktionsparadigmen für den Umgang mit hierarchisch strukturierten Daten auf der externen Ebene ein intuitiv verständliches und benutzerfreundliches Konzept darstellen. In der Einleitung zu dieser Arbeit wurde allerdings gleichzeitig festgestellt, daß das eNF^2-Datenmodell als logisches Datenmodell nicht ausdrucksstark genug ist, um die in Anwendungen enthaltene Semantik in adäquater Weise und vollständig zu erfassen.

Thema dieser Arbeit war daher die ***semantische Anreicherung des eNF^2-Datenmodells***, um zu einer weitergehenden Kongruenz zwischen zu modellierendem Ausschnitt der Realwelt und logischem Schema zu gelangen. Das in dieser Arbeit entwickelte Datenmodell haben wir **ESCHER$^+$** genannt.

Im folgenden fassen wir die einzelnen Beiträge dieser Arbeit zusammen.

Erweiterung um Typkonzepte

Als eines der Defizite des eNF^2-Datenmodells ist das Fehlen eines Typkonzeptes zu nennen, in dessen Rahmen die Definition anwendungsspezifischer Typen möglich ist. Die Notwendigkeit der Integration benutzerdefinierter Typen in das eNF^2-Datenmodell ist unumstritten, und dazu sind in der Vergangenheit auch bereits verschiedene Erweiterungsvorschläge unterbreitet worden in [LKD+91, WPC93, The93, Pau94]. In dieser Arbeit wird die in [The93] vorgestellte Erweiterung des Datenmodells um *benutzerdefinierte Objekttypen* vertieft. Hinsichtlich der Fragestellung, ob es sich bei benutzerdefinierten Typen um Objekt- oder Wertetypen handeln soll, entscheiden wir uns also für Objekttypen. In ESCHER$^+$ stellen lediglich Aufzählungstypen eine Form benutzerdefinierter Wertetypen dar. Es handelt sich bei ESCHER$^+$ also um eine objektorientierte Erweiterung des eNF^2-Modells.

Wie in vergleichbaren Datenmodellen unterstützt ESCHER$^+$ die Vererbung struktureller und operationaler Eigenschaften auf der Basis einer *isa*-Beziehung. Damit verbunden ist die für eine Programmiersprachenschnittstelle relevante Substituierbarkeit einer Instanz des Subtyps überall dort, wo eine Instanz des direkten oder eines indirekten Supertyps erwartet wird.

Eine Besonderheit unseres Modells stellt die Möglichkeit der *multiplen und variablen Typzugehörigkeit* eines Objektes dar. Die verbreiteten objektorientierten Datenmodelle lassen nach der Erzeugung eines neuen Objektes keine Veränderung der Typzugehörigkeit mehr zu. Jedem Objekt ist ein konstanter Typ zugeordnet. Dies ist der Typ, zu dem das Objekt instanziiert wurde. In ESCHER$^+$ ist hingegen die sog. *dynamische Spezialisierung* möglich: Danach kann man einem Objekt während seiner Lebenszeit Subtypen seiner bisherigen Typen hinzufügen, wobei die Identität des Objektes erhalten bleibt. Ein Objekt kann so gleichzeitig zu mehreren Subtypen eines Typs gehören (parallele Spezialisierung).

ESCHER$^+$ bietet die Möglichkeit, *variante Typen* zu definieren, die in zwei Situationen sinnvoll sind: Zum einen dienen sie der Modellierung varianter Teile innerhalb von Anwendungsstrukturen (z.B. die Alternativen Straße oder Postfach bei einer Adresse), zum anderen können mit ihrer Hilfe inhomogene Kollektionen gebildet werden, d.h. Kollektionen, deren Elemente unterschiedliche Typen besitzen. Im letzteren Fall spielen die sog. *Objekttypvarianten* eine besondere Rolle. Eine Objekttypvariante ist der Spezialfall eines varianten Typs, bei dem alle Alternativen Objekttypen sind. Es lassen sich z.B. verschiedenartige geometrische Objekte in einer Kollektion zusammenfassen, sofern ihre Typen zu den Alternativen einer Objekttypvariante gehören. Die Objekttypvarianten entsprechen dem semantischen Konzept der disjunkten Generalisierung mit Überdeckungseigenschaft.

Formalisierung der Modellbeschreibung

Von anderen Vorschlägen zur Erweiterung des eNF^2-Datenmodells unterscheidet sich diese Arbeit durch eine wesentlich strengere *Formalisierung*. Während für das flache und auch für das geschachtelte Relationenmodell „genormte“ formale Modellbeschreibungen seit langem vorliegen, ist dies für neuere Datenmodelle, insbesondere auch für objektorientierte Datenmodelle, nicht der Fall. Mit dieser Arbeit wollen wir daher auch einen Beitrag zu den formalen Grundlagen objektorientierter Datenmodelle liefern.

Instanzen benutzerdefinierter Objekttypen sind abstrakte Objekte ω aus einem unendlichen Objektvorrat Ω. Ein zentrales Konzept unseres Datenmodells bilden die Interpretationen für die abstrakten Objekte. Für zwei uninterpretierte Objekte ω und ω' kann lediglich festgestellt

werden, ob sie identisch sind oder nicht. Gehört ein abstraktes Objekt ω einem Typ t an, dann kann es relativ zu diesem Typ interpretiert werden. Jeder Objekttyp t stellt dazu eine *Interpretationsfunktion* $I(t)$ bereit. Der Funktionswert $I(t)(\omega)$ gibt den strukturellen Zustand des abstrakten Objektes ω zum Typ t an, der die „konkrete“ Information über ein abstraktes Objekt enthält. Da ein Objekt mehr als einen Typ besitzen kann, sind ihm ggf. auch mehrere Zustände zugeordnet.

Im Rahmen der Formalisierung des Strukturteils des Datenmodells wurden *Baumrepräsentationen* für komplexe Datenobjekte eingeführt. Damit wurde eine Verbindung zwischen einer auf Begriffen wie Menge, Tupel, Liste usw. basierenden Modellbeschreibung, wie sie üblicherweise für das eNF^2-Modell angegeben wird, und einer anschaulicheren graphbasierten Beschreibung hergestellt. Auch für das graphbasierte Modell wurde großer Wert auf eine formale Darstellung gelegt. Datenobjekte lassen sich durch *beschriftete Bäume* repräsentieren. Eine Baumrepräsentation eines Datenobjektes ist ein Paar beschrifteter Bäume, das aus einem Wertebaum und einem Strukturbaum gebildet wird, wobei letzterer die Information über die Struktur bzw. den Typ des repräsentierten Datenobjektes wiedergibt.

Eine Datenbankinstanz zu einem Datenbankschema ist gegeben durch eine Menge von Baumrepräsentationen von Datenobjekten sowie einer Menge von Interpretationsfunktionen $I(t)$. Die Interpretationsfunktionen sind dabei konzeptionell mit „Hyperlinks“ vergleichbar: Von einem Knoten, der ein abstraktes Objekt ω mit dem Typ t repräsentiert, wird mittels $I(t)(\omega)$ auf einen anderen Wertebaum verwiesen, der den Zustand von ω relativ zu t repräsentiert.

Für ESCHER$^+$ wurde ein *Meta-Schema* angegeben, das als „Schablone“ für alle benutzerdefinierten Datenbankschemata dient. Dabei zahlte es sich aus, daß alle Elemente eines Datenbankschemas – wie z.B. Typen, Tabellen und Operationen – bereits zuvor als abstrakte Objekte eingeführt wurden. Sie sind nun Instanzen spezieller Objekttypen des Meta-Schemas. Auch alle Datenbankschemata sind Instanzen eines solchen Objekttyps. Die Datenbankinstanz zum Meta-Schema enthält neben allen benutzerdefinierten Datenbankschemata auch das Meta-Schema selbst, womit eine wünschenswerte Abschlußeigenschaft der Metamodellierung erfüllt ist.

Angabe der formalen Semantik des Operationenteils

Baumrepräsentationen bildeten in dieser Arbeit auch die Grundlage für die formale Beschreibung der Semantik des operationalen Teils des ESCHER$^+$-Datenmodells. Es wurden die für das Datenmodell relevanten *Basisoperationen* eingeführt. Die Semantik einer Basisoperation ist gegeben durch die mit ihrer Ausführung verbundenen Manipulationen von Baumrepräsentationen (Hinzufügen bzw. Löschen von Ecken oder Kanten, Modifikation von Beschriftungen, Generierung neuer Baumrepräsentationen). Für die generischen Updateoperationen muß gelten, daß sie abgeschlossen sind in der Menge der Baumrepräsentationen, da nicht jeder beliebige beschriftete Baum ein Datenobjekt repräsentiert.

Zum operationalen Teil des Datenmodells gehören neben den vordefinierten Basisoperationen auch *benutzerdefinierte Operationen*. Es werden *Methoden*, d.h. an einen Objekttyp „gebundene“ Operationen, und *freie Operationen* unterschieden. Zu letzteren gehören u.a. durch den Benutzer aufrufbare Hauptprogramme. Als Grundlage für die formale Beschreibung der Semantik benutzerdefinierter Operationen wurde zunächst der Begriff der *Laufzeitinstanz* eingeführt. Sie berücksichtigt neben den persistenten Datenobjekten einer Datenbankinstanz auch

transiente Datenobjekte sowie einen Stack zur Verwaltung der Variablenbindungen bei geschachtelten Aufrufen von Operationen. Es wurde dann die Sprache BASESCRIPT eingeführt, die als „primitive“ Kernsprache für das ESCHER$^+$-Datenmodell konzipiert ist und eine sehr einfache Syntax und Semantik besitzt. Die Semantik einer „höheren“ persistenten Programmiersprache für ESCHER$^+$ läßt sich dann durch ihre Übersetzung nach BASESCRIPT angeben.

Als „höhere“ persistente Programmiersprache für ESCHER$^+$ wurde in dieser Arbeit die Sprache SCRIPT$^+$ verwendet, die eine Weiterentwicklung der Sprache SCRIPT$^+$ aus [Pau94] darstellt. Neben geringfügigen Modifikationen in der Syntax, die jedoch die Ausdrucksfähigkeit der Sprache nicht verändern, wurden einige Erweiterungen vorgenommen. Sie betreffen zum einen die dynamische Spezialisierung, zum anderen die Behandlung von Objekttypvarianten in SCRIPT$^+$. Bekanntlich sind variante Typen für Anwendungsprogramme problematisch, da sie vergleichsweise aufwendige Fallunterscheidungen (`case`-Statements) notwendig machen. Im Fall von Objekttypvarianten kann jedoch der Code deutlich vereinfacht werden. Soll auf ein Attribut zugegriffen oder eine Methode ausgeführt werden, die in allen Alternativen der Objekttypvariante existiert, dann ist keine Fallunterscheidung notwendig, sondern es wird durch dynamisches Binden dafür gesorgt, daß der Zugriff bzw. Aufruf immer in korrekter Weise erfolgt. Bei der Definition einer Objekttypvariante wird zu diesem Zweck angegeben, daß alle ihre Alternativen bestimmte Attribute oder Methoden haben *müssen*.

Integritätsbedingungen und Konsistenzerhaltung

Ein weiterer wichtiger Baustein, der maßgeblich zur semantischen Anreicherung beiträgt, ist die Ergänzung des Datenmodells um Integritätsbedingungen. In den Arbeiten zum eNF^2-Datenmodell und seinen Erweiterungen wurden Integritätsbedingungen bislang stark vernachlässigt. Für ESCHER$^+$ haben wir verschiedene Klassen deklarativer Integritätsbedingungen eingeführt, die eine Einschränkung noch bestehender „Freiheitsgrade“ in einem Datenbankschema darstellen und somit zur weiteren Kongruenz zwischen Realwelt und logischem Schema beitragen. Damit verbunden war die Integration der (flachen) Transaktion in das formale Datenmodell. Es wurde vorgeschlagen, die Integritätsbedingungen in interne Event-Condition-Action-Regeln (ECA-Regeln) zu transformieren, um sie dann in dieser Form von einem Integritätsmonitor automatisch überwachen zu lassen.

Zur weiteren Unterstützung der Konsistenz führten wir für ESCHER$^+$ eine Einkapselung von Attributen und Operationen ein, die mit dem *public*/*private*-Konzept objektorientierter Programmiersprachen wie z.B. C++ vergleichbar ist. Durch selektive Einkapselung von Attributen können direkte Update-Operationen auf diesen Attributen oder sogar der direkte Zugriff auf sie untersagt werden. Unter den Operationen sollten nur diejenigen als öffentlich gelten, für die verifiziert werden kann, daß sie konsistenzerhaltend sind.

Beziehungen als fundamentaler Baustein des Datenmodells

Der semantischen Anreicherung diente schließlich auch die Erweiterung des Datenmodells um ein explizites Beziehungskonstrukt. Beziehungen gehören somit zu den fundamentalen Bausteinen des Datenmodells, wie dies u.a. bereits für Typen und Tabellen der Fall ist. Mit dem zusätzlichen Konstrukt lassen sich Beziehungen (Assoziationen) zwischen Anwendungsobjekten in einem ESCHER$^+$-Datenbankschema direkt erfassen. Für jede Beziehung können verschiedene Integritätsbedingungen angegeben werden. Damit unterscheidet sich ESCHER$^+$ von

vergleichbaren Datenmodellen für komplexe Objekte, die darauf angewiesen sind, Beziehungen mit Hilfe anderer Konstrukte im Schema auszudrücken.

Andere Datenmodelle setzen die Assoziation oftmals einfach mit der Aggregation gleich. Die zu einem Objekt ω in Beziehung stehenden Objekte werden zu Subobjekten von ω, d.h. die Beziehung wird „gerichtet“ bzw. „hierarchisiert“. Auch im eNF^2-Modell, das sich ja in erster Linie zur Modellierung streng hierarchischer Strukturen eignet, wird dieser Weg eingeschlagen. Daraus resultiert eine „Einseitigkeit“ der Modellierung, die die Traversierung der Beziehung ausgehend von den mit ω in Beziehung stehenden Objekten benachteiligt. Diese „Einseitigkeit“ kann in einigen Datenmodellen (nicht aber im eNF^2-Modell!) dadurch vermieden werden, daß für die andere Richtung genauso verfahren wird: Das Objekt ω wird zu einem Subobjekt aller Objekte, zu denen es in Beziehung steht. Diese Methode zur Abbildung von Beziehungen in ein Datenbankschema hatten wir *verteilte Aggregation* genannt. Die damit einhergehende Redundanz auf der Instanzenebene kann in den uns bekannten Datenmodellen nur für binäre Beziehungen ohne Attribute kontrolliert werden.

In ESCHER$^+$ führt die Definition einer Beziehung nicht zur Erzeugung einer flachen Relation („Beziehungstabelle“), in der alle Beziehungsinstanzen gesammelt werden. Von entscheidender Bedeutung für die Instanzenebene sind die mit jeder Beziehung verbundenen Beziehungssichten. Mit ihnen läßt sich das Prinzip der verteilten Aggregation, das in anderen Datenmodellen nur für binäre Beziehungen ohne Attribute anwendbar ist, auf beliebige Beziehungen übertragen. Eine Beziehung wird über Interpretationsfunktionen für die einzelnen Beziehungssichten instanziiert. Über die Interpretationsfunktion einer Beziehungssicht kann ein direkter Zugriff auf die für die jeweilige Sicht benötigte Information erfolgen, ohne daß zuerst auf eine flache Beziehungstabelle zugegriffen wird und danach die relevanten Beziehungsinstanzen auf die Struktur abgebildet werden müssen, die von der Beziehungssicht verlangt wird. Beziehungssichten stellen Attribute für den Zugriff aus der Perspektive einer Komponente der Beziehung zur Verfügung, die wie jedes andere Attribut des jeweiligen Objekttyps benutzt werden können. Die Redundanz auf der Instanzenebene, die sich durch überlappende Beziehungssichten ergibt, wird zur Laufzeit kontrolliert, indem nach Updates in einer Beziehungssicht entsprechende Update-Operationen in den anderen Beziehungssichten ausgelöst werden.

7.2 Ausblick

Wir wollen nun noch auf einige Ansatzpunkte für weitere Forschungsarbeiten hinweisen, die die Resultate dieser Arbeit ergänzen und zu einer Weiterentwicklung des ESCHER-Prototyps beitragen können.

- Entwicklung eines *Sichten-Konzepts* für die externe Ebene: Zu diesem Punkt liegen bereits erste Ansätze vor. *Sicht-Typen* (*view types*) werden von Objekttypen abgeleitet, indem eine Projektion auf die für den Sicht-Typ relevanten Attribute und Methoden durchgeführt wird. Daneben können für den Sicht-Typ Attribute und Methoden umbenannt sowie zusätzliche, abgeleitete Attribute und Methoden definiert werden. Die Instanzen eines Sicht-Typs sind die Instanzen der ihm zugrundeliegenden Objekttypen, d.h. Sicht-Typen verändern das „Interface“ von Objekten unter Beibehaltung ihrer Identität.

Sicht-Tabellen (*view tables*) sind „virtuelle“ Tabellen, die – wie die aus SQL bekannten *Views* – über Anfragen definiert sind.

- Weiterentwicklung der *Benutzeroberfläche* des ESCHER-Prototyps: In dieser Arbeit wurde bewußt auf die Behandlung von Eingabe- und Ausgabeoperationen verzichtet, um uns ausschließlich auf die logische Ebene der Datenmodellierung zu konzentrieren. Auch für das Datenmodell ESCHER$^+$ kann an der Visualisierung der Daten in Form von geschachtelten Tabellen festgehalten werden, so daß die Benutzerinteraktion weiterhin über ESCHER-Arbeitsfenster erfolgt. Es bedarf jetzt jedoch flexiblerer Darstellungsformen. Abstrakte Objekte sollen i.d.R. „expandiert“ dargestellt werden, d.h. es werden anstelle des Identifikators ω die in einem Anwendungskontext relevanten Attribute des Objektes angezeigt. Bei einem Expandieren „on demand“ sollte unterschieden werden können, ob dies „in place“ oder in einem neuen Arbeitsfenster geschieht. Es ist daher an eine ESCHER-spezifische „Layout-Sprache“ gedacht, mit der sich die Gestaltung eines Arbeitsfensters konfigurieren läßt. Dabei ist auch vorstellbar, daß sich über die Layout-Spezifikation benutzerdefinierte Operationen an Ereignisse der Benutzeroberfläche (Tastendrücke, Mausklicks usw.) binden lassen und auf diese Weise eine noch weitergehende Anpassung der Oberfläche an eine konkrete Anwendungssituation möglich wird.
- Operationen zur *Schema-Modifikation*: Eine nachträgliche Veränderung eines ESCHER$^+$-Datenbankschemas mit automatischer Anpassung einer bereits existierenden Datenbankinstanz ist vielfach wünschenswert, um mit der Evolution einer Anwendung schrittzuhalten. Als Beispiele bisher nicht unterstützter Schema-Modifikationen seien das Hinzufügen oder Entfernen von Attributen aus einem Tupel oder Objektzustand oder auch das nachträgliche Hinzufügen oder Verändern von Integritätsbedingungen genannt.
- Weitere Arbeiten zur *Metamodellierung*: Das in dieser Arbeit vorgestellte Meta-Schema deckt nur den strukturellen Teil von ESCHER$^+$ ab. Das Meta-Schema ist daher um alle weiteren Komponenten des Datenmodells zu ergänzen. Ferner steht noch die formale Definition der DDL-Operationen als „gewöhnliche“ Update-Operationen auf den Datenobjekten des Meta-Schemas aus.
- Erweiterung der Sprache SCRIPT$^+$ um Konstrukte zur *Ausnahmebehandlung*: In Kapitel 5 wurde bereits ein `handle violation`-Konstrukt skizziert, mit dem in Anschluß an eine Transaktion auf eine Integritätsverletzung reagiert werden kann. Denkbar ist auch ein allgemeineres Konzept, unter dem Ausnahmen (wie z.B. die Division durch Null), Integritätsverletzungen sowie benutzerdefinierte und explizit ausgelöste Ereignisse als Spezialfälle allgemeiner Ereignisse zusammengefaßt werden.
- *Indexunterstützung* auf interner Ebene: Indexe sind für die effiziente Auswertung von Anfragen, aber auch für die effiziente Überprüfung einiger Integritätsbedingungen von Bedeutung. In einer gerade begonnenen Arbeit sollen Untersuchungen zu verschiedenen Indizierungsverfahren durchgeführt werden.
- *Ausführungspläne* für Anfragen und *Anfrageoptimierung*: Anfragen werden nicht direkt von SCRIPT$^+$ in eine Folge von BASESCRIPT-Anweisungen übersetzt, sondern sie sollen an eine spezielle Komponente des DBMS, den *Query Optimizer*, übergeben werden, der sie in einen möglichst optimalen Ausführungsplan transformiert. Dabei spielen Gegebenheiten der internen Ebene, wie z.B. Indexdateien, eine zentrale Rolle. Die oben erwähnte Arbeit zur Indexunterstützung wird daher auch diesen Punkt aufgreifen.

Anhang A

Mathematische Grundlagen

A.1 Elementare Notationen und Definitionen

Es werden nun einige Bezeichnungen und Schreibweisen eingeführt, die in dieser Arbeit verwendet werden.

- Mit $\mathcal{Set}$ soll eine hinreichend große[1] Menge von Mengen bezeichnet werden.
- $\mathcal{FSet}$ (M) sei die Menge aller endlichen Teilmengen einer Menge M.
- Die Menge aller partiellen (totalen) Funktionen f von A nach B wird mit $\mathcal{PFun}$ (A, B) ($\mathcal{Fun}$ (A, B))bezeichnet.
  Für die Menge aller partiellen (totalen) Funktionen schreiben wir $\mathcal{PFun}$ ($\mathcal{Fun}$).
- Mit $\mathcal{Fun}^{bij}$(A, B) wird die Menge aller bijektiven Abbildungen von A nach B bezeichnet.
- Ist f $\in$ $\mathcal{PFun}$ (A, B), dann schreiben wir f: A ···> B.
  Ist f $\in$ $\mathcal{Fun}$ (A, B), dann wird f: A $\rightarrow$ B notiert.
  Für f: A ···> B sei ferner *dom* (f) := A, *range* (f) := B.
- Ist f $\in$ $\mathcal{PFun}$ (A, B), dann ist Def(f) := { a $\in$ A | $\exists$ b $\in$ B : f(a) = b } $\subseteq$ *dom* (f) der Definitionsbereich von f. Ist f eine totale Funktion, so gilt bekanntlich Def(f) = *dom* (f).
- Es ist Bild(f) := { b $\in$ B | $\exists$ a $\in$ A : f(a) = b } $\subseteq$ *range* (f) der Bildbereich von f.
  Ist A' $\subseteq$ Def(f), dann ist f(A') := { f(a) | a $\in$ A' } das Bild von f unter A'.
- Sind f und g zwei partielle Funktionen, dann ist
  f = g $\Leftrightarrow$ Def(f) = Def(g) $\wedge$ $\forall$ a $\in$ Def(f): f(a) = g(a).
  Für f $\in$ $\mathcal{PFun}$ (A, B) schreiben wir f(a) = $\perp$ gdw. a $\notin$ Def(f), d.h. $\perp$ steht für „undefiniert“.
- Eine Funktion f: A ···> B mit endlichem Definitionsbereich Def(f) = {a_1, …, a_n} kann auch als Relation f = {(a_1, f(a_1)), …, (a_n, f(a_n))} notiert werden.
- Für Updates von Funktionen wird eine Schreibweise verwendet, die an die Syntax für Zuweisungen in Programmiersprachen angelehnt ist. Mit b $\in$ *range* (f) $\cup$ {$\perp$} schreiben wir

1. Das Kriterium „hinreichend groß“ ist so zu interpretieren, daß alle in Kapitel 3 auftretenden Mengen in $\mathcal{Set}$ liegen, d.h. $\mathcal{Set}$ enthält neben endlich vielen Grundmengen M_i, die als einfache Wertebereiche fungieren, auch alle komplexen Wertebereiche, die gemäß Abschnitt 3.2.1 konstruierbar sind.

$f(a) := b,$

wenn ein Übergang von f_{alt} zu f_{neu} stattfindet mit

$$f_{neu}(x) = \begin{cases} b & \text{, falls } x = a \\ f_{alt}(x) & \text{, falls } x \neq a, \end{cases}$$

für $x \in Def(f_{alt}) \cup \{a\}$.

A.2 Definitionen aus der Graphentheorie

Wir beginnen mit der Definition endlicher, gerichteter Graphen. Danach konzentrieren wir uns auf den Begriff des Baumes und insbesondere des beschrifteten Baumes, der bei der Formalisierung des Datenmodells eine zentrale Rolle einnimmt.

Es sei eine abzählbar unendliche Menge $\mathcal{V} = \{r_0, r_1, \ldots\}$ gegeben, die einen Vorrat an Knoten $r_0, r_1, \ldots$ zur Konstruktion von Graphen umfaßt. Ferner gebe es einen besonders ausgezeichneten Nullknoten *null*, der selbst nie Bestandteil eines Graphen sein wird.

Es fällt auf, daß für die Elemente der Knotenmenge $\mathcal{V}$ wird nicht der Buchstabe „v“, sondern „r“ verwendet wird. Dies hat zweierlei Gründe: Zum einen wird „v“ bereits für Werte (values) verwendet, und es sollen Verwechselungen mit Werten ausgeschlossen werden, zum anderen bietet sich die Wahl von „r“ an, da die r_i als Identifikatoren von Knoten gerade eine Entsprechung in den RIDs (record identifiers) der Implementation des ESCHER-Prototyps finden.

Definition A.1 (gerichteter Graph)

Ein endlicher gerichteter Graph G ist ein Paar (V, E) mit $V \in \mathcal{FS}et(\mathcal{V})$ und $E \subseteq V \times V$. Die Elemente von V heißen *Knoten*, die Elemente von E heißen *Kanten* des Graphen G.

Ist G = (V, E) ein gerichteter Graph, dann sei $\rightarrow^*$ die reflexive und transitive Hülle der Relation E. Wir schreiben also $r \rightarrow^* r'$, wenn r = r' oder wenn es eine Folge von Kanten aus E gibt, die die Knoten r und r' verbindet. ❒

Definition A.2 (Baum)

Ein gerichteter Graph B = (V, E) ist ein *Baum*, wenn gilt:

(i) $\forall r \in V: (r_1, r), (r_2, r) \in E \Rightarrow r_1 = r_2 \neq r$

(ii) $\exists r_0 \in V: \{r \in V \mid \forall r' \in V: (r', r) \notin E \wedge r_0 \rightarrow^* r'\} = \{r_0\}$

Die Definition besagt gerade, daß ein Baum ein zusammenhängender, kreisloser gerichteter Graph ist.

- Der nach (ii) eindeutig bestimmte Knoten $r_0 \in V$ heißt *Wurzel* des Baumes und wird auch mit *root* (B) bezeichnet.
- Ist $(r_1, r_2) \in E$, dann heißt r_2 *Sohn* von r_1, und umgekehrt ist r_1 *Vater* von r_2. Ein Knoten ohne Söhne heißt *Blatt*.
- Für einen Baum B = (V, E) setzen wir V(B) := V und E(B) := E.

- Zwei Bäume B_1 und B_2 heißen *disjunkt*, wenn $V(B_1) \cap V(B_2) = \emptyset$.
- Die Menge aller Bäume bezeichnen wir mit $\mathcal{Tree}$.
- Ist B = (V, E) ein Baum, dann soll

  $Tree : V \rightarrow \mathcal{Tree}$

  diejenige Funktion sein, die für $r \in V$ den *Unterbaum* B' = (V', E') von B mit Wurzel r liefert, d.h. $V' = \{ r' \in V \mid r \rightarrow^* r' \}$ und $E' = E \cap (V' \times V')$.

❒

Bislang ist ein Baum zu „abstrakt“, wenn er allein durch die Angabe der Knoten- und Kantenmenge beschrieben wird. „Konkreter“ wird ein Baum dadurch, daß man den Knoten eines Baumes sog. Beschriftungen zuordnet. Diese sind Gegenstand der folgenden Definition. Grob gesprochen sind Beschriftungen (Label, Wert)-Paare, die an einen Baumknoten geheftet werden. Da dieselben „Labels“ in verschiedenen Bäumen auftauchen können, werden wir Beschriftungen gleich für eine Menge von Bäumen definieren.

Definition A.3 (explizite Beschriftungen)

Es sei eine endliche Menge $\mathcal{B} = \{b_1, \ldots, b_n\}$ von partiellen Funktionen

$$b_i : \mathcal{V} \cdots> D \tag{A.1}$$

gegeben, wobei $D \in \mathcal{Set}$ eine beliebige Menge sei. Die Funktionen b_i werden *explizite Beschriftungen* genannt.

Für $b \in \mathcal{B}$ und $r \in \mathrm{Def}(b)$ wird der Wert $b(r)$ die *explizite Beschriftung* (*Label*) des Knotens r bezüglich *b* genannt.

Ist B = (V, E) ein beliebiger Baum und $L \subseteq \mathcal{B}$, dann gilt

$$B \in \mathcal{Tree}(L) \Leftrightarrow \forall\, b \in \mathcal{B} - L\ \forall\, r \in V : r \notin \mathrm{Def}(b),$$

d.h. die Knoten von B haben höchstens Beschriftungen aus L. ❒

Definition A.4 (implizite Beschriftungen)

Es gibt in jedem Baum B = (V, E) neben den expliziten auch *implizite* Beschriftungen, die sich allein aus der Struktur des Baumes ergeben:

$$children : V \rightarrow \wp(V) \text{ für die Menge der Söhne eines Knotens v,} \tag{A.2}$$

mit $children(r) := \{ r' \in V \mid (r, r') \in E \}$

$$degree : V \rightarrow \mathbb{N}_0 \text{ für die Anzahl der Söhne des Knotens v} \tag{A.3}$$

mit $degree(r) = |\, children(r) \,|$

$$parent : V \cup \{null\} \cdots> V \cup \{null\} \text{ für den Vater eines Knotens v,} \tag{A.4}$$

$$\text{mit } parent(r) := \begin{cases} r' & \text{, falls } \exists\, r' \in V\text{: } (r', r) \in E \\ null & \text{, sonst} \end{cases}$$

$$isRoot : V \rightarrow \{true, false\} \text{ als Test, ob ein Knoten Wurzel eines Baumes ist,} \tag{A.5}$$

mit $isRoot(r) \Leftrightarrow parent(r) = null$

❒

Laut Definition eines Baumes wird zunächst keine Aussage über Ordnungsrelationen auf den Knoten eines Baumes gemacht. Oftmals ist jedoch eine lineare Ordnung der Söhne eines Knotens gewünscht. Die nächste Definition führt daher geordnete Bäume ein.

Definition A.5 (geordneter Baum)

Es sei B = (V, E) ein Baum. Auf B sei eine implizite Beschriftung

$$ord : V \cdots> IN, \tag{A.6}$$

mit Def(*ord*) = V − {*root* (B)} definiert, wobei gelte:

> Für alle Knoten r ∈ V mit *children* (r) ≠ ∅ bildet *ord* die Elemente der Menge *children* (r) bijektiv auf {1, …, *degree* (r)} ab.

Diese Bedingung bedeutet gerade, daß für die Söhne eine lineare Ordnung „<" definiert ist, indem man setzt:

$$r_1 < r_2 \Leftrightarrow ord\ (r_1) < ord\ (r_2).$$

Ein Baum mit der impliziten Beschriftung *ord* heiß *geordneter* Baum.

❒

In dieser Arbeit wird, sofern nicht ausdrücklich anders vermerkt, immer von geordneten Bäumen ausgegangen. Geordnete Bäume sind vorteilhaft, wenn Bäume zur Implementation von komplexen Datenstrukturen verwendet werden – wie es in dieser Arbeit der Fall ist – und dabei insbesondere der Zugriff ausgehend von einem Knoten r auf einen Sohnknoten r'∈ *children* (r) zu einer gegebenen Ordnungszahl adäquat unterstützt wird.

Wir definieren die Funktion

$$get\_child: V \times IN \cdots> V \cup \{r_{null}\} \tag{A.7}$$

mit

$$get\_child\ (r, i) := \begin{cases} r' & \text{, falls } \exists\ r' \in children\ (r): \\ & \quad ord\ (r') = i \\ r_{null} & \text{, sonst} \end{cases}$$

Die „Umkehrfunkion" zu *get_child* ist die implizite Beschriftung *parent*.

Anhang B

Syntaxregeln

B.1 Vereinbarungen zur Angabe von Grammatiken

Die in dieser Arbeit vorkommenden Grammatikregeln werden in der üblichen erweiterten Backus-Naur-Form notiert. Es sollen folgende Vereinbarungen gelten:

- Nichtterminale werden kursiv gedruckt.
  Beispiele: *struct*, *defType*
- Terminale werden im Courier-Font gesetzt. Einzelne Terminalzeichen werden von Apostrophen umgeben.
  Beispiele: `define type`, `'('` *expr* `')'`
- Als Metazeichen werden verwendet:
  - ::= trennt die linke und rechte Seite einer Produktion
  - […] für die optionale Angabe der zwischen den Klammern stehenden Folge von Nichtterminalen und Terminalen
  - { … } für 0- bis n-fache Wiederholung der zwischen den Klammern stehenden Folge von Nichtterminalen und Terminalen
  - | trennt alternative rechte Seiten einer Produktion
  - (… | …) bezeichnet Alternativen innerhalb der rechten Seite eine Produktion

Sind nt_i Nichtterminale einer Grammatik ($1 \le i \le k$), dann bezeichnen wir mit $\mathcal{L}(nt_i)$ die mit nt_i als Startsymbol erzeugte Sprache. $\mathcal{L}(nt_1, \ldots, nt_k)$ soll die Vereinigung aller dieser Sprachen sein.

Der *Namensvorrat* $\mathcal{N}$ werde definiert durch $\mathcal{N} := \mathcal{L}$ (*identifier*) und den Produktionen

identifier ::= *alpha* {*symbol*}
symbol ::= *alpha* | *digit* | *special*
alpha ::= 'a' | … | 'z' | 'A' | … | 'Z'
digit ::= '0' | '1' | … | '9'
special ::= '_' (evtl. weitere Sonderzeichen)

B.2 Die DDL für ESCHER+

DDL ::= *simpleDDL* |
`DDL_transaction begin` { *simpleDDL* }
`end DDL_transaction ';'`

simpleDDL ::= *enumDef* | *variantDef* | *typeDef* | *tableDef* | *opDef* | *relDef*
| *modify* | *drop*

enumDef ::= `define enumtype` *name enum* `end define ';'`

variantDef ::= `define variant` *name variants* [*constraint*] `end define ';'`

typeDef ::= `define type`
name typeStruct [*default*] [*isa*] [*ops*] [*constraint*]
`end define ';'`

tableDef ::= `define table` *name tableStruct* [*init*] [*constraint*] `end define ';'`

opDef ::= `define operation`
`-name` *qualifName signDef* [*opEncDef*] [*codeDef*]
`end define ';'`

relDef ::= `define relship`
name relComp [*relAttr*] [*relKeys*] [*relCard*] [*relVal*] [*relIncl*] *relViews*
`end define ';'`

modify ::= `modify`
(*modOp* | *modEnc* | *modDefault* | *modEnum* | *modVariant*)
`end modify ';'`

modOp ::= *opName codeDef*

modEnc ::= (*opName opEncDef* | `-structpath` *structPath encDef*)

modDefault ::= `-type` *defType default*

modEnum ::= `-enumtype` *identifier* [*addEnum*] [*dropEnum*] [*renameEnum*]

modVariant ::= `-variant` *identifier* [*addVar*] [*dropVar*]

drop ::= `drop`
((`enumtype` | `variant | type` | `table` | `relship | constraint`)
identifier | `operation` *qualifName*) `';'`

name ::= `-name` *identifier*

enum ::= `-enum` *identifier* { `','` *identifier* }

variants ::= `-variant` (*labelStruct* {`','` *labelStruct*} | *defType* {`','` *defType*})

typeStruct ::= `-struct` *tupleStruct*

tableStruct ::= `-struct` *struct*

default ::= `-default` *attrName* `'='` *expr* { `','` *attrName* `'='` *expr* }

isa ::= `-isa` *defType*
init ::= `-init` *expr*
constraint ::= `-constraint` *constrDef* { `,` *constrDef* }
signDef ::= `-sign` *sign*
encDef ::= `-enc` *enc*
opEncDef ::= `-enc` *opEnc*
codeDef ::= `-code` *script*
opName ::= `-op` *qualifName*
ops ::= `-ops` *opList*
addEnum ::= `-add` *identifier*
dropEnum ::= `-drop` *identifier*
renameEnum ::= `-rename` *identifier* `to` *identifier*
addVar ::= `-add` (*labStruct* | *defType*
| `ops needed` *identifier* | `struct needed` *attrName*)
dropVar ::= `-drop` (*label* | `ops needed` *identifier* | `struct needed` *attrName*)
relComp ::= `-comp` *tupleStruct*
relAttr ::= `-attr` *tupleStruct*
relKeys ::= `-key` *relKey* { `,` *relKey* }
relCard ::= `-card` *relCardItem* { `,` *relCardItem* }
relVal ::= `-val` *expr* { `,` *expr* }
relIncl ::= `-incl` *inclCondition* { `,` *inclCondition* }
relViews ::= `-views` *relViewsItem* { `,` *relViewsItem* }

qualifName ::= [*defType* `.`] *identifier*
opList ::= *opDecl* {`,` *opDecl* }
opDecl ::= *identifier* `(` *sign* `)` [*opEnc*]
sign ::= *argList* `->` *result*
result ::= *struct* | `void`
argList ::= [*arg* { `,` *arg* }]
arg ::= [*identifier* `:`] *struct*
enc ::= `readwrite`|`read`|`private`
opEnc ::= `public`|`private`
labelStruct ::= *label* `:` *struct*
relKey ::= *attrName* | `[` *attrName* { `,` *attrName* } `]`
relCardItem ::= *identifier* `(` *minCard* `,` *maxCard* `)`
minCard ::= *natInt*
maxCard ::= *natInt* | `*`
inclCondition ::= *identifier* `in` *structPathExpr* [*cascade*]

relViewsItem ::= *identifier* '->' *refTupleStruct*
refTupleStruct::= '[' *attrName* ':' *refStruct* { ',' *attrName* ':' *refStruct* } ']'
refStruct ::= *refTupleStruct* | '{' *refStruct* '}' | '<' *refStruct* '>' | '@' *identifier*

struct ::= (*simpleStruct* | *complexStruct* | *structVar*) [*enc*]
simpleStruct ::= `int[eger]` | `float` | `string` | `char` | `boolean` | *defType*
complexStruct ::= *tupleStruct* | '{' *struct* '}' | '<' *struct* '>' | '{*' *struct* '*}'
| '<' *struct* '|' *int* ':' *int* { ',' *int* ':' *int* } '>' | *vector* | *matrix*
tupleStruct ::= '[' [*attrName* ':' *struct* { ',' *attrName* ':' *struct* }] ']'
vector ::= `vector` '(' *natInt* ')'
matrix ::= `matrix` '(' *natInt* ',' *natInt* ')'

structVar ::= '\' *identifier*
attrName ::= *identifier*
label ::= *identifier*
defType ::= *identifier*

structPathExpr ::= *identifier path*
path ::= {*step* }
step ::= ('.#' | '.' *attrName* | ':' *identifier*)

constrDef ::= [*identifier*] '(' *condition* ')' [*constrMode*]
condition ::= *keyConstraint* | *inclConstraint* | *disjConstraint* | *valueConstraint*
constrMode ::= `hard` | `soft`
keyConstraint ::= *structPathExpr* (`has key` *key* | `has unique elements`)
key ::= *keyPath* | '[' *keyPath* { ',' *keyPath* } ']'
keyPath ::= *attrName* { '.' *attrName* }
inclConstraint ::=
(*structPathExpr* `in` *structPathExpr*
| *structPathExpr* `is included in` *structPathExpr*) [*cascade*]
disjConstraint ::=
(*structPathExpr* `not in` *structPathExpr*
| *structPathExpr* `is disjoint to` *structPathExpr*) [*cascade*]
cascade ::= `cascade` *cascadeOp* [',' *cascadeOp*]
cascadeOp ::= `add` | `rem`
valueConstraint ::= *expr* `with` *identifier* '=' *structPathExpr*

B.3 Die Syntax von SCRIPT+

script ::= '[' { *params* } '->' *result* '|' { *decls* } '|' *statements* ']'
params ::= *identifier* ':' *extStruct* { ',' *identifier* ':' *extStruct* }
decls ::= *identifier* ':' *struct* ['=' *expr*] ';'
statements ::= *statement* {*statement*}
statement ::= (*assign* | *opCall* | *update* | *transaction* | *if* | *case* | *for*
| *while* | *return* | `abort`) ';'
assign ::= *pathExpr* `:=` (*expr* | *addTypeExpr*)
pathExpr ::= *identifier* { *pathStep* } | `self` '.' *attrName* { *pathStep* }
pathStep ::= ('.' *attrName* | '[' *expr* { ',' *expr* } ']')
update ::= *pathExpr* `add` (*expr* | '(' *addTypeExpr* ')') [(`at` *expr* | `as last`)]
| *pathExpr* `rem` (*expr* | `at` (*expr* | `last`))
| *pathExpr* `move` (*expr* | `last`) `to` (*expr* | `last`)
addTypeExpr ::= *expr* `add type` *defType* '(' *initVals* ')'
transaction ::= `transaction begin` *statements* `end transaction`
if ::= `if` *expr* `then` *statements*
{ `elseif` *expr* `then` *statements* }
[`else` *statements*] `end if`
case ::= `case` *expr* `of`
identifier ':' *statements* { *identifier* ':' *statements* }
[`else` *statements*] `end case`
for ::= `for` *identifier* '=' *expr* '..' *expr* `do` *statements* `end for`
| `for` *rangeExpr* `do` *statements* `end for`
while ::= `while` *expr* `do` *statements* `end while`
return ::= `return` [*expr*]

expr ::= [*relExpr relOp*] *relExpr*
relOp ::= '=' | '<>' | '<' | '>' | '<=' | '>=' | `in` | `is` (`proper`) `subset of`
relExpr ::= [*relExpr addOp*] *term*
addOp ::= '+' | '-' | `or` | `union` | `minus` | `append`
term ::= [*term multOp*] *factor*
multOp ::= '*' | '/' | `div` | `mod` | `and` | `sect`
factor ::= `the` '(' *singleRange* ')'
| '(' *quantifExpr* ')' | `not` *expr*
| `card` '(' *expr* ')' | `empty` '(' *expr* ')' | `pos` '(' *identifier* ')'
| *expr* `as` (*label* | *defType*) | *expr* `is` *label*
| *expr* `is` [`transient` | `persistent`] *defType*
| *expr pathStep* | { *expr* '.' } *opCall*
| *former* | *literal* | *identifier* | '(' *expr* ')' | `null` | `dne` | `self`

quantifExpr ::= *quantor identifier* `in` *expr* {`,` *quantor identifier* `in` *expr* } [`:` *expr*]
quantor ::= `exists` | `all`
former ::= *setFormer* | *listFormer* | *bagFormer*
setFormer ::= `{` *targetExpr* `|` *rangeExpr* `}`
listFormer ::= `<` *targetExpr* `|` *rangeExpr* `>`
bagFormer ::= `{*` *targetExpr* `|` *rangeExpr* `*}`
targetExpr ::= *expr*
rangeExpr ::= *identifier* `in` *expr* [`with` `null`] { `,` *identifier* `in` *expr* [`with` `null`]}
[`:` *expr*]
singleRange ::= *identifier* `in` *expr* [`:` *expr*]
opCall ::= *identifier* `(` [*expr* { `,` *expr* }] `)`
literal ::= *atomic* | *complex* | *object*
atomic ::= *int* | *float* | *char* | *string* | *bool*
int ::= *natInt* | `-` *digit* {*digit*}
natInt ::= [`+`] *digit* {*digit*}
float ::= [(`+` | `-`)] *digit* { *digit* } `.` {*digit*} [`E` (`+` | `-`) *digit* {*digit*}]
char ::= `'` druckbares Zeichen `'`
string ::= `"` Folge druckbarer Zeichen `"`
bool ::= `true` | `false`
complex ::= *tuple* | *set* | *list* | *bag* | *array*
tuple ::= `[` [*attrName* `:` *expr* { `,` *attrName* `:` *expr* }] `]`
set ::= `{` [*expr* { `,` *expr* }] `}`
list ::= `<` [*expr* { `,` *expr* }] `>`
bag ::= `{*` [*expr* { `,` *expr* }] `*}`
array ::= `<` [*expr* { `,` *expr* }] `|` *int* `:` *int* { `,` *int* `:` *int* } `>`
object ::= *defType* `(` *initVals* `)`
variant ::= `(+` *label* `:` *expr* `+)`
initVals ::= [*attrName* `=` *expr* { `,` *attrName* `=` *expr* }]

Anhang C

Beispiele zur Ableitung von ECA-Regeln aus Beziehungsdefinitionen

C.1 Regeln für die Beziehung `Teilnahme`

Es sei die Beziehung `Teilnahme` gemäß

```
define relship
  -name Teilnahme
  -comp [P: Person, K: Kurs]
  -attr [Anm: date, Erg: integer]
  -key  [P, K]
  -val  Erg = null or (0 <= Erg and Erg <= 10)
  -incl K in DV_Kurse cascade add, rem
  -view P -> [Kurse:{@K}],
        K -> [T_Liste: <[Teilnehmer:@P,
              Anmeldung:@Anm, Ergebnis: @Erg]>]
end define;
```

definiert.

Sofern nicht anders angegeben, soll für die *coupling*-Komponente und die *condition*-Komponente der nachfolgenden ECA-Regeln gelten:

coupling : `immediate`
condition : `true`

Aus der Beziehung `Teilnahme` leiten sich die folgenden ECA-Regeln ab:

name : `T1`
event : `after remove(K, k) with K = Teilnahme:P.Kurse`
action : `suspend(T3);`
`p := getObject(K);`

```
          k.T_Liste rem
             the(t in k.T_Liste: t.Teilnehmer = p);

name     :T2
event    :after insert(K, k) with K = Teilnahme:P.Kurse
action   :suspend(T4, T7);
          k.T_Liste add
          [Teilnehmer: p, Anmeldung: null, Ergebnis: null]
          as last; // Einfügen am Ende der Liste

name     :T3
event    :after remove(T, t) with T = Teilnahme:K.T_Liste
action   :suspend(T1);
          k := getObject(T);
          t.Teilnehmer.Kurse rem k;

name     :T4
event    :after insertInList(T, t, n)
          with T = Teilnahme:K.T_Liste
action   :suspend(T2, T6);
          k := getObject(T);
          t.Teilnehmer.Kurse add k;

name     :T5
event    :after update(p_neu, p_alt)
          with p_neu = Teilnahme:K.T_Liste.Teilnehmer
action   :suspend(T1, T2, T6);
          k := getObject(p_neu);
          p_alt.Kurse rem k;
          p_neu.Kurse add k;

name     :T6
event    :before insert(K, k) with K = Teilnahme:P.Kurse
condition:k = null
action   :abort;

name     :T7
event    :before insertInList(T, t, n)
          with T = Teilnahme:K.T_Liste
condition:(t.Teilnehmer = null) or
          (exists u in T: t.Teilnehmer = u.Teilnehmer)
action   :abort;
```

```
name     : T8
event    : before update(p, p_neu)
           with p_neu = Teilnahme:K.T_Liste.Teilnehmer
prepare  : T := getParent(p, 2);
condition: (p_neu = null) or
           (exists t in T: t.Teilnehmer = p_neu)
action   : abort;

name     : T9
event    : before update(e, e_neu)
             with e = Teilnahme:K.T_Liste.#.Ergebnis
condition: not (e_neu = null or (0 <= e_neu and e_neu <= 10))
action   : abort;

name     : T10
event    : after insertInList(T, t, n)
           with T = Teilnahme:K.T_Liste
prepare  : k = getObject(T);
condition: (card(T) = 1) and not (k in DV_Kurse)
action   : DV_Kurse add k;

name     : T11
event    : after remove(K, k) with K = DV_Kurse
action   : for t in k.T_Liste do
             k.T_Liste rem t;
           end for;
```

C.2 Regeln für die Beziehung `Ausleihe`

Es sei die Beziehung `Ausleihe` gemäß

```
define relship
  -name Ausleihe
  -comp [L: Leser, E: Exemplar]
  -attr [LDat: date]
  -key  E
  -view L -> [Ausleihen:{[Exemplar: @E, Leihdatum: @LDat]}],
        E -> [ausgeliehen_von: @L, Leihdatum: @LDat]
end define;
```

definiert (vgl. Beispiel 6.7).

Daraus leiten sich die nachfolgenden ECA-Regeln ab:

```
name   : A1
event  : after remove(A, t) with A = Ausleihe:L.Ausleihen
action : suspend(A5, A6);
         t.Exemplar.entliehen_von := dne;

name   : A2
event  : after insert(A, t) with A = Ausleihe:L.Ausleihen
action : suspend(A5, A6);
         leser := getObject(A);
         t.Exemplar.entliehen_von := leser;
         t.Exemplar.entliehen_am := t.Leihdatum;

name   : A3
event  : after update(e_neu, e_alt)
             with e_neu = Ausleihe:L.Ausleihen.#.Exemplar
action : suspend(A5, A6);
         leser := getObject(e_neu);
         t := getParent(e_neu, 1);
         e_alt.entliehen_von := dne;
         e_neu.entliehen_von := leser;
         e_neu.entliehen_am := t^.Leihdatum;

name   : A4
event  : after update(d_neu, d_alt)
         with d_neu = Ausleihe:L.Ausleihen.#.Leihdatum
action : suspend(A6);
         t := getParent(d_neu, 1);
         t^.Exemplar.entliehen_am := d_neu;

name   : A5
event  : after update(l_neu, l_alt)
         with l_neu = Ausleihe:E.entliehen_von
action : suspend(A1, A2, A6, A7, A9);
         e := getObject(l_neu);
         if l_alt <> dne and l_alt <> null then
           l_alt.Ausleihen rem
               the(t in l_alt.Ausleihen: t.Exemplar = e);
         end if;
         if l_neu <> null then
           l_neu.Ausleihen add
                [Exemplar: e, Leihdatum: e.entliehen_am];
         end if;
```

```
name     :A6
event    :after update(d_neu, d_alt)
          with d = Ausleihe:E.entliehen_am
action   :suspend(A1, A4, A5);
          e := getObject(d_neu);
          leser := e.entliehen_von;
          if d_neu = dne then
           if leser <> null then
             for t in leser.Ausleihen: t.Exemplar = e do
              leser.Ausleihen rem t;
             end for;
           end if;
          else
           if leser <> null then
             for t in leser.Ausleihen: t.Exemplar = e do
              t.Leihdatum := d_neu;
             end for;
           end if;
          end if;

name     :A7
event    :before insert(A, t) with A = Ausleihe:L.Ausleihen
condition:t.Exemplar = null
action   :abort;

name     :A8
event    :before update(e, e_neu)
              with e = Ausleihe:L.Ausleihen.#.Exemplar
condition:e_neu = null
action   :abort;

name     :A9
event    :before insert(A, t) with A = Ausleihe:L.Ausleihen
condition:t.Exemplar <> dne
action   :abort;

name     :A10
event    :before update(e, e_neu)
              with e = Ausleihe:L.Ausleihen.#.Exemplar
condition:e_neu <> dne
action   :abort;
```

Literatur

[AB84] Serge Abiteboul, Nicole Bidoit: *Non First Normal Form Relations to represent hierarchically organized data.* – Proc. of the 3rd ACM SIGACT/SIGMOD Symposium on Principles of Database Systems, S. 191-200, 1984

[AB87] M. P. Atkinson, O. P. Buneman: *Types and Persistence in Database Programming Languages.* – ACM Computing Surveys, Vol. 19, No. 2, S. 105-191, 1987

[AB91] Serge Abiteboul, Anthony Bonner: *Objects and Views*. – Proc. of the ACM SIGMOD International Conference on Management of Data, S. 238-247, 1991

[ABC+76] M.M. Astrahan, M. W. Blasgen, D.D. Chamberlin et. al.: *System R: A Relational Approach to Data Base Management.* – ACM Transactions on Database Systems, Vol. 1, No. 2, S. 97-137, 1976

[ABD+89] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, Stanley Zdonik: *The Object-Oriented Database System Manifesto.* – Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD), S. 40-57, 1989, auch in [BDK92], S. 3-20

[ABG+93] Antonio Albano, , R. Bergamini, Giorgio Ghelli, Renzo Orsini: *An Object Data Model with Roles*. – Proc. of the 19th International Conference on Very Large Data Bases, S. 39-51, 1993

[ACC81] M. P. Atkinson, K. J. Chisholm, W. P. Cockshott: *PS-Algol: an algol with a persistent heap.* – SIGPLAN Notices, Vol. 17, No. 7, S. 24-31, 1981

[ACL91] Rakesh Agrawal, Roberta Cochrane, Bruce Lindsay: *On Maintaining Priorities in an Production Rule System*. – Proceedings of the 17th International Conference on Very Large Data Bases, S. 479-487, 1991

[AD92] R. Ahad, D. Dedo: *OpenODB from Hewlett-Packard: A commercial object-oriented database management system.* – Journal of Object-Oriented Programming, Vol. 4, No. 9, S. 31-35, 1992

[AD93] Paolo Atzeni, Valeria De Antonellis: *Relational Database Theory.* – Redwood City (CA): Benjamin/Cummings, 1993

[ADG+87] K. Abramowicz, K. R. Dittrich, W. Gotthard, R. Längle, P. C. Lockemann, T. Raupp, S. Rehm, T. Wenner: *Datenbankunterstützung für Software-Produktionsumgebungen.* – Proceedings Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'87), Informatik-Fachberichte 136, Berlin: Springer, S. 116-131, 1987

[ADL+91] K. Abramowicz, K. R. Dittrich, R. Längle, M. Ranft, T. Raupp, S. Rehm: *DAMOKLES - Architektur, Implementierung, Erfahrungen.* – Informatik - Forschung und Entwicklung, Vol. 6, No. 1, S. 1-13, 1991

[AFS89] S. Abiteboul, P.C. Fischer, H.-J. Schek (Hrsg.): *Nested Relations and Complex Objects in Databases*. – Lecture Notes in Computer Science, Band 361, Berlin: Springer, 1989

[AG89] R. Agrawal, N.H. Gehani: *ODE (Object Database Environment): The Language and the Data Model.* – Proceedings of the ACM SIGMOD Conference on Management of Data, S. 36-45, 1989

[AGO90] Antonio Albano, Giorgio Ghelli, Renzo Orsini: *Objects and Classes for a Database Programming Language*. – Progetto finalizzato sistemi informatici e calcolo parallelo, sottoprogetto 5: Sistemi evoluti per Basi di Dati, Rapporto Technico N. 5/24, Università di Pisa, 1990

[AGO91] Antonio Albano, Giorgio Ghelli, Renzo Orsini: *A Relationship Mechanism for a Strongly Typed Object-Oriented Database*. – Proceedings of the 17th International Conference on Very Large Data Bases, S. 565-575, 1991

[AGO95] Antonio Albano, Giorgio Ghelli, Renzo Orsini: *Fibonacci: A Programming Language for Object Databases.* – VLDB Journal, Vol. 4, Nr. 3, S. 403-444, 1995

[AH87] Serge Abiteboul, Richard Hull: *IFO: A Formal Semantic Database Model*. – ACM Transactions on Database Systems, Vol. 12, No. 4, S. 525-565, 1987

[AH88] Serge Abiteboul, Richard Hull: *Restructuring Hierarchical Database Objects*. – Theoretical Computer Science, Vol. 62, S. 3-38, 1988

[AHS91] T. Andrews, C. Harris, K. Sinkel: *ONTOS: a persistent database for C++*. – In: R. Gupta, E. Horowitz (Hrsg.): Object-Oriented Databases with Applications to CASE, Networks, and VLSI Design. Englewood Cliffs (NJ): Prentice-Hall, S. 387-406, 1991

[Alb91] Joseph Albert: *Algebraic Properties of Bag Data Types*. – Proceedings of the 17th International Conference on Very Large Data Bases, S. 211-219, 1991

[AWH92] Alexander Aiken, Jennifer Widom, Joseph M. Hellerstein: *Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism.* – Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, S. 59-68, 1992

[Bau93] Peter Baumann: *Ein konzeptuelles Informationsmodell für Visualisierungsdatenbanken*. – Dissertation, TH Darmstadt. Shaker-Verlag, Aachen, 1993

[BCG+87] Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, Nat Ballou: *Data Model Issues for Object-Oriented Applications.* – ACM Transactions on Office Information Systems, Vol. 5, No. 1, S. 3-26, 1987

[BCN92] C. Batini, S. Ceri, S. B. Navathe: *Conceptual Database Design: An Entity Relationship Approach.* – Redwood City (CA): Benjamin Cummings, 1992

[BD77] Charles W. Bachmann, Manilal Daya: *The Role Concept in Data Models.* – Proc. of the 3rd International Conference on Very Large Data Bases, S. 464-476, 1977

[BD95] V. Benzaken, A. Doucet: *Thémis: a Database Programming Language Handling Integrity Constraints.* – VLDB Journal, Vol. 4, No. 3, S. 493-517, 1995

[BDK92] François Bancilhon, Claude Delobel, Paris Kannelakis (Hrsg.): *Building an Object-Oriented Database System. The Story of O_2.* – San Mateo, California: Morgan Kaufmann, 1992

[Bee90] Catriel Beeri: *A formal approach to object-oriented databases*. – Data & Knowledge Engineering, Vol. 5, S. 353-382, 1990

[Ber92] Elisa Bertino: *A View Mechanism for Object-Oriented Databases.* – Proceedings of the International Conference on Extending Database Technology (EDBT'92). Lecture Notes in Computer Science 580. Berlin: Springer, S. 136-151, 1992

[BK93] Peter J. Barclay, Jessie B. Kennedy: *Viewing Objects.* – Proceedings of the 11th British National Conference on Databases, Lecture Notes in Computer Science No. 696. Berlin: Springer, S. 93-110, 1993

[BLT86] J. A. Blakeley, P.-Å. Larson, F. W. Tompa: *Efficiently Updating Materialized Views.* – Proceedings of the ACM SIGMOD International Conference on Management of Data, S. 61-71, 1986

[Boo91] G. Booch: *Object Oriented Design.* – Redwood City (CA): Benjamin Cummings, 1991

[BPR88] Michael R. Blaha, William J. Premerlani, James E. Rumbaugh: *Relational Database Design using an Object-Oriented Methodology.* – Communications of the ACM, Vol. 31, No. 4, S. 414-427, 1988

[BR84] M. L. Brodie, D. Ridjanovic: *On the design and specification of database transactions.* – In: M. L. Brodie, J. Mylopoulos, J. W. Schmidt (Hrsg.): On Conceptual Modelling. New York: Springer, S. 277-306, 1984

[Bro84] M. L. Brodie: *On the Development of Data Models.* – In: M. L. Brodie, J. Mylopoulos, J. W. Schmidt (Hrsg.): On Conceptual Modeling, New York: Springer, S. 19-47, 1984

[Bru94] Marlis Brunk: *Eine Anfragesprache und ein Sichtenkonzept für objektorientierte Datenbanksysteme.* – Fortschritt-Berichte VDI Reihe 10 Nr. 290, Düsseldorf: VDI-Verlag, 1994

[Brü94] H. H. Brüggemann: *Object-oriented Authorization.* – in: [PT94], S. 139-160

[BS81] F. Bancilhon, N. Spyratos: *Update Semantics and Relational Views.* – ACM Transactions on Database Systems, Vol. 12, No. 4, S. 525-565, 1981

[BSK+91] Thierry Barsalou, Niki Siambela, Arthur M. Keller, Gio Wiederhold: *Updating Relational Databases through Object-Based Views.* – Proceedings of the ACM SIGMOD International Conference on Management of Data, S. 248-257, 1991

[Buc94] A. P. Buchmann: *Active Object Systems.* – In: [DÖB+94], S. 201-224, 1994

[Bur+86] Thomas Burns et. al.: *Reference Model for DBMS Standardization. Final Report by the Database Architecture Framework Taskgroup (DAFTG) of the ANSI/X3/SPARC Database System Study Group.* – SIGMOD Record, Vol. 15, No. 1, S. 19-58, 1986

[Car84] L. Cardelli: *A semantics of multiple inheritance.* – in: International Symposium on Semantics of Data Types. LNCS 173, Berlin: Springer, S. 51-67, 1984 (ebenfalls in: S. Zdonik, D. Maier (Hrsg.): Readings in Object-Oriented Database Systems. San Francisco: Morgan Kaufmann, 1990)

[Cat91] R. G. G. Cattell: *Object Data Management.* – Reading (MA): Addison-Wesley, 1991

[Cat+94] R. G. G. Cattell (Hrsg.): *The Object Database Standard: ODMG-93. Release 1.1.* – San Francisco: Morgan Kaufmann, 1994

[CBS94] Roger H.L. Chiang, Terence M. Barron, Veda C. Storey: *Reverse engineering of relational databases: Extraction of an EER model from a relational database.* – Data & Knowledge Engineering, 12, S. 107-142, 1994

[CCS94] C. Collet, T. Coupaye, T. Svensen: *NAOS. Efficient and modular reactive capabilities in an Object-Oriented Database System.* – Proceedings of the 20th International Conference on Very Large Data Bases, S. 132-143, 1994

[CFP+94] S. Ceri, P. Fraternali, S. Paraboschi, L. Tanca: *Automatic Generation of Procedure Rules for Integrity Maintenance.* – ACM Transactions on Database Systems, Vol. 19, No. 3, S. 367-422, 1994

[CG85] S. Ceri, G. Gottlob: *Translating SQL into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries.* – IEEE Transactions on Software Engineering, Vol. 11, No. 4, S. 324-345, 1985

[Cha+76] D. D. Chamberlin et. al.: *SEQUEL2: a unified approach to data definition, manipulation, and control.* – IBM J. Research and Development, Vol. 20, No. 6, S. 560-575, 1976

[Che76] P. Chen: *The Entity-Relationship Model – Toward a Unified View on Data.* – ACM Transactions on Database Systems, Vol. 1, No. 1, S. 9-36, 1976

[Cod70] E. F. Codd: *A Relational Model for Large Shared Data Banks.* – Communications of the ACM, Vol. 13, No. 6, S. 377-387, 1970

[Cod79] E. F. Codd: *Extending the database relational model to capture more meaning.* – ACM Transactions on Database Systems, Vol. 4, S. 397-434, 1979

[Cod90] E. F. Codd: *The Relational Model for Database Management: Version 2.* – Reading (Mass.): Addison-Wesley, 1990

[COD71] CODASYL Database Task Group: *April 1971 Report.* – Amsterdam: IFIP Adm. Data Processing Group, 1971

[CT94] Daniel K.C. Chan, Phil W. Trinder: *Object Comprehensions: A Query Notation for Object-Oriented Databases.* – Proceedings of the British National Conference on Databases. Lecture Notes in Computer Science, Vol. 826. Berlin: Springer, S. 55-72, 1994

[CW85] L. Cardelli, P. Wegner: *On understanding types, data abstraction, and polymorphism.* – ACM Computing Surveys, 17 (4), S. 471-522, 1985

[CW90] Stefano Ceri, Jennifer Widom: *Deriving Production Rules for Constraint Maintenance*. – Proceedings of the 16th International Conference on Very Large Data Bases, S. 566-577, 1990

[CW91] Stefano Ceri, Jennifer Widom: *Deriving Production Rules for Incremental View Maintenance*. – Proceedings of the 17th International Conference on Very Large Data Bases, S. 577-589, 1991

[Das93] S. K. Das: *Deductive Databases and Logic Programming*. – Reading (Mass.): Addison-Wesley, 1993

[Dat81] C. J. Date: *Referential Integrity.* – Proceedings of the 7th International Conference on Very Large Data Bases, 1981

[Dat83] C. J. Date: *An Introduction to Database Systems, Volume II.* – Reading (Mass.): Addison Wesley Systems Programming Series, 1983

[Dat90] C. J. Date: *Referential Integrity and Foreign Keys, Part 1 & Part 2.* – In: Relational Database Writings 1985-1989. Addison-Wesley, 1990

[Dat95] C. J: Date: *An Introduction to Database Systems, Volume I, 6th Edition.* – Addison-Wesley, 1995

[DB82] U. Dayal, P. A. Bernstein: *On the Correct Translation of Update Operations on Relational Views.* – ACM Transactions on Database Systems, Vol. 7, No. 3, 381-416, 1982

[DBM88] Umeshwar Dayal, Alejandro P. Buchmann, Dennis R. McCarthy: *Rules are Objects too: A Knowledge Model for an Active, Object-OrientedDatabase System*. – Proc. Adv. in OODBS, Bad Munster 1988, S. 129-143, 1988

[DD93] C.J. Date, Hugh Darwen: *A Guide to the SQL Standard.* – Reading (Mass.): Addison Wesley, 1993

[DD95] Hugh Darwen, C. J. Date: *The Third Manifesto.* – SIGMOD RECORD, Vol. 24, No. 1, S. 39-49, 1995

[DDB91] Klaus R. Dittrich, Umeshwar Dayal, Alejandro P. Buchman (Hrsg.): *On Object-Oriented Database Systems.* – Springer Series Topics in Information Systems, Berlin: Springer, 1991

[Deu91] O. Deux: *The O_2 System*. – Communications of the ACM, Vol. 34, No. 10, S. 34-48, 1991

[DG90] O. Díaz, P. M. D. Gray: *Semantic-rich User-defined Relationships as a Main Constructor in Object Oriented Database*. – Proceedings of the IFIP TC2/WG

2.6 Working Conference on Object-Oriented Databases: Analysis, Design & Construction (DS-4), Windermere, UK, S. 207-224, 1990

[Dit91] K. R. Dittrich: *Object-Oriented Database Systems: The Notions and the Issues.* – In: [DDB91], S. 3-10, 1991

[Dit94] K. R. Dittrich: *Object-Oriented Data Model Concepts.* – in [DÖB+94], S. 29-46, 1994

[DKA+86] Peter Dadam, Klaus Küspert, Flemming Andersen, H. Blanken, R. Erbe, J. Günauer, V. Lum, P. Pistor, G. Walch: *A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies*. – Proc. of the ACM SIGMOD International Conference on Management of Data, S. 356-367, 1986

[DKS+88] Peter Dadam, Klaus Küspert, N. Südkamp, R. Erbe, V. Linnemann, P. Pistor, G. Walch: *Managing Complex Objects in R^2D^2*. – IBM Technical Report TR 88.03.004, IBM Heidelberg, 1988

[DM94a] C. J. Date, David McGoveran: *Updating Union, Intersection, and Difference Views.* – Database Programming & Design 7 (6), S. 46-53, 1994

[DM94b] C. J. Date, David McGoveran: *Updating Joins and Other Views.* – Database Programming & Design 7 (8), S. 43-49, 1994

[DN66] O.-J. Dahl, K. Nygaard: *Simula - an ALGOL-based simulation language.* – Communications of the ACM, Vol. 9, S. 671-678, 1966

[DÖB+94] Asuman Dogac, M. Tamer Özsu, Alexandros Biliris, Timos Sellis (Hrsg.): *Advances in Object-Oriented Database Systems.* – NATO ASI Series, Series F: Computer and Systems Sciences, Vol. 130. Berlin: Springer, 1994

[DV92] Scott Danforth, Patrick Valduriez: *A FAD for Data Intensive Applications*. – IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 1, S. 34-51, 1992

[EGH+92] Gregor Engels, Martin Gogolla, Uwe Hohenstein, Klaus Hülsmann, Perdita Löhr-Richter, Gunter Saake, Hans-Dieter Ehrich: *Conceptual modelling of database applications using an extended ER model*. – Data & Knowledge Engineering, Vol. 9, 1992/93, S. 157-204, 1992

[EGL89] H.-D. Ehrich, M. Gogolla, U.W. Lipeck: *Algebraische Spezifikation Abstrakter Datentypen - Eine Einführung in die Theorie*. Leitfäden und Monographien der Informatik. – Teubner, Stuttgart, 1989

[EKW92] D. W. Embley, B. D. Kurtz, S. N. Woodfield: *Object-Oriented System Analysis: A Model-Driven Approach.* – Englewood Cliffs (NJ): Prentice-Hall, 1992

[EN94] Ramez Elmasri, Shamkant B. Navathe: *Fundamentals of Database Systems.* – 2nd Edition, Redwood City, California: Benjamin/Cummimgs, 1994

[Esw76] K. P. Eswaran: *Specification, Implementations, and Interactions of a Trigger Subsystem in an Integrated Database System*. – IBM Research Report RJ 1820, IBM Research Laboratory, San José (CA), 1976

[ESW88] R. Erbe, Norbert Südkamp, G. Walch: *Advanced Information Management Prototype. Application Program Interface User Manual*. – IBM Technical Report TN 88.03, IBM Wiss. Zentrum Heidelberg , 1988

[EWH85] R. A. Elmasri, J. Weeldreyer, A. Hevner: *The category concept: An extension to the Entity-Relationship Model*. – Data & Knowledge Engineering, Vol. 1, S. 75-116, 1985

[FAC+89] D. H. Fishman, J. Annevelink, E. Chow, T. Connors, J. W. Davis, W. Hasan, C. G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. Risch, M. C. Shan, W. K. Wilkinson: *Overview of the Iris DBMS*. – in [KL89], S. 219-250 (auch: Hewlett Packard Technical Report HPL-SAL-89-15), 1989

[FC85] A. L. Furtado, M. A. Casanova: *Updating Relational Views*. – in: W. Kim, D. S. Reiner, D. S. Batory: Query Processing in Database Systems, Berlin: Springer, S. 127-142, 1985

[FPT93] P. Fraternali, S. Paraboschi, L. Tanca: *Automatic Rule Generation for Constraint Enforcement in Active Databases*. – in: U. Lipeck, B. Thalheim (Hrsg.): Modelling Database Dynamics, Selected Papers from the Fourth International Workshop on Foundations of Models and Languages for Data and Objects, Volkse, Oktober 1992. Workshops in Computing, Berlin: Springer, S. 153-173, 1993

[FV85] P. C. Fischer, D. van Gucht: *Determining when a structure is a nested relation*. – Proccedings of the 11th International Conference on Very Large Data Bases, S. 171-180, 1985

[FV95] Christian Fahrner, Gottfried Vossen: *A survey of database design transformations based on the Entity-Relationship model*. – Data & Knowledge Engineering 15, S. 213-250

[GD95] Andreas Geppert, Klaus R. Dittrich: *Specification and Implementation of Consistency Constraints in Object-Oriented Database Systems: Applying Programming-by-Contract*. – in: G. Lausen (Hrsg.): Datenbanksysteme in Büro, Technik und Wissenschaft, GI-Fachtagung, Dresden. Berlin: Springer, S. 322-337, 1995

[GGD91] Stella Gatziu, Andreas Geppert, Klaus R. Dittrich: *Integrating Active Concepts into an Object-Oriented Database System*. – in: [KS91], S. 399-415, 1991

[GGF93] N. Gal-Oz, E. Gudes, E. B. Fernandez: *A Model of Methods Access Authorization in Object-Oriented Databases*. – Proceedings of the 19th International Conference on Very Large Databases, S. 52-61, 1993

[GH91] Martin Gogolla, Uwe Hohenstein: *Towards a Semantic View of an Extended Entity-Relationship Model*. – ACM Transactions on Database Systems, Vol. 16, No. 3, S. 369-416, 1991

[GJ91] N. Gehani, H. V. Jagadish: *Ode as an Active Database: Constraints and Triggers*. – Proceedings of the 17th International Conference on Very Large Data Bases, S. 327-336, 1991

[GJS92] N. H. Gehani, H. V. Jagadish, O. Shmueli: *Composite Event Specification in Active Databases: Model & Implementation*. – Proceedings of the 18th International Conference on Very Large Data Bases, S. 327-338, 1992

[GKP92] Peter M.D. Gray, Krishnarao G. Kulkarni, Norman W. Paton: *Object-Oriented Databases. A Semantic Data Model Approach.* – London: Prentice-Hall, 1992

[Gog94] Martin Gogolla: *An Extended Entity-Relationship Model - Fundamentals and Pragmatics*. Lecture Notes in Computer Science 767. – Springer, Berlin/Heidelberg/New York, 1994

[GOO95] The GOODSTEP Team: *The GOODSTEP Project. Final Report.* – GOODSTEP ESPRIT-III project 6115, Technical Report, 1995

[GPB+94] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, Dirk van Gucht: *A Graph-Oriented Object Database Model*. – IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No. 4, S. 572-586, 1994

[GR83] A. Goldber, D. Robson: *Smalltalk-80: The Language and its Implementation.* – Reading (MA): Addison-Wesley, 1983

[GR93] Jim Gray, Andreas Reuter: *Transaction Processing: Concepts and Techniques.* – San Mateo, CA: Morgan Kaufmann, 1993

[Gra84] Peter M. D. Gray: *Logic, Algebra and Databases.* – Chichester: Ellis Horwood Ltd., 1984

[GS82] A. Guttmann, M. Stonebraker: *Using a Relational Database Management System for Computer Aided Design.* – IEEE Data Engineering, Vol. 5, No. 2, S. 21-28, 1982

[Gut77] J. Guttag: *Abstract Data Types and the Development of Data Structures.* – Communications of the ACM, Vol. 20, No. 6, S. 396-404, 1977

[Gut84] Antonin Guttman: R-Trees: *A dynamic index structure for spatial searching*. – Proc. of the ACM SIGMOD International Conference on Management of Data, S. 47-57, 1984

[Hal95] T. A. Halpin: *Conceptual Schema and Relational Database Design.* – Sydney: Prentice-Hall, 1995

[Han95] Magdy S. Hanna: *A Close Look at the IFO Data Model.* – SIGMOD RECORD, Vol. 24, No. 1, S. 21-26, 1995

[Hel93] Dan Heller: *Motif programming manual for OSF/Motif version 1.1*. – Sebastopol (CA) : O'Reilly, 1993

[Her93] Hector J. Hernandez: *Extended nested relations*. – Acta Informatica, Vol. 30, S. 741-771, 1993

[Heu89] Andreas Heuer: *A Data Model for Complex Objects Based on a Semantic Database Model and Nested Relations.* – In: [AFS89], S. 297-312, 1987

[Heu92] Andreas Heuer: *Objektorientierte Datenbanken*. – Bonn: Addison-Wesley, 1992

[HK87] Richard Hull, Roger King: *Semantic Database Modeling: Survey, Applications, and Research Issues*. – ACM Computing Surveys, Vol. 19, No.3, S. 201-260, 1987

[HK95] Uwe Hohenstein, Christian Körner: *Semantische Anreicherung relationaler Datenbanken.* – In: G. Lausen (Hrsg.): Datenbanksysteme in Büro, Technik und Wissenschaft, GI-Fachtagung, Dresden. Berlin: Springer, S. 130-149, 1995

[HKU95] M. Hein, C. Kersten, G. Unbescheid: *Oracle7.* – Bonn: Addison-Wesley, 1995

[HRS91] Tina M. Harvey, Mark A. Roth, Craig W. Schnepf: *The Design of the Triton Nested Relational Database System*. – SIGMOD RECORD, Vol. 20, No.3, S. 62-72, 1991

[HS95] Andreas Heuer, Gunter Saake: *Datenbanken. Konzepte und Sprachen.* – Bonn, Albany (u.a.): International Thomson Publ., 1995

[HSJ+94] Thorsten Hartmann, Gunter Saake, Ralf Jungclaus, Peter Hartel, Jan Kusch: *Revised Version of the Modelling Language TROLL (TROLL Version 2.0).* – Informatik-Berichte 94-03, Techn. Universität Braunschweig, 1993

[HZ90] Sandra Heiler, Stanley Zdonik: *Object Views: Extending the Vision.* – Proceedings of the 6th International Conference on Data Engineering, S. 86-93, 1990

[IBM75] IBM: *Information Management System Virtual Store.* – IMS/VS Reference Manual, 1975

[Ill96a] Illustra: *Hersteller-Information.* – Illustra Information Technology Inc. , 1111 Broadway, Oakland, CA 94607, oder auch http://www.illustra.com

[Ill96b] Illustra Information Technology Inc.: *Illustra's Web DataBlade Module.* – SIGMOD Record, Vol. 25, No. 1, S. 105-112, 1996

[Ing91] Ingres Corporation: *INGRES Object Management Extension User's Guide.* – Handbuch für Ingres Release 6.4, Ingres Corporation, Alameda (CA), 1991

[ISO92a] International Organization for Standardization: *Database Language SQL.* – Document ISO/IEC 9075:1992

[ISO92b] International Organization for Standardization: *(ISO Working Draft) Database Language SQL3.* – Document ISO/IEC JTC1/SC21/WG3 DBL OTT-003 (May 1992)

[JK84] Matthias Jarke, Jürgen Koch: *Query Optimization.* – ACM Computing Surveys, Vol. 16, No. 2, S. 111-152, 1984

[JQ92] H. V. Jagadish, Xiaolei Qian: *Integrity Maintenance in an Object-Oriented Database*. – Proceedings of the 18th International Conference on Very Large Data Bases, S. 469-480, 1992

[JS82] G. Jaeschke, Hans-Jörg Schek: *Remarks on the algebra of non first normal form relations.* – Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, S. 124-138, 1982

[KA90] Setrag Khoshafian, Razmik Abnous: *Object Orientation. Concepts, Languages, Databases, User Interfaces.* – John Wiley & Sons, New York, 1990

[KC86] Setrag N. Koshafian, George P. Copeland:
Object Identity. – OOPSLA ‘86 Proceedings, S. 406-416, 1986

[KD91] Ullrich Keßler, Peter Dadam: *Auswertung komplexer Anfragen an hierarchisch strukturierte Objekte mittels Pfadindexen*. – Proc. GI-Fachtagung „Datenbanksysteme für Büro, Technik und Wissenschaft“, Kaiserslautern, S. 218-237, 1991

[KD95] Christian Kalus, Peter Dadam: *Flexible Relations - Operational Support of Variant Relational Structures.* – Proceedings of the 21st International Conference on Very Large Data Bases, 1995

[Kel85] Arthur M. Keller: *Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins.* – Proceedings of the 4th Symposium on Principles of Database Systems, S. 154-163, 1985

[Kel86] Arthur M. Keller: *Choosing a View Update Translator by Dialog at View Definition Time.* – Proceedings of the 12th International Conference on Very Large Data Bases, S. 467-474, 1986

[Kim93] Won Kim: *Object-Oriented Database Systems: Promises, Reality, and Future*. – Proc. of the 19th International Conference on Very Large Data Bases, S. 676-687, 1993

[Kin89] Roger King: *My Cat is Object-Oriented.* – in [KL89], S. 23-30

[KK89] Hiroyuki Kitagawa, Tosiyasu L. Kunii: *The Unnormalized Relational Data Model For Office Form Processor Design.* – Tokyo: Springer, 1989

[KK93] Peter Kandzia, Hans-Joachim Klein: *Theoretische Grundlagen relationaler Datenbanksysteme.* – Mannheim: BI Wissenschaftsverlag, 1993

[KM94] Alfons Kemper, Guido Moerkotte: *Object-Oriented Database Management. Applications in Engineering and Computer Science.* – London: Prentice-Hall, 1994

[KL89] Won Kim, Frederick H. Lochovsky (Hrsg.): *Object-Oriented Concepts, Databases, and Applications.* – New York: ACM Press, 1989

[KS91] Paris Kanellakis, Joachim W. Schmidt (Hrsg.): Proceedings of the 3rd International Workshop on Database Programming Languages: Bulk Types & Persistent Data. San Mateo: Morgan Kaufmann, 1991

[KSW89] Klaus Küspert, Gunter Saake, Lutz M. Wegner: *Duplicate Detection and Deletion in the Extended NF^2 Data Model*. – Proceedings of the 3rd International Conference on Foundations of Data Organization and Algorithms. Lecture Notes in Computer Science, Vol. 367. Berlin: Springer, S. 83-100, 1989

[KTW90] Klaus Küspert, Jukka Teuhola, Lutz Wegner:
Design Issues and First Experience with a Visual Database Editor for the Extended NF^2 Data Model. – Proc. 23rd Int. Conf. System Sciences, Hawaii, Januar 1990, S. 308-317

[KW87] A. Kemper, M. Wallrath: *An analysis of geometric modeling in database systems.* – ACM Computing Surveys, Vol. 19, No. 1, S. 47-91, 1987

[Lar88] Per-Åke Larson: *The Data Model and Query Language of LauRel.* – IEEE Bulletin on Data Engineering, Vol. 11, S. 23-30, 1988

[Lev92] Mark Levene: *The Nested Universal Relation Database Model.* – Lecture Notes in Computer Science, Vol. 595. Berlin: Springer, 1992

[LEW93] Stephen W. Liddle, David W. Embley, Scott N. Woodfield: *Cardinality constraints in semantic data models*. – Data & Knowledge Engineering, Vol. 11, S. 235-270, 1993

[Lin85] T.-W. Ling: *A Normal Form for Entity-Relationship diagrams.* – Proceedings of the 4th Intenational Conference on the Entity-Relationship Approach, S. 24-35, 1985

[Lip89] Udo W. Lipeck: *Dynamische Integrität von Datenbanken.* – Informatik Fachberichte, Band 209, Berlin: Springer, 1989

[LKD+91] Volker Linnemann, Klaus Küspert, P. Dadam, R. Erbe, Peter Pistor, Norbert Südkamp: *User Defined Data Types and Functions in an Extensible Database Management System Supporting Complex Objects*. – IBM Technical Report TR 75.91.10, IBM Wiss. Zentrum Heidelberg, 1991

[LL95] Marc Levene, George Loizou: *A Graph-Based Data Model and its Ramifications.* – IEEE Transactions on Knowledge and Data Engineering, Vol. 7, No. 5, S. 809-823, 1995

[LLO+91] C. Lamb, G. Landis, J. Orenstein, D. Weinreb: *The ObjectStore Database System.* – Communications of the ACM, Vol. 34, No. 10, S. 50-63, 1991

[LLP+91] Guy M. Lohman, Bruce G. Lindsay, Hamid Pirahesh, K. Bernhard Schiefer: *Extensions to Starburst: Objects, Types, Functions, and Rule*s. – Communications of the ACM, Vol. 34, No. 10, S. 94-109, 1991

[Loc+85] P.C. Lockemann et. al.: *Anforderungen technischer Anwendungen an Datenbanksysteme.* – Proceedings GI-Fachtagung Datenbanken für Büro, Technik und Wissenschaft. Informatik-Fachberichte 94. Berlin: Springer, S. 1-26, 1985

[Loo94] Mary E. S. Loomis: *Making objects persistent.* – Journal on Object-Oriented Programming, Vol. 6, No. 6, S. 25-28, 1994

[LPS91] Volker Linnemann, Norbert Südkamp, Peter Pistor: *User Manual of the AIM-P Online Interface*. – IBM Wiss. Zentrum Heidelberg, S. , 1991

[LRV88] Christophe Lécluse, Philippe Richard, Fernando Vélez: *O_2, an Object-Oriented Data Model.* – Proceedings of the 7th ACM SIGACT-SIGMOD Conference on Management of Data, S. 424-433, 1988

[LS92] Christian Laasch, Marc H. Scholl: *Generic Update Operations Keeping Object-Oriented Databases Consistent*. – Proc. of the 2nd GI-Workshop on Information Systems and Artificial Intelligence, Ulm. Informatik-Fachberichte 303, Berlin: Springer, 1992

[LZ92] P. Loucopolous, R.Zicari (Hrsg.): *Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Developement*. – John Wiley, 1992

[Mai83] D. Maier: *The Theory of Relational Databases.* – Rockville (MD): Computer Science Press, 1983

[Mai89] D. Maier: *Making database systems fast enough for CAD applications.* – In: [KL89], S. 573-582, 1989

[Mak77] Akifumi Makinouchi: *A Consideration on Normal Form of Not-Necessarily-Normalized Relation in theRelational Data Model*. – Proceedings of the 3rd International Conference on Very Large Databases, S. 447-453, 1977

[Mar90] Viktor M. Markowitz: *Referential Integrity Revisited: An Object-Oriented Perspective*. – Proc. of the 16th Very Large Data Bases Conference, S. 578-589, 1990

[MBW80] J. Mylopoulos, P. A. Bernstein, H. K. T. Wong: *A language facility for designing database-intensive applications.* – ACM Transactions on Database Systems, Vol. 5, No. 2, S. 185-207, 1980

[McG94] David McGoveran: *Nothing from Nothing.* – Database Programming & Design, in 4 parts: 6 (12), S. 32-41, 1993 (part 1); 7 (1), S. 54-61, 1994 (part 2); 7 (2), S. 42-48 (part 3); 7 (3), S. 54-63 (part 4)

[MD89] D. McCarthy, U. Dayal: *The Architecture of an Active Data Base Management System.* – Proceedings of the ACM SIGMOD International Conference on Management of Data, S. 215-224, 1989

[Mit88] Bernhard Mitschang: *Ein Molekül-Atom-Datenmodell für Non-Standard-Anwendungen*. – Informatik-Fachberichte 185, Berlin: Springer, 1988

[MMR86] V. M. Markowitz, J. A. Makowsky, N. Rotics: *Entity-Relationship Consistency for Relational Schemas.* – Proceedings of the 1st International COnference on Database Theory. Berlin: Springer (LNCS 243), S. 306-322, 1986

[Mos85] J. E. B. Moss: *Nested Transactions: An Approach to Reliable Distributed Computing.* – Boston: MIT Press, 1985

[MP94] Bernhard Mitschang, Hamid Pirahesh: *Integration of Composite Objects into Relational Query Processing: The SQL/XNF Approach.* – in: J. C. Freytag, D. Maier, G. Vossen (Hrsg.): Query Processing for Advanced Database Systems. San Mateo: Morgan Kaufmann, S. 35-62, 1994

[MPP+93] Bernhard Mitschang, Hamid Pirahesh, Peter Pistor, Bruce Lindsay, Norbert Südkamp: *SQL/XNF - Processing Composite Objects as Abstractions over Relational Data*. – Proceedings of the 9th IEEE International Conference on Data Engineering, S. 272-282, 1993

[MR92] Heikki Mannila, Kari-Jouko Räihä: *The Design of Relational Databases.* – Wokingham: Addison-Wesley, 1992

[MS90] D. Maier, J. Stein: *Development and Implementation of an Object-Oriented DBMS.* – In: S. B. Zdonik, D. Maier (Hrsg.): Readings in Object-Oriented Data-

base Systems. San Mateo (CA): Morgan Kaufmann, S. 167-185, 1990

[MS91] F. Matthes, J. W. Schmidt: *Bulk types: Built-in or add-on?* - in: [KS91], S. 33-54, 1991

[MS92] Victor M. Markowitz, Arie Shoshani: *Representing Extended Entity-Relationship Structures in Relational Databases: A Modular Approach.* – ACM Transactions on Database Systems, Vol. 17, No. 3, S. 423-464, 1992

[MUV84] D. Maier, J. D. Ullman, M. Y. Vardi: *On the foundations of the universal relation model.* – ACM Transactions on Database Systems 8 (1), S. 1-14, 1984

[Nav92] Shamkant B. Navathe: *Evolution of Data Modeling for Databases*. – Communications of the ACM, Vol. 35, No. 9, S. 112-123, 1992

[Ngu89] Anne H. H. Ngu: *Conceptual Transaction Modeling.* – IEEE Transactions on Knowledge and Data Engineering, Vol. 1, No. 4, S. 508-518, 1989

[NH89] G. M. Nijssen, T. A. Halpin: *Conceptual Schema and Relational Database Design, a Fact Oriented Approach.* – Sydney: Prentice-Hall, 1989

[NLL+87] B. Nixon, C. Lawrence, D. Lauzon, A. Borgida, J. Mylopoulos, M. Stanley: *Implementation of a compiler for a semantic data model: experience with TAXIS.* – Proceedings of the ACM SIGMOD International Conference on Management of Data, S. 118-131, 1987

[NPS91] Mauro Negri, Giuseppe Pelagatti, Licia Sbattella: *Formal Semantics of SQL Queries*. – ACM Transactions on Database Systems, Vol. 17, No. 3, S. 513-534, 1991

[ÖPS+95] M. Tamer Özsu, Randal J. Peters, Duane Szafron, Boman Irani, Anna Lipka, Adriana Muñoz: *TIGUKAT: A Uniform Behavioral Objectbase Management System.* – VLDB Journal, Vol. 4, No. 3, S. 445-492, 1995

[Özs94] M. Tamer Özsu: *Transaction Models and Transaction Management in OBject-Oriented Database Management Systems.* – in: [DÖB+94], S. 147-168, 1994

[Osb79] S. L. Osborn: *Towards a universal relation interface.* – Proceedings of the 5th International Conference on Very Large Data Bases, S. 52-60, 1979

[PA86] Peter Pistor, Fleming Andersen: *Designing a Generalized NF^2-Model with an SQL-type Language Interface.* – Proceedings of the 12th International Conference on Very Large Databases, S. 278-285, 1986

[Pau94] Manfred Paul: *Typerweiterungen im eNF^2-Datenmodell.* – Universität Gesamthochschule Kassel, Dissertation, 1994 (zugl.: Aachen: Shaker, 1995)

[PD89] P. Pistor, P. Dadam: *The Advanced Information Management Prototype*. – In: [AFS89], S. 3-26, 1989

[PDG+89] Jan Paredaens, Paul De Bra, Marc Gyssens, Dirk Van Gucht: *The Structure of the Relational Database Model.* – Berlin: Springer, 1989

[Per90] Barbara Pernici: *Object with Roles*. – Proc. of the IEEE/ACM Conference on Office Information Systems, S. 205-215, 1990

[Pis89] Peter Pistor: *Variante Strukturen in HDBL*. – TU Braunschweig, Bericht Nr. 89-02, 1989

[Pis93] Peter Pistor: *Objektorientierung in SQL3: Stand und Entwicklungstendenzen*. – Informatik-Spektrum, Vol. 16, S. 89-94, 1993

[PM88] Joan Peckham, Fred Maryanski: *Semantic Data Models*. – ACM Computing Surveys, Vol. 20, No. 3, S. 153-189, 1988

[Poe96] POET Software Corp.: *POET 3.0 Technical Overview - Programming with POET*. – Technical Report, POET Software Corp., 999 Baker Way, Suite 100, San Mateo, CA 95054, siehe auch http://www.poet.com/techover, 1996

[PT86] Peter Pistor, R. Traunmüller: *A Database Language for Sets, Lists and Tables*. – Information Systems, Vol. 11, No. 4, S. 323-336, 1986

[PT94] J. Paredaens, L. Tenenbaum: *Advances in Database Systems. Implementations and Applications.* – CISM Courses and Lectures No. 347, International Centre for Mechanical Sciences, Berlin: Springer, 1994

[PTC+93] Pascal Poncelet, M. Teisseire, R. Cicchetti, Lotfi Lakhal: *Towards a Formal Approach for Object Database Design*. – Proc. of the 19th International Conference on Very Large Data Bases, S. 278-289, 1993

[PTW94] Manfred Paul, Sven Thelemann, Lutz M. Wegner: *Darstellungsmethoden für visuelle NF^2-Datenbankschnittstellen*. – Proc. GI-Workshop „Benutzungsschnittstellen für Datenbanken“, in: Datenbankrundbrief 13, S. 55-57, Mai 1994

[PS89] Christine Parent, Stefano Spaccapietra: *Complex objects modeling: an entity-relationship approach*. – In: [AFS89], S. 272-296, 1989

[PS92] Christine Parent, Stefano Spaccapietra: *ERC+: an Object Based Entity-Relationship Approach*. – Technical Report, Departement d’Informatique, Ecole Polytechnique Federal, Lausanne, 1992 ; auch in [LZ92]

[Rei93] Joachim Reinert: *Referentielle Integrität*. – Informatik Forschung und Entwicklung, Band 8, S. 79-96, 1993

[RBK+91] F. Rabitti, E. Bertino, W. Kim, D. Woelk: *A Model of Authorization for Next-Generation Database Systems*. – ACM Transactions on Database Systems, Vol. 16, No. 1, S. 88-131, 1991

[RBP+94] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Objektorientiertes Modellieren und Entwerfen.* – München, Wien: Hanser, 1994

[RKB87] Mark A. Roth, Henry F. Korth, Don S. Batory: *SQL/NF: A Query Language for ¬ 1NF Relational Databases*. – Information Systems, Vol. 12, No. 1, S. 99-114, 1987

[RKS88] Mark A. Roth, Henry F. Korth, Abraham Silberschatz: *Extended Algebra and Calculus for Nested Relational Databases*. – ACM Transactions on Database Systems, Vol. 13, No. 4, S. 389-417, 1988

[RKS89] Mark A. Roth, Henry F. Korth, Abraham Silberschatz: *Null Values in Nested Relational Databases*. – Acta Informatica, Vol. 26, S. 615-642, 1989

[RS87] Lawrence A. Rowe, Michael R. Stonebraker: *The POSTGRES Data Model*. – Proceedings of the 13th International Conference on Very Large Data Bases, S. 83-96, 1987

[RS91] Joel E. Richardson, Peter Schwarz: *Aspects: Extending Objects to Support Multiple, Independent Roles*. – Proc. of the ACM SIGMOD International Conference on Management of Data, S. 298-307, 1991

[Rum87] James E. Rumbaugh: *Relations as Semantic Constructs in an Object-Oriented Language*. – Proc. of the ACM Conference on Object-Oriented Progamming Systems, Languages, and Applications (OOPSLA ‘87), S. 466-481, 1987

[Run92] Elke A. Rundensteiner: *MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases*. – Proceedings of the 18th International Conference on Very Large Data Bases, S. 187-198, 1992

[Saa91] Gunter Saake: *Descriptive specification of database object behaviour.* – Data & Knowledge Engineering, Vol. 6, S. 47-73, 1991

[SAB+89] Michel Scholl, Serge Abiteboul, Francois Bançilhon, Nicole Bidoit, S. Gamerman, D. Plateau, P. Richard, A. Verroust: *VERSO: A Database Machine Based On Nested Relations*. – In: [AFS89], S. 27-49

[SAD+93] M. Stonebraker, R. Agrawal, U. Dayal, E. Neuhold, A. Reuter: *DBMS Research at a Crossroads: The Vienna Update*.- Proceedings of the 19th International Conference on Very Large Databases, S. 688-692, 1993

[SAD94] C. Santos, S. Abiteboul, C. Delobel: *Virtual Schemas and Bases.* – Proceedings of the 4th International Conference on Extending Database Technology, LNCS 779, Berlin: Springer, S. 81-94, 1994

[San94] C. Santos: *Design and Implementation of an Object-Oriented View Mechanism.* – GOODSTEP ESPRIT-III project No. 6115, Technical Report No. 7, 1994

[SC88] F. N. Springsteel, P.-J. Chuang: *ERDDS: The intelligent E-R-based database design system.* – Proceedings of the 7th International Conference on the Entity-Relationship Approach, S. 349-368, 1989

[Sch77] J. W. Schmidt: *Some high level language constructs for data of type relation.* – ACM Transactions on Database Systems, Vol.2, No.3, S. 247-261, 1977

[Sch93] Bernhard Schiefer: *Eine Umgebung zur Unterstützung von Schemaänderungen und Sichten in objektorientierten Datenbanksystemen.* – Dissertation, Universität Karlsruhe, 1993

[Shi81] D. W. Shipman: *The Functional Data Model and the Data Language DAPLEX*. – ACM Transactions on Database Systems, Vol. 6, No. 1, S. 140-173, 1981

[Shn87] B. Shneidermann: *Designing the User-Interface: Strategies for Effective Human-Computer Interaction.* – Reading, Mass.: Addison-Wesley, 1987

[SHP88] Michael R. Stonebraker, Eric N. Hanson, Spyros Potamianos: *The POSTGRES Rule Manager.* – IEEE Transactions on Software Engineering, Vol. 14, No. 7, S. 897-907, 1988

[SJG+90] Michael R. Stonebraker, Anant Jhingran, Jeffrey Goh, Spyros Potamianos: *On Rules, Procedures, Caching and Views in Data Base Systems.* – Proc. of the ACM SIGMOD International Conference on Management of Data, S. 281-290, 1990

[SK91] Michael R. Stonebraker, Greg Kemnitz: *The Postgres Next-Generation Database Management System*. – Communications of the ACM, Vol. 34, No. 10, S. 78-92, 1991

[SLP+89] G. Saake, V. Linnemann, P. Pistor, L. Wegner: *Sorting, Grouping, and Duplicate Elimination in the Advanced Information Management Prototype.* – Proceedings of the 15th International Conference on Very Large Data Bases, S. 307-316, 1989

[SLR+92] Marc H. Scholl, Christian Laasch, Christian Rich, Hans-Jörg Schek, Markus Tresch: *The COCOON Object Model*. – Department of Computer Science, ETH Zürich, Technical Report No. 192, 1992

[SLT91] Marc H. Scholl, Christian Laasch, Markus Tresch: *Updatable Views in Object-Oriented Databases*. – Proc. of the Conference on Deductive and Object-Oriented Databases, S. , 1991

[SM92] J. W. Schmidt, F. Matthes: *The database programming language DBPL - Rationale and Report.* – Technical Report FIDE/92/46, FB Informatik, Universität Hamburg, 1992

[SM93] Ernest Szeto, Victor M. Markowitz: *ERDRAW 5.3, a Graphical Editor for Extended Entity-Relationship Schemas, Reference Manual.* – Technical Report LBL-PUB 3084, Lawrence Berkeley Laboratory, Berkeley (CA), 1993

[SP94] Stefano Spaccapietra, Christine Parent: *View integration: A Step Forward in Solving Structural Conflicts*. –IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No. 2, S. 258- 274, 1994

[SPS87] M. H. Scholl, H.-B. Paul, H.-J. Schek: *Supporting Flat Relations by a Nested Relational Kernel.* – Proceedings of the 13th International Conference on Very Large Data Bases, S. 137-146, 1987

[SPS+90] Hans-Jörg Schek, Heinz-Bernhard Paul, Marc H. Scholl, Gerhard Weikum: *The DASDBS Project: Objectives, Experiences, and Future Prospects*. – IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, S. 25-43, 1990

[SRL+90] The Committee for Advanced DBMS Function (Michael R. Stonebraker, Lawrence A. Rowe, Bruce G. Lindsay, James Gray, Michael Carey, Michael Brodie, Philip Bernstein, David Beech): *Third-Generation Data Base System Manifesto.* – ACM SIGMOD Record, Vol. 19, No. 3, S. 495-511, 1990

[SS77] J. M. Smith, D. C. P. Smith: *Database Abstractions: Aggregation and Generalization.* – ACM Transactions on Database Systems, Vol. 2, No. 2, S. 105-133, 1977

[SS83] Gunter Schlageter, Wolffried Stucky: *Datenbanksysteme: Konzepte und Modelle.* – 2. Aufl., Stuttgart: Teubner, 1983

[SS86] Hans-Jörg Schek, Marc H. Scholl: *The relational model with relation-valued attributes*. – Information Systems, Vol. 11, No. 2, S. 137-147, 1986

[SS89a] Hans-Jörg Schek, Marc H. Scholl: *The Two Roles of Nested Relations in the DASDBS Project*. – In: [AFS89], S. 50-68

[SS89b] John J. Shilling, Peter F. Sweeney: *Three Steps to Views: Extending the Object-Oriented Paradigma*. – ACM SIGPLAN Notices, Vol. 24, S. 353-361, 1989

[ST93] K.-D. Schewe, B. Thalheim: *Fundamental Concepts of Object-Oriented Databases.* – Acta Cybernetica, Vol. 11 (1-2), S. 49-83, 1993

[Sto75] Michael R. Stonebraker: *Implementation of Integrity Constraints and Views by Query Modification*. – Proc. of the ACM SIGMOD International Conference on Management of Data, S. 65-78, 1975

[Str91] Bjarne Stroustrup: *The C++ Programming Language, 2nd Edition.* – Reading (MA): Addison-Wesley, 1991

[SW93] C. Santos, E. Waller: *O_2Views User Manual.* – GOODSTEP ESPRIT-III project No. 6115, Technical Report No. 26, 1994

[SWK+76] M. Stonebraker, E. Wong, P. Kreps, G. Held: *The Design and Implementation of INGRES.* – ACM Transactions on Database Systems, Vol. 1, No. 3, S. 189-222, 1976

[SZ89] L. A. Stein, S. B. Zdonik: *Clovers: The dynamic behaviour of types and instances.* – Technical Report No. CS-89-42, Brown University, 1989

[TF76] R. W. Taylor, R.L. Frank: *CODASYL data-base management systems.* – ACM Computing Surveys, Vol. 8, S. 67-103, 1976

[TF86] S. J. Thomas, P. C. Fischer: *Nested relational structures.* – In: P. C. Kanellakis, F. Preparata (Hrsg.): Advances in Computing Research, Vol. 3 - Greenwich, CT: JAI Press, S. 269-307, 1986

[Tha90] Bernhard Thalheim: *Generalizing the Entity-Relationship-Model for Database Modelling*. – Journal on New Gener. Comput. Syst., Vol. 3, No. 3, S. 197-211, 1990

[Tha91] Bernhard Thalheim: *Konzepte des Datenbank-Entwurf*s. – In: [VW91b], S. 1-48

[The92] Sven Thelemann: *ESCHER: Ein Editor für ein erweitertes NF^2-Datenmodell: Konzepte der Benutzerinteraktion*. – Kurzfassung des 4. GI-Workshops ‘Grundlagen von Datenbanken’, Technical Report ECRC-92-13, ECRC, München, 1992

[The93] Sven Thelemann: *Ein Modell für komplexe Werte mit Objektidentität*. – In: Bernhard Thalheim (Hrsg.): Tagung des Arbeitskreises „Grundlagen von Informationssystemen“ der GI-Fachgruppe „Datenbanken“, Universität Rostock, FB Informatik, Bericht 3-93, S. 117-121, 1993 (the 468)

[The95] Sven Thelemann: *Assertion of Consistency Within a Complex Object Database Using a Relationship Construct*. – in: M. P. Papazoglou (Hrsg.): OOER’95: Object-Oriented and Entity-Relationship Modeling. Proceedings of the 14th International Conference, Gold Coast, Australia, LNCS Band 1021, S. 32-43, 1995

[The96] Sven Thelemann: *A Relationship Construct Supporting Consistency of Multiple Views in a Complex Object Database*. – eingereicht für: Data & Knowledge Engineering, 1996

[Tod76] S. J. P. Todd: *The Peterlee relational test vehicle - a system overview.* – IBM Systems Journal, Vol. 15, No. 4, S. 285-308

[TPC93] M. Teisseire, Pascal Poncelet, R. Cicchetti: *A Tool based on a Formal Approach for Object-Oriented Database Modeling and Design*. – Proceedings of the 6th International Workshop on Computer-Aided Software Engineering (CASE’93), 1993

[Tri91] Phil Trinder: Comprehensions, a Query Notation for DBPLs. – in: [KS91], S. 55-68, 1991

[TS92] Markus Tresch, Marc H. Scholl: *Meta Object Management and its Application to Database Evolution*. – Proceedings of the 11th International Conference Entity-Relationship Approach, S. 299-321, 1992

[TWB+89] Toby J. Teorey, Guangping Wei, Deborah L. Bolton, John A. Koenig: *ER Model Clustering as an Aid for User Communiction and Documentation in Database Design*. – Communications of the ACM, Vol. 32, No. 8, S. 975-987, 1989

[TYF86] T. J. Teorey, D. Yang, J. P. Fry: *A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model.* – ACM Computing Surveys, VOl. 18, No. 2, S. 197-222, 1986

[TYI88] Katsumi Tanaka, Masatoshi Yoshikawa, Kozo Ishihara: *Schema Virtualization in Object-Oriented Databases.* – Proc. of the 4th IEEE CS Conference on Data Engineering, S. 23-30, 1988

[Ull87] Jeffrey D. Ullman: *Database theory: past and future*. – Proceedings of the 6th ACM Symposium on Principles of Database Systems, S. 1-10, 1987

[Ull88] Jeffrey D. Ullman: *Database and Knowledge-Base Systems, Volume I.* – Rockville, Maryland: Computer Science Press, 1988

[Ull89] Jeffrey D. Ullman: *Database and Knowledge-Base Systems, Volume II.* – Rockville, Maryland: Computer Science Press, 1989

[Vos91] Gottfried Vossen: *Data models, database languages and database management systems*. International computer science series. – Addison-Wesley, 1991

[VW91a] Gottfried Vossen; Kurt-Ulrich Witt: *SUXESS: towards a sound unification of extensions of the relational data model*. – Data & Knowledge Engineering, Vol. 6, S. 75-92, 1991

[VW91b] Gottfried Vossen/K.-U. Witt (Hrsg.): *Entwicklungstendenzen bei Datenbank-Systemen*. – München, Wien: Oldenbourg, 1991

[WCL91] Jennifer Widom, Roberta Jo Cochrane, Bruce G. Lindsay: *Implementing Set-Oriented Production Rules as an Extension to Starburst*. – Proceedings of the 17th International Conference on Very Large Data Bases, S. 275-285, 1991

[Web81] Wolfgang Weber: *Ein Subsystem zur Aufrechterhaltung der semantischen Integrität in Datenbanken*. – Dissertation, Karlsruhe: Heizmann Verlag, 1981

[Web93] Reinhold Weber: *SQL-Standards in Vergangenheit, Gegenwart und Zukunft*. – Datenbankrundbrief, Ausg. 12, November 1993, S. 15-20, 1993

[Weg89] Lutz M. Wegner: *ESCHER - interactive, visual handling of complex objects in the extended NF^2 database model*. – Proceedings of the IFIP TC-2 Working Conference on Visual Database Systems, S. 277-297, 1989

[Weg90] Lutz M. Wegner: *A Portable Record Manager*. – Mathematische Schriften Kassel, Preprint No. 11/90, Vordruck-Reihe des Fachbereichs 17 der GhK, 1990

[Weg91a] Lutz M. Wegner: *Managing Persistence in ESCHER*. – Mathematische Schriften Kassel, Preprint No. 7/91, Vordruck-Reihe des Fachbereichs 17 der GhK, 1991

[Weg91b] Lutz M. Wegner: *Let the Fingers Do the Walking: Object Manipulation in an NF^2 Database Editor*. – Proc. of the Symposium on New Results and New Trends in Computer Science, Graz, Austria. Lecture Notes in Computer Science, Band 555, Berlin: Springer, S. , 1991

[WJ91] Roel Wieringa, Wiebren de Jonge: *The identification of objects and roles*. – Technical Report IR-267, Faculty of Mathematics and Computer Science,Vrije Universiteit, Amsterdam, 1991

[WJS94] Roel Wieringa, Wiebren de Jonge, Paul Spruit: *Roles and dynamic subclasses: a modal logic approach*. – Proceedings of the 8th European Conference on Object-Oriented Programming. Lecture Notes in Computer Science No. 821. Berlin: Springer, S. 32-59, 1994

[WPC92] Lutz M. Wegner, Manfred Paul, R. Colomb: *Variants and Recursive Types in the eNF^2 Data Model*. – University of Queensland, Department of Computer Science, Technical Report No. 233, 1992

[WTW+96] L. Wegner, S. Thelemann, S. Wilke, R. Lievaart: *QBE-like Queries and Multimedia Extensions in a Nested Relational DBMS*. – Proceedings of the 1st International Conference on Visual Information Systems, S. 437-446, 1996

[Zeh89] C. A. Zehnder: *Informationssysteme und Datenbanken*. – Stuttgart: Teubner, 1989