Schemaevolution in objektorientierten Datenbanksystemen auf der Basis von Versionierungskonzepten

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich Biologie und Informatik
der Johann Wolfgang Goethe — Universität
in Frankfurt am Main

von
Sven-Eric Lautemann
aus Karlsruhe

Frankfurt (2000)
DF1

vom Fachbereich Biologie und Informatik der Johann Wolfgang Goethe — Universität als Dissertation angenommen.

Dekan:	Prof. Dr. Karl-Dieter Entian
Gutachter:	Prof. Dott. Ing. Roberto Zicari
	Prof. Dr. Markku J. Sakkinen
Datum der Disputation:	23.6.2000

Kurzfassung

Diese Arbeit beschäftigt sich mit der Durchführung evolutionärer Schemaänderungen in Datenbanksystemen und berücksichtigt dabei insbesondere bereits in Benutzung befindliche Objekte und Applikationen.

Jedes Datenbanksystem basiert auf der Abbildung der betrachteten Diskurswelt auf die technischen Konzepte eines gegebenen Datenmodells. Das resultierende Datenbankschema stellt die Grundlage jeglichen Zugriffs auf die Informationen der Datenbank dar. Es muß bei unvorhersehbaren Änderungen der Diskurswelt, des zu modellierenden Ausschnittes oder des Detaillierungsgrades angepaßt werden. Die Entwicklung, die es dabei durchläuft, wird durch den Begriff der Schemaevolution subsumiert.

Grundlegend für diese Arbeit ist die Feststellung, daß man die Existenz eines eindeutigen und für alle Anwendungen gleichermaßen geeigneten Schemas nicht voraussetzen darf. Die Schemaevolution verläuft nicht in linearen Bahnen, sondern sie entwickelt sich in alternativen Modellbildungen verschiedener Perspektiven der Diskurswelt. Diese Alternativen besitzen zur gleichen Zeit Gültigkeit und werden jeweils von verschiedenen Anwendungsbereichen bevorzugt.

Die Motivation zur Auseinandersetzung mit der Thematik der Schemaevolution ergibt sich aus der Feststellung, daß gegenwärtige Modelle und Systeme hier erhebliche Defizite aufweisen. Trotz generell steigenden Änderungsbedarfes bieten sie keine ausreichende Unterstützung für Entwicklungsprozesse, die in inkrementellen und evolutionären Zyklen verlaufen.

Das Ziel dieser Arbeit gliedert sich in die folgenden vier, grundlegenden Aspekte:

- Die jederzeitige Durchführbarkeit beliebiger Schemaänderungen und die Sicherstellung der Konsistenz des Schemas und der Datenbank durch, nötigenfalls automatisch zu ergreifende, korrigierende Maßnahmen ist zu gewährleisten.
- Vorhandene Applikationen müssen ohne Anpassung ausführbar bleiben.
- Um Kooperation zu ermöglichen, müssen vorhandene Objekte der Datenbank von Applikationen verschiedener Schemazustände zugreifbar sein.
- Für den Einsatz mit großen Datenbanken ist eine effiziente Realisierung evolutionärer Schemaänderungen, die einen unterbrechungsfreien Betrieb ermöglicht, Voraussetzung.

Die Grundlage für die Erreichung dieser Ziele wird durch die Anwendung von bereits auf Objektebene erfolgreich eingesetzten Versionierungskonzepten auf das Datenbankschema gelegt. Der daraus resultierende Ansatz der Schemaversionierung wird im COAST-Projekt (**C**omplex **O**bject **A**nd **S**chema **T**ransformation) der Universität Frankfurt am Main entwickelt und realisiert. Dabei gehen wir in den folgenden Arbeitsschritten vor: Analyse der Aufgabenstellung und Verfeinerung der fundamentalen Ziele zu technischen Teilzielen, Sichten und Analysieren der Literatur, detaillierte Modellbildung auf Schema- und Objektebene, Realisierungsbetrachtungen durch eine prototypische Implementierung, Evaluierung.

Vorwort

Die vorliegende Dissertation ist während meiner Tätigkeit an der Professur für Datenbanken und Informationssysteme (DBIS) der Johann Wolfgang Goethe–Universität in Frankfurt am Main entstanden und wurde auszugsweise bereits auf verschiedenen Konferenzen publiziert [BGL97, FL96, Lau96a, Lau96b, Lau97b, Lau97a, LEW97, LADH99]. Ich möchte an dieser Stelle die Gelegenheit wahrnehmen, mich bei all denjenigen zu bedanken, die direkt oder indirekt zum Gelingen der Arbeit beitrugen.

Zuallererst möchte ich Herrn Prof. Dott. Ing. Roberto Zicari danken für seine Aufgeschlossenheit gegenüber meinen wissenschaftlichen Ideen und seine konstruktiv-kritische Förderung dieser Arbeit. An meiner fachlichen und persönlichen Entwicklung der letzten sechs Jahre hat er maßgeblichen Anteil. Mein weiterer Dank gilt Herrn Prof. Dr. Markku J. Sakkinen für die Übernahme des Koreferates und die daraus resultierende, sehr gute Zusammenarbeit, vor allem in der Endphase meiner Arbeit.

Allen heutigen und ehemaligen Mitarbeitern der Professur danke ich für ihre Diskussionsbereitschaft in Bezug auf meine Dissertation, aber auch in anderweitigen Fragen. In besonderer Weise gilt dies für Thomas Behrens, Philippe Brèche, Frank Buddrus, Fabrizio Ferrandina, Frau Dr. Ling Liu, Claus Peter Priese, Herrn Dr. Christoph Schommer und Peter Werner. Auch meinen Koautoren von anderen Universitäten und aus der Industrie gebührt mein Dank für die intensive und fruchtbare Zusammenarbeit.

Ich danke allen Studenten, die mich bei der Entwicklung und Erprobung des COAST-Modells unterstützten. Dies sind Sabbas Apostolidis, Alexander Doll, Patricia Eigner, Michael Großmann, Jan Haase, Detlef Herchen, Manfred Prien, Christian Wöhrle und Kay Wölfle. Für die Schaffung der technischen Voraussetzungen danke ich Felix Gaethgens, Rainer Konrad und Markus Michalek.

Nicht zu vergessen sind die zahlreichen Personen, die mich durch meine Schul- und Studienzeit hindurch begleitet und damit erst die Grundlage für meine selbständige Weiterentwicklung gelegt haben. In bester Erinnerung sind mir die Mathematikstunden mit Herrn Studienrat Wolfgang Bechlars geblieben, ebenso die Datenbankvorlesungen bei Herrn Prof. Dr. Peter C. Lockemann und die Besprechungen mit dem Betreuer meiner Diplomarbeit, Herrn Dr. Stefan M. Lang. Ihr Vertrauen in meine Fähigkeiten hat mich stets begleitet.

Meine Lebensgefährtin Claudia Knoblich hat mich stets motiviert und mich von vielen Aufgaben befreit. Dafür danke ich ihr ganz herzlich.

Meinen Eltern schließlich gebührt besonderer Dank für ihre ununterbrochene Unterstützung durch alle Phasen meiner Entwicklung hindurch.

Frankfurt, im Januar 2000 — Sven-Eric Lautemann

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Umfeld der Arbeit, Problemstellung und Motivation 1

1.2 Ziele und Vorgehensweise 3

1.3 Gliederung der Arbeit 5

1.4 Notationen 6

2 Grundlagen **7**

2.1 Standard-Konzepte objektorientierter Datenbanksysteme 7

2.1.1 Objekte 7

2.1.2 Klassen 8

2.1.3 Vererbung 8

2.1.4 Datenabstraktion 10

2.1.5 Strenge Typisierung 10

2.1.6 Nebenläufigkeit 11

2.1.7 Persistenz 11

2.1.8 Weitere nach dem Manifesto verpflichtende Eigenschaften eines OODBMS 12

2.1.8.1 Überschreiben, Überladen und spätes Binden 12

2.1.8.2 Vollständigkeit 13

2.1.8.3 Erweiterbarkeit 13

2.1.8.4 Hintergrundspeicher-Management 13

2.1.8.5 Ausfallsicherheit 14

2.1.8.6 Ad-hoc Anfragemöglichkeit 14

2.1.9 Weitere nach dem Manifesto optionale Eigenschaften eines OODBMS . . 14

2.1.9.1 Verteilung 14

2.1.9.2 Designtransaktionen 14

2.1.9.3 Versionen 14

2.1.10 Applikationsanbindung in Datenbanksystemen 14

2.2 Versionierungskonzepte 15

2.3 Standard-Konzepte der Graphentheorie 17
2.4 Zusammenfassung 20

3 Problemanalyse, Anforderungen und Lösungsmodell 21
3.1 Problemanalyse 21
3.1.1 Das GOODSTEP-Projekt 21
3.1.2 Ein durchgängiges Beispiel aus dem Bereich der Softwareentwicklung . . . 23
3.1.3 Motivation für die Notwendigkeit von Schemaänderungen 28
3.2 Technische Teilziele zur Beschreibung der Anforderungen 30
3.2.1 Flexibilität bei der Durchführung von Schemaänderungen 31
3.2.2 Berücksichtigung von Applikationen 39
3.2.3 Flexibilität bei der Propagation von Schemaänderungen auf die Objektebene 41
3.2.4 Effizienz und Handhabung der Mechanismen 44
3.3 Schemaversionierung als Lösungsmodell 46
3.3.1 Flexibilität bei der Durchführung von Schemaänderungen 47
3.3.2 Berücksichtigung von Applikationen 50
3.3.3 Flexibilität bei der Propagation von Schemaänderungen auf die Objektebene 51
3.3.4 Effizienz und Handhabung der Mechanismen 52
3.4 Zusammenfassung und Bewertung 55

4 Bestehende Konzepte zur Erreichung der technischen Teilziele 57
4.1 Schemaänderungen ohne persistente Objekte 57
4.2 Isolierte Datenbanken 58
4.2.1 Grundsätzliche Vorgehensweise 58
4.2.2 Bewertung der grundsätzlichen Vorgehensweise 58
4.2.3 Konkrete Vertreter der Vorgehensweise 60
4.3 Direkte Schemaevolution 60
4.3.1 Grundsätzliche Vorgehensweise 60
4.3.1.1 Angebotene Schemaänderungsprimitive 61
4.3.2 Bewertung der grundsätzlichen Vorgehensweise 62
4.3.3 Vergleich der Vorgehensweise mit der Schemaversionierung 63
4.3.4 Konkrete Vertreter der Vorgehensweise 66
4.3.4.1 Der Ansatz von Penney und Stein 66
4.3.4.2 Der Ansatz von Ferrandina und Zicari 67
4.3.4.3 Anpassung der Applikationen mit Demeter 69
4.3.4.4 Der externe Ansatz von Lerner und Habermann 71
4.4 Simulation von Schemaänderungen durch Sichten 73

4.4.1 Sichten in objektorientierten Datenbanksystemen 73
4.4.2 Grundsätzliche Vorgehensweise 73
4.4.3 Bewertung der grundsätzlichen Vorgehensweise 74
4.4.4 Vergleich der Vorgehensweise mit dem manuellen Ansatz 77
4.4.5 Vergleich der Vorgehensweise mit der direkten Schemaevolution 78
4.4.6 Vergleich der Vorgehensweise mit der Schemaversionierung 79
4.4.7 Konkrete Vertreter der Vorgehensweise 80
4.4.7.1 Der Ansatz von Tresch und Scholl 80
4.4.7.2 Der Ansatz von Ra und Rundensteiner 80
4.5 Einsatz von Versionen 82
4.5.1 Grundsätzliche Vorgehensweise 83
4.5.2 Bewertung der grundsätzlichen Vorgehensweise 84
4.5.3 Konkrete Vertreter der Vorgehensweise 85
4.5.3.1 Der Datenbankversionierungsansatz von Cellary und Jomier . . 85
4.5.3.2 Der Typversionierungsansatz von Skarra und Zdonik 86
4.5.3.3 Der Klassenversionierungsansatz von Monk und Sommerville . . 90
4.5.3.4 Der Schemaversionierungsansatz von Clamen 93
4.5.3.5 Der Schemaversionierungsansatz von Labib und Saunders 95
4.5.3.6 Der Ansatz von Odberg 97
4.5.3.7 Der Schemaversionierungsansatz von Kim und Chou 103
4.5.4 Integration von Schemata 111
4.6 Zusammenfassung und Bewertung 112

5 Schemaversionierung auf Schemaebene 115

5.1 Grundlegende Aspekte der Schemaversionierung 115
5.2 Formale Definition des COAST-Objektmodells für unversionierte Schemata . . . 117
5.2.1 Identifikatoren, Namen, atomare Wertebereiche, Werte und Objekte . . . 117
5.2.2 Typen, Wertebereiche von Typen und Typhierarchien 120
5.2.3 Klassen, Vererbung und (strukturelle und verhaltensmäßige) Schemata . . 123
5.2.4 Der Konsistenzbegriff für unversionierte Schemata 127
5.2.4.1 Schemainvarianten 128
5.2.4.2 Konsistenz unversionierter Schemata 136
5.2.5 Datenbanken 138
5.3 Erweiterung des COAST-Objektmodells auf versionierte Schemata 138
5.3.1 Schemaversionen 139
5.3.1.1 Spezifikation von Schemaversionen 140
5.3.1.2 Relative Spezifikation von Schemaversionen 141

5.3.2 Klassenversionen 145
5.3.2.1 Die Klassenableitungsbeziehung 146
5.3.3 Der Konsistenzbegriff für versionierte Schemata 149
5.3.3.1 Schemainvarianten 149
5.3.3.2 Konsistenz versionierter Schemata 150
5.4 Aspekte des praktischen Umgangs mit der Schemaversionierung 151
5.4.1 Der Schemaableitungsprozeß 151
5.4.2 Reduzierung der Zahl notwendiger Schemaversionen bei kapazitätserweiternden Schemaänderungen 152
5.4.3 Applikationsanbindung an Schemaversionen 153
5.4.4 Auswahl von Schemaversionen für Applikationen 155
5.5 Die Schemabeschreibungssprache COAST-ODL 155
5.5.1 Primitive der COAST-ODL auf Schemaebene 157
5.5.2 Primitive der COAST-ODL auf Schemaversionsebene 158
5.5.2.1 Ableitung von Schemaversionen 158
5.5.2.2 Verändern von Schemaversionen 159
5.5.2.3 Umbenennen und Löschen von Schemaversionen 159
5.5.2.4 Einfrieren und Auftauen von Schemaversionen 159
5.5.3 Primitive der COAST-ODL auf Klassenebene 160
5.5.3.1 Integration von Klassen existierender Schemaversionen 160
5.5.3.2 Anlegen von Klassen 165
5.5.3.3 Verändern von Klassen 165
5.5.3.4 Umbenennen und Löschen von Klassen 165
5.5.4 Primitive der COAST-ODL auf Attributebene 166
5.5.4.1 Spezifikation und Veränderung der Oberklassen einer Klasse . . 167
5.5.4.2 Spezifikation und Veränderung der Attribute einer Klasse 170
5.5.4.3 Spezifikation und Veränderung der Methoden einer Klasse . . . 171
5.5.5 Einige Schemaänderungen in unserem Beispielszenario 172
5.5.6 Die Erzeugende ODL 175
5.5.7 Vergleich der allgemeinen Integrationsproblematik mit der Klassenintegration in COAST 176
5.6 Zusammenfassung und Bewertung 178

6 Schemaversionierung auf Objektebene 181
6.1 Objektzugriffsbereiche 182
6.2 Versionierte Objekte und Konvertierungsfunktionen 183
6.3 Die Propagationssteuerung 189

6.3.1 Analyse und erste Modellbildung 189
6.3.2 Propagation des Zustandes zum Ableitungszeitpunkt 192
6.3.2.1 Das Schnappschußflag (s-Flag) 192
6.3.3 Propagation des Zustandes nach dem Ableitungszeitpunkt 193
6.3.3.1 Das Erzeugungsflag (c-Flag) 194
6.3.3.2 Das Modifikationsflag (m-Flag) 195
6.3.3.3 Das Löschungsflag (d-Flag) 200
6.3.4 Einige Propagationsflags in unserem Beispielszenario 201
6.3.5 Kombinationsmöglichkeiten der Propagationsflags 201
6.4 Die Propagationssprache 202
6.5 Transitive Objektkonvertierung und -propagation 204
6.5.1 Bestimmung des Konvertierungspfades 206
6.5.2 Eindeutigkeit des Konvertierungspfades 209
6.5.3 Löcher in Konvertierungspfaden 214
6.6 Extrakonvertierungsfunktionen 218
6.6.1 Motivation 218
6.6.2 Formale Beschreibung von Extrakonvertierungsfunktionen 219
6.6.2.1 Eingeschränkte Spezifikation von Extrakonvertierungsfunktionen 220
6.6.2.2 Eingeschränkte Benutzung von Extrakonvertierungsfunktionen . 221
6.6.3 Bestimmung des Konvertierungspfades 223
6.6.4 Eindeutigkeit des Konvertierungspfades 225
6.7 Entwurfsunterstützung bei der Ableitung neuer Schemaversionen 226
6.8 Löschen von Schemaversionen und deren Zugriffsbereichen 227
6.9 Zusammenfassung und Bewertung 229

7 Prototypische Realisierung 233

7.1 Die Architektur des COAST-OODBMS Prototypen 233
7.1.1 Die Datenbankmaschine 235
7.1.1.1 Der Objektmanager 235
7.1.1.2 Der Schemamanager 236
7.1.1.3 Der Propagationsmanager 236
7.1.2 Weitere Komponenten 237
7.1.2.1 Der Schemaeditor 237
7.1.2.2 Der Schemaassistent und weitere Werkzeuge 237
7.1.2.3 Der ODL-Parser und der ODL-Generator 242
7.2 Die Implementierung des COAST-OODBMS Prototypen 244
7.2.1 Implementierung des Objektmanagers 244

7.2.1.1 Die Datenstrukturen für Objekte 244
7.2.1.2 Klassenextensionen 246
7.2.2 Implementierung des Schemamanagers 247
7.2.2.1 Das Metaschema 248
7.2.2.2 Alternativen bei der Speicherung des Schemas 248
7.2.3 Implementierung des Propagationsmanagers 249
7.2.3.1 Verzögerte physikalische Propagation 250
7.2.3.2 Transitive Propagation von Löschungen und Veränderungen . . 255
7.2.3.3 Die Objektpropagationsalgorithmen 258
7.2.4 Implementierung von ODL-Parser und ODL-Generator 268
7.3 Zusammenfassung und Bewertung 269

8 Validierung des COAST-Modells 271

8.1 Flexibilität bei der Durchführung von Schemaänderungen 272
8.2 Berücksichtigung von Applikationen 274
8.3 Flexibilität bei der Propagation von Schemaänderungen auf die Objektebene . . 274
8.4 Effizienz und Handhabung der Mechanismen 275
8.5 Zusammenfassung und Bewertung 277

9 Zusammenfassung und Ausblick 279

9.1 Erreichte Ziele 279
9.2 Übertragbarkeit der Ergebnisse 282
9.3 Ausblick 284

A Übersicht: Definitionen, Invarianten und Regeln 287

A.1 Definitionen 287
A.2 Invarianten 289
A.3 Regeln 289

B Die COAST-ODL 291

Literaturverzeichnis 295

Index 321

Kapitel 1

Einleitung

The affinities of all the beings of the same class
have sometimes been represented by a great tree.
(Darwin, 1859)

Dieses Kapitel führt in die Aufgabenstellung ein, die wir in dieser Abhandlung bearbeiten. Eine zentrale Rolle spielt die Änderung vorhandener Datenbankschemata beispielsweise zum Zwecke der Anpassung an eine veränderte Diskurswelt. Betrachtungsgegenstand der Untersuchungen sind insbesondere die Konsequenzen, die sich aus einer Schemaänderung für Objekte der Datenbank und darauf zugreifende Applikationen ergeben. Dies wird als die zentrale Problemstellung dieser Arbeit formuliert. Die Vorgehensweise zur Lösung des Problems sowie eine kapitelorientierte Gliederung der Arbeit und eine kurze Auflistung verwendeter Notationen schließen sich an.

1.1 Umfeld der Arbeit, Problemstellung und Motivation

Datenbanksysteme dienen der Verwaltung und Speicherung von Informationen ihres jeweiligen Einsatzgebietes. Der dabei relevante Ausschnitt der (realen oder einer virtuellen) Welt wird als *Miniwelt* oder auch als *Diskurswelt* (engl. *domain of discourse*) bezeichnet und muß für die Abbildung in ein technisches System scharf begrenzt und genau spezifiziert sein. Daher geht dem Einsatz eines Datenbanksystems stets eine Modellbildung voran, in der die relevanten Aspekte ausgewählt und auf die technischen Möglichkeiten des eingesetzten Systems abgebildet werden müssen.

Die Modellierung der Diskurswelt geschieht typischerweise in zwei Schritten. Zunächst wird im konzeptuellen Entwurf ein systemunabhängiges Modell (beispielsweise in Form eines ER-Diagramms) erstellt, das anschließend in ein logisches Modell umgesetzt wird. Für diesen zweiten Schritt der Modellbildung sind die von dem einzusetzenden Datenbanksystem angebotenen, technischen Konzepte, also das Datenmodell, von erheblicher Bedeutung. Daher wird bei der Kategorisierung von Datenbankmanagementsystemen zunächst die Zuordnung zu den Bereichen hierarchisches Datenmodell, Netzwerkdatenmodell, relationales Datenmodell oder objektorientiertes Datenmodell vorgenommen. Allen gemeinsam ist die Notwendigkeit der Festlegung eines

Modells der betrachteten Diskurswelt durch die jeweils angebotenen technischen Konzepte, bevor das Datenbanksystem eingesetzt werden kann.

Der Einsatzbereich von Datenbanksystemen hat sich in den letzten Jahren stark ausgedehnt. Wurden Datenbanken zunächst nur in den klassischen Anwendungsgebieten wie Bank- und Versicherungswesen, Flugbuchungssysteme, etc. eingesetzt, so finden sie heute verstärkt auch Verwendung in den als modern oder fortschrittlich bezeichneten Umfeldern zu denen u.a. die Anwenderunterstützung in verschiedenen Bereichen (z.B. CAD (engl. *computer aided design*), CAE (engl. *computer aided engineering*), CAM (engl. *computer aided manufacturing*), CBT (engl. *computer based training*), CSCW (engl. *computer supported cooperative work*)), die Telekommunikation und die Büroautomatisierung gehören. Während die klassischen Anwendungsgebiete in der Regel sehr große Mengen homogener und einfach strukturierter Daten zu verwalten haben, modellieren die neuen Datenbankapplikationen (engl. *advanced applications*) oft komplex strukturierte und hochdynamische Zustände und Prozesse und benötigen daher weitaus mächtigere und flexiblere Modellierungskonzepte zur Beschreibung ihrer Datenbestände und deren dynamischen Verhaltens.

Objektorientierte Datenbankmanagementsysteme (engl. *object oriented database management systems, OODBMS*) bieten die von den fortschrittlichen Anwendungsgebieten benötigten, leistungsfähigen Modellierungskonzepte an und eignen sich daher besonders gut für die oben genannten Anwendungsgebiete. Dabei haben sie erhebliche Vorteile gegenüber den bisher sehr viel häufiger eingesetzten, hierarchischen und relationalen Datenbanksystemen. Es hat sich sogar gezeigt, daß sich objektorientierte Konzepte oft auch in den üblicherweise als klassisch bezeichneten Anwendungsgebieten vorteilhaft einsetzen lassen.

Das Modell einer Diskurswelt beschreibt die Struktur — im Falle des objektorientierten Datenmodells auch das Verhalten — ihrer Datenobjekte und wird als *Datenbankschema* bezeichnet. Dieses Schema muß beim Datenbankentwurf erstellt und dem Datenbankmanagementsystem bekannt gemacht werden, bevor sich eine Datenbank erstellen und benutzen läßt. Aus diversen Gründen kommt es jedoch oft auch noch nach dem initialen Datenbankentwurf, also während des Betriebs, zu Änderungen, die nachträgliche Anpassungen des Schemas, sog. *Schemaänderungen*, notwendig machen:

- *Anpassung an unvorhersehbare Änderungen in der Diskurswelt:* Datenbanken bleiben mitunter Jahrzehnte erhalten und überdauern zahlreiche Generationen von Applikationen, die auf ihnen operieren. Während dieser Zeitspannen finden zahlreiche, unvorhersehbare Veränderungen der modellierten Diskurswelt statt, die in neuen oder angepaßten Applikationen bearbeitet werden sollen und die damit im Modell, d.h. im Datenbankschema entsprechende Anpassungen erforderlich machen.
- *Vielfältigkeit der Konstrukte objektorientierter Datenmodelle:* Die Objektorientierung bietet im Vergleich zu anderen Datenmodellen eine große Vielzahl an Modellierungskonzepten und nimmt damit sowohl dem Schemaentwickler als auch dem Applikationsentwickler viel Arbeit ab. Hierbei entsteht jedoch bei Änderungsbedarf entsprechend häufig die Notwendigkeit, die Verwendung von Modellierungskonstrukten anzupassen oder zu anderen Konstrukten wechseln zu müssen.
- *Komplexität typischer Einsatzgebiete:* Typische Vertreter der sog. fortschrittlichen Applikationen verarbeiten große und komplex strukturierte Objekte, die zu entwerfende Systeme oder Geräte modellieren. Die dafür benötigten Datenbankschemata sind damit selbst große und komplexe Objekte, bei deren Entwurf Fehler nicht ausgeschlossen werden können. Oft ist die Qualität eines Entwurfs in frühen Projektphasen schwer einschätzbar und wird erst durch Ausprobieren bewertet. Die verwendete Entwurfsmethodik hat also oft auch den Charakter des Modellierens verschiedener Alternativen mit anschließender Selektion

(sehr ähnlich dem von Charles Darwin [Dar1859] beschriebenen Selektionsprozeß, wie er in der Natur stattfindet). Im Bereich der Software-Entwicklung wird hier auch von inkrementellen und evolutionären Entwicklungsprozessen (engl. *incremental and evolutionary development*) [AFY98, Boo95] gesprochen.

- *Gleichzeitige Existenz verschiedener Perspektiven der Diskurswelt:* Datenbanksysteme werden typischerweise von zahlreichen Benutzern über verschiedene Applikationen genutzt. Daher ist es nicht vernünftig anzunehmen, daß eine einzige Schemaspezifikation alle Anforderungen sämtlicher Benutzer gleichermaßen gut erfüllen kann.

- *Ein- und Auschecken an Knoten föderierter Datenbanksysteme:* Vor dem Transport von Daten in einem heterogenen, föderierten Datenbanksystem muß sichergestellt werden, daß im Zielknoten ein Schema vorliegt, das für die Aufnahme der Daten geeignet ist. Ist diese Voraussetzung nicht erfüllt, so ist sie vor dem eigentlichen Datentransport durch Schemaänderungen auf Quell- oder Zielseite sicherzustellen.

- *Angleichung und Integration vormals isolierter Systeme:* Bei Zusammenschluß oder Übernahme von Firmen müssen deren unabhängig voneinander entstandene Informationssysteme integriert werden. Dies setzt eine Konsolidierungsphase voraus, in der verschiedene Modellierungen derselben oder zumindest überlappender Ausschnitte der Diskurswelt harmonisiert werden. Unterschiede der Modellierungen auf Schemaebene werden durch Schemaänderungen konsolidiert.

Eine Schemaänderung entspricht einer Entwicklung der Datenbankstruktur und der Prozeß der Veränderung wird als *Schemaevolution* bezeichnet. Wie wir gerade dargelegt haben, ist eine Unterstützung für die Schemaevolution durch das Datenbanksystem eine oftmals benötigte Eigenschaft. Dabei betrachten wir die Erzeugung und Änderung eines Schemas als komplexe Entwurfsaufgabe, vergleichbar dem Software-Engineering.

1.2 Ziele und Vorgehensweise

Die heutezutage verfügbaren Datenbanksysteme bieten keine oder nur rudimentäre Unterstützung evolutionärer Schemaänderungsprozesse. Dies verteuert notwendige Anpassungen des Datenmodells an veränderte Voraussetzungen und führt in der Praxis letztlich häufig dazu, daß Schemaänderungen aus Aufwands- und Kostengründen verspätet oder gar nicht durchgeführt werden. Ziel dieser Arbeit ist daher eine Umgebung zu konzeptionieren und zu erstellen, in der Schemaänderungen bedarfsgerecht vollziehbar sind, so daß das Schema evolutionäre Entwicklungen der Diskurswelt stets angemessen abbilden kann.

Im Vergleich zum Entwurf neuer Datenbankschemata, der in der Literatur enorme Beachtung findet, wurde die Durchführung von Änderungen an bestehenden Schemata relativ wenig untersucht. Dies liegt sicher zum einen daran, daß der initiale Entwurf stets den ersten Schritt beim Einsatz eines Datenbanksystems darstellt und demzufolge auf jeden Fall durchgeführt werden muß. Zum anderen mag der erhöhte Schwierigkeitsgrad bei der Änderung eines gegebenen Schemas sehr wohl ein weiterer Grund dafür sein. Im Gegensatz zum Entwurf neuer Schemata, wo ausschließlich die für den geplanten Einsatz gestellten Anforderungen berücksichtigt werden müssen, gilt es bei bereits in Benutzung befindlichen Systemen verschieden geartete Abhängigkeiten zu berücksichtigen und bereits vorhandene Datenbestände zu migrieren. Eine neue, vom bisherigen Entwurf unabhängige Modellierung entsprechend den bekannten Konzepten des Software-Engineering ist hier nicht mehr möglich, weil existierende Objekte und Applikationen

dabei nicht berücksichtigt werden können. Im Allgemeinen werden sog. *korrigierende Maßnahmen* erforderlich, die nach Änderungen erforderliche Anpassungen bei abhängigen Komponenten vornehmen, d.h.

- bei anderen Teilen des Schemas,
- bei bereits existierenden Objekten in der Datenbank und
- bei in der Entwicklung befindlichen oder bereits verwendeten Applikationen.

Die Beachtung dieser Abhängigkeiten stellt einen wesentlichen Teil der in der vorliegenden Abhandlung entwickelten Konzepte dar. Dabei konzentrieren wir uns auf das objektorientierte Datenmodell, da dieses mit Abstand die leistungsfähigsten Modellierungskonzepte bietet und daher für die hier behandelten Einsatzgebiete am besten geeignet ist.

Wir beschreiben in dieser Arbeit typische Anforderungen bei der Durchführung evolutionärer Schemaänderungsprozesse und entwickeln Konzepte zu deren Erfüllung. Dazu gehört insbesondere die Möglichkeit, Schemaänderungen auch dann noch durchzuführen, wenn eine Datenbank bereits benutzt wird, d.h. wenn sie bereits Objekte enthält und wenn bereits Applikationen auf ihr arbeiten. Die fundamentalen Ziele dieser Arbeit leiten sich damit wie folgt aus den Einschränkungen bisher existierender Modelle und Systeme ab:

- jederzeitige Durchführbarkeit beliebiger Schemaänderungen und Sicherstellung der Konsistenz durch korrigierende Maßnahmen
- fortlaufende Ausführbarkeit vorhandener Applikationen ohne Anpassung
- fortlaufende Zugreifbarkeit vorhandener Objekte der Datenbank mit der Möglichkeit zur Kooperation von Applikationen verschiedener Schemazustände
- effiziente Realisierung evolutionärer Schemaänderungen und unterbrechungsfreier Betrieb insbesondere auch bei großen Datenbanken.

Mit der Erreichung dieser Ziele wird ein Szenario ermöglicht, in dem für unterschiedliche Anwendungen zugeschnittene Ausprägungen eines Datenbankschemas bedarfsgerecht und unabhängig voneinander erstellt und benutzt werden können. Trotz dieser Entkopplung der Applikationen voneinander bzw. vom Schema bleibt die Möglichkeit zur Kooperation auf gemeinsamen Datenbeständen erhalten.

Entsprechend unserer Aufgabenstellung folgen wir einer zielgerichteten Vorgehensweise. Anhand einer Problemanalyse leiten wir eine Detaillierung der oben aufgeführten fundamentalen Ziele dieser Arbeit in Form technischer Teilziele ab, die dann zur Evaluierung in der Literatur beschriebener Ansätze herangezogen werden. Die zentrale Modellbildung findet anschließend in zwei Teilschritten statt. Zunächst konzentrieren wir unsere Betrachtungen auf das Schema selbst und berücksichtigen durch den Versionierungsansatz bereits elementare, durch die Existenz von Applikationen entstehende Anforderungen. Darauf aufbauend widmen wir uns dann der Instanzenebene, wo wir die ununterbrochene und fortwährende Zugreifbarkeit der Objekte sicherstellen. Daraufhin ist der Aufwand für die Realisierung und die Benutzung unserer Konzepte anhand einer prototypischen Realisierung abzuschätzen und eine Validierung unserer Ergebnisse entsprechend der zu Beginn detailliert beschriebenen Zielsetzung durchzuführen.

1.3 Gliederung der Arbeit

Aufgrund der im vorangegangenen Abschnitt skizzierten Vorgehensweise gliedert sich die vorliegende Arbeit wie folgt.

Kapitel 2 legt die in dieser Arbeit benötigten Grundlagen in den Bereichen objektorientierter Datenbanksysteme und Versionierung auf Objektebene.

Die Einzelheiten unserer Aufgabenstellung werden in Kapitel 3 erläutert. Die Problemanalyse basiert dabei auf zwei Säulen. Zum einen untersuchen wir spezielle, projektbezogene Anforderungen, die wir in Form eines durchgängigen Beispiels durch die gesamte Arbeit hindurch verfolgen. Zum anderen erläutern wir allgemeine Arten von Aufgaben, die die Durchführung von Schemaänderungen notwendig machen. Daran schließt sich eine detaillierte Analyse der Anforderungen an ein System zur Unterstützung von Schemaevolutionsprozessen an. Diese Anforderungen werden in Gruppen technischer Teilziele formuliert, welche weitestgehend unabhängig voneinander untersucht und zur Bewertung vorliegender Konzepte und Systeme herangezogen werden können. Den Abschluß des Kapitels bildet ein erster Grobentwurf eines Lösungsmodells als Rahmen für die nächsten Arbeitsschritte.

Kapitel 4 gibt eine Literaturübersicht und eine Kategorisierung möglicher Ansätze im Bereich der Schemaevolution. Dabei werden verschiedene konzeptionelle Vorgehensweisen identifiziert, jeweils bedeutende konkrete Vertreter davon vorgestellt und anhand der technischen Teilziele auf die Erfüllung der hier gestellten Anforderungen hin untersucht, bewertet und verglichen.

Die Entwicklung des Schemaversionierungsansatzes auf der Abstraktionsebene des Schemas und der damit verbundenen Begriffe der Schemaversion und der Klassenversion nimmt Kapitel 5 vor. Dabei wird das COAST-Objektmodell zunächst für unversionierte Schemata formal eingeführt und die Konsistenz eines Datenbankschemas durch Angabe einer Reihe einzuhaltender Schemainvarianten definiert. Daraufhin werden Objektmodell und Konsistenzbedingungen auf versionierte Schemata ausgedehnt. Nach dieser zumeist formalen Einführung gehen wir auf einige Aspekte im praktischen Umgang mit versionierten Schemata ein. Abschließend wird die Schemabeschreibungs- und -änderungssprache von COAST entwickelt und die Wirkungsweise der einzelnen Primitive insbesondere mit Blick auf ggf. auftretende, temporäre Inkonsistenzen, d.h. Verletzungen der Schemainvarianten hin analysiert.

Kapitel 6 untersucht die Auswirkungen der Schemaversionierung auf existierende Instanzen der Objektebene. Dabei werden den Schemaversionen zunächst eigene Objektzugriffsbereiche zugeordnet. Diese können sich jedoch überlappen und gestatten so die Kooperation von Applikationen verschiedener Schemaversionen. Für die Steuerung der Weitergabe von Informationen zwischen überlappenden Zugriffsbereichen werden sog. Propagationsflags eingeführt. Die Abbildung von Objekten zwischen verschiedenen Versionen ihrer Klassen geschieht durch Angabe von Konvertierungsfunktionen.

Kapitel 7 stellt die prototypische Implementierung des COAST-OODBMS vor, wobei zunächst auf die Architektur und die Funktionsweise allgemein eingegangen wird. Danach werden insbesondere die zentralen Module des Prototypen, nämlich Objektmanager, Schemamanager und Propagationsmanager detaillierter vorgestellt. Dabei gehen wir auch auf die Speicherung und Verwaltung versionierter Objekte (Objektmanager) und des Schemas (Schemamanager), sowie auf die zur Propagation der Objekte verwendeten, verzögert wirkenden Algorithmen (Propagationsmanager) ein.

Die zur Beurteilung der Tragfähigkeit der hier entwickelten Konzepte notwendige Validierung erfolgt in Kapitel 8.

Eine Zusammenfassung und Bewertung der in dieser Arbeit erreichten Ergebnisse gibt Kapitel 9. Darüberhinaus stellen wir einige Überlegungen an, welche Teilergebnisse sich als Lösungskonzepte in angrenzenden Problembereichen anwenden lassen und wie dies erfolgen kann. Die Betrachtungen schließen wir mit einem Ausblick auf zukünftige Arbeiten.

1.4 Notationen

Wo die Entsprechung eines hier gewählten deutschen Begriffes mit der in der Regel englischsprachigen Originalliteratur nicht direkt ersichtlich ist, wird der Originalbegriff aufgeführt, beispielsweise verwendet [BKKK87] den Begriff der *Klassenattribute* (engl. *shared-value variables*). Dabei kann es passieren, daß für verschiedene Begriffe der Originalliteratur derselbe deutschsprachige Begriff verwendet wird. *Versionierte Datenbanken* werden beispielsweise von Kim (engl. *versioned database*) anders bezeichnet als von Jomier (engl. *multiversion database*).

Neben *Definitionen*, die neue Sprechweisen, Symbole und Formalismen einführen, unterscheiden wir in dieser Arbeit Invarianten und Regeln. Als *Invarianten* bezeichnen wir wie in [BKKK87, TS92] allgemein Zustandsbedingungen, deren Einhaltung für die spezifikationsgemäße Funktionsweise vorgestellter Mechanismen und die Erhaltung der Konsistenz eine notwendige und hinreichende Voraussetzung darstellt.[1] Durch die Beachtung von *Regeln* bei Zustandsübergängen kann die Gültigkeit der Invarianten garantiert werden. Sämtliche Definitionen, Invarianten und Regeln sind in Anhang A aufgeführt.

Durch die Verwendung verschiedener Schriftsätze möchten wir die Bedeutung einzelner Textpassagen verdeutlichen. Im Einzelnen unterscheiden wir Textabschnitte durch die folgenden Darstellungen: fortlaufender Text, `benutzerdefinierte Namen` und `Quellcode` sowie vordefinierte **Schlüsselworte** wie z.B. **Methoden-**, **Operations-** und **Typnamen**.

Um längere, zusammenhängende Textpassagen in einer schräggestellten Schriftform zu vermeiden, drucken wir Beispiele und technische Teilziele aufrecht und markieren ihr Ende mit einem kleinen Quadrat (□).

[1] In [PÖ95] werden die Invarianten als *Axiome* bezeichnet.

Kapitel 2

Grundlagen

Relationale Datenbanksysteme basieren durchweg auf dem 1970 von E. F. Codd entwickelten Datenmodell [Cod70]. Damit besteht im Bereich des relationalen Modells eine allgemein akzeptierte Grundlage, auf der aufgebaut werden kann. Der objektorientierte Bereich offeriert jedoch erheblich mehr Modellierungskonzepte und entsprechend existieren zahlreiche Varianten des objektorientierten Modells. Daher sollen in diesem Kapitel zunächst die in dieser Arbeit verwendeten Begriffe definiert werden. Dies geschieht in Anlehnung an die 1987 von P. Wegner vorgeschlagenen „Dimensionen der Sprachgestaltung“ (engl. *dimensions of language design*), welche nach [Weg87][2] folgende Begriffe umfassen: Objekte, Klassen, Vererbung, Datenabstraktion, strenge Typisierung, Nebenläufigkeit und Persistenz. Das sog. „Manifesto“ für OODBMS [ABDW+89], das den Versuch unternimmt eine generelle Übereinkunft über die von einem OODBMS zu erfüllenden Anforderungen zu erzielen, nimmt eine abweichende Einteilung vor. Wir werden die dort aufgeführten *verpflichtenden*[3] und *optionalen*[4] Eigenschaften eines OODBMS im Vergleich zu der Kategorisierung von Wegner betrachten. Das etwas neuere „Manifesto für Datenbanksysteme der dritten Generation“ [SRL+90] fordert im wesentlichen Eigenschaften, die auch schon in [ABDW+89] verlangt werden. Weitere Anforderungen werden in [LV96] gestellt. Eine sehr umfangreiche Bibliographie zu OODBMS gibt [Vos91].

Der Arbeitsschritt der Beschreibung elementarer Grundlagen endet mit einer Vorstellung von aus dem Bereich der Objektversionierung bekannten Konzepten.

2.1 Standard-Konzepte objektorientierter Datenbanksysteme

Die folgende Auswahl grundlegender Konzepte der Objektorientierung folgt Wegner [Weg87]. Wir gehen auf jede der „Dimensionen der Sprachgestaltung“ kurz ein.

2.1.1 Objekte

Wie sich bereits aus dem Namen erahnen läßt, spielen die *Objekte* (engl. *objects*) die zentrale Rolle in allen objektorientierten Systemen. Objekte repräsentieren Konzepte der zu modellierenden Diskurswelt und verfügen dazu über einen inneren Zustand und eine Menge von Operationen, die als *Methoden* bezeichnet werden. Jedes Objekt besitzt also *strukturelle* und *operationelle*

[2] Eine neuere Arbeit zu diesem Thema ist [Weg90].

[3] Diese umfassen komplexe Objekte, Objektidentifikatoren, Kapselung, Typen und Klassen, Klassen oder Typhierarchie, Überschreiben, Überladen und spätes Binden, Vollständigkeit, Erweiterbarkeit, Persistenz, Hintergrundspeicher-Management, Nebenläufigkeit, Ausfallsicherheit und ad-hoc Anfragemöglichkeit.

[4] Diese umfassen Mehrfachvererbung, Typsystem, Verteilung, Designtransaktionen und Versionen.

(verhaltensmäßige) Merkmale[5] (engl. *structural and behavioural properties*). Zu den strukturellen Merkmalen eines Objektes zählt die Menge seiner *Attribute*, d.h. der Variablen, die seinen inneren Zustand beschreiben. Dieser Zustand reflektiert einerseits die bisherige Entwicklung des Objektes, andererseits beeinflußt er dessen zukünftiges Verhalten. Die operationellen Merkmale beinhalten die Menge der Methoden, die das Objekt ausführen kann und die als seine Schnittstelle (engl. *interface*) bezeichnet wird. Jedes Objekt ist über einen unveränderlichen Objektidentifikator (engl. *object identity*) eindeutig identifizierbar.

Das Manifesto [ABDW+89] fordert insbesondere die Unterstützung für komplexe Objekte (engl. *complex objects*). Diese sind aus Komponentenobjekten zusammengesetzt (**is_part_of**-Beziehung), welche existentiell von den komplexen Objekten abhängen.

Objekte interagieren durch Kommunikation, d.h. durch den Austausch von Nachrichten mit bestimmten Namen. Wenn ein Senderobjekt einem Empfängerobjekt eine solche Nachricht schickt, so löst diese beim Empfänger die Ausführung eines Verhaltensmusters aus. Dieses Verhaltensmuster wird durch diejenige Methode des Empfängers bereitgestellt, die denselben Namen wie die empfangene Nachricht trägt.

2.1.2 Klassen

Um den Aufwand für die Spezifikation der Eigenschaften von Objekten zu verringern und um existierende Software wiederverwendbar zu machen, wird das Konzept der *Klassen* (engl. *classes*) eingeführt. Eine Klasse erlaubt die gemeinsame Beschreibung einer beliebigen Anzahl von Objekten mit vergleichbaren strukturellen und operationellen Merkmalen und dient gleichsam als Schablone aus der Objekte *instanziiert* werden können. Wir sagen ein Objekt gehöre einer Klasse an, wenn es die von der Klasse spezifizierten Merkmale aufweist. Diese Beschreibung der gemeinsamen Merkmale wird als *Intension* der Klasse bezeichnet, während die Menge aller Objekte, die einer Klasse angehören, ihre *Extension* genannt wird.

Die Klasse `Person` könnte beispielsweise der Modellierung von Menschen der Diskurswelt dienen. Die dazu spezifizierten strukturellen Merkmale könnten Vorname (`first_name`), Nachname (`family_name`), Geburtsdatum (`date_of_birth`) und Wohnort (`address`) umfassen. Zu den operationellen Merkmalen könnten Methoden gehören, die die Geburt (`birth`), die Einschulung (`start_school`), eine Heirat (`marry`) oder den Tod (`death`) einer Person beschreiben.

```
class Person {
    type tuple (                              method {
        first_name      : string,                 birth ();
        family_name     : string,                 start_school ();
        date_of_birth   : date,                   marry (spouse: Person);
        address         : Address)                death (); }
                                              } /* class Person */
```

Die Klasse `Person`.

2.1.3 Vererbung

Die durch die Einführung des Klassenkonzeptes erreichte Reduzierung notwendigen Spezifikationsaufwandes und Erhöhung des Wiederverwendungsgrades wird durch das Konzept der *Vererbung* (engl. *inheritance*) noch gesteigert. Dieses sieht nämlich vor, daß bei der Beschreibung einer

[5]Synonym verwenden wir auch den Begriff der Eigenschaften.

Klasse c auf bereits existierende Klassenbeschreibungen zurückgegriffen werden kann, wodurch sich die dort beschriebenen Eigenschaften auf c übertragen. Wir sagen dann, die Klasse c *erbe* die strukturellen und operationellen Merkmale einer oder mehrerer Klassen $c_1, \ldots, c_n$. Die Klassen $c_1, \ldots, c_n$ heißen dann *Oberklassen* (engl. *superclasses*) von c und c heißt *Unterklasse* (engl. *subclass*) von $c_1, \ldots, c_n$. Die geerbten Merkmale können in der Klasse c modifiziert oder ergänzt werden. Die Vererbungsbeziehung ist transitiv, dementsprechend unterscheiden wir *direkte* und *indirekte* Ober- und Unterklassen.

Häufig ist es nützlich, in einem Schema eine ausgezeichnete Klasse zu haben, die stets Oberklasse aller Klassen ist. Diese als *Wurzelklasse* bezeichnete Klasse wird von vielen Systemen unter dem Namen **Object** automatisch angelegt. In Systemen mit *Einfachvererbung* (engl. *single inheritance*) hat jede Klasse mit Ausnahme dieser Wurzelklasse genau eine direkte Oberklasse, so daß eine Vererbungshierarchie entsteht, die graphisch als *Vererbungsbaum* dargestellt werden kann. Ist die Einschränkung auf höchstens eine direkte Oberklasse aufgehoben, spricht man von *Mehrfachvererbung* (engl. *multiple inheritance*). Hier ensteht eine Vererbungshierarchie, die als gerichteter, zyklenfreier Graph (engl. *directed acyclic graph, DAG*) dargestellt werden kann. Bei der Mehrfachvererbung ggf. auftretende *Vererbungskonflikte* müssen jedoch mit einer geeigneten Strategie aufgelöst werden. Im Manifesto wird die Einfachvererbung unter den verpflichtenden Eigenschaften aufgeführt, während die Mehrfachvererbung als optional eingestuft ist. In beiden Fällen entsteht eine Vererbungsstruktur mit der Wurzelklasse als Wurzel,[6] wobei die Vererbung von Eigenschaften entlang der Kanten dieser Struktur geschieht.

Eine Unterklasse stellt eine *Spezialisierung* ihrer Oberklassen dar, während umgekehrt eine Oberklasse eine *Generalisierung* ihrer Unterklassen ist. Demzufolge kann ein Objekt einer Klasse c durch ein Objekt einer Unterklasse von c *substituiert* werden. Die Extension einer Unterklasse ist eine Teilmenge der Extension ihrer Oberklassen.

Die Klasse `Employee` wird beispielsweise in der Regel als Unterklasse der Klasse `Person` modelliert, womit zum Ausdruck gebracht wird, daß jeder Angestellte auch eine Person ist. Durch die Vererbung erhält jeder Angestellte automatisch die strukturellen (Vorname, Nachname, Geburtsdatum und Wohnort) und operationellen Merkmale (Methoden für Geburt, Einschulung, Heirat und Tod) der Oberklasse `Person`. Zusätzlich werden Angestellte weitere Attribute wie Abteilung (`department`) und Gehalt (`salary`) und weitere Methoden beispielsweise für die Aufnahme oder Beendigung des Angestelltenverhältnisses (`hire` und `fire`) und für die Änderung des Gehaltes (`change_salary`) besitzen, die nicht von der Oberklasse `Person` geerbt sondern in der Klasse `Employee` lokal definiert sind.

```
class Employee: Person {
    type tuple (                    method {
        department  : int,              hire ();
        salary      : int)              fire ();
                                        change_salary (new_salary : int);
                                    } /* class Employee */
```

Die Klasse `Employee` mit Vererbung.

Im Manifesto wird ebenfalls ein Vererbungsmechanismus verpflichtend gefordert. Allerdings kann sich dieser auf Typen anstelle von Klassen beziehen. Dieser Alternative folgend entsteht dann eine Typhierarchie anstelle der Klassenhierarchie.

[6] Damit wird die Bezeichnung der ausgezeichneten und vom System angelegten Klasse als *Wurzel*klasse verständlich.

2.1.4 Datenabstraktion

Einem Objekt der Diskurswelt wird bei der Modellierung ein technisches Objekt o zugeordnet und die Zustände, die das Objekt der Diskurswelt annehmen kann, werden auf die o zugeordnete Datenstruktur abgebildet. Wenn das Konzept der *Datenabstraktion* (engl. *data abstraction*) eingehalten wird, dann können keine direkten Zugriffe anderer Objekte auf die Datenstruktur von o stattfinden. Lediglich die Methoden von o können dessen Zustand direkt auslesen oder verändern und nur über sie können andere Objekte auf den Zustand von o zugreifen. Damit ist die Repräsentation des Zustandes in der Datenstruktur von o vor fremden Zugriffen geschützt. Jedes Objekt erhält so eine durch seine Methoden spezifizierte Schnittstelle, die nicht umgangen werden kann. Im Manifesto wird dies als Kapselung (engl. *encapsulation*) der Objekte bezeichnet. Die Datenabstraktion erleichtert somit die Konsistenzerhaltung in der Datenstruktur des Objektes und die interne Repräsentation kann ohne Seiteneffekte beliebig verändert werden, solange die Schnittstelle des Objektes unverändert bleibt.

Durch die Kapselung ist der Zugriff auf den Zustand eines Objektes von außerhalb nur über seine Methoden möglich.

```
Schmidt.salary = 5500;           // direkter Attributzugriff ist nicht zulässig
Schmidt.change_salary (5500);    // Methodenaufruf ist zulässig
```

Zugriffe auf den Zustand von Objekten.

Das Manifesto besteht für Datenbanksysteme nicht auf der geschilderten strengen Form der Datenabstraktion, die ihren Ursprung in den abstrakten Datentypen von Programmiersprachen hat. Prinzipiell kann die Struktur eines Datentyps auch seiner Schnittstelle (und nicht seiner Implementierung) zugeordnet werden. Damit wird die Struktur nach außen sichtbar, was beispielsweise die Formulierung von ad-hoc Anfragen vereinfacht.

2.1.5 Strenge Typisierung

Das Typkonzept unterstützt in Programmiersprachen die Entwicklung fehlerfreier Programme, indem es die automatische Erkennung einer — allerdings sehr kleinen — Gruppe von Programmfehlern erlaubt. Es zwingt den Programmierer zur expliziten Deklaration aller benutzter Variablen,[7] wobei jeder Variablen ein *Datentyp* zugeordnet werden muß, der den Wertebereich und das Verhaltensrepertoire zulässiger Variablenbelegungen bestimmt. Bei Wertzuweisungen an eine Variable kann damit die *Typkompatibilität* zwischen der Variablen und dem zuzuweisenden Wert automatisch geprüft werden.

In objektorientierten Systemen können in der Regel Werte spezielleren Typs an Variablen allgemeineren Typs zugewiesen werden. Daher muß zwischen dem *statischen* und dem *dynamischen* Typ einer Variablen unterschieden werden. Der statische Typ entspricht dem bei der Deklaration der Variablen angegebenen Typ und ist damit in einem Programm unveränderlich festgelegt. Der dynamische Typ einer Variablen beschreibt den in der Variablen zu einem bestimmten Zeitpunkt der Programmausführung enthaltenen Wert und kann sich daher mit jeder Zuweisung

[7] Der Begriff der Variablen soll hier auch die Parameter von Unterprogrammen einschließen.

an die Variable verändern. Der dynamische Typ einer Variablen muß jedoch stets ein Untertyp ihres statischen Typs sein.

```
Schmidt : Person;           // statischer Typ von Schmidt ist Person
Schmidt = new (Person);     // dynamischer Typ von Schmidt ist Person
Schmidt = new (Employee);   // dynamischer Typ von Schmidt ist Employee
```

Statischer und dynamischer Typ eines Objektes.

Der Typ der Basisklasse **Object** wird meist **any** genannt und ist Obertyp aller systemdefinierter und benutzerdefinierbarer Typen.

Je nach Typsystem können Typprüfungen zur Übersetzungszeit eines Programmes vollständig oder nur teilweise durchgeführt werden. Im letzteren Fall müssen weitere Typprüfungen noch zur Programmlaufzeit durchgeführt werden. Schlagen diese dann jedoch fehl, so muß das Programm mit einem Typfehler abgebrochen werden. Können Typfehler zur Laufzeit bereits durch eine Analyse der statischen Programmstruktur ausgeschlossen werden, so spricht man von *strenger Typisierung* (engl. *strong typing*).

Im Manifesto werden Typprüfung und Typermittlung (engl. *type checking and type inferencing*) als optional eingestuft.

2.1.6 Nebenläufigkeit

Können verschiedene Applikationen auf eine Datenbank quasi gleichzeitig zugreifen, so spricht man von *Nebenläufigkeit* (engl. *concurrency*). Die Folgen von Lese- und Schreiboperationen, mit denen die Applikationen auf die Datenbank zugreifen, können dabei zeitlich verzahnt ausgeführt werden. Ein Datenbanksystem muß i.Allg. die Eigenschaft der *Serialisierbarkeit* garantieren, d.h. die zeitlich verzahnte Ausführungsreihenfolge muß einer strikt seriellen Hintereinanderausführung der verschiedenen Applikationen im Ergebnis äquivalent sein.

2.1.7 Persistenz

Im Gegensatz zu herkömmlichen Programmiersprachen, bei denen Variableninhalte mit der Terminierung eines Programmes verloren gehen, müssen Objekte in der Datenbank *persistent* gespeichert werden. Damit können Objekte die sie erzeugenden Programme überleben und stehen für später gestartete Programme in der Datenbank zur Verfügung.

```
name The_employees : set of Employee; // persistente Menge
Schmidt = new (Employee);             // Objekt Schmidt der Klasse Employee erzeugen
The_employees += (Schmidt);           // Schmidt wird durch Einfügen in die
                                         persistente Menge persistent
The_employees -= (Schmidt);           // Schmidt wird durch Ausfügen aus der
                                         persistenten Menge transient
```

Persistenz und Transienz von Objekten.

Persistenz (engl. *persistency*) sollte orthogonal zum Typkonzept sein, d.h. jedes Objekt sollte unabhängig von seinem Typ transient oder persistent gemacht werden können.

2.1.8 Weitere nach dem Manifesto verpflichtende Eigenschaften eines OO-DBMS

Über [Weg87] hinausgehend fordert das Manifesto in den sog. „Golden Rules“ weitere Eigenschaften, die jedes OODBMS notwendigerweise besitzen muß. Auf diese soll hier kurz eingegangen werden.

2.1.8.1 Überschreiben, Überladen und spätes Binden

In imperativen Programmiersprachen müssen Funktionsnamen des einfacheren Modells wegen global eindeutig sein. Objektorientierte Programmiersprachen und Datenbanksysteme heben diese Einschränkung auf, womit dem Programmierer die Möglichkeit gegeben wird, Methoden, die vergleichbare Aufgaben erfüllen, denselben Namen zu geben. Damit kann im Vergleich zu imperativen Programmiersprachen mehr Semantik ausgedrückt werden, womit sich der zur Erreichung bestimmter Ziele notwendige Programmieraufwand teilweise erheblich verringern läßt. Dies wird bei der Arbeit mit heterogenen Mengen, die also Objekte verschiedener Klassen enthalten, besonders deutlich. Soll nämlich dasselbe Verhaltensmuster von allen Objekten einer solchen heterogenen Menge ausgeführt werden und sind für alle Klassen von in der Menge enthaltenen Objekten Methoden für das gewünschte Verhaltensmuster mit demselben Namen implementiert, so genügt es über die Menge zu iterieren und für jedes Objekt die Methode einmal aufzurufen. Die Unterscheidung welcher Klasse ein einzelnes Objekt angehört und welche der gleichnamigen Methoden demzufolge eingesetzt werden muß, kann dann nämlich vom System übernommen werden. Es kommt hier als Reaktion auf eine verschickte Nachricht immer nur genau eine der gleichnamigen Methoden zum Einsatz.

Die Unterscheidung zwischen den gleichnamigen Methoden kann auf zwei verschiedenen Wegen erfolgen, welche als *Überschreiben* (engl. *overriding*) und als *Überladen* (engl. *overloading*) bezeichnet werden. Beim Überschreiben geschieht die Unterscheidung durch die Klasse, in der eine Methode implementiert ist. Dabei kann insbesondere eine in einer Oberklasse definierte Methode in einer Unterklasse in einer spezialisierteren Form angegeben werden. Bei jedem Objekt wird dann die speziellste Implementierung der Methode verwendet. Wenn die direkte Klasse des Objektes eine entsprechende Methode enthält, so wird diese verwendet, ansonsten wird die Suche nach einer passenden Implementierung rekursiv in den Oberklassen der direkten Klasse fortgesetzt.

Im folgenden Beispiel definieren die Klassen `Person` und `Employee` jeweils eigene Implementierungen für die Methode `marry`. Die Methode zum Heiraten der Klasse `Employee` überschreibt die gleichnamige, von der Oberklasse `Person` geerbte Methode. In Erweiterung der Methode `marry` der Klasse `Person` könnte bei Angestellten das Gehalt um einen Bonus erhöht werden.

```
class Person {
    ...
    method {
        marry (spouse : Person); }
    } /* class Person */

class Employee: Person {
    ...
    method {
        marry (spouse : Person); }
    } /* class Employee */
```

Überschreiben von Methoden.

Beim Überladen geschieht die Unterscheidung durch die Anzahl oder durch die Typen der übergebenen Parameter. Damit kann sogar eine einzelne Klasse mehrere gleichnamige Methoden enthalten, die sich jedoch anhand von Parameteranzahl oder -typ unterscheiden müssen, so daß bei jedem Methodenaufruf die zu verwendende Implementierung eindeutig bestimmt werden kann.

Die Klasse `Employee` definiert zwei überladene Methoden namens `hire`, die sich durch die Zahl ihrer Parameter unterscheiden.

```
class Employee: Person {
    ...
    method {
        hire (salary : int);
        hire (salary : int, dept: int); }
    } /* class Employee */
```

Überladen von Methoden.

Da eine Variable zur Laufzeit nicht nur Objekte ihres direkten Typs sondern ebenso Objekte aller Untertypen beinhalten darf, kann die bei einem Methodenaufruf zu verwendende Implementierung einer überschriebenen oder überladenen Methode zur Übersetzungszeit noch nicht ermittelt werden. Dieser als *Binden* bezeichnete Zuordnungsvorgang kann also erst zur Laufzeit des Programmes durchgeführt werden und wird daher auch *spätes Binden* (engl. *late binding*) genannt.

2.1.8.2 Vollständigkeit

Unter Vollständigkeit (engl. *computational completeness*) wird verstanden, daß jede berechenbare Funktion in der Datenmanipulationssprache eines Datenbanksystems oder in einer verwendbaren externen Sprache ausgedrückt werden kann.

2.1.8.3 Erweiterbarkeit

Als Erweiterbarkeit (engl. *extensibility*) fordert das Manifesto die Möglichkeit der Definition neuer Typen und neuen Verhaltens durch den Programmierer. Dabei soll bei der Verwendung von benutzerdefinierten und systemdefinierten Typen kein Unterschied erkennbar sein. Die Erweiterbarkeit der Menge verwendbarer Typkonstruktoren wird jedoch explizit von den verpflichtenden Eigenschaften ausgenommen. Als Beispiel führen wir die Definition eines neuen Typs zur Repräsentation von Adressen an.

```
type Address_type = tuple (
    street   : string,
    zip_code : int,
    city     : string);
```

Ein benutzerdefinierter Typ.

2.1.8.4 Hintergrundspeicher-Management

Datenbanksysteme sollten in der Lage sein, auch sehr große Datenmengen zu verwalten. Dazu gehört insbesondere, daß beim Hintergrundspeicher-Management (engl. *secondary storage management*) effiziente Mechanismen eingesetzt werden, wie Indexierung (engl. *index management*), Blockung (engl. *data clustering*), Pufferung (engl. *data buffering*), Auswahl von Zugriffspfaden (engl. *access path selection*) und Anfrageoptimierung (engl. *query optimization*). Diese Mechanismen müssen für den Benutzer der Datenbank allerdings transparent sein.

2.1.8.5 Ausfallsicherheit

Die Ausfallsicherheit (engl. *recovery*) fordert nach Hard– oder Softwarefehlern die Möglichkeit zur Wiederherstellung des zuletzt erreichten, konsistenten Datenbankzustandes.

2.1.8.6 Ad-hoc Anfragemöglichkeit

Jedes OODBMS muß laut Manifesto über eine Möglichkeit verfügen ad-hoc Anfragen zu stellen (engl. *ad hoc query facility*). Diese kann durch eine Anfragesprache oder durch eine graphische Schnittstelle realisiert sein und sollte deklarativ, effizient und applikationsunabhängig sein.

2.1.9 Weitere nach dem Manifesto optionale Eigenschaften eines OODBMS

Das Manifesto schlägt einige als „Goodies“ bezeichnete Eigenschaften vor, die hier kurz aufgeführt werden sollen, sofern sie nicht schon zuvor Erwähnung fanden.

2.1.9.1 Verteilung

Die Möglichkeit zur Verteilung (engl. *distribution*) eines Datenbanksystems auf verschiedene Knoten eines Rechnernetzes ist orthogonal zu seinen sonstigen Eigenschaften.

2.1.9.2 Designtransaktionen

Insbesondere für die Unterstützung von Entwurfsaufgaben kann eine Beschränkung auf dem ACID-Paradigma [HR83] folgende Transaktionen zu restriktiv sein. Einige neue Einsatzgebiete für Datenbanksysteme erfordern daher die Unterstützung von Designtransaktionen (engl. *design transactions*), zu denen langandauernde und verschachtelte Transaktionen zählen.

2.1.9.3 Versionen

Eine weitere Eigenschaft, die von vielen neuen Anwendungsgebieten gefordert wird, ist die Unterstützung von Versionen (engl. *versions*). Da die Versionierungsidee einen zentralen Punkt in dieser Arbeit darstellt, gehen wir auf die damit verbundenen Mechanismen separat in Abschnitt 2.2 ein.

2.1.10 Applikationsanbindung in Datenbanksystemen

Wie zuvor dargestellt, beschreibt das Schema einer objektorientierten Datenbank nicht nur strukturelle, sondern auch verhaltensmäßige Eigenschaften von Instanzen. Dies geschieht durch Methoden, die generelles Verhalten implementieren und daher von mehreren Applikationen eines (unversionierten) Schemas gewinnbringend genutzt werden können. Applikationsspezifisches Verhalten wird dann oberhalb des gemeinsamen Schemas von den verschiedenen Applikationen selbst realisiert. Damit besteht die Implementierung einer Applikation aus zwei Teilen: einem generellen Teil, der als verhaltensmäßige Komponente des Schemas implementiert wird, und einem applikationsspezifischen Teil, der die Methoden des Schemas intensiv nutzt. Daher muß einem Applikationsentwickler die Semantik des Schemas und insbesondere der darin enthaltenen Methoden vollständig klar sein.

Die Implementierung einer Applikation kann in verschiedenen Programmiersprachen erfolgen, sie wird jedoch stets für ein spezielles, fest vorgegebenes Schema realisiert und gegen dieses Schema compiliert.

2.2 Versionierungskonzepte

Wir gehen hier etwas näher auf die Unterstützung von Objektevolution durch Versionierungskonzepte in Datenbanksystemen ein. In Abschnitt 2.1.9.3 hatten wir bereits erwähnt, daß diese vom Manifesto als optionale Eigenschaft eingestuft wird. Die Unterstützung für Versionierung wird also nicht als notwendige Eigenschaft eines jeden OODBMS erachtet, sie wird aber als nutzbringende Ergänzung insbesondere für die sog. neuen Datenbankapplikationen angesehen.

Der Begriff der *Version* eines Objektes bezeichnet einen Zustand, in dem sich das Objekt befindet, eine Ausprägung des Objektes oder auch eine Variante desselben. Dabei geht man oftmals von sog. *Designobjekten* aus. Diese beschreiben oder modellieren Objekte, die in ihrer Diskurswelt entworfen werden. Aufgrund der zumeist hohen Komplexität von Designobjekten zeichnet sich deren Entwurfsprozeß durch einen hohen Grad an Dynamik aus und bedarf zahlreicher Änderungen, Verbesserungen und Korrekturen. In der Praxis sind die Konsequenzen von Designänderungen oft a priori nicht vollständig erkennbar. Daher wird beim Entwurf eine verschiedene Alternativen probierende und vergleichende Vorgehensweise gewählt, die sich als untauglich erweisende Alternativen verwirft (engl. *trial and error*). Dies entspricht ungefähr dem Konzept der biologischen Evolution, die nach Darwin durch *Variation* Alternativen schafft und durch *Selektion* die besten darunter auswählt. Die biologische Evolution kann somit als ein spezieller Entwurfsprozeß betrachtet werden. Eine weitere Ähnlichkeit ist dadurch gegeben, daß in der Evolution der Natur wie beim Entwurf von Designobjekten nicht notwendigerweise eine Alternative als Optimallösung existiert, die alle Anforderungen bestmöglich befriedigt. Stattdessen bestehen in verschiedenen Situationen, unter verschiedenen Voraussetzungen oder Bedingungen und aus verschiedenen Blickwinkeln unterschiedliche, unter den jeweils gegebenen Anforderungen zu bevorzugende Alternativen, die jeweils nur die Anforderungen einer einzelnen Nische optimal erfüllen.

Zur Unterstützung des Entwurfsprozesses werden also Konzepte benötigt, die es erlauben, sich als unvorteilhaft herausgestellte Änderungen rückgängig zu machen. Weiterhin muß die Modellierung von Versionen möglich sein, welche sowohl die Zustände zu repräsentieren vermögen, die ein Objekt im Laufe der Zeit *nacheinander* annimmt, als auch die Alternativen, die von einem Objekt *gleichzeitig* existieren. Einem Versionierungsmechanismus unterworfene Objekte, die demzufolge in verschiedenen Versionen vorliegen können, werden als *versionierte Objekte* bezeichnet.

Versionierungskonzepte zur Unterstützung von Entwurfsprozessen wurden intensiv untersucht. Neben allgemeinen, applikationsunabhängigen Untersuchungen [BM88, CJ90, KSW86, Lau91, Sci91a, Sci91b, Sci94, TG92, Wed94] wurden sie in verschiedensten Anwendungsbereichen eingesetzt, z.B. im Bereich CAD [AN91, CK86, KCB86, RC96], im Bereich geographischer Informationssysteme [MJ94], im Bereich Engineering allgemein [AJ89, DL88, Kat90, RR96, SS94a], im Bereich concurrency control [CJ94], im Bereich Hypertext [DV95], im Bereich Konsistenzkontrolle [DFG+97, DGJM96a], im Bereich komplexer Objekte [KS92, TOC93], etc. Seit einigen Jahren wird Objektversionierung[8] von verschiedenen Datenbanksystemen angeboten, u.a. von ODE [ABGS91], von ORION [BCG+87, CK86, CK88, KBC+89] und von O2 [O2 96c]. Diese etablieren in der Regel eine Ableitungsbeziehung zwischen den verschiedenen Versionen eines

[8] Eine kommentierte Bibliographie zu Versionierung und Konfigurationsverwaltung hat Dan Conde [Con86] erstellt, [Ken89] faßt einige Beiträge einer Panel-Sitzung zusammen.

Objektes. Eine Ableitungsbeziehung von einer Version ov_1 zu einer Version ov_2 drückt aus, daß ov_2 aus ov_1 abgeleitet wurde. Anstatt eine anstehende Veränderung direkt auf der Objektversion ov_1 durchzuführen und deren vorherigen Zustand dabei zu überschreiben, bleibt ov_1 unverändert erhalten. Stattdessen wird bei der Ableitung zunächst eine Kopie unter der Bezeichnung ov_2 angelegt und nur diese Kopie wird verändert. Damit existieren nun zwei Versionen, die den Zustand des Objektes vor und nach der Veränderung repräsentieren und die durch die Ableitungsbeziehung verbunden sind. Damit kann die Veränderung durch Rücksetzen auf ov_1 rückgängig gemacht werden, wenn dies erwünscht ist. Im Allgemeinen kann ein versioniertes Objekt natürlich aus beliebig vielen Versionen bestehen.

Die Ableitungsbeziehung zwischen den Versionen eines Objektes kann graphisch durch eine Liste veranschaulicht werden, wenn nur zeitlich aufeinanderfolgende Versionen existieren, die quasi die lineare, historische Entwicklung des Objektes widerspiegeln (siehe Abbildung 2.1a). Dabei besteht eine Vollordnung zwischen den Versionen eines Objektes, d.h. es kann immer genau eine neueste Version gefunden werden und diese ist auch die einzige, die modifiziert werden darf. Wird die zuletztgenannte Einschränkung aufgehoben, so können auch ältere Versionen modifiziert werden, wodurch Alternativen zu bestehenden Versionen erzeugt werden. Die dabei entstehende Ableitungsbeziehung enthält paralelle Stränge und läßt sich durch einen Baum repräsentieren (siehe Abbildung 2.1b). Sie stellt lediglich eine Halbordnung dar und enthält i.Allg. mehrere neueste Versionen, wobei neu nicht im zeitlichen, sondern im logischen Sinne zu verstehen ist. An diesen logisch neuesten Versionen wurden noch keine Veränderungen vorgenommen und sie stellen genau die Blätter des Ableitungsbaumes dar. Mitunter sind Objekte so umfangreich, daß verschiedene Versionen unterschiedliche Aspekte am geeignetsten repräsentieren. Dann wird die Verschmelzung zweier (oder mehr) Objektversionen wünschenswert, wobei die jeweils besten Komponenten zu einer neuen Objektversion integriert werden. Damit werden also Alternativen zusammengeführt und es entsteht der allgemeinste Fall eines (gerichteten, azyklischen) Ableitungsgraphen (siehe Abbildung 2.1c).

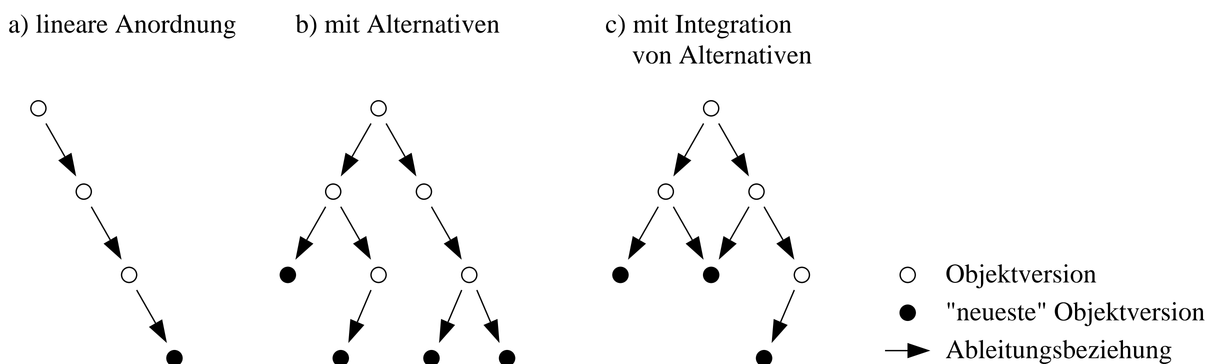


Abbildung 2.1: Verschiedene Typen von Ableitungsbeziehungen zwischen Versionen eines Objektes.

Bei der Auswahl der bei einem Zugriff zu benutzenden Objektversion wird zwischen statischem und dynamischem Binden unterschieden. Beim *statischen Binden* wählt die zugreifende Applikation bereits zur Übersetzungszeit explizit eine bestimmte Version anhand eines Versionsidentifikators (wie ov_1) fest aus. Dahingegen findet beim *dynamischen Binden* keine explizite Auswahl der zuzugreifenden Version statt. Die Auswahl geschieht stattdessen für alle unspezifisch zugreifenden Applikationen auf dieselbe Art und Weise nach einem dynamischen Kriterium. Dies kann beispielsweise besagen, daß immer die zeitlich neueste Version zu benutzen ist. In Erweiterung dessen ist die Partitionierung der Versionen eines Objektes dem Entwicklungsgrad entsprechend in Teilmengen wie *unfertig* (Anfangsstadium), *vorläufig* (Zwischenstadium) und

stabil (Endstadium, freigegeben) möglich. Damit kann dynamisch immer an die neueste unter den stabilen Versionen gebunden werden. Eine weitere Möglichkeit besteht darin, für jedes versionierte Objekt manuell genau eine Defaultversion zu bestimmen, die dann benutzt wird. Beim dynamischen Binden wird die zu benutzende Version eines Objektes also erst zur Laufzeit bestimmt. Applikationen, die nicht berücksichtigen, daß ein von ihnen zugegriffenes Objekt versioniert ist, können nur dynamisch gebunden werden. Für sie kann der Versionierungsmechanismus vollkommen transparent bleiben, weil die Auswahl der zuzugreifenden Objektversion ohne ihr Zutun durchgeführt wird.

Ein wichtiger Punkt bei der Verwendung von Datenbanken ist die Konsistenz der darin gespeicherten Daten. Im unversionierten Fall beschreibt eine Datenbank nur genau einen Zustand der Diskurswelt und dessen Konsistenz wird gefordert. Durch das Versionierungskonzept liegen nun beliebig viele Objekte der Datenbank in mehreren Versionen vor. Aus der damit entstehenden kombinatorischen Vielfalt beschriebener Zustände der Diskurswelt sind i.Allg. nur die wenigsten konsistent. Daher ist ein Mechanismus zu etablieren, der es gestattet, von jedem Objekt der Datenbank genau eine Version auszuwählen, so daß der durch diese Auswahl beschriebene Zustand konsistent ist. Solche Mechanismen werden unter der Bezeichnung *Konfigurationsverwaltung* zusammengefaßt. Eine Konfiguration selektiert für eine Menge von Objekten untereinander konsistente Versionen. Damit reduziert sich das Problem der Auswahl konsistenter Objektversionen auf die Auswahl konsistenter Konfigurationen. Im Extremfall enthält eine Konfiguration alle Objekte der Datenbank und wird dann auch als *Datenbankversion* bezeichnet (siehe Abschnitt 4.5.3.1).

Versionierungskonzepte bieten zahlreiche Vorteile. Dazu gehört die Möglichkeit, durchgeführte Änderungen nachträglich analysieren und nötigenfalls rückgängig machen zu können, womit insbesondere auch das Testen von Entwürfen unterstützt wird, da keine Gefahr eines Datenverlustes besteht. Weiterhin können Entwurfsalternativen für dasselbe Objekt gleichzeitig von verschiedenen Designern erstellt und anschließend miteinander verglichen werden. Schließlich besteht die Möglichkeit jeweils partiell als bestmöglich angesehene Teile verschiedener Alternativen zu integrieren.

Die beschriebenen Konzepte sind für beliebige Objekte in derselben Art und Weise einsetzbar. Anders als die oben genannten Literaturstellen, die Objekte der Datenbank (wie beispielsweise Textdokumente, CAD-Zeichnungen oder Programmcode) versionieren, wenden wir unseren Versionierungsmechanismus auf die Objekte des Metaschemas an. Diese repräsentieren das Schema und beschreiben damit die Schnittstelle zwischen den Applikationen und der Datenbank. Damit bestehen mit den Objekten der Datenbank Instanzen, die existentiell von der in den versionierten Objekten enthaltenen Information abhängen, da diese u.a. deren physikalisches Speicherungsformat bestimmt. Daher unterscheiden wir Versionierung auf der Objekt- und auf der Schemaebene. Damit sind Klassen- und Schemaversionierung analog zu Objekt- und Datenbankversionierung zu sehen.

2.3 Standard-Konzepte der Graphentheorie

Wir werden im Verlaufe dieser Arbeit mehrfach diverse Notationen und Definitionen verwenden, die wir der Eindeutigkeit halber hier kurz einführen. Die gewählten Bezeichnungen entstammen u.a. [Chr75, Har72, Nol76].

Definition 2.1 {**Relation** (engl. ***relation***), (R, M)}
Eine Relation R *auf einer Menge* M *ist eine Menge von Paaren* (a, b) *mit* $a, b \in M$*, d.h.* $R \subseteq M^2$*.*
Die Schreibweise $(a\ R\ b)$ *bedeute* $(a, b) \in R$*.*

Definition 2.2 {**Eigenschaften von Relationen**}
Gegeben sei eine Relation R auf einer Menge M.

- *R heißt* symmetrisch, *wenn gilt:* $\forall a, b \in M : (a\ R\ b) \Rightarrow (b\ R\ a)$.
- *R heißt* antisymmetrisch, *wenn gilt:* $\forall a, b \in M : (a\ R\ b \wedge b\ R\ a) \Rightarrow (a = b)$.
- *R heißt* reflexiv, *wenn gilt:* $\forall a \in M : (a\ R\ a)$.
- *R heißt* transitiv, *wenn gilt:* $\forall a, b, c \in M : (a\ R\ b \wedge b\ R\ c) \Rightarrow (a\ R\ c)$.
- *R heißt* azyklisch, *wenn gilt:* $\nexists\, a_1, \ldots, a_n$ *paarweise verschieden:*$a_1\ R\ a_2\ R\ \cdots\ R\ a_n\ R\ a_1$.
- *R heißt* zusammenhängend, *wenn gilt:* $\forall a, b \in M : (a\ R\ b) \vee (b\ R\ a)$.

Definition 2.3 { **partielle** und **totale Ordnungsrelation** (engl. ***partial*** und ***total order***), $(\leq, M)$ }
Eine antisymmetrische, reflexive und transitive Relation $\leq$ auf einer Menge M heißt Ordnungsrelation *auf M (auch Halbordnung oder Hierarchie). Eine Ordnungsrelation heißt* total, *wenn sie zusammenhängend ist.*

Lemma 2.1 (Zyklenfreiheit von Ordnungsrelationen (engl. ***acyclicity of orders***))
Aus der Antisymmetrie und der Transitivität folgt, daß jede Ordnungsrelation stets auch azyklisch ist.

Definition 2.4 {**abgeleitete Relationen** (engl. ***derived relations***)}
Sei $(\leq, M)$ eine Ordnungsrelation. Dann definieren wir weitere Relationen auf M wie folgt:

- $(a < b) :\Leftrightarrow (a \leq b \wedge a \neq b)$
- $(a <^1 b) :\Leftrightarrow (a < b \wedge\ \nexists c \in M : a < c < b)$
- $(a \leq^1 b) :\Leftrightarrow (a = b \vee a <^1 b)$
- $(a \geq b) :\Leftrightarrow (b \leq a)$
- $(a > b) :\Leftrightarrow (b < a)$
- $(a >^1 b) :\Leftrightarrow (b <^1 a)$
- $(a \geq^1 b) :\Leftrightarrow (b \leq^1 a)$

Definition 2.5 {**gerichteter Graph** (engl. ***directed graph***)}
Ein Paar $G = (V, E)$ wird gerichteter Graph *genannt, wenn V eine Menge ist und E eine Relation auf V. Wir nennen V die* Knotenmenge (engl. vertices)*, E die* Kantenmenge (engl. edges)*, $v \in V$ einen* Knoten *und $e \in E$ eine* Kante.[9]

Definition 2.6 {**Weg** (engl. ***walk***)}
Eine Sequenz $\langle v_0, v_1, \ldots, v_l \rangle$ heißt Weg *von v_0 nach v_l* der Länge *l in $G = (V, E)$, wenn $(v_{i-1}, v_i) \in E$ für $i = 1, \ldots, l$.*

[9] Diese Definition gerichteter Graphen geht bereits von der vereinfachenden Annahme aus, daß je zwei Knoten durch maximal eine Kante verbunden sind. Eine allgemeinere Definition beschreibt einen Graphen als 4-Tupel $G = (V, E, s, z)$, wobei V und E Mengen sind und s und z Abbildungen von E nach V, die jeder Kante $e \in E$ ihren Start- und Zielknoten ($s(e)$ und $z(e)$) zuweisen.

Definition 2.7 {**einfacher Weg** (engl. ***simple path*** oder ***trail***)}
Wir bezeichnen einen Weg $\langle v_0, v_1, \ldots, v_l \rangle$ *als* einfach, *wenn keine Kante doppelt benutzt wird.*

Definition 2.8 {**elementarer Weg** oder **Pfad** (engl. ***elementary path*** oder kurz ***path***)}
Wir bezeichnen einen Weg $\langle v_0, v_1, \ldots, v_l \rangle$ *als* elementar, *wenn kein Knoten doppelt benutzt wird.*

Definition 2.9 {**zusammenhängender Graph** (engl. ***connected graph***)}
Ein Graph $G = (V, E)$ *heißt* (schwach) zusammenhängend, *wenn für je zwei Knoten aus* G *ein Pfad in dem korrespondierenden (ungerichteten) Graphen* $G' = (V, E')$ *existiert, der diese verbindet.* E' *ist dabei als die symmetrische Erweiterung von* E *definiert.*

Definition 2.10 {**Zyklus** (engl. ***cycle***)}
Ein Weg $\langle v_0, v_1, \ldots, v_l, v_0 \rangle$ *wird* Zyklus (oder geschlossener Weg) der Länge $l+1$ *genannt, wenn* $(v_{i-1}, v_i) \in E$ *für* $i = 1, \ldots, l$ *und* $(v_l, v_0) \in E$.

Definition 2.11 {**gerichteter azyklischer Graph,** (engl. ***directed acyclic graph, DAG***)}
Ein gerichteter Graph G *heißt* azyklisch, *wenn kein Zyklus in* G *existiert.*

Definition 2.12 {**Teilgraph** (engl. ***subgraph***)}
Ein Teilgraph *eines Graphen* $G = (V, E)$ *ist definiert als ein beliebiger Graph* $G' = (V', E')$ *für den gilt:* $V' \subseteq V$ *und* $E' \subseteq E$. *(Man beachte, daß hieraus* $\forall (u, v) \in E' : u, v \in V'$ *folgt.)*

Definition 2.13 {**partieller** und **vollständiger Teilgraph** (engl. ***partial*** und ***complete subgraph***)}
Ein Teilgraph $G' = (V', E')$ *von* $G = (V, E)$ *heißt* vollständig, *wenn gilt* $\forall a, b \in V' : (a, b) \in E \Rightarrow (a, b) \in E'$, *sonst* partiell.

Definition 2.14 {**direkte Vorgänger** (engl. ***direct predecessors***), **direkte Nachfolger** (engl. ***direct successors***)}
In einem Graphen $G = (V, E)$ *ist die Menge der* direkten Vorgänger *einer Knotenmenge* $V' \subseteq V$ *definiert als*

$$dpred(V') = \{p \in V - V' \mid \exists v \in V' : (p, v) \in E\}$$

Für einen einzelnen Knoten $v \in V$ *ist die Menge der direkten Vorgänger definiert als*

$$dpred(v) = \{p \in V - \{v\} \mid (p, v) \in E\}.$$

Analog wird die Menge der direkten Nachfolger *einer Knotenmenge* $V' \subseteq V$ *und eines einzelnen Knotens* $v \in V$ *definiert als*

$$dsucc(V') = \{s \in V - V' \mid \exists v \in V' : (v, s) \in E\}$$

$$dsucc(v) = \{s \in V - \{v\} \mid (v, s) \in E\}.$$

Definition 2.15 {**reflexive Hülle** (engl. ***reflexive closure***)}
Die reflexive Hülle $RC(G)$ *eines Graphen* $G = (V, E)$ *ist definiert als* $RC(G) = (V, E^{RC})$ *mit* $E^{RC} = E \cup \{(u, u) \mid u \in V\}$. E^{RC} *wird die* reflexive Hülle *der Relation* E *genannt.*

Definition 2.16 {**transitive Hülle** (engl. ***transitive closure***)}
Die transitive Hülle $TC(G)$ *eines Graphen* $G = (V, E)$ *ist definiert als* $TC(G) = (V, E^{TC})$ *mit* $E^{TC} = \{(u, v) \in V^2 \mid$ es existiert ein Pfad von u nach v in $G\}$. E^{TC} *wird die* transitive Hülle *der Relation* E *genannt.*

Definition 2.17 $\left\{ \begin{array}{l} \textbf{reflexive und transitive Hülle} \\ \text{(engl. } \textbf{\textit{reflexive and transitive closure}}\text{)} \end{array} \right\}$

Die reflexive und transitive Hülle *$RTC(G)$ eines Graphen $G = (V, E)$ ist definiert als die reflexive Hülle der transitiven Hülle von G, d.h. $RTC(G) = RC(TC(G))$.*[10] *Gleichbedeutend gilt* $RTC(G) = (V, E^{RTC})$ *mit* $E^{RTC} = \{(u, v) \in V^2 |$ es existiert ein Pfad von u nach v in $G\} \cup \{(u, u) |\ u \in V\}$. *$E^{RTC}$ wird die* reflexive und transitive Hülle *der Relation E genannt.*

Lemma 2.2 (reflexive und transitive Hülle)
Die reflexive und transitive Hülle eines gerichteten, azyklischen Graphen (oder genauer dessen Kantenmenge) ist eine partielle Ordnungsrelation.

Definition 2.18 {**Vorgänger** (engl. ***predecessors***), **Nachfolger** (engl. ***successors***)}
Die Vorgänger (Nachfolger) *einer Knotenmenge V' oder eines einzelnen Knotens v in einem Graphen G (notiert als $succ(V')$, $pred(V')$, $succ(v)$ und $pred(v)$) sind definiert als die direkten Vorgänger (Nachfolger) derselben Knotenmenge V' bzw. des einzelnen Knotens v in der reflexiven und transitiven Hülle von G.*

Definition 2.19 {**verwurzelter gerichteter Graph** (engl. ***rooted graph***)}
Ein gerichteter Graph $G = (V, E)$ heißt verwurzelt, *wenn es einen eindeutig bestimmten Knoten $w \in V$ gibt, der Vorgänger sämtlicher Knoten von G ist, d.h. wenn gilt: $\exists_1 w \in V : succ(w) \cup \{w\} = V$. Der Knoten w wird dann auch die* Wurzel *des Graphen G genannt.*

Definition 2.20 {**gerichteter Wald** (engl. ***forest***)}
Ein gerichteter azyklischer Graph $G = (V, E)$ heißt gerichteter Wald, *wenn jeder Knoten maximal einen direkten Vorgänger hat. Formal: $\forall v \in V : |dpred(v)| \leq 1$.*

Definition 2.21 {**gerichteter Baum** (engl. ***tree***)}
Ein verwurzelter gerichteter Wald heißt gerichteter Baum.

2.4 Zusammenfassung

In diesem Kapitel haben wir die elementaren Konzepte objektorientierter Datenbanksysteme insbesondere nach [ABDW+89, Weg87] vorgestellt. Eine weitere wichtige Grundlage für den Fortgang unserer Arbeit haben wir mit der Vorstellung von Versionierungskonzepten und deren Anwendung auf Objektebene gelegt. Darauf aufbauend werden wir im folgenden das Datenbankschema als in Versionen vorliegende Instanz der Metaebene betrachten und dann ein Modell zur Durchführung von Schemaänderungen entwickeln.

[10] Wie man leicht sieht, ist dies gleichbedeutend mit der transitiven Hülle der reflexiven Hülle von G. Dies liegt daran, daß die durch die beiden Hüllenbildungen hinzugefügten Kantenmengen stets disjunkt und voneinander unabhängig sind.

Kapitel 3

Problemanalyse, Anforderungen und Lösungsmodell

In diesem Kapitel gehen wir näher auf die verfolgte Aufgabenstellung ein. Wir beginnen den Arbeitsschritt mit einer Analyse, die das vorliegende Problem zum einen beispielhaft und zum anderen einer generellen Betrachtung folgend studiert. Aus den Resultaten der Analyse leiten wir detaillierte Anforderungen an eine Umgebung zur Unterstützung von Schemaänderungen in Form technischer Teilziele ab und entwerfen einen groben Lösungsansatz auf der Basis eines versionierten Datenbankschemas.

3.1 Problemanalyse

In der Problemanalyse betrachten wir Aufgaben im Bereich der Schemaänderung sowohl aus einem speziellen als auch aus einem allgemeinen Blickwinkel. Dabei stellen wir zunächst ein internationales Projekt vor, in dessen Rahmen eine Plattform zur Unterstützung von Softwareentwicklungsprozessen auf der Basis eines objektorientierten Datenbanksystems entwickelt und realisiert wurde. Aus dem damit verbundenen Szenario leiten wir ein durchgängiges Beispiel ab, mit dem wir uns im weiteren Verlauf dieser Arbeit noch häufiger befassen werden. Nach der Untersuchung dieses speziellen Anwendungsfalles motivieren wir die Notwendigkeit zur Durchführung von Schemaänderungen anhand einer allgemein gefaßten Menge von Aufgaben.

3.1.1 Das GOODSTEP-Projekt

Zur Veranschaulichung unserer Aussagen verwenden wir in dieser Arbeit ein durchgängiges Szenario, dem die Mehrzahl der nachfolgenden Beispiele entstammt. Dieses Szenario basiert auf Erfahrungen, die wir bei der Durchführung des Esprit III-Projektes GOODSTEP[11] (General Object Oriented Database for SofTware Engineering Processes) [AAA+94, AAA+96, GT95] gemacht haben.

Im Rahmen von GOODSTEP wurde eine integrierte Softwareentwicklungsumgebung konzipiert und realisiert, die alle Phasen des Entwicklungsprozesses unterstützt, indem sie entsprechende CASE-Werkzeuge (engl. *computer aided software engineering*) wie graphische Editoren für die Erstellung von Diagrammen, syntaxgesteuerte Texteditoren sowie Prozeßmodellierungswerkzeuge bereitstellt. Die Plattform läßt sich sehr flexibel an die Anforderungen des Endanwenders

[11] Die Partner des GOODSTEP-Konsortiums sind British Airways (UK), Cefriel (I), Engineering (I), INRIA (F), LogOn Technology Transfer (D), O_2Technology (F), The Technology Broker (UK), Universität Dortmund (D), Universität Frankfurt (D) und Universität Grenoble (F).

bezüglich des gewünschten Prozeßmodells sowie der zu verwendenden Programmiersprachen und Modellierungskonzepte anpassen. Dies wurde u.a. dadurch erreicht, daß die einzelnen Werkzeuge über Werkzeugkonstruktionswerkzeuge (Werkzeuggeneratoren) aus sehr abstrakten Beschreibungen, welche sich leicht ändern lassen, erstellt werden.

Die Einsatzfähigkeit der Werkzeuge der GOODSTEP-Plattform konnten wir im Rahmen des Esprit IV-Projektes DOOR[12] (Developing Object Oriented applications Rapidly) [CCC+99] im kommerziellen Umfeld zweier Endanwender erfolgreich evaluieren.

Technisch wurden die Werkzeuge der GOODSTEP-Plattform auf einer speziellen Version des Datenbanksystems O_2 erstellt, welche als gemeinsames Repository für alle beim Software-Entwicklungsprozeß entstehenden Dokumente verwendet wurde (siehe Abbildung 3.1). Um diese Aufgabe zu erfüllen, mußte die kommerziell verfügbare Version von O_2 in Kooperation mit dem Hersteller um Leistungsmerkmale in den Bereichen Änderungsmanagement, Sichten, Versionen und Trigger (Aktive Regeln) erweitert werden. Zu dem Bereich des Änderungsmanagements gehörten neben Sicherheitsmodellen insbesondere die Unterstützung für Schemaänderungen mit entsprechend leistungsfähigen Schemaänderungsprimitiven und die Durchführung der erforderlichen Anpassungen der Datenbank. Auf das objektorientierte Datenbanksystem O_2 werden wir bei der Untersuchung seiner Fähigkeiten im Bereich der Schemaevolution noch näher eingehen (siehe Abschnitt 4.3.4.2).

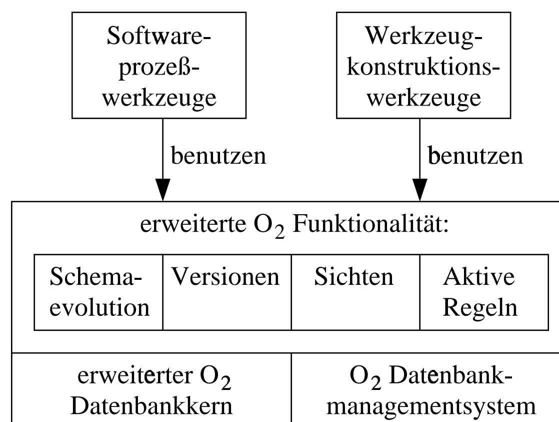


Abbildung 3.1: Die GOODSTEP Architektur.

Wir wollen im folgenden kurz die Rolle von Schemaänderungen für GOODSTEP beschreiben und dann ein Szenario aus dem Umfeld einer Softwareentwicklungsumgebung vorstellen. Wie bereits kurz erläutert sind viele Werkzeuge der GOODSTEP-Plattform Werkzeuggeneratoren, die sich bezüglich verschiedener Aspekte auf die Bedürfnisse des Anwendungsumfeldes einstellen lassen. Dies schließt selbstverständlich auch solche Anpassungen ein, die erst während des laufenden Softwareentwicklungsprozesses durchgeführt werden, etwa weil sich die Anforderungen erst im Verlaufe des Softwareentwicklungsprozesses ergeben oder weil Anpassungen und Verbesserungen der Werkzeuge notwendig werden.

Es entsteht also eine Situation, in der vorhandene Werkzeuge eingesetzt werden, um den Softwareentwicklungsprozeß zu beginnen. Demzufolge sind auch bereits Dokumente verschiedenster Typen (Anforderungskataloge, Analyse- und Design-Dokumente, Use-Cases, Programmquellen, Dokumentation, Projektpläne mit Meilensteinen, Zwischenberichte, etc.) in der Datenbank gespeichert. Das dafür benötigte Datenbankschema wird von den Werkzeugen generiert und in O_2 angelegt, so daß die Werkzeuge entsprechende Dokumente persistent ablegen und darauf kooperieren können.

[12] Die Partner des DOOR-Konsortiums sind Cefriel (I), ENEL (I), Engineering (I), Mannesmann Datenverarbeitung (D) und Universität Frankfurt (D).

Wird nun die Spezifikation eines Werkzeuges verändert und eine neue Version des Werkzeuges generiert, so wird diese neue Version in der Regel auch andere Klassen in der Datenbank benötigen, um die behandelten Dokumente zu verwalten. Dabei ist klar, daß die im bisherigen Verlauf des Softwareentwicklungsprozesses erstellten Dokumente nicht verworfen werden dürfen. Stattdessen müssen sie von nun an sowohl der neuen Version des veränderten Werkzeuges als auch den anderen, unveränderten Werkzeugen zur Verfügung stehen. Eventuell werden sogar, zumindest während einer Test- und Umstellungsphase, verschiedene Versionen desselben Werkzeuges gleichzeitig eingesetzt.

Das beschriebene Umfeld der GOODSTEP-Plattform ist demzufolge ein sehr gutes Bespiel für die Notwendigkeit von Schemaänderungen und entsprechender Anpassungen der Datenbank. Trotz unserer Erfahrungen während der Projekte GOODSTEP und DOOR sind die in dieser Arbeit entwickelten Konzepte nicht auf die besonderen Belange von Softwareentwicklungsumgebungen eingeschränkt. Die Tatsache, daß viele der Werkzeuge nicht händisch erstellt sondern generiert werden, hätte sich ggf. nutzen lassen, um hier bessere Konzepte zu entwickeln, deren Einsatzbereich dann jedoch stark spezialisiert gewesen wäre. Auch die Motivation unserer Arbeiten geht weit über die speziellen Belange der Unterstützung von Softwareentwicklungsprozessen hinaus. Trotzdem stellt das hier beschriebene Umfeld eine sehr typische Instanz des von uns bearbeiteten Aufgabenspektrums dar und ist damit als durchgängiges Beispiel gut zu gebrauchen. Dies zeigt sich zunächst daran, daß verschiedene Werkzeuge auf gemeinsamen Dokumenten kooperieren. Weiterhin entwickeln sich die Werkzeuge auch noch während sie sich bereits im Einsatz befinden weiter und diese Entwicklung macht Anpassungen des Datenbankschemas erforderlich. Bereits existierende Dokumente müssen aber auch unter den genannten Umständen allen Applikationen zur Verfügung stehen und schließlich handelt es sich bei den Dokumenten der Softwareentwicklung um komplex strukturierte Entwurfsdokumente, die von den Modellierungskonzepten des Objektmodells intensiv Gebrauch machen.

3.1.2 Ein durchgängiges Beispiel aus dem Bereich der Softwareentwicklung

Die Modellierung einer Softwareentwicklungsumgebung stellt eine sehr umfangreiche Aufgabe dar; es gilt Dokumente verschiedenster Typen und in verschiedensten Entwicklungsstadien zu beschreiben. Außerdem sind Prozeßmodelle zu erstellen, nach denen der Softwareentwicklungsprozeß insgesamt und die Entwicklung der einzelnen Dokumente fortschreitet. Entsprechend groß ist die Vielfalt benötigter Applikationen, welche auf einem gemeinsamen Datenbestand kooperieren:

- Für das Erstellen und Bearbeiten von Dokumenten der verschiedenen Typen werden entsprechende Programme benötigt. Dazu zählen neben auf die verschiedenen Dokumenttypen spezialisierten Text- und Graphikeditoren auch allgemein verfügbare Werkzeuge wie Compiler, Linker, Debugger, Versionierungs- und Konfigurationswerkzeuge, etc.

- Neben dem Erstellen und Verändern von Prozeßmodellen ist deren Ausführung von zentraler Bedeutung. Dabei werden den definierten Prozessen entsprechende Tätigkeiten zum gegebenen Zeitpunkt automatisch ausgeführt oder notwendige manuelle Arbeitsschritte zumindest angestoßen. Nach Abschluß einzelner Tätigkeiten werden Prozesse in Abhängigkeit von den erreichten Zwischenergebnissen entsprechend dem Prozeßmodell fortgesetzt.

- Im Rahmen des Projektmanagementes müssen Aufgabenbeschreibungen entsprechend den Terminen des Zeitplans an die betroffenen Angestellten verschickt werden. Die Fortschritte bei der Durchführung eines Projektes können anhand von Zustandsattributen oder Versionsnummern kontrolliert und durch Reportgeneratoren dokumentiert werden.

Aufgrund der großen Vielfalt und der zum Teil komplexen Struktur involvierter Dokumente sind viele verschiedene Schemata zu deren Verwaltung in einem Datenbanksystem möglich und wir zeigen hier nur einige sehr rudimentäre Ausschnitte, die unseren Bedürfnissen eines durchgängigen Beispiels genügen. Eine erste Fassung eines Schemas mit Klassen zur Beschreibung textlicher und bildlicher Dokumente könnte beispielsweise wie in Abbildung 3.2 aussehen.

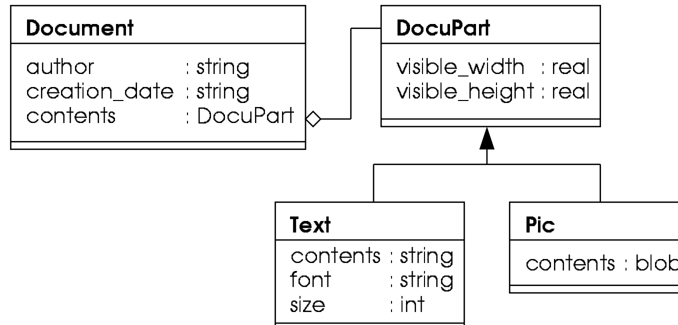


Abbildung 3.2: Graphische Darstellung der Schemaversion sv_1.

Die textuelle Beschreibung dieser ersten Schemaversion sieht wie folgt aus:[13]

```
schema sv1 {
    class Document {
        type tuple (
            author          : string,
            creation_date   : string,
            contents        : DocuPart);
        method ...
        } /* class Document */
    class DocuPart {
        type tuple (
            visible_width   : real,
            visible_height  : real);
        method ...
        } /* class DocuPart */
    class Text : DocuPart {
        type tuple (
            contents: string,
            font    : string,
            size    : int);
        method ...
        } /* class Text */
    class Pic : DocuPart {
        type tuple (
            contents: blob);
        method ...
        } /* class Pic */
} /* sv1 */
```

Textuelle Darstellung der Schemaversion sv_1.

Eine zweite Version sv_2 des Schemas (siehe Abbildung 3.3) könnte u.a. die Klasse `Embedded` hinzufügen, um baumstrukturierte, hierarchische Dokumente modellieren zu können. Diese Vorgehensweise folgt dem Composite-Pattern [GHJV95]. `Embedded` entspricht dabei der Klasse `composite`, `DocuPart` entspricht der Klasse `component` und die übrigen Unterklassen von `DocuPart` entsprechen den Blättern eines baumstrukturierten Dokumentes.

Weiterhin wurden in der Klasse `Document` Attribute zur Beschreibung von Titel und Status eingefügt. Diese werden beispielsweise benötigt, um Statusberichte über den Zustand und die Fortschritte eines Projektes zu erstellen.

[13] Die in dieser Arbeit verwendete Syntax wird erst in den Kapiteln 5 und 6 vollständig beschrieben. Sie wurde jedoch bereits bei der nicht formalen Einführung der Grundlagen in Abschnitt 2.1 verwendet. Damit sollten auch die hier für die Modellierung des Beispiels verwendeten Schemabeschreibungen, insbesondere im Zusammenhang mit den graphischen Darstellungen problemlos intuitiv verständlich sein.

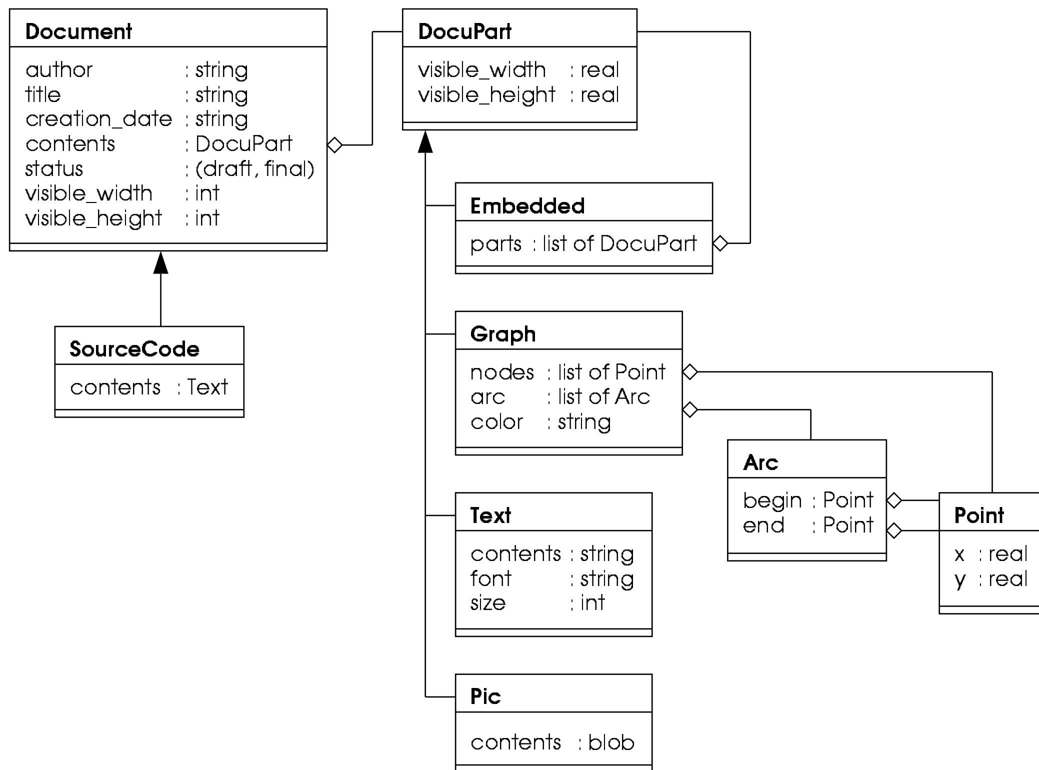


Abbildung 3.3: Graphische Darstellung der Schemaversion sv_2.

```
schema sv2 {
    class Document {
        type tuple (
            author          : string,
            title           : string,
            creation_date   : string,
            contents        : DocuPart,
            status          : (draft, final),
            visible_width   : int,
            visible_height  : int);
        method ...
        } /* class Document */
    class SourceCode : Document {
        type tuple (
            contents: Text);
        method ...
        } /* class SourceCode */
    class DocuPart {
        type tuple (
            visible_width   : real,
            visible_height  : real);
        method ...
        } /* class DocuPart */
    class Embedded : DocuPart {
        type tuple (
            parts    : list of DocuPart);
        method ...
        } /* class Embedded */
    class Text : DocuPart {
        type tuple (
            contents: string,
            font    : string,
            size    : int);
        method ...
        } /* class Text */
    class Pic : DocuPart {
        type tuple (
            contents: blob);
        method ...
        } /* class Pic */
    class Graph : DocuPart {
        type tuple (
            nodes    : list of Point,
            arcs     : list of Arc,
            color    : string);
        method ...
        } /* class Graph */
```

Textuelle Darstellung der Schemaversion sv_2 (Teil 1).

```
class Arc {
    type tuple (
        begin   : Point,
        end     : Point);
    method ...
    } /* class Arc */

class Point {
    type tuple (
        x   : real,
        y   : real);
    method ...
    } /* class Point */
} /* sv2 */
```

Textuelle Darstellung der Schemaversion sv_2 (Teil 2).

Für manche Applikationen mag es vorteilhaft sein, Punkte alternativ auch mit Polarkoordinaten angeben zu können. Eine mögliche Modellierung findet sich in Version sv_3 unseres Beispielschemas.

Weiterhin wurde in sv_3 (siehe Abbildung 3.4) die Klasse `Employee` hinzugefügt, um die an einem Prozeß beteiligten Angestellten zu beschreiben, deren Arbeitszeit zu planen und ihnen Aufgaben und Verantwortlichkeiten im Rahmen der Softwareentwicklung zuweisen zu können.

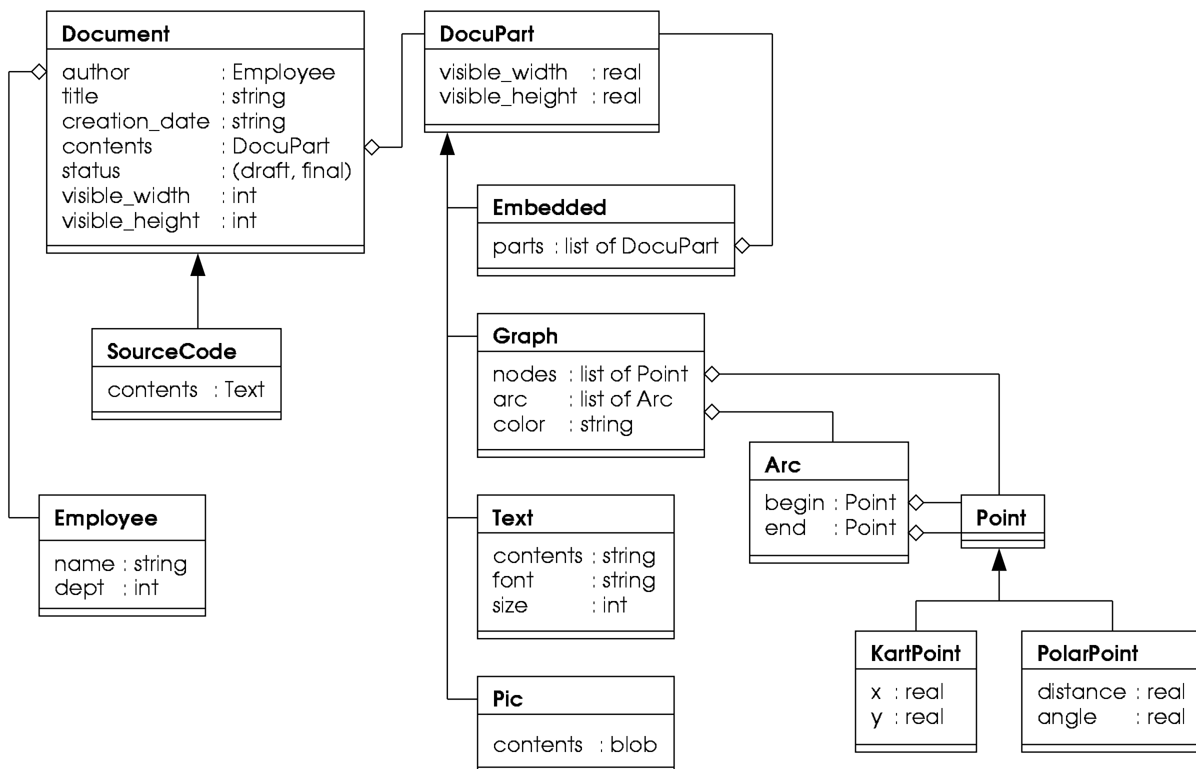


Abbildung 3.4: Graphische Darstellung der Schemaversion sv_3.

Sofern feststeht, daß alle Dokumente der Softwareentwicklung von eigenen Angestellten erstellt werden und diese in der Klasse `Employee` enthalten sind, so könnte der Typ des Attributes `author` der Klasse `Document` von **string** auf `Employee` geändert werden.

```
schema sv3 {
    class Document {
        type tuple (
            author          : Employee,
            title           : string,
            creation_date   : string,
            contents        : DocuPart,
            status          : (draft, final),
            visible_width   : int,
            visible_height  : int);
        method ...
        } /* class Document */
    ...
    class Employee {
        type tuple (
            name    : string,
            dept    : int);
        method ...
        } /* class Employee */
    class Point {
        type tuple ();
        method ...
        } /* class Point */
    class KartPoint : Point {
        type tuple (
            x   : real,
            y   : real);
        method ...
        } /* class KartPoint */
    class PolarPoint : Point {
        type tuple (
            distance: real,
            angle   : real);
        method ...
        } /* class PolarPoint */
    } /* sv3 */
```

Textuelle Darstellung der Schemaversion sv_3.

Um das Layout von Dokumenten zu verbessern, könnte die Beschreibung der Randbreiten flexibilisiert werden, indem die Attribute `visible_height` und `visible_width` aus sv_3 durch `border_left`, `border_right`, `border_top` und `border_bottom` in der neuen Schemaversion sv_4 ersetzt werden und ein Attribut zur Repräsentation des Papierformates (`page_size`) ergänzt wird.

Während sv_3 insgesamt drei Klassen (`Point`, `KartPoint` und `PolarPoint`) zur Darstellung von Punkten verwendet, könnte man auch mit nur einer Klasse auskommen, die die beiden Attribute des Typs **real** alternativ für beide Koordinatensysteme benutzt. Diese Klasse `Point` wurde zur Unterscheidung der möglichen Koordinatensysteme um ein Attribut eines Aufzählungstyps ergänzt, das genau zwei Zustände (`kart` und `polar`) annehmen kann.

```
schema sv4 {
    ...
    class Document {
        type tuple (
            author        : Employee,
            title         : string,
            creation_date : string,
            contents      : DocuPart,
            status        : (draft, final),
            page_size     : string,
            border_left   : int,
            border_right  : int,
            border_top    : int,
            border_bottom : int);
        method ...
        } /* class Document */
    class Point {
        type tuple (
            a          : real,
            b          : real,
            point_type : (kart, polar));
        method ...
        } /* class Point */
    } /* sv4 */
```

Textuelle Darstellung der Schemaversion sv_4.

Die hier beschriebenen Schemaversionen dienen uns lediglich als einfache Anschauungsobjekte für die späteren Betrachtungen und die Erläuterungen verschiedener Konzepte und Mechanis-

men. Auch wenn sie in der hier vorgestellten Form für Modellierung von Softwareentwicklungsumgebungen, wie sie etwa im Rahmen des GOODSTEP-Projektes entwickelt wurden, sicherlich bei weitem nicht ausreichen, so dürfen sie doch als Kristallisationspunkte verstanden werden, die einige wesentliche Aspekte aufzeigen und uns als Beispiel mit realistischem Hintergrund dienen können. Bereits an dieser Stelle läßt sich erahnen, wie groß die Vielfalt möglicher Modellierungen bei Verwendung der Konstrukte eines objektorientierten Datenmodells ist und welcher Bedarf für verschiedenste Anpassungen des Schemas daraus in einer dynamischen Umgebung erwachsen kann.

3.1.3 Motivation für die Notwendigkeit von Schemaänderungen

Wir beschäftigen uns in dieser Arbeit mit Änderungen an Datenbankschemata und setzen dabei eine Vorgehensweise ein, die sogar die Verwaltung und Verwendung mehrerer Schemaausprägungen zur selben Zeit erlaubt. Schemaänderungen werden in zahlreichen, komplexen Anwendungsbereichen wie Entwurfsunterstützung oder Büroautomatisierung häufig benötigt und sind Gegenstand zahlreicher Veröffentlichungen, auf die wir zum Teil bei der Literaturanalyse im nächsten Kapitel eingehen werden. An dieser Stelle belegen wir zunächst die Bedeutung der hier verfolgten Aufgabenstellung, indem wir verschiedene Umstände beschreiben, die die Durchführung evolutionärer Schemaänderungen erfordern.

- *Anpassung an unvorhersehbare Änderungen in der Diskurswelt:* Sowohl der Umfang als auch der Inhalt der durch eine Datenbank zu beschreibenden Diskurswelt (engl. *domain of discourse*) können sich im Laufe der Zeit verändern. Diese Veränderungen sind naturgemäß unvorhersehbar und können nicht geplant werden. Sie können in beliebigen Richtungen ablaufen, folgen nicht notwendigerweise einem strukturierten Evolutionspfad und erfordern damit entsprechende Anpassungen in der Modellierung, d.h. im Datenbankschema. Durch die Betrachtung zusätzlicher oder erweiterter Geschäftsfelder dehnt sich der Umfang der Diskurswelt aus; durch neue Technologien, gesetzliche und sonstige Rahmenbedingungen verändert sich der bisherige Inhalt der Modellierung. Damit ergeben sich zahlreiche Situationen, die eine Erweiterung, Detaillierung oder Anpassung von Datenbankschemata erfordern. Insbesondere für große und lang lebende Datenbanken kann nicht davon ausgegangen werden, daß eine initiale Spezifikation alle Bedürfnisse für die Datenspeicherung und -manipulation erfüllt.

- *Vielfältigkeit der Konstrukte objektorientierter Datenmodelle:* Nach Dag Sjøberg [Sjø93a, Sjø93b] werden Schemaänderungen bereits in relationalen Datenbanksystemen sehr häufig benötigt, obwohl relationale Schemata nur strukturelle Eigenschaften besitzen und nur beschränkte Modellierungskonzepte anbieten. Schemata objektorientierter Datenbanksysteme verfügen über mächtigere Konzepte und spezifizieren zusätzlich verhaltensmäßige Eigenschaften, womit sich die Zahl notwendiger Schemaänderungen noch erhöht. Damit besteht insbesondere bei OODBMS eine starke Notwendigkeit für die Unterstützung von Schemaevolution. Dies gilt selbst unter der Einschränkung, daß wir uns in dieser Arbeit hauptsächlich auf die strukturellen Veränderungen beziehen, da nur diese direkte Konsequenzen für persistente Objekte haben.

- *Komplexität typischer Einsatzgebiete:* Weiterhin läßt sich für objektorientierte Systeme im Vergleich zu relationalen ein erhöhter Bedarf für die Unterstützung von Schemaänderungen erkennen, weil diese typischerweise für die sog. fortschrittlichen Applikationen eingesetzt werden. Diese sind in Bereichen angesiedelt, welche forschenden Charakter haben und welche sich sowohl durch experimentelle als auch kooperative Aktivitäten auszeichnen. Im Gegensatz zu traditionellen Datenbanken, die typischerweise *datenintensiv* sind und eine

große Zahl von Instanzen relativ weniger Klassen (bzw. Typen) enthalten, sind OODBMS typischerweise *berechnungsintensiv* und enthalten weniger Instanzen einer größeren Menge von Klassen. Dadurch steigt die Wahrscheinlichkeit einer Klassen- bzw. Schemaänderung. Ein weiterer Grund hierfür liegt darin, daß in den genannten komplexen Anwendungsbereichen Schemata häufig nur nach der Trial-and-Error Vorgehensweise [WKL86], die auch als *exploratory programming* bezeichnet wird, erstellt werden können.

- *Gleichzeitige Existenz verschiedener Perspektiven der Diskurswelt:* Die bisher besprochenen Änderungen zeichnen sich durch ihre korrigierende Natur aus: Das Schema und die Datenbank werden an sich in der Diskurswelt vollzogene Änderungen angepaßt. Die dabei zugrunde liegende Betrachtungsweise geht davon aus, daß stets eine für alle Anwendungen gleichermaßen ideale Modellierung der Diskurswelt existiert und eine Änderung demzufolge stets das Ziel einer globalen Verbesserung verfolgt. Jedoch haben verschiedene Benutzergruppen oft sehr unterschiedliche Betrachtungsweisen derselben Objekte. Die Herstellung eines Produktes hat beispielsweise u.a. technische, betriebswirtschaftliche und juristische Aspekte und demnach ergeben sich entsprechend grundlegend verschiedene Perspektiven, die bei der Modellierung zu berücksichtigen sind.

  Die logische Konsequenz aus dieser Feststellung besagt, daß eine Änderung nicht notwendigerweise eine Korrektur beinhaltet, die eine bestehende Modellierung verbessert und deren Ausgangszustand damit hinfällig macht. Im Gegenteil werden Änderungen häufig das Ziel verfolgen, andere Sichtweisen auf die Diskurswelt zu beschreiben, ohne die Absicht, die bisherige Modellierung zu entfernen. Stattdessen können mehrere Perspektiven zur selben Zeit geeignet sein, die Anforderungen jeweils einer Benutzergruppe ideal zu erfüllen, ohne Kompromisse mit den Anforderungen anderer Benutzergruppen eingehen zu müssen. Verschiedene Sichtweisen können sich überlappen, vergleichbar einem Softwaresystem, das sich in verschiedenen Revisionen und Varianten entwickelt.

  Weiterhin werden die Anforderungen verschiedener Benutzergruppen oft in divergierende Richtungen wachsen [Odb94a], so daß mehrere Evolutionsketten entstehen, die verschiedene, aber überlappende Spezifikationen derselben Datenbank reflektieren und die alle verwaltet werden müssen. Ein einzelnes Datenbankschema reicht sicher nicht aus, um alle Perspektiven einer umfangreichen Diskurswelt zu modellieren.

- *Ein- und Auschecken an Knoten föderierter Datenbanksysteme:* Föderierte Datenbanksysteme verwalten neben einer zentralen, gemeinsam genutzten Datenbank auch lokale, privat genutzte Datenbanken auf verschiedenen Knoten eines Rechnernetzwerkes. In [KBGW91] wird eine Einteilung vorgenommen in Systeme, bei denen alle Knoten dasselbe Datenbankschema haben (engl. *single-schema approach*) und solche, bei denen die lokalen Datenbanken eigene Schemata haben können (engl. *multiple-schema approach*). Die Übertragung von Daten aus einer lokalen Datenbank in die Zentrale und zurück geschieht durch das sog. *Ein-* und *Auschecken* (engl. *checkin* und *checkout*). Wenn eine lokale Datenbank ein Schema hat, das von dem der gemeinsamen Datenbank abweicht (multiple-schema approach), dann muß das Schema der lokalen Datenbank vor dem Auschecken von Daten ggf. um deren Schema angereichert werden. Entsprechend muß das Schema der zentralen Datenbank eventuell beim Einchecken von Daten erweitert werden.

- *Angleichung und Integration vormals isolierter Systeme:* Die Notwendigkeit zur Anpassung von Datenbankschemata entsteht nicht nur in eng begrenzten Anwendungbereichen, welche nur Applikationen mit einem hohen Verwandtschaftsgrad enthalten. Auch thematisch weiter voneinander entfernt angesiedelte Applikationen können der Anpassung bedürfen, beispielsweise, um die Wiederverwendung elementarer Klassen oder die Kooperation verschiedener Systeme zu ermöglichen.

Als Extremfall ist hierbei die betriebliche Zusammenlegung organisatorischer Einheiten zu sehen. In der Regel ist damit auch die Forderung nach einer Integration zuvor isolierter Informationssysteme verbunden. Zur Vorbereitung der Integration wird in einer Konsolidierungsphase eine Angleichung der Schemata der zu integrierenden Datenbanken angestrebt [BP95]. Das Ziel dieser Konsolidierung ist insbesondere die Erreichung einer einheitlichen Modellierung gemeinsamer Aspekte der Diskurswelt in den zu integrierenden Teilschemata. Damit kann die sich anschließende Integrationsphase durch eine Vereinigung der Teile sowohl auf Schema- als auch auf Objektebene erreicht werden. Gemeinsame Aspekte der Einzelsysteme sind auf diese Weise nach der Integration wunschgemäß nur einmal vorhanden.

Den oben genannten Argumenten analoge Gründe gibt auch [GTC+90] an. In [Ari91] ordnet Gad Ariav die Umstände, die zur Durchführung von Schemaänderungen führen können zeitlich wie folgt zu: Erzeugung (engl. *Initiation*), Erweiterung (engl. *Extension*), Anpassung (engl. *Redefinition*), Integration (engl. *Integration*) und Beendigung (engl. *Termination*). Die in [HKV94] untersuchte Änderung des zugrunde liegenden Datenmodells fällt nicht in den Bereich der Schemaevolution und bildet daher keinen Gegenstand unserer Betrachtungen.

Wir haben hier eine Auflistung verschiedener Umstände erstellt, die die Durchführung von Schemaänderungen auch bei bereits in Betrieb befindlichen Datenbanken erforderlich machen. Davon ausgehend werden wir im nächsten Abschnitt analysieren, welche konkreten Anforderungen an ein System zur Unterstützung von Schemaänderungen sich aus den dargestellten Umständen ableiten lassen.

3.2 Technische Teilziele zur Beschreibung der Anforderungen

Das Ziel dieser Arbeit ist die Entwicklung von Konzepten, mit denen Datenbanksysteme evolutionäre Prozesse in den von ihnen modellierten Diskurswelten reflektieren können. Demzufolge muß ein Datenbankschema, wie es typischerweise zur technischen Beschreibung einer Diskurswelt verwendet wird, im Verlaufe der Evolution immer wieder an Veränderungen der Diskurswelt angepaßt werden.

Dieser Abschnitt beschreibt die Anforderungen an ein Datenbanksystem, das Schemaänderungen unterstützt (engl. *schema modification management, SMM*). Dabei betrachten wir insbesondere solche Veränderungen, die die im Schema einer Datenbank spezifizierte Struktur der gespeicherten Daten betreffen. Die Konsequenzen solcher Schemaänderungen beschränken sich jedoch nicht auf die Schemaebene sondern betreffen gleichermaßen Objekte und Applikationen, wobei insbesondere die Effizienz und die einfache Handhabung von Datenbanksystemen beeinträchtigt werden können.

Die Anforderungen werden hier in Form technischer Teilziele formuliert, die entsprechend der genannten Konsequenzen von Schemaänderungen den folgenden groben Bereichen zugeordnet werden: Ziele bezüglich der Durchführung von Änderungen auf Schemaebene, Ziele bezüglich der davon betroffenen Applikationen, Ziele bezüglich der Konsequenzen auf Objektebene und Ziele bezüglich Effizienz und Handhabung. Wir gehen in den folgenden Abschnitten auf die vier genannten Kategorien näher ein. Die hier zu gebenden Erläuterungen sind allgemein gehalten,[14] um keine Konzepte für ihre Lösung zu implizieren und somit auch einen Vergleich mit Zielen der Literatur vornehmen zu können.

[14] Wenn wir hier stellenweise doch von Schemaversionen sprechen, so sind damit lediglich die Zustände vor und nach der Durchführung von Schemaänderungen gemeint.

3.2.1 Flexibilität bei der Durchführung von Schemaänderungen

Zunächst beschränken wir unsere Betrachtungen der Spezifikation von Änderungen auf die Schemaebene. Die hier verfolgten Ziele basieren im wesentlichen auf dem Verständnis des Schemaentwicklungs- und -änderungsprozesses als eine komplexe Entwurfsaufgabe, zu deren Unterstützung aus dem Bereich der Softwareentwicklung bekannte Mechanismen einzusetzen sind.

Technisches Teilziel 3.1 {Angebot an Schemaänderungsprimitiven}
Eine vollständige Taxonomie von Primitiven sollte angeboten werden, um beliebige Änderungen an einem vorliegenden Schema durchführen zu können.

Veränderte Anforderungen an ein Schema sollten nicht notwendigerweise eine komplett neue Spezifikation desselben erfordern. Stattdessen muß die Möglichkeit bestehen, ein vorhandenes Schema zu modifizieren und damit an veränderte Anforderungen anzupassen. Grundsätzlich kann die Veränderung eines Datenbankschemas innerhalb oder außerhalb eines Datenbanksystems geschehen; wir sprechen demzufolge von *internen* bzw. von *externen Schemaänderungen* oder auch vom *internen* bzw. vom *externen Ansatz* zur Durchführung von Schemaänderungen.

Ein Datenbanksystem, das interne Schemaänderungen erlaubt, muß eine Menge von Operationen anbieten, die wir in dieser Arbeit *Schemaänderungsprimitive* (engl. *schema update primitives*) nennen. Die Spezifikation eines solchen Primitives beschreibt quasi ein Delta, um das ein gegebenes Schema zu ändern ist und seine Ausführung durch einen Schemaentwickler bewirkt die entsprechende Veränderung des im Datenbanksystem intern verwalteten Schemas.

Ein Datenbanksystem, das externe Schemaänderungen erlaubt, muß in der Lage sein, eine textuelle Beschreibung eines Schemas einzulesen und auszugeben. Dabei sollte die für die Ausgabe eines Schemas verwendete Sprache selbstverständlich eine Teilmenge der Sprache sein, die das Datenbanksystem als Eingabe akzeptiert. Die Durchführung einer externen Schemaänderung geschieht i.Allg. in drei Schritten. Zunächst wird das gerade intern verwaltete Schema vom Datenbanksystem ausgegeben. Die dabei entstandene, textuelle Schemabeschreibung kann dann beispielsweise mit einem herkömmlichen Texteditor außerhalb der Kontrolle des Datenbanksystems in beliebiger Art und Weise verändert werden. Sind die Änderungen abgeschlossen, so muß die veränderte Schemabeschreibung vom Datenbanksystem wieder eingelesen werden. Dabei ist neben der Syntax der Schemabeschreibung auch die Konsistenz des beschriebenen Schemas zu prüfen. Im Erfolgsfalle wird das zuvor intern verwaltete Schema durch das eingelesene ersetzt, anderenfalls wird die Änderung komplett abgelehnt und das vorherige Schema beibehalten.

Die Verwendung spezieller Schemaänderungsprimitive bietet gegenüber externen Schemaänderungen zahlreiche Vorteile. Zum einen betreffen einzelne Schemaänderungen oft zahlreiche Teile eines Schemas. Das Umbenennen einer Klasse beispielsweise erfordert überall dort Anpassungen, wo diese Klasse als Ober- oder als Komponentenklasse Verwendung findet. Solche Anpassungen können nicht nur umfangreich, sondern auch sehr komplex sein. Die Anpassung lokal definierter Attribute einer Klasse (und ihrer Unterklassen) bei Hinzufügen oder Entfernen einer Oberklasse erfordert insbesondere bei Mehrfachvererbung die Berücksichtigung zahlreicher Faktoren. Während ein vom Datenbanksystem angebotenes Schemaänderungsprimitiv die genannten korrigierenden Maßnahmen automatisch oder nach Rückfrage beim Schemaentwickler vollständig und korrekt durchführen kann, ist eine manuelle Anpassung in der Praxis oft nicht durchführbar.

Zum anderen kann die Konsistenz des Schemas nach jeder internen Änderung sofort automatisch geprüft werden. Bei externen Schemaänderungen hingegen wird die Konsistenzprüfung erst nach Abschluß aller Änderungen durchgeführt, was eine frühzeitige Erkennung von Problemen unmöglich macht. Dadurch kann unnötige Mehrarbeit des Schemaentwicklers entstehen. Die sofortige Konsistenzprüfung bei internen Schemaänderungen bedingt aber nicht notwendigerweise, daß jeder bei Ausführung einer Liste von Schemaänderungen entstehende Zwischenzustand des

Schemas konsistent sein muß. Stattdessen können temporäre Inkonsistenzen beispielsweise in Form von Unvollständigkeiten (benutzte Komponenten sind noch nicht spezifiziert) geduldet werden. Dazu kann eine Liste primitiver Schemaänderungen mittels eines Transaktionskonzeptes zu einer komplexen Gesamtoperation zusammengefaßt werden, wobei die Konsistenz erst am Transaktionsende gegeben sein muß [SGD93].

Weiterhin bietet der interne Ansatz den Vorteil, daß das System die mit den durchgeführten Änderungen verbundene Semantik kennt und daraus entsprechende Konsequenzen ziehen kann. Dies ist beispielsweise für die Analyse der durchgeführten Änderungen [Sjø93a, Sjø93b] oder für die Anpassung existierender Objekte der Datenbank notwendig. Dahingegen sind bei Verwendung des externen Ansatzes gewisse, semantisch bedeutungsvolle Unterschiede nicht erkennbar. Beispielsweise kann nicht unterschieden werden, ob ein Attribut einer Klasse im Verlaufe einer Änderung lediglich umbenannt wurde, oder ob es gelöscht und ein neues Attribut hinzugefügt wurde. Dabei besteht nur im erstgenannten Fall eine Beziehung zwischen dem vorherigen und dem späteren Attribut. Dieser Unterschied wird bei der Behandlung existierender Objekte der Datenbank jedoch eine erhebliche Bedeutung erlangen.

Damit sich ein Schemaentwickler auf die Durchführung interner Schemaänderungen beschränken und somit die beschriebenen Vorteile nutzen kann, ist eine gewisse Vollständigkeit der angebotenen Primitive notwendig. Diese bezieht sich zum einen auf die Schema-, zum anderen aber auch auf die Objektebene [CPLZ92d]. Auf der Schemaebene sind minimal Primitive zum Anlegen und Löschen aller Schemakomponenten anzubieten. Damit ist die Schemaänderungstaxonomie *vollständig* im Sinne der von Banerjee et al. in [BKKK87] gegebenen Definition,[15] d.h. jede beliebige Schemaversion kann in jede beliebige andere Schemaversion überführt werden. Dies kann trivialerweise stets durch vollständiges Löschen des gegebenen Schemas und anschließende, vollständige Spezifikation des gewünschten Schemazustandes erreicht werden. Damit werden allerdings die oben genannten Vorteile bezüglich der Semantik der Schemaänderungsoperationen nicht erreicht. Genaugenommen stehen dann ja auch keine eigentlichen Änderungsprimitive zur Verfügung, sondern nur solche zum Anlegen und Löschen. Demzufolge muß eine Schemaänderungstaxonomie zur Erreichung der obigen Vorteile über den von Banerjee et al. geforderten Umfang einer vollständigen Taxonomie hinausgehende Primitive anbieten. Dazu gehören beispielsweise Primitive zum Umbenennen von Schemakomponenten. Damit kann dann nämlich erst zwischen dem Löschen einer vorhandenen Schemakomponente und dem Anlegen einer neuen Schemakomponente einerseits und dem Umbenennen einer Schemakomponente andererseits unterschieden werden. Nur im zweiten Fall hat die alte Schemakomponente etwas mit der neuen zu tun, es handelt sich dann nämlich tatsächlich um *dieselbe* Komponente. Im Falle der Löschung einer alten und späteren Erzeugung einer neuen Komponente muß davon ausgegangen werden, daß diese beiden nichts miteinander zu tun haben, d.h. daß es sich um zwei verschiedene Schemakomponenten handelt, zwischen denen keinerlei Ableitungsbeziehung besteht.

Der Vorteil der Erkennung zusätzlicher Semantik kann allerdings nur dann erreicht werden, wenn der Schemaentwickler die Primitive auch entsprechend des beschriebenen Verwendungszweckes einsetzt. Er muß in dem genannten Beispiel also zwischen Löschen und Anlegen einerseits und Umbenennen andererseits unterscheiden und jeweils den seiner Semantik entsprechend richtigen Weg zur Veränderung eines gegebenen Schemazustandes in einen neuen wählen.

Über diese Schemaänderungsprimitive im engeren Sinne hinausgehend, sind weitere, sog. *komplexe Schemaänderungsoperationen* sinnvoll. Deren Resultat könnte einschließlich der aus der Durchführung erhaltenen Ableitungsbeziehungen bei einer isolierten Betrachtung der Schemaebene in gleicher Weise auch durch Verwendung der bisherigen Primitive erreicht werden. Auf der Objektebene können diese komplexen Schemaänderungsoperationen allerdings mit einer bisher

[15] Wir werden die von Banerjee et al. für das Datenbanksystem ORION realisierte Taxonomie in Abschnitt 4.3.1.1 vorstellen.

nicht ausdrückbaren Semantik verbunden werden. Ein typisches Beispiel hierfür sind etwa die Teleskopoperatoren **nest** und **unnest**, die Attribute entlang von Aggregationsbeziehungen zwischen Klassen verschieben. Diese erreichen auf Schemaebene zunächst keine andere Wirkung als das Löschen eines Attributes aus einer Klasse und das Hinzufügen eines gleichnamigen Attributes mit demselben Typ in einer anderen Klasse. Auf Objektebene macht es aber einen erheblichen Unterschied, ob bei der Propagation eines Objektes ein Attributwert eines anderen Objektes übertagen oder ob einfach nur ein Defaultwert zugewiesen wird. Auch hier gilt natürlich wieder, daß der Schemaentwickler um die Semantik der Primitive wissen und diese entsprechend korrekt einsetzen muß. Dabei kann ihm auch nicht geholfen werden, da das Datenbanksystem natürlich nicht um die Semantik durchzuführender Schemaänderungen wissen kann. Es kann diese jedoch aufgrund von Heuristiken vermuten. Wird in einer neuen Schemaversion etwa eine Klasse angelegt, die einer Klasse einer älteren Schemaversion bezüglich Name, Oberklassen, Attribute (und deren Typen) sowie Methoden (und deren Signatur) ähnelt oder gar gleicht, so könnte man von der Vermutung ausgehen, daß es sich in der Tat um zwei Versionen derselben Klasse handelt und den Schemaentwickler auf diesen Umstand aufmerksam machen.

Eine aus unserer Sicht vollständige Schemaänderungstaxonomie enthält damit Primitive, die drei verschiedenen Kategorien zugeordnet werden können:

- Primitive, die bei isolierter Betrachtung von Schemaversionen jede beliebige Änderung erlauben und damit die Vollständigkeit der Taxonomie entsprechend Banerjee et al. [BKKK87] erreichen. Hierzu genügen bereits Primitive zum Anlegen und Löschen der verschiedenen Schemakomponenten.
- Primitive, die die Integration von Schemakomponenten anderer Schemaversionen oder Schemaänderungen im engeren Sinne, also beispielsweise Umbenennungen, durchführen. Damit wird die Erreichung desselben Zielschemas auf verschiedenen Wegen möglich, was dem Schemaentwickler erlaubt, eine gewisse Semantik mit seiner Schemaänderung auszudrücken, die vom Datenbanksystem durch die Etablierung von Ableitungsbeziehungen zwischen Schemaversionen manifestiert wird.
- Primitive, die im Vergleich zu den obigen bezüglich ihrer Auswirkungen nicht mehr auf Schema- wohl aber auf Objektebene hinausgehen. Diese Gruppe umfaßt u.a. die komplexen Schemaänderungsoperationen.

□

Technisches Teilziel 3.2 $\left\{ \begin{array}{l} \textbf{uneingeschränkte Ausführbarkeit} \\ \textbf{sämtlicher Schemaänderungen} \end{array} \right\}$

Sämtliche Schemaänderungsprimitive müssen zu jeder Zeit ausführbar sein, wenn sie die Konsistenz des Schemas erhalten. Es sollten keine durch die zur Unterstützung der Schemaevolution gewählten Konzepte bedingten Beschränkungen existieren.

Da nicht nur zur Veränderung sondern ebenso zur initialen Erzeugung eines Schemas eine Schemabeschreibungssprache (engl. *object definition language, ODL*) benötigt wird, sollte diese sowohl erzeugende als auch verändernde Primitive enthalten. Diese Integration von erzeugenden und modifizierenden Primitiven in einer einzigen Schemabeschreibungssprache ist durch zwei weitere Aspekte motiviert. Zum einen kann man stets ein leeres Schema als Grundlage vorgeben und damit auch die initiale Erzeugung des ersten vom Benutzer spezifizierten Schemazustandes als Änderung eben des vorgegebenen leeren Schemas interpretieren.[16] Damit wird die Notwendigkeit der Unterscheidung zwischen Erzeugung und Veränderung hinfällig, was nicht nur die

[16] Wir werden dies später tun, indem wir von einer systemdefinierten Schemaversion sv_0 ausgehen, die keinerlei benutzerdefinierte Komponenten enthält.

Beschreibung sondern auch die Benutzung eines Systems vereinfacht. Unser Begriff der Schemaänderungsprimitive schließt daher auch erzeugende Operationen ein. Zum anderen sind zur Veränderung eines Schemas oft Erzeugungen von Komponenten verschiedener Ebenen (Klassen in einem Schema, Attribute in einer Klasse, etc.) notwendig. Die Primitive zur Erzeugung der Komponenten müßten also ohnehin sowohl zur Erzeugung als auch zur Veränderung eines Schemas angeboten werden. Damit würde sich im Falle zweier getrennter Sprachen eine weitreichende Überlappung ergeben, welche sich auf mehrere Ebenen eines Schemas erstreckt. Lediglich auf der Ebene des gesamten Schemas würden sich die beiden Sprachen unterscheiden. Insbesondere bei der späteren Betrachtung unserer Schemabeschreibungssprache wird klar werden, daß eine Trennung nicht sinnvoll ist. Daher fordern wir eine Schemabeschreibungssprache, die Erzeugungs- und Änderungsprimitive auf allen Ebenen integriert.

Die Menge der angebotenen Schemaänderungsprimitive muß dabei zumindest in dem Sinne vollständig sein, daß jeder beliebige Schemazustand erzeugt werden kann. Das bedeutet, jeder gegebene Schemazustand muß in jeden gewünschten Schemazustand transformierbar sein. □

Technisches Teilziel 3.3 $\{$**Durchführung von Änderungen auf Schemaebene**$\}$
Die Anwendung der Schemaänderungsprimitive führt einen Schemazustand in einen zweiten (nicht notwendigerweise verschiedenen) Schemazustand über. Die Primitive operieren also auf Schemaebene, nicht auf einzelnen Klassen.

Die technischen Teilziele 3.1 und 3.2 betreffen die Existenz und Anwendbarkeit einer Menge von Schemaänderungsprimitiven. Darüber hinausgehend ist eine Forderung hinsichtlich der Qualität der Primitive angebracht. Diese legt die Granularität der Primitive auf die Schemaebene fest, d.h. ein Primitiv setzt auf einer kompletten Beschreibung eines vorgegebenen Schemazustandes auf, verändert diesen im Erfolgsfalle und ergibt wiederum eine vollständige Schemabeschreibung [Odb95]. Schemaänderungsprimitive, die lediglich auf Komponenten eines Schemas, also beispielsweise auf Klassenebene operieren, sind aus drei Gründen nicht wünschenswert, die wir an dieser Stelle kurz erläutern.

Zum einen ist das Schema die Einheit der Beschreibung struktureller und verhaltensmäßiger Datenbankeigenschaften. Aus der Perspektive eines Applikationsentwicklers ist ein Datenbankschema in dem Sinne atomar, daß isolierte Teile daraus notwendigerweise unvollständig sind und daher keinen eigenständigen Nutzen haben. Insbesondere die zwischen Klassen bestehenden Generalisierungs- und Aggregationsbeziehungen können bei der Betrachtung isolierter Klassen nicht berücksichtigt werden. Demzufolge sollten Schemaänderungen analog zu Applikationen auf Schemata insgesamt aufsetzen und nicht auf einzelnen Bruchstücken daraus, d.h. das Schema sollte auch die Einheit für die Beschreibung von Änderungen sein. Somit resultiert jede Schemaänderung in einem konsistenten, also insbesondere in einem vollständigen Schema, das wie üblich von Applikationen benutzt werden kann. Werden Schemakomponenten hingegen separat modifiziert, so ist eine Konfigurationsverwaltung notwendig [Cla92], damit eine Applikation eine konsistente Menge von Schemakomponenten (also beispielsweise von Ausprägungen verschiedener Klassen) als Schema sieht.

Desweiteren können auch solche Konsistenzverletzungen, die erst bei der gemeinsamen Betrachtung mehrerer Klassen zutage treten, bei Änderungen auf Schemaebene sofort erkannt werden. Beispielsweise das Umbenennen des Attributes `first_name` einer Klasse `Person` zu `name` könnte bei isolierter Betrachtung als korrekt empfunden werden, obwohl eine Unterklasse `Employee` existiert, die ein Attribut `name` zur Modellierung der Position eines Angestellten lokal definiert hat und somit ein Vererbungskonflikt vorliegt. Anders als bei Betrachtung der Schemaebene können derartige, mehrere Klassen betreffende Konsistenzverletzungen bei Beschränkung auf die Klassenebene erst zum Zeitpunkt der Übersetzung einer Applikation festgestellt werden. Erst dann wird in solchen Systemen nämlich die zu verwendende Konfiguration von Ausprägungen

verschiedener Klassen spezifiziert und zwar nicht durch den Schemaentwickler, sondern durch den Applikationsentwickler, d.h. letztlich durch die Anforderungen einer einzelnen Applikation. Derartige Komplikationen vermeidet die Arbeit auf Schemaebene dadurch, daß das gesamte zu verwendende Datenbankschema vor Beginn der Applikationsentwicklung feststeht und seine Konsistenz garantiert ist.

Schließlich können Konzepte, die sich auf die Modifikation von Schemakomponenten beschränken, keine simultanen Änderungen an mehreren solcher Komponenten handhaben. Beispiele für Änderungen, die mehrere Komponenten gleichzeitig betreffen, sind etwa das Aufteilen oder Zusammenlegen von Klassen, das Verschieben von Eigenschaften entlang von Vererbungs- oder Aggregationsbeziehungen, etc. Im Bereich der Programmiersprachen werden solche Restrukturierungsmaßnahmen für Klassenhierarchien als sehr wichtig erachtet [Cas95, JF88]. OTGen und Tess [LH90, Ler94, Ler96, Ler97, Ler00] (siehe Abschnitt 4.3.4.4) diskutieren einige Aspekte von Änderungen auf Schemaebene. □

Technisches Teilziel 3.4 {Möglichkeit externer Schemaänderungen}
Die Möglichkeit zur Durchführung von Schemaänderungen entsprechend dem externen Ansatz sollte gegeben sein.

Idealerweise werden Schemaänderungen in einem Datenbanksystem ausschließlich durch den internen Ansatz, d.h. durch die Ausführung der angebotenen Primitive bewirkt, weil diese unter der Kontrolle des Datenbanksystems durchgeführt werden und so das benötigte Wissen über Zusammenhänge zwischen den Komponenten des Schemas vor und nach einer Änderung erfaßt werden kann. Die Spezifikation einer neuen Schemaausprägung geschieht dabei nämlich gleichsam durch Angabe einer Differenz (eines Deltas) zu der vorliegenden Fassung des Schemas. Die Angabe dieser Differenz erfolgt implizit durch Anwendung der Schemaänderungsprimitive und findet unter der Kontrolle des Datenbanksystems statt. Damit kann das Datenbanksystem bei Verwendung des internen Ansatzes nach jeder Teiländerung die Konsistenz prüfen, dem Schemaentwickler ggf. Verletzungen melden und Strategien zu deren Behebung anbieten.

Die angebotene Menge von Schemaänderungsprimitiven für den internen Ansatz ist allerdings vom Datenbanksystem fest vorgegeben und kann, auch wenn sie entsprechend obiger Interpretation (siehe Teilziel 3.1) als vollständig bezeichnet werden darf, nie alle spezifischen Anforderungen der verschiedensten Einsatzgebiete eines Datenbanksystems optimal abdecken [CJR98]. Ein weiterer Grund für den Wunsch, Schemaänderungen unabhängig von den Primitiven des Datenbanksystems durchführen zu wollen, kann darin bestehen, daß ein bestimmtes, externes Werkzeug zur Erstellung von Datenbankschemata verwendet werden soll. Alternativ zu der in [CJR98] vorgestellten Idee einer dynamisch zur Laufzeit erweiterbaren Bibliothek von, insbesondere auch komplexen Schemaänderungsoperationen (in [CJR98] *templates* genannt) sollte zumindest konzeptionell die Möglichkeit bestehen, eine Schemaänderung außerhalb der Kontrolle des Datenbanksystems durchzuführen. Abbildung 3.5 zeigt die drei wesentlichen Schritte auf Schemaebene. Zunächst ist der im Datenbanksystem verwaltete Schemazustand in eine Datei des Betriebssystems auszugeben (1). Diese wird dann in beliebiger Art und Weise außerhalb der Kontrolle des Datenbanksystems modifiziert (2) und schließlich wieder in das System eingebracht (3).

Der Entnahmeprozeß (1) beinhaltet hier die Ausgabe des Schemas oder von Teilen daraus in textueller Form entsprechend der Schemabeschreibungssprache. Durch die unkontrollierte Durchführung der Veränderungen an der textuellen Schemabeschreibung, die z.B. mit einem herkömmlichen Texteditor durchgeführt werden kann, ist die Konsistenz des beschriebenen Schemas in keinster Weise garantiert und muß beim Einbringen (3) zunächst geprüft werden.

Ein weiteres Problem beim externen Ansatz ist, daß das System dabei keinerlei Information darüber erhält, wie die neue Schemaausprägung aus der bisherigen hervorgegangen ist. Dieser

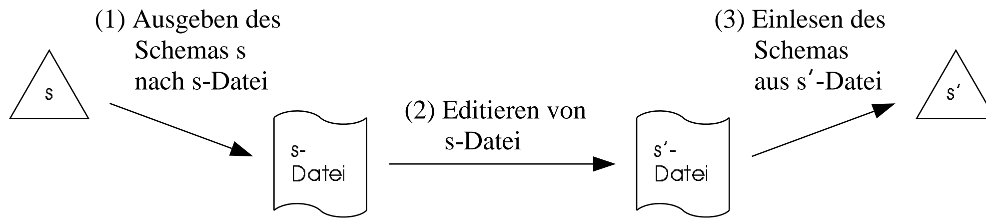


Abbildung 3.5: Vorgehensweise bei der Durchführung externer Schemaänderungen.

semantische Zusammenhang ist jedoch bei Schemaänderungen in Datenbanksystemen besonders wichtig, da sich hieraus Konsequenzen für die Behandlung gespeicherter Objekte ergeben. Hier ist die Unterstützung des Schemaentwicklers durch ein Werkzeug sinnvoll, das die beiden Schemaausprägungen vergleicht, Unterschiede und Gemeinsamkeiten analysiert und dem Schemaentwickler bei der Spezifikation der semantischen Beziehungen hilft, z.B. indem es potentiell sinnvolle Vorschläge macht. Ein solches Werkzeug kann jedoch nie ohne die Interaktion mit dem Schemaentwickler auskommen, da die semantischen Beziehungen zwischen den Komponenten der alten und der neuen Schemaausprägung nicht vollkommen automatisch erkannt werden können. □

Technisches Teilziel 3.5 {Erhaltung bisheriger Schemazustände und Rücksetzbarkeit}
Vorhandene Schemazustände sollten bei der Durchführung von Schemaänderungen unverändert erhalten bleiben und insbesondere das Rücksetzen auf vorherige Zustände ist zu ermöglichen.

Aufgrund von Umfang und Komplexität eines Schemas kann dessen Änderung ein schwieriger Prozeß sein, dessen Konsequenzen im voraus nicht immer vollständig überschaubar sind. Daher wird die Durchführung einer beabsichtigten Schemaänderung i.Allg. nicht immer auf Anhieb glücken. Wie oben dargestellt, machen Schemaänderungen generell korrigierende Maßnahmen notwendig, welche sich auf andere Teile des Schemas, auf Objekte der Datenbank und auf existierende Applikationen beziehen können. Insbesondere durch diese „Seiteneffekte“ können unvorhergesehene Probleme auftreten, die die Durchführung beabsichtigter Schemaänderungen als nachteilig oder gar untragbar erkennen lassen. Solche Nachteile sind zum Teil erst nach der Durchführung der Änderung, eventuell sogar erst mehrere Evolutionsschritte später erkennbar. Daher müssen zum einen alle Zwischenschritte, d.h. alle bisherigen Schemazustände unverändert erhalten bleiben. Insbesondere müssen bestehende Zustände bei der Erzeugung neuer Schemazustände unverändert bleiben. Die Erhaltung kann zum einen der Dokumentation früherer Schemazustände und der durchlaufenen Evolution dienen. Zum anderen kann damit die dringend erforderliche Möglichkeit realisiert werden, jederzeit auf beliebige der bisherigen Zustände zurückzusetzen und damit die gemachten Änderungen gleichsam rückgängig zu machen. Dies schließt natürlich die durch eine Schemaänderung modifizierten Objekte ein, d.h. die Rücksetzbarkeit muß auch auf der Objektebene gewährleistet sein.[17] Wir werden auf diesen Punkt in Abschnitt 3.2.3 zurück kommen. Entsprechendes gilt für die Ebene der Applikationen. Dort wird jedoch in der Regel ohnehin keine automatische Anpassung durchgeführt (siehe Abschnitt 3.2.2). □

Technisches Teilziel 3.6 {Ableitung alternativer Schemaversionen}
Neue Schemaversionen sollten von beliebigen, existierenden Schemaversionen ableitbar sein, so

[17] Die Rücksetzbarkeit auf Objektebene fordern wir allerdings nur für durch Schemaänderungen verursachte Anpassungen in der Datenbank. Änderungen der Datenbank, die Applikationen durch erfolgreich beendete Transaktionen mit herkömmlichen, schreibenden Zugriffen auf Objekte durchführen, sind hier nicht gemeint. Derartige Änderungen reflektieren nämlich Zustandsänderungen in der Diskurswelt und sollten idealerweise auch nach dem Rücksetzen auf einen früheren Schemazustand erhalten bleiben.

daß neben historischen, aufeinanderfolgenden auch alternative, nebeneinander gültige Modellierungen erstellt und verglichen werden können.

Die Erstellung eines Datenbankschemas ist ein komplexer Entwurfsprozeß. Dies gilt insbesondere in objektorientierten Datenbanksystemen, da diese zahlreiche und mächtige Modellierungskonzepte anbieten, um Struktur und Verhalten einer gegebenen Diskurswelt zu modellieren. Dabei sind im Entwurfsprozeß zwei Motivationen für Änderungsanforderungen zu unterscheiden, die als direkte Konsequenzen der problemimmanenten Komplexität verstanden werden können. Wir gehen im folgenden kurz auf diese beiden Motivationen ein.

Zum einen stellt ein komplexer Entwurf stets eine Aufgabe dar, bei deren Bewältigung anfängliche Schwachstellen und Fehler fast unvermeidlich sind. Dies liegt im Umfang und im Detaillierungsgrad des zu erstellenden Modells begründet. Eine weitere Ursache für das Entstehen von Diskrepanzen zwischen der vorgegebenen Diskurswelt und dem dafür erstellten Modell kann in nachträglichen Veränderungen der Diskurswelt liegen, an die das Modell dann anzupassen ist. Unabhängig von der Ursache der Diskrepanzen ist für die hier besprochene Motivation einer Änderungsanforderung charakteristisch, daß unter den Benutzern eines Datenbanksystems Konsens über das Vorliegen einer Diskrepanz besteht. Ihre Beseitigung wird also stets als qualitativer Fortschritt des Datenbankschemas bewertet. Wir sagen in solchen Fällen auch, Änderungen aus den genannten Gründen haben *korrigierenden* Charakter.

Zum anderen kann eine Änderungsanforderung an einem Datenbankschema dadurch begründet sein, daß verschiedene Benutzergruppen unterschiedliche Sichtweisen der Diskurswelt haben, beispielsweise weil ihre Anwendungen unterschiedliche Schwerpunkte besitzen. Hier ist charakteristisch, daß kein Konsens aller Benutzer besteht und demzufolge auch nicht generell über die Korrektheit eines Schemas geurteilt werden kann. Stattdessen haben Benutzergruppen unterschiedliche Anforderungen an die Speicherung und Verarbeitung von Informationen der Diskurswelt.

Aus den beiden genannten Motivationen entstehen zwei Anforderungen an ein System zur Unterstützung von Schemaänderungen [Odb95].

Bei Änderungen mit korrigierendem Charakter liegt eine Diskrepanz zwischen Diskurswelt und modellierendem Schema vor. Durch deren Beseitigung entsteht ein neuer Schemazustand, der als ein verbesserter Nachfolger des vorherigen Zustandes betrachtet werden kann und der seinen Vorgänger ersetzt. Damit entsteht im Laufe mehrerer Änderungen eine *Historie* von entsprechend ihrer Entstehungszeit linear angeordneten Zuständen (siehe sv_1, sv_2, sv_3 in Abbildung 3.6a).

Abbildung 3.6: Abstrakte Darstellung der Ableitung neuer Schemazustände.

Liegt die Ursache für eine Änderungsanforderung jedoch in unterschiedlichen Perspektiven einer gemeinsamen Diskurswelt begründet, so macht die Änderung den bisherigen Zustand nicht hinfällig. Demzufolge kann dieser nicht einfach durch das Resultat der Veränderung ersetzt werden. Stattdessen müssen die Modellierungen der unterschiedlichen Perspektiven als *Alternativen* ne-

beneinander existieren, ohne daß zwischen ihnen automatisch eine semantikbehaftete Anordnung erzeugt werden kann (siehe sv_1, sv_2, sv_3 in Abbildung 3.6b).

Um dem zuletztgenannten Aspekt Rechnung zu tragen, muß die Möglichkeit bestehen, nicht nur die neueste sondern alle existierenden Schemazustände zu modifizieren, so daß eine baumartige Ableitungsstruktur zwischen den verschiedenen Schemazuständen entsteht. Dies erlaubt für viele Anwendungsgebiete eine natürlichere Modellierung [MS92, Mon93, MS93]. □

Technisches Teilziel 3.7 {Integration alternativer Schemaversionen}
Die Integration von Teilen alternativer Schemaversionen ist zu ermöglichen.

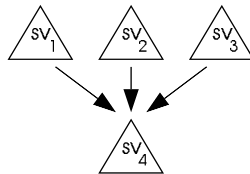


Abbildung 3.7: Abstrakte Darstellung der Integration alternativer Schemazustände.

Durch die Möglichkeit der Ableitung alternativer Schemazustände können diese unabhängig voneinander entwickelt und dann auch miteinander verglichen werden. Dabei wird in aller Regel eine Situation entstehen, in der keine der Alternativen alle Anforderungen einer Benutzergruppe optimal erfüllt. Stattdessen werden gewisse Ausschnitte der Modellierung in der einen, andere Ausschnitte in anderen Alternativen vorteilhafter modelliert sein. Damit entsteht die Notwendigkeit der Integration von Ausschnitten bestehender Schemazustände zu einem neuen Zustand (siehe Abbildung 3.7). Dieselbe Anforderung resultiert aus der Kooperation mehrerer Schemaentwickler, also wenn eine Modellierungsaufgabe aufgeteilt wird und verschiedene Personen jeweils andere Ausschnitte der Diskurswelt modellieren. Die Integration von Schemazuständen entspricht damit dem letzten Schritt bei der Konsolidierung verschiedener Modelle beim Datenbankentwurf. Dabei muß natürlich auch hier Einigkeit bei der Interpretation der Modellierungen herrschen, d.h. die Beseitigung von Synonymen, Homonymen, verschiedenen Maßeinheiten etc. muß der Integration vorausgegangen sein. □

Technisches Teilziel 3.8 {Verwaltung der Ableitungsbeziehung}
Die Ableitungsbeziehung, die ausdrückt, wie eine Schemaversion und ihre Komponenten aus bestehenden hervorgegangen sind, stellt eine wichtige Informationsquelle dar und ist entsprechend zu verwalten.

In den Teilzielen 3.6 und 3.7 werden verschiedene Varianten der Erzeugung neuer Schemazustände vorgestellt. Diese umfassen die sequentielle Entwicklung einer Liste historischer Schemazustände, die ggf. paralelle Entwicklung alternativer Schemazustände und die Integration mehrerer Schemazustände zu einem neuen. Die Erzeugung neuer Zustände einer Schemabeschreibung auf der Basis existierender bezeichnen wir auch als *Ableitung*. Bei der Erzeugung eines neues Schemazustandes wird demzufolge eine *Ableitungsbeziehung* etabliert, die in den Abbildungen 3.6 und 3.7 bereits durch Pfeile graphisch veranschaulicht wurde.

Die Ableitungsbeziehung zwischen verschiedenen Zuständen eines Schemas beinhaltet wichtige semantische Informationen und muß daher vom Datenbanksystem verwaltet werden. Diese Informationen dienen nicht nur Zwecken der Dokumentation, sondern wirken sich auch auf die Anpassung existierender Applikationen (siehe Abschnitt 3.2.2) und auf die Behandlung von Objekten der Datenbank (siehe Abschnitt 3.2.3) aus. □

3.2.2 Berücksichtigung von Applikationen

Die beiden in diesem Abschnitt zu besprechenden technischen Teilziele betreffen die Applikationen, die auf die Objekte einer gemeinsamen Datenbank zugreifen. Bei diesen Zugriffen benutzen die Applikationen die im Schema zugesicherten Eigenschaften der Objekte der Datenbasis. Dabei sind Applikationen jedoch normalerweise *statisch*, d.h. sie gehen davon aus, daß das zur Übersetzungszeit der Applikation vorhandene Schema unverändert erhalten bleibt und können daher nicht mehr arbeiten, wenn die zugesicherten Eigenschaften nach einer Schemaänderung nicht mehr in der benötigten Form vorhanden sind.

Es existieren zwar auch *dynamische* Applikationen, die im Gegensatz zu den statischen nicht von bestimmten Eigenschaften des Schemas ausgehen, sondern sich vom Metaschema Informationen über das vorliegende Schema beschaffen. Beispiele hierfür finden sich unter den allgemeinen Datenbankwerkzeugen, wie graphische Schemaeditoren, Programme zum Einlesen und Ausgeben textueller Schemabeschreibungen sowie zur Analyse und Optimierung vorgegebener Schemata. Diesen, mitunter vom Datenbankhersteller mitgelieferten dynamischen Werkzeugen ist jedoch gemeinsam, daß sie ohne weitere Programmierung, die dann wiederum statisch wäre, keine für einen Anwendungsbereich semantisch sinnvollen Operationen mit den Objekten der Datenbank durchführen können. Stattdessen beschränken sie sich auf Zugriffe auf der Schemaebene.

Selbst wenn eine sehr aufwendige dynamische Implementierung für anwendungsspezifische Applikationen in der Praxis in Kauf genommen werden könnte, so würde sich das Problem dabei nur verlagern, nämlich von der Ebene des Schemas auf die des Metaschemas. Dann müßten dort statische Annahmen gemacht werden, ohne die eine Applikation die Semantik einer in ihrer Entwurfsphase noch unvorhersehbaren Schemaänderung gar nicht erfassen könnte.

Technisches Teilziel 3.9 {Vermeidung manueller Applikationsanpassungen}
Der Zwang zur sofortigen Anpassung von Applikationen nach Schemaänderungen ist aufzuheben, so daß der laufende Betrieb eines Datenbanksystems bei Durchführung von Schemaänderungen nicht unterbrochen werden muß.

Im Allgemeinen müssen auch Applikationen an ein verändertes Schema angepaßt werden. Um dieses Problem zu lösen, wäre zum Zeitpunkt einer Schemaänderung eine sofortige Anpassung sämtlicher Applikationen notwendig oder der alte Zustand des Schemas muß aufbewahrt werden und bestehenden Applikationen weiterhin verfügbar bleiben.

Der erstere Ansatz ist jedoch in der Praxis nicht praktikabel. Eine automatische Anpassung (siehe Abschnitt 4.3.4.3) vorhandener Applikationen kann lediglich unter ganz bestimmten, sehr selten anzutreffenden Voraussetzungen durchgeführt werden. Dazu ist es beispielsweise erforderlich, daß dem Datenbanksystem alle darauf operierenden Applikationen (insbesondere inklusive ihres Quellcodes) bekannt gemacht werden und daß der Anwender mit einer sehr begrenzten Menge von Schemaänderungsprimitiven auskommt [Hür95, HS96, LH89, BH93, Ber94, Ber97]. Eine manuelle Anpassung (insbesondere eine sofortige) verbietet sich allein aus Aufwandsgründen, da der Umfang existierender Applikationen sehr groß sein kann. Wenn Quelldateien von Applikationen gar nicht vorliegen, was beispielsweise bei eingekauften Applikationen in der Regel der Fall ist, wird der Ansatz der Applikationsanpassung gänzlich undurchführbar. Wenn Applikationen, wie im Falle von GOODSTEP (siehe Abschnitt 3.1.1) automatisch generiert werden, dann ist deren Anpassung zum einen sehr aufwendig, weil die automatisch erzeugten Quellen nur für eine anschließende Übersetzung, nicht aber für eine spätere, manuelle Anpassung gedacht sind. Dies zeigt sich darin, daß sie normalerweise länger und für Applikationsentwickler schwerer lesbar sind als händisch implementierter Code. Zum anderen ist die manuelle Anpassung der automatisch erzeugten Quellen gar nicht erwünscht, da sie nach jeder Generierung eines Werkzeuges erneut durchgeführt werden muß. Auch die Steuerung der Generierung durch

Vorgabe des zu verwendenden Datenbankschemas wird oft aufgrund mangelnder Flexibilität der Generatoren unmöglich sein. Die dritte und letzte Alternative, nämlich die Anpassung der Generatoren selbst, wird zumeist an dem bereits angesprochenen Problem scheitern, daß die Quellen des Generators nicht verfügbar sind, da es sich um ein gekauftes Werkzeug „von der Stange“ handelt. Liegen allerdings unter besonders glücklichen Umständen sowohl die Quellen des Generators selbst als auch die notwendige Detailkenntnis von seiner Funktionsweise vor, so ist durch eine geeignete Modifikation des Generators sicher eine gewisse Erleichterung manuell noch durchzuführender Applikationsanpassungen möglich. Einschränkend ist hier jedoch festzuhalten, daß der Generator zumindest die *wiederholte Generierung*[18] unterstützen muß und daß sich die Erleichterung nur auf den tatsächlich generierten Anteil der Applikationsquellen beschränkt, d.h. manuell ergänzter Code muß natürlich auch manuell angepaßt werden.

Aus der vorangegangenen Erläuterung alternativer Ansätze wird deutlich, daß zumindest i.Allg. alle von Applikationen benutzten Schemazustände aufbewahrt werden und beliebig lange verwendbar bleiben müssen (siehe technisches Teilziel 3.5).

Obwohl aufgrund der hier genannten Anforderungen keine systembedingten Anpassungen von Applikationen an ein verändertes Schema notwendig sein sollten, wird sich eine Anpassung aufgrund anwendungsspezifischer Anforderungen manchmal trotzdem empfehlen. Insbesondere wenn die Notwendigkeit einer Schemaänderung auf neue Anforderungen an eine bestehende Applikation zurück geht, so wird zumindest diese Applikation nach der Schemaänderung möglichst schnell angepaßt werden, um die Vorteile des veränderten Schemas zur Implementierung der neuen Anforderungen in der Applikation ausnutzen zu können. Dies bedeutet jedoch keine Einschränkung des hier vorgestellten Teilzieles. Dieses fordert nämlich Flexibilität dahingehend, daß solche Anpassungen zum einen nicht für alle existierenden Applikationen erforderlich sind, zumindest nicht aufgrund systembedingter Gründe, und zum anderen, daß die Anpassungen nicht sofort durchgeführt werden müssen.

Dieses Ziel entspricht dem der Typänderungstransparenz (engl. *type change transparency*) von Odberg [Odb95]. □

Die im vorangegangenen Abschnitt dargelegten technischen Teilziele ergeben bereits ein neuartiges Bild von einem Datenbanksystem. Die Summe der Forderungen beschreibt eine Situation, in der mehrere Zustände eines Schemas s koexistieren. Für dieses Datenbankschema s können im Laufe der Zeit zahlreiche Applikationen entwickelt werden, die zunächst auf verschiedenen Zuständen von s aufsetzen. In Kombination mit Ziel 3.9 ergibt sich darüber hinaus, daß diese Applikationen auch dann noch ausführbar bleiben, wenn der jeweils zugrunde liegende Schemazustand nicht mehr aktuell ist. Demzufolge koexistieren Gruppen von Applikationen, die jeweils auf verschiedenen Zuständen des Schemas s aufsetzen, wobei die gleichzeitige Ausführung von Applikationen all dieser Gruppen erlaubt sein soll.

Technisches Teilziel 3.10 {Kooperation}
Applikationen müssen auch dann die Möglichkeit zur Kooperation auf gemeinsamen Daten einer Datenbank haben, wenn sie auf verschiedenen Versionen eines Schemas aufsetzen.

[18] An dieser Stelle sprechen wir von Generatoren, die aus einem irgendwie gearteten Modell ein Generat in Form von Code der Zielsprache erzeugen. In diesen Code sind dann i.Allg. manuell Ergänzungen zu integrieren. Die Eigenschaft der wiederholbaren Generierung stellt sicher, daß diese manuell integrierten Ergänzungen nicht verloren gehen, wenn das Modell verändert und die Generierung erneut durchgeführt wird. Dies kann ein Generator z.B. durch Integration sog. *Codeschutzblöcke* in das Generat erreichen. Diese Codeschutzblöcke kann ein Generator bei wiederholter Generierung wieder erkennen und den durch sie eingeschlossenen, manuell integrierten Code in das neue Generat übertragen. Änderungen außerhalb der Codeschutzblöcke sind nicht erlaubt. Die wiederholbare Generierung stellt also eine Vorstufe zur Technologie des Reverse-Engineering dar. Diese erlaubt beliebige Änderungen im Code und führt ggf. notwendige Anpassungen des Modells sogar automatisch durch.

Ein zentraler Zweck jedes Datenbanksystems ist die Speicherung und Verwaltung gemeinsam genutzter Daten, so daß verschiedene Applikationen darauf kooperieren können. Dabei muß insbesondere unter Berücksichtigung möglicher nebenläufiger Zugriffe Konsistenzerhaltung garantiert werden. In der hier zu betrachtenden Situation liegen insbesondere Teilziel 3.9 zufolge gleichzeitig mehrere Zustände einer Schemaspezifikation vor, die i.Allg. von Applikationen verwendet werden, die auf verschiedenen dieser Zustände aufsetzen.

Selbst für Applikationen verschiedener Zustände eines Schemas muß die Möglichkeit zur Kooperation gegeben sein. Ein physikalisch gespeichertes Objekt entspricht zunächst nur einem einzigen Typ und wird daher i.Allg. nicht unter allen Zuständen eines Schemas korrekt interpretiert werden können. Dadurch können sich auch die Mengen der Objekte, die durch verschiedene Zustände eines Schemas zugreifbar sind, voneinander unterscheiden. Da sich der Typ einer Klasse von einer Schemabeschreibung zur nächsten ändern kann, kann sich sogar die Extension einer Klasse beim Zugriff über verschiedene Schemazustände unterschiedlich darstellen. □

3.2.3 Flexibilität bei der Propagation von Schemaänderungen auf die Objektebene

Wie bereits dargestellt, hängen in einem Datenbanksystem verschiedene Komponenten vom Schema ab und demzufolge sind dort nach Schemaänderungen korrigierende Maßnahmen erforderlich. Nachdem in den beiden vorangegangenen Abschnitten u.a. auf Komponenten des veränderten Schemas selbst und auf die darauf operierenden Applikationen eingegangen wurde, besprechen wir nun Konsequenzen für die Handhabung der von diesen Applikationen benötigten Objekte der Datenbank. Die hier darzustellenden Ziele leiten sich daher teilweise von denen des vorherigen Abschnittes über Applikationen ab.

Technisches Teilziel 3.11 {vollständige Propagation}
Die Propagation von Objekten muß zwischen den Zugriffsbereichen beliebiger Versionen eines Schemas möglich sein.

Die Möglichkeit zur Durchführung von Schemaänderungen muß auch dann noch bestehen, wenn sich bereits Objekte in der Datenbank befinden. Aus den Veränderungen im Schema kann sich jedoch ergeben, daß alter und neuer Schemazustand verschiedene physikalische Speicherformate für Objekte derselben, allerdings veränderten Klasse implizieren.

Selbst wenn wir die Forderung aus Teilziel 3.9 vorübergehend ignorieren und von der vereinfachenden Annahme ausgehen, daß alle Applikationen zum Zeitpunkt einer Schemaänderung sofort an den neuen Schemazustand angepaßt werden, so dürfen zuvor existierende Objekte bei der Schemaänderung natürlich nicht einfach verloren gehen, sondern müssen weiterhin für die dann angepaßten Applikationen zugreifbar sein. Da die vor der Schemaänderung existierenden Objekte den ggf. veränderten Vorgaben an ihre Struktur nicht mehr notwendigerweise entsprechen, müssen sie wie die Applikationen an das veränderte Schema angepaßt werden. Dazu ist eine Umsetzung der Objekte entsprechend der durchgeführten Schemaänderungen, *Konvertierung* genannt, notwendig.

Unter Berücksichtigung von Teilziel 3.9 ist die Situation etwas komplizierter. Dann liegen nämlich Applikationen vor, die auf verschiedenen Zuständen des Schemas aufsetzen und die Teilziel 3.10 folgend auf gemeinsamen Objekten kooperieren sollen. Wir betrachten hier zunächst nur die Situation, daß das Schema in genau zwei Zuständen vorliegt (siehe Abbildung 3.8). Dabei bezeichnen wir Objekte, je nachdem welchem der beiden Schemazustände entsprechend sie vorliegen, als *alt* oder *neu*. Analog nennen wir Applikationen in Abhängigkeit von dem Schemazustand, den sie erwarten, *alt* oder *neu*. Liegt nun ein von einer Applikation zuzugreifendes Objekt nicht in der passenden Form vor, so unterscheiden wir zwei Fälle. Der erste Fall entspricht dem oben

gesagten: Ein altes Objekt soll von einer neuen Applikation zugegriffen werden. Umgekehrt beschreibt der zweite Fall die Situation, daß eine alte Applikation auf ein neues Objekt zugreifen will. Demzufolge besteht hier in Erweiterung des oben gesagten nicht nur die Notwendigkeit zur Konvertierung von alten Objekten für neue Schemazustände, sondern umgekehrt auch die Notwendigkeit der Konvertierung neuer Objekte für alte Schemazustände. Wir sprechen hierbei entsprechend der zeitlichen Richtung von *Vorwärts-* bzw. von *Rückwärtskonvertierung*.[19]

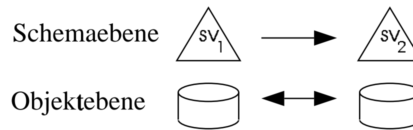


Abbildung 3.8: Abstrakte Darstellung der Propagation.

Die Konvertierung transformiert den vorliegenden Zustand eines Objektes, so daß dieses danach den Anforderungen eines anderen Schemazustandes entspricht. Es ist jedoch aus verschiedenen Gründen, auf die wir später noch näher eingehen werden, nicht immer sinnvoll, den vorliegenden Zustand dabei zu überschreiben. Stattdessen ist es oft besser, zunächst eine Kopie des Objektes anzulegen und diese dann zu konvertieren, so daß das Objekt gleichzeitig für verschiedene Schemazustände in passenden Ausprägungen vorliegt. Bei Änderungen eines Objektes durch eine Applikation müssen dann ggf. andere Ausprägungen desselben Objektes berücksichtigt werden. Von den eher technischen Details abstrahierend, führen wir den Begriff der *Propagation* ein, der allgemein die Weitergabe von Zustandsänderungen einer Datenbank beschreibt. Solche Zustandsänderungen können die Erzeugung neuer Objekte oder die Modifikation oder Löschung existierender Objekte durch Applikationen sein. Entsprechend müssen ggf. neue Objektausprägungen erzeugt oder existierende modifiziert oder gelöscht werden. Die Propagation schließt die bei Objekterzeugung und Modifikation notwendige Konvertierung mit ein. Analog dem oben gesagten unterscheiden wir auch zwischen *Vorwärts-* und *Rückwärtspropagation*.

Die Verfügbarkeit aller Objekte für Applikationen verschiedener Schemazustände ist ein besonders wichtiges Ziel und bedingt die Möglichkeit, Objekte zwischen beliebigen Zuständen ihres Schemas frei hin- und herpropagieren zu können. Dazu ist allerdings nicht nur die (Vorwärts- und Rückwärts-) Propagation zwischen in der Ableitungsstruktur benachbarten Schemazuständen notwendig, sondern auch die Propagation zwischen nur indirekt in Verbindung stehenden Schemazuständen. □

Technisches Teilziel 3.12 {flexible Propagationssteuerung}
Der Grad an durchzuführender Propagation sollte flexibel einstellbar sein.

Teilziel 3.11 fordert, daß eine vollständige Propagation zwischen allen Zuständen eines Schemas möglich sein muß. Eine solche Propagation von jedem Zustand eines Schemas zu jedem anderen ist in der Praxis allerdings nicht immer wünschenswert. Wie bereits dargelegt, kann die Entwicklung eines Datenbankschemas eine sehr komplexe Aufgabe sein, die insbesondere im objektorientierten Datenmodell aufgrund dessen vielfältiger und mächtiger Modellierungskonzepte oft nicht auf Anhieb in idealer Art und Weise bewältigt werden kann. Auf Schemaebene hatten wir daher eine Rücksetzbarkeit auf beliebige vorherige Zustände gefordert (Teilziel 3.5), um eine Schemaänderung rückgängig machen zu können, sofern sich unerwünschte Konsequenzen ergeben. Ein Rücksetzen allein des Schemas reicht jedoch nicht aus, wenn sich die angesprochenen Nachteile einer Datenmodellierung, d.h. eines Schemazustandes, erst beim Umgang mit dessen Applikationen herausstellen. Dann wird die Datenbank nämlich durch Ausführung der Applikationen bereits modifiziert sein, bevor die Mängel der Modellierung entdeckt werden. Die logische

[19] Mit den Begriffen der Softwaretechnik könnte man hier wohl auch von *Aufwärts-* und *Abwärtskompatibilität* sprechen.

Fortsetzung von Teilziel 3.5 ist demzufolge die Rücksetzbarkeit auch auf Objektebene. Dies bedingt aber gerade, daß die in Teilziel 3.11 geforderte vollständige Propagation gezielt abschaltbar sein muß. Dies ermöglicht weiterhin den Test neuer Applikationen mit realen Daten, ohne die Gefahr unerwünschter Modifikationen von in der Produktion eingesetzten Datenbanken.

Aufgrund der dargelegten Problematik ergibt sich die Forderung nach einer flexiblen Steuerbarkeit der Propagation. Insbesondere in Rückwärtsrichtung muß eine vollständige Abschaltung möglich sein, um eine Rücksetzbarkeit auf einen früheren und unveränderten Zustand gewährleisten zu können. □

Technisches Teilziel 3.13 {mehrfache, getrennte Datenwerte}
Von einem Objekt sind mehrere logische Objektwerte zuzulassen.

Da die Versionen eines Objektes i.Allg. verschiedenen Typen angehören, sind sie auch unmittelbar nach der Durchführung einer Propagation nicht vollkommen gleich, selbst wenn sie denselben Zustand des Objektes in der Diskurswelt modellieren. Um eine solche Ähnlichkeit bezüglich der mit verschiedenen Versionen eines Objektes verbundenen Semantik auszudrücken, sagen wir, die Objektversionen repräsentieren denselben *logischen Objektwert*.

Aus dem vorangegangenen Teilziel 3.12 leitet sich ab, daß bei eingeschränkter Propagation Änderungen, die von Applikationen eines Schemazustandes durchgeführt werden, für Applikationen anderer Schemazustände möglicherweise transparent sind, obwohl diese mit denselben Objekten operieren. Daher muß die Möglichkeit bestehen, daß ein Objekt für Applikationen verschiedener Schemazustände verschiedene logische Objektwerte aufweist.

Analog zu der Trennung zwischen verschiedenen Zuständen eines Schemas ergeben sich hier verschiedene Ausprägungen der Datenbank, wobei jedem Schemazustand genau eine Datenbankausprägung zugeordnet ist. Für eine Applikation eines bestimmten Schemazustandes sind dann nur die Objekte der zugehörigen Datenbankausprägung sichtbar und zugreifbar. Der Grad der Isolation zwischen den verschiedenen Ausprägungen einer Datenbank ist über die Propagationssteuerung (siehe Teilziel 3.12) einstellbar. □

Technisches Teilziel 3.14 {semantische Beziehungen zwischen beliebigen Schemaausprägungen}
Propagationsbeziehungen auf Objektebene müssen zwischen beliebigen Schemaausprägungen etablierbar sein.

Die Ableitungsbeziehung zwischen verschiedenen Ausprägungen eines Schemas ergibt sich durch die Integration von Komponenten aus bestehenden Schemaausprägungen. Dabei wird die Auswahl einer zu integrierenden Komponente aus mehreren Alternativen im wesentlichen dadurch beeinflußt, wie ähnlich die integrierte Komponente der gewünschten Komponente ist, damit nach der Integration möglichst wenige Anpassungen an der integrierten Komponente vorgenommen werden müssen. Dabei wird i.Allg. noch keine Rücksicht auf die später zu spezifizierende Propagation genommen.

Aus der geschilderten Situation ergibt sich die Forderung nach Flexibilität bei der Integration von Schemakomponenten und damit auch bei der durch die Ableitung implizit festgelegten semantischen Beziehung zwischen verschiedenen Ausprägungen eines Schemas. Neben der bereits durch Teilziel 3.7 ausgedrückten Forderung, verschiedene Komponenten eines Schemas aus verschiedenen Schemaausprägungen integrieren zu können, muß auch bei der Spezifikation von Propagationsbeziehungen relativ frei vorgegangen werden können. Insbesondere sollte die Ableitungsbeziehung auf Schemaebene nicht die Propagationsbeziehungen auf Objektebene vollständig festlegen, d.h. die Propagation sollte nicht nur zwischen direkt voneinander abgeleiteten Schemaausprägungen etablierbar sein. Dies reicht nämlich beispielsweise dann nicht mehr aus,

wenn ein zwischenzeitlich aus einer Klasse gelöschtes Attribut, in einem späteren Schemazustand wieder auftaucht [Sjø93a, Sjø93b]. Hier besteht eine semantische Abhängigkeit zwischen der Ausprägung vor der Löschung und derjenigen nach der Wiedereinführung des Attributes, obwohl diese nicht direkt voneinander abgeleitet wurden.

Dieses Ziel wird u.a. auch von Odberg verfolgt. □

3.2.4 Effizienz und Handhabung der Mechanismen

Nachdem die vorangegangenen drei Abschnitte konzeptionelle Anforderungen an ein System zur Unterstützung von Schemaänderungen vorstellten, soll an dieser Stelle noch auf einige, eher als technisch einzustufende Aspekte eingegangen werden. Diese betreffen die praktische Nutzbarkeit des resultierenden Systems durch den Anwender.

Technisches Teilziel 3.15 {Effizienz}
Die Propagation von Objekten muß effizient ablaufen, damit auch große Datenbestände bewältigt werden können.

Das technische Teilziel 3.10 hatte die Möglichkeit zur Kooperation verschiedener Applikationen auf gemeinsamen Daten gefordert, selbst wenn diese auf verschiedenen Zuständen eines Schemas aufsetzen. Im Zusammenhang mit der Forderung nach vollständiger Propagation (Teilziel 3.11) ergibt sich damit als Konsequenz ein weiteres Ziel. Bei der Ableitung einer neuen Schemaausprägung müssen die Objekte der vorangegangenen für neue Applikationen sofort verwendbar sein, d.h. eine bestehende Ausprägung der Datenbank muß bei Ableitung eines neuen Schemazustandes für dessen Applikationen propagiert werden. In Abhängigkeit von der Komplexität der dabei zu verwendenden Konvertierungsfunktionen und von der Größe der Datenbank kann diese initiale Propagation sehr viel Zeit in Anspruch nehmen und die Datenbank ggf. inakzeptabel lange blockieren. Daher sollte die Möglichkeit gegeben sein, die initiale Propagation einer kompletten Ausprägung der Datenbank *physikalisch verzögert* durchzuführen [Bar91, TK89, FMZ94b, FMZ94a, FMZ+95b]. Während *logisch*, d.h. aus der Sicht der Anwender, keinerlei Unterschied im Vergleich zu einer sofortigen Propagation erkennbar ist, propagiert der verzögerte Mechanismus ein Objekt physikalisch erst dann, wenn darauf tatsächlich von einer Applikation zugegriffen wird (und es im schlechtesten Fall sowieso mit einem erheblichen Zeitaufwand vom Hintergrundspeicher geladen werden müßte). Das bedeutet, daß Zeit und Platz für die Propagation eines Objektes erst dann in Anspruch genommen werden, wenn tatsächlich Bedarf nach diesem Objekt besteht. □

Technisches Teilziel 3.16 {geringer Spezifikationsaufwand für Schemaänderung, Konvertierungsfunktionen und Propagationssteuerung}
Der Aufwand für die Spezifikation von Schemaänderungen, Konvertierungsfunktionen und Propagationsflags sollte möglichst gering sein.

Der Aufwand des Anwenders für die Spezifikation von Schemaänderungen und die Steuerung der Propagation sollte möglichst gering gehalten werden. Die Spezifikation von Schemaänderungen wird bereits durch das Angebot an Schemaänderungsprimitiven entsprechend Teilziel 3.1 erreicht. Die Spezifikation der Propagation auf Objektebene kann beispielsweise dadurch erleichtert werden, daß dem Anwender automatisch erstellte Funktionen zur Durchführung der Objektkonvertierung angeboten werden. Solche Defaultkonvertierungsfunktionen sollten, soweit möglich, semantisch sinnvoll sein. Dies kann u.a. durch eine Analyse der während der Ableitung der neuen Schemaausprägung durchgeführten Änderungen erreicht werden. Darauf aufbauend

kann weiterhin ein sinnvoller Vorschlag für die Einstellung der Propagationssteuerung gemacht werden. □

Technisches Teilziel 3.17 {Lokalität bei der Spezifikation neuer Schemazustände}
Die Spezifikation neuer Schemazustände sollte ohne die Kenntnis der Gesamtheit existierender Schemaversionen möglich sein.

Die Spezifikation eines neuen Schemazustandes geschieht zumindest bei internen Schemaänderungen dadurch, daß Komponenten bereits vorliegender Zustände ausgewählt und in den neuen Zustand integriert werden. Wenn bereits zahlreiche Zustände des Schemas existieren, kann die Übersichtlichkeit leiden. Daher fordern wir eine gewisse Lokalität bei der Spezifikation neuer Schemazustände. Insbesondere die Möglichkeit zur Propagation zwischen allen Ausprägungen der Datenbank (siehe Teilziel 3.11) sollte nicht erfordern, daß bei Ableitung einer neuen Schemaversion für eine Klasse, die zuvor bereits in n verschiedenen Zuständen vorliegt, weitere $2n$ Konvertierungsfunktionen (Vorwärts- und Rückwärtsrichtung) angegeben werden müssen. Dazu wäre neben dem hohen Spezifikationsaufwand, der bereits aufgrund von Teilziel 3.16 vermieden werden sollte, insbesondere die Kenntnis aller existierenden Zustände des Schemas notwendig. Stattdessen sollte einem Schemaentwickler die Kenntnis einer Teilmenge der Schemazustände genügen, um eine sinnvolle Ableitung und Propagation sicherstellen zu können. Diese Teilmenge wird gerade diejenigen Schemazustände enthalten, die eine hohe semantische Ähnlichkeit mit dem neu abzuleitenden Schemazustand aufweisen. □

Die Forderung nach Lokalität bei der Definition einer neuen Schemaausprägung widerspricht zunächst Teilziel 3.14, welches vom Schemaentwickler einen gewissen Überblick über alle vorliegenden Ausprägungen des Schemas verlangt. Dieser Konflikt kann jedoch in gewissem Maße durch ein Werkzeug aufgelöst werden, das dem Schemaentwickler (in Kenntnis aller Schemaausprägungen) Vorschläge für Verbesserungen der von ihm spezifizierten Ableitungsbeziehungen macht.

Technisches Teilziel 3.18 {Konsistenz des Datenbanksystems}
Die Konsistenz des Schemas und der Datenbank ist stets zu erhalten.

Eine Datenbank modelliert typischerweise eine komplex strukturierte Diskurswelt aus einer Vielzahl von Komponenten, zwischen denen zahlreiche, sichtbare und unsichtbare Querbeziehungen und Abhängigkeiten bestehen. Daraus ergeben sich für das Datenbankschema entsprechende Konsistenzbedingungen, die nur zum Teil mit den Mitteln des verwendeten Datenmodells ausgedrückt und damit vom Datenbankmanagementsystem sichergestellt werden können. Objektorientierte Systeme erhöhen durch ihre im Vergleich zu älteren Datenmodellen fortgeschrittenen Modellierungskonzepte zwar den Anteil systematisch ausdrückbarer Konsistenzbedingungen, damit wird die Konsistenzerhaltung bei Schemaänderungen allerdings auch erschwert.

Die Durchführung von Schemaänderungen betrifft i.Allg. sowohl das Schema als auch die Datenbank. Bei unkontrollierten Änderungen kann es dabei auf beiden Ebenen zu Inkonsistenzen kommen, was zu vermeiden ist [Tre95].

Auf der Ebene des Schemas macht eine Änderung i.Allg. korrigierende Maßnahmen bei indirekt betroffenen Schemakomponenten erforderlich. Solche Korrekturen sollten weitestgehend automatisch durchgeführt werden, um den Schemaentwickler von ggf. komplexen Arbeiten zu entlasten. Wenn der Schemaentwickler eine Änderung vornimmt, welche Korrekturen erforderlich macht, die er nicht notwendigerweise als offensichtliche Konsequenzen seines Handelns erkennen kann, so sollte er zumindest im interaktiven Betrieb gewarnt und um eine Bestätigung ersucht werden. Schemaänderungen, bei denen keine offensichtliche und eindeutige Möglichkeit zur Korrektur ggf. entstehender Inkonsistenzen existiert, sind abzulehnen. Bei der nicht–interaktiven Durchführung von Schemaänderungen ist zunächst die Durchführbarkeit aller spezifizierter Änderungen zu prüfen, bevor mit ihrer tatsächlichen Durchführung begonnen wird.

Auf der Ebene der Datenbank müssen ggf. Anpassungen persistenter Objekte an durchgeführte Schemaänderungen vorgenommen werden. Diese Anpassung sollten nach Möglichkeit verlustfrei und vollständig sein, so daß keine Semantik verloren geht. In diesem Sinne ist eine umkehrbare Anpassung ideal. □

Technisches Teilziel 3.19 {Transparenz für Applikationsentwickler}
Die zur Ermöglichung von Schemaänderungen eingesetzten Mechanismen sollten den Applikationsentwicklern transparent sein, um deren Aufgabe nicht zu erschweren.

Wir unterscheiden in dieser Arbeit zwei Gruppen von Datenbankanwendern, nämlich *Schemaentwickler* und *Applikationsentwickler*.

Die Aufgabe der Schemaentwickler umfaßt den Entwurf und die Implementierung von Datenbanken, die durch mehrere Applikationen genutzt werden. Während des Datenbankbetriebs führen sie ggf. notwendige Schemaänderungen durch. Demzufolge müssen Sie mit den in dieser Arbeit behandelten Problemen umgehen und dazu die hier vorgestellten Konzepte kennen und einsetzen.

Die Aufgabe der Applikationsentwickler besteht hingegen im Entwurf und in der Implementierung von Applikationen. Dazu greifen sie aus den Applikationen heraus auf gemeinsam genutzte Datenbanken wie auf andere Informationsquellen zu, sie sind jedoch nicht mit deren Entwurf beschäftigt. Stattdessen bekommen sie die zum Zugriff auf eine Datenbank notwendigen Informationen in Form eines Datenbankschemas von den Schemaentwicklern vorgegeben und beschränken sich im wesentlichen[20] auf die Nutzung der Datenbank aus den von ihnen entwickelten Applikationen heraus. Daher sollten die Aspekte der Schemaevolution für die Applikationsentwickler soweit wie möglich verborgen bleiben.

Dieses Ziel wird u.a. auch von Odberg und Rundensteiner verfolgt. □

3.3 Schemaversionierung als Lösungsmodell

Die in Abschnitt 2.2 vorgestellten Versionierungskonzepte sind in vielen Anwendungsbereichen ideal geeignet zur Modellierung von Objekten, die einem Evolutionsprozeß unterworfen sind und sich daher häufig ändern. Das Schema eines Datenbanksystems wird oft als ein komplexes Objekt der Metaebene betrachtet und weist aus dieser Perspektive im Rahmen der Schemaevolution genau die bezeichneten Eigenschaften auf. Daher verfolgen wir in dieser Arbeit den naheliegenden Weg der Anwendung von Versionierungskonzepten auf das Datenbankschema. Wir stellen diesen Ansatz hier zunächst grob vor, um einen Überblick über unser Lösungsmodell zu geben und die Zuordnung der später im Detail zu besprechenden Konzepte in den Gesamtzusammenhang zu erleichtern. Dabei werden wir die technischen Teilziele des vorangegangenen Abschnittes in ihrer dortigen Reihenfolge wieder aufgreifen und jeweils darlegen, wie diese durch Anwendung von Versionierungskonzepten zur Unterstützung von Schemaevolutionsprozessen erreicht werden können.

Heutzutage verfügbare, kommerzielle Datenbanksysteme verfolgen, sofern sie überhaupt die Durchführung von Schemaänderungen erlauben, durchweg einen direkten Ansatz. Dabei wird ein vorliegendes Schema modifiziert, wobei dessen bisheriger Zustand überschrieben wird und damit unwiederbringlich verloren geht. Damit ergeben sich jedoch erhebliche Probleme bzw.

[20] Datenbanken sind kein Selbstzweck, sondern bieten den sie benutzenden Applikationen Dienste an. Daher ist es natürlich, daß die Applikationen als Dienstnehmer Anforderungen an die Qualität des Dienstes, hier insbesondere an das Datenbankschema, stellen. Die Notwendigkeit zur Durchführung von Schemaänderungen kommt also von neuen oder veränderten Applikationen her und demzufolge sind sich doch zumindest einige Applikationsentwickler der sich im Datenbankschema vollziehenden Änderungen bewußt.

Einschränkungen beim Einsatz der angebotenen Mechanismen, insbesondere, wenn sich das veränderte Schema bereits in Benutzung befindet. Das Schema stellt nämlich quasi die Schnittstelle zwischen den in der Datenbank abgelegten Objekten und den darauf zugreifenden Applikationen dar und darf daher im laufenden Betrieb keinerlei Veränderungen unterzogen werden. Mehrere unserer technischen Teilziele repräsentieren jedoch gerade Aspekte einer Vorgehensweise, die man als *dynamische Schemaänderung* bezeichnen könnte und die eben jene Möglichkeit zu Schemaänderungen auch noch im bereits laufenden Betrieb eines Systems gestattet. Die dabei vorliegende Situation zeichnet sich durch die Existenz der beiden genannten Elemente aus, welche beide eine unveränderliche Schnittstelle benötigen. Dies sind zum einen die Objekte der Datenbank, die physikalisch in einem Format abgelegt sind, das durch ein entsprechendes Schema beschrieben wird und die ohne diese Beschreibung nicht mehr korrekt interpretiert werden könnten. Zum anderen befinden sich Applikationen im Einsatz, die auf die Datenbank zugreifen und dabei notwendigerweise davon ausgehen müssen, daß die dort abgelegten Objekte die durch das Schema zugesagten Eigenschaften auch garantiert aufweisen.

Die Grundlage unseres Lösungsmodells ist die Anwendung von Versionierungskonzepten auf das Schema, das als Entwurfsobjekt bzw. als Metaobjekt verstanden und mit den in Abschnitt 2.2 dargestellten Konzepten versioniert wird. Damit erhalten wir ein *versioniertes Schema*, das sich aus einer nichtleeren Menge von *Schemaversionen* zusammensetzt. Die Grundidee unseres Ansatzes besteht also darin, alte Zustände des Schemas in Form der Schemaversionen aufzubewahren. Das heißt bestehende Schemaversionen werden nie gelöscht und stehen damit unbegrenzt zur Verfügung, um weiterhin eine korrekte Interpretation gespeicherter Objekte gestatten und bestehenden Applikationen weiterhin die von ihnen benötigte Form des Schemas, d.h. die passende Schemaversion anbieten zu können.

Für die angesprochene Problematik dynamischer Schemaänderungen ist dabei von besonderem Interesse, daß eine Ausprägung des Schemas, also eine Schemaversion, modifiziert werden kann, ohne ihren vorherigen Zustand zu überschreiben. Stattdessen wird eine neue Schemaversion abgeleitet, d.h. sie wird neu angelegt und erhält zunächst dieselben Eigenschaften wie die Ausgangsversion. Die beabsichtigten Schemaänderungen werden dann auf der neuen Schemaversion ausgeführt, wobei die Ausgangsversion unverändert erhalten bleibt, so daß der laufende Betrieb ohne Unterbrechung fortgesetzt werden kann. Einzelne Applikationen arbeiten dabei auf der Basis je einer fest ausgewählten Schemaversion, die für sie genau dieselbe Rolle spielt wie ein unversioniertes Datenbankschema in herkömmlichen Systemen.[21] Weitere Vorteile des Einsatzes von Versionierungskonzepten resultieren aus der bereits in der Einleitung angesprochenen Komplexität des Schemaentwurfs und aus der Feststellung, daß oft mehrere verschiedene, aber trotzdem adäquate und korrekte Schemaausprägungen zur Modellierung derselben Diskurswelt existieren. Die Flexibilität des Versionierungskonzeptes und insbesondere die Möglichkeiten der schrittweisen Entwicklung eines Entwurfs kommen der tatsächlichen Vorgehensweise eines Schemaentwicklers sehr entgegen. Wir gehen nun etwas näher auf einzelne Aspekte ein und legen dar, wie diese zur Erreichung der technischen Teilziele beitragen.

3.3.1 Flexibilität bei der Durchführung von Schemaänderungen

(Ziel 3.1: Angebot an Schemaänderungsprimitiven) Ein versioniertes Schema kann eine beliebige Anzahl von Schemaversionen enthalten, die gleichzeitig zur Benutzung zur Verfügung stehen. Der Begriff der Benutzung schließt hier auch die Möglichkeit der Ableitung neuer Schemaversionen ein, die dann den Bedürfnissen entsprechend verändert werden können. Wir hatten in Teilziel 3.1 bereits festgehalten, daß die Spezifikation neuer Schemaausprägungen auf der Basis bestehender Ausprägungen erfolgen sollte, um damit u.a. den Spezifikationsaufwand für

[21] Die Schemaversionierung ist damit für die Applikationen transparent, so daß wir aus deren Blickwinkel mitunter von Schema sprechen, auch wenn wir genau genommen nur eine einzelne Version dieses Schemas meinen.

den Schemaentwickler zu verringern. Durch das in der Ableitung einer neuen Schemaversion bereits enthaltene Kopieren des Zustandes der Ausgangsversion sind die Voraussetzungen für die Erreichung von Teilziel 3.1 bereits gegeben.

Was nun noch fehlt sind die Schemaänderungsprimitive, die eine Modifikation einer gegebenen Schemaversion gestatten. Zu diesem Zweck führen wir eine Taxonomie ein, die sowohl erzeugende als auch verändernde Primitive enthält. Diese Primitive sind dann Teil einer kombinierten Schemabeschreibungs- und -änderungssprache, die sowohl die Beschreibung neuer Schemakomponenten als auch die Veränderung vorhandener Komponenten erlaubt. Da die Hinzufügung neuer Komponenten auch als Veränderung eines Schemas bzw. einer Schemaversion verstanden werden kann, lag die Entwicklung einer integrierten Sprache für die beiden, ohnehin schwer zu trennenden Aspekte nahe. Obwohl sie genau genommen nicht die Objekte selbst, sondern deren Struktur und Verhalten, also ihr Schema wiedergibt, wird die Schemabeschreibungssprache eines objektorientierten Datenbanksystems mitunter auch als Objektbeschreibungssprache (engl. *object definition language, ODL*) bezeichnet. Daher verwenden wir auch in dieser Arbeit die Kurzform ODL für unsere Schemabeschreibungs- und -änderungssprache. Unsere ODL basiert auf einer Schemaänderungstaxonomie und enthält die entsprechenden Primitive mit den benötigten Parametern. Neben den aus anderen Ansätzen der Literatur bekannten Primitiven, die in der Schemaversionierung für die Erstellung und Modifikation einzelner Schemaversionen benutzt werden (intra-Schemaversion), bieten wir Primitive zum Austausch von Komponenten und zur Etablierung und Änderung von Beziehungen zwischen Schemaversionen (inter-Schemaversion).

(Ziel 3.2: uneingeschränkte Ausführbarkeit sämtlicher Schemaänderungen) Bei Veränderungen werden durch Anwendung der Schemaänderungsprimitive stets neue Versionen des Schemas erzeugt (siehe Abbildung 3.10). Umgekehrt entstehen neue Schemaversionen stets durch Ableitung von bestehenden Versionen desselben Schemas. Dabei ist zu beachten, daß die Existenz der Ableitungsbeziehung zwischen den Schemaversionen, die zur Erreichung von Teilziel 3.8 benötigt werden, keinerlei Einschränkung bezüglich der Änderbarkeit einzelner Schemaversionen bedeutet. Das heißt für die Durchführung von Änderungen kann eine Schemaversion isoliert betrachtet werden und demzufolge dienen bestehende Einschränkungen nur der Konsistenzerhaltung des Schemas bzw. einzelner Schemaversionen. Damit gehen die hier zu berücksichtigenden Restriktionen nicht über das notwendige und bereits von unversionierten Systemen bekannte Maß hinaus. Weiterhin ist das Schema auch von bestehenden Objekten und Applikationen entkoppelt und kann daher zu beliebigen Zeiten und in beliebiger Art und Weise verändert werden.

(Ziel 3.3: Durchführung von Änderungen auf Schemaebene) Normalerweise werden zusammengehörige, primitive Schemaänderungen in Listen gesammelt und gemeinsam als Einheit ausgeführt, so daß nicht jede Einzeloperation eine neue Schemaversion erzeugt. Auf diese Art und Weise beinhalten Schemaversionen komplette, natürliche und konsistente Beschreibungen einer Diskurswelt. Demzufolge ist das Schema die Einheit der Änderung. Eine Beschränkung einer Änderung auf eine einzelne Klasse ist meist auch gar nicht möglich. Stattdessen betrifft eine Veränderung in der Regel mehrere Klassen direkt oder, über Abhängigkeiten zwischen den Klassen, auch indirekt.

Entsprechend Teilziel 3.3 wenden wir den Versionierungsansatz auf Schemaebene (und nicht auf Klassenebene) an. Damit können wir die Konsistenz und insbesondere die gegenseitige „Verträglichkeit“ zwischen den jeweils ausgewählten Versionen der verschiedenen Klassen bereits bei der Erzeugung einer neuen Schemaversion prüfen und sicherstellen (und nicht erst, wenn eine Applikation die von ihr erwünschten Versionen der verschiedenen Klassen zusammenstellt). Damit können die Applikationsentwickler auf Schemaversionen zurückgreifen, die garantiert jeweils eine konsistente Konfiguration von Klassenversionen beinhalten.

(Ziel 3.4: Möglichkeit externer Schemaänderungen) Obwohl unsere Schemaänderungstaxonomie in dem Sinne vollständig ist, daß sich jede beliebige Schemaversion durch Anwendung

der Primitive erstellen läßt,[22] kann es in Einzelfällen erwünscht sein, eine Schemaversion in Ausschnitten oder sogar vollständig neu beschreiben oder überarbeiten zu können, ohne auf eine vorgegebene Menge von Primitiven festgelegt zu sein.

Die geforderte Möglichkeit zur Durchführung externer Schemaänderungen läßt sich bei Verwendung des Versionierungskonzeptes in besonders allgemeiner Weise schaffen und erreicht einen besonders hohen Grad an Flexibilität, insbesondere wenn die Möglichkeit zur Ableitung alternativer Versionen nach Teilziel 3.6 gegeben ist.

Im Rahmen der Objektversionierung werden neue Objektversionen außerhalb der Kontrolle des Versionierungsmechanismus erstellt, z.B. indem Versionen aus der Datenbank entnommen werden (der Entnahmeprozeß wird als *check out* bezeichnet), daran beliebige Veränderungen unkontrolliert durchgeführt werden und die entstandenen Ergebnisse schließlich wieder in die Datenbank eingebracht werden (engl. *check in*).

Im Rahmen der Schemaversionierung bietet sich eine ähnliche Vorgehensweise an, die als *externer Schemaversionierungsansatz* bezeichnet wird. Anders als bei der Durchführung externer Schemaänderungen in Systemen, die jeweils nur eine aktuelle Ausprägung verwalten, besteht hier bei der Entnahme des Schemas die Wahl zwischen allen bisher existierenden Schemaversionen. Dadurch reduziert sich i.Allg. der Änderungsaufwand zur Erreichung des gewünschten Ergebnisses.

Beim Einbringen kann das System in der Regel annehmen, daß eine wiedereingebrachte Version ein Nachfolger des zuvor entnommenen Originals ist und sie damit an der entsprechenden Position in einer Versionsableitungsstruktur eintragen. Dieser Mechanismus kann auch mit mehreren, gleichzeitig entnommenen Schemaversionen und mit mehrmaligem Wiedereinbringen derselben Version genauso problemlos umgehen wie mit entnommenen Schemaversionen, die nie wieder eingebracht werden. Durch diese Robustheit gegenüber außerhalb der eigenen Kontrolle durchgeführten Operationen eignet sich der Versionierungsmechanismus besonders gut für den Umgang mit externen Schemaänderungen.

(Ziel 3.5: Erhaltung bisheriger Schemazustände und Rücksetzbarkeit) Dem Versionierungsansatz entsprechend werden einmal erstellte Schemaversionen nie überschrieben oder gelöscht, es sei denn, dies würde ausdrücklich gewünscht. Damit bleiben alle bisherigen Schemaversionen erhalten, wodurch u.a. die durchlaufenen Evolutionsschritte dokumentiert werden. Frühere Schemaversionen unterscheiden sich konzeptionell nicht von der neuesten Schemaversion und stehen damit auch weiterhin für die Benutzung zur Verfügung. Dies umfaßt sowohl die Benutzung für die Ableitung neuer Schemaversionen als auch die Benutzung durch bereits im Einsatz befindliche oder noch zu entwickelnde Applikationen. Auf diese Weise ist die Rücksetzbarkeit auf frühere Zustände des Schemas jederzeit gegeben. Dies ist nicht nur dann sinnvoll, wenn sich an dem früheren Zustand vorgenommene Änderungen als unvorteilhaft herausgestellt haben und rückgängig gemacht werden sollen. Eine unserer elementaren Beobachtungen ist nämlich, daß in einem bestimmten Anwendungsgebiet durchaus mehrere „richtige“ Schemaversionen existieren können. Unterschiedliche Schemaversionen können durchaus verschiedene Perspektiven derselben Diskurswelt modellieren und sie tun dies in der Praxis sogar recht häufig. Die Verwaltung dieser verschiedenen Perspektiven wird durch die Schemaversionierung in idealer Weise bewerkstelligt.

(Ziel 3.6: Ableitung alternativer Schemaversionen) Dabei kann in natürlicher Weise die Verwaltung sowohl historischer, d.h. sequentiell voneinander abgeleiteter, als auch die Verwaltung von Alternativen, d.h. parallel abgeleiteter Schemaversionen unterstützt werden.

---

[22] Diese Aussage bezieht sich lediglich auf die Ebene des Schemas. Mit Blick auf die aus einer Schemaänderung resultierenden Konsequenzen auf der Objektebene kann eine vorgegebene Menge von Schemaänderungsprimitiven wohl nie alle Anforderungen erfüllen, die von unterschiedlichsten Anwendungen gestellt werden. Wir verwenden daher eine konzeptionell erweiterbare Menge von Primitiven.

(Ziel 3.7: Integration alternativer Schemaversionen) In Ergänzung zur Unterstützung bei der Ableitung neuer Schemaversionen, die in möglicherweise divergierende Designalternativen resultiert, muß auch Unterstützung für die Integration solcher Alternativen angeboten werden. Während das allgemeine Problem der Integration unabhängiger Datenbanken (engl. *multidatabases*) [Bre90, TS93b, TS93a] sehr schwierig ist, kann die Integration von Schemaversionen vergleichsweise einfach durchgeführt werden, da das Datenbanksystem Kenntnis der semantischen Beziehungen zwischen den Komponenten verschiedener Schemaversionen hat. Unsere Schemaänderungstaxonomie enthält auch solche Primitive, die eine Integration verschiedener Versionen eines Schemas erlauben.

(Ziel 3.8: Verwaltung der Ableitungsbeziehung) Durch die Anwendung allgemeiner Versionierungskonzepte auf das Schema werden dessen Versionen in einer Struktur gespeichert, die die Ableitungsbeziehung zwischen ihnen darstellt (engl. *derivation relationship*), d.h. diese Struktur gibt für jede Schemaversion wieder, aus welchen Schemaversionen sie durch die Anwendung von Schemaänderungsprimitiven hervorgegangen ist. Abbildung 3.9 zeigt einen *Schemaableitungsbaum*, der beispielsweise beinhaltet, daß Schemaversion sv_4 von sv_3 abgeleitet wurde, welche wiederum aus sv_1 hervorging. Zur Beschreibung der Ableitungsbeziehung verwenden wir die Begriffe *Vorgängerschemaversion* (engl. *parent schema version*) und Nachfolgerschemaversion (engl. *child schema version*) analog ihrer Bedeutung bei der Objektversionierung.

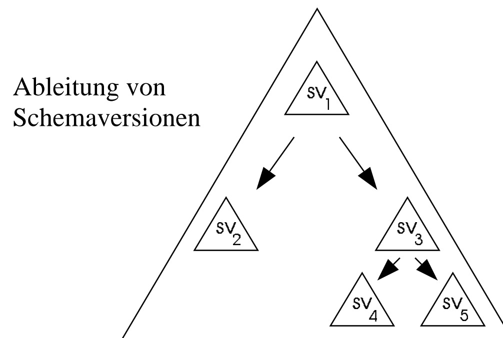


Abbildung 3.9: Abstrakte Darstellung eines Schemaableitungsbaumes.

Wenn eine neue Schemaversion sv_y von einer bestehenden sv_x abgeleitet wird, können existierende Klassen verändert oder gelöscht oder neue Klassen für sv_y erzeugt werden. Wird eine Klasse c dabei verändert, so sagen wir sv_x und sv_y enthalten verschiedene *Versionen der Klasse* c, die (in dieser Reihenfolge) als $sv_x.c$ und $sv_y.c$ notiert werden. Wie bereits zuvor etablieren wir auch hier eine Ableitungsbeziehung, diesmal aber nicht zwischen den Versionen eines Schemas sondern zwischen den Versionen einer Klasse. Die in einem solchen *Klassenableitungsbaum* dargestellten Beziehungen dokumentieren wie die verschiedenen Versionen einer Klasse auseinander hervor gingen und sind für die später zu betrachtende Konvertierung von Objekten von elementarer Bedeutung.

3.3.2 Berücksichtigung von Applikationen

Unser Ansatz erlaubt die Durchführung dynamischer Schemaänderungen, d.h. Schemaänderungen können auch während des Betriebs der Datenbank vorgenommen werden.

Jede Applikation arbeitet auf genau einer Schemaversion und durch jede Schemaversion sv ist eine bestimmte Menge von Objekten zugreifbar. Diese Menge wird der Zugriffsbereich (engl. *instance access scope, IAS*) der Schemaversion genannt.

(Ziel 3.9: Vermeidung manueller Applikationsanpassungen) Damit ist es im Gegensatz zu anderen Ansätzen nicht notwendig, alle Applikationen bei jeder Schemaänderung sofort an-

zupassen. Stattdessen können alle existierenden Applikationen unverändert (auch ohne Recompilierung) weiterverwendet werden und die Anpassung einer oder mehrerer Applikationen kann, sofern das als vorteilhaft erachtet wird, zu einem beliebigen späteren Zeitpunkt durchgeführt werden.

Das verbleibende Problem ist auf der Objektebene angesiedelt. Dabei muß bestimmt werden, wie eine Schemaänderung auf die Objektebene übertragen, d.h. wie existierende Instanzen an neue Versionen ihrer Klassen in neueren Schemaversionen angepaßt werden können. Es ist zu klären, wie Applikationen kooperieren können, obwohl sie auf verschiedenen Versionen eines Schemas und damit auf verschiedenen Klassenversionen und auf verschiedenen Typen aufsetzen.

(Ziel 3.10: Kooperation) Überlappen sich die Zugriffsbereiche zweier Schemaversionen, so haben diese gemeinsame Objekte auf die ihre Applikationen gemeinsam zugreifen und damit kooperieren können. Ein Objekt, das in den Zugriffsbereichen verschiedener Schemaversionen enthalten ist, muß sich dabei allerdings je nach Anforderung der zugreifenden Applikation anders darstellen, d.h. es muß immer dem von der Applikation erwarteten Typ entsprechen. Daher liegt die Idee nahe, von einem Objekt für jede Schemaversion in der dieses sichtbar ist, eine passende Objektversion vorzuhalten.

Im Gegensatz zu den allgemein gängigen Modellen der Objektversionierung (siehe Abschnitt 2.2), wo alle Versionen eines Objektes dieselben strukturellen Eigenschaften besitzen müssen, benötigt die Verwaltung der Objekte der Datenbank unter dem Schemaversionierungsmechanismus ein verallgemeinertes Modell. Hier entsprechen die verschiedenen Versionen eines Objektes nämlich den Ausprägungen dieses Objektes unter den verschiedenen Versionen seines Schemas (zumindest denjenigen, in denen das Objekt sichtbar ist) und da dieselbe Klasse in verschiedenen Schemaversionen verschiedene Typen spezifizieren kann, sind die Versionen eines mit dem Objektversionierungsmechanismus gespeicherten Objektes i.Allg. von verschiedenen Typen.

3.3.3 Flexibilität bei der Propagation von Schemaänderungen auf die Objektebene

(Ziel 3.11: vollständige Propagation) Der vorgestellte Propagationsansatz basiert auf Konvertierungsfunktionen, die Objekte zwischen verschiedenen Typen abbilden und die zur Unterstützung von Schemaevolution und -integration verwendet werden können.

Wenn eine Applikation einer Schemaversion sv ein Objekt erzeugt, so wird dieses zunächst dem Zugriffsbereich $IAS(sv)$ dieser Schemaversion hinzugefügt. Weiterhin kann es automatisch vorwärts und rückwärts in die Zugriffsbereiche aller anderen Schemaversionen propagiert werden (siehe Abbildung 3.10). Auf diesem Wege kann ein Objekt verschiedene Versionen erhalten, die entsprechend verschiedener Schemaversionen verschiedene Typen besitzen.

Wir bieten verschiedene Arten von Konvertierungsfunktionen an, mit denen Objekte sowohl (zeitlich) vorwärts als auch rückwärts zwischen den Zugriffsbereichen verschiedener Schemaversionen propagiert werden können. Durch die automatische Hintereinanderausführung mehrerer Konvertierungsfunktionen entlang von Pfaden durch die Ableitungsstruktur einer Klasse erreichen wir eine vollständige Propagation mit der geringst möglichen Anzahl notwendiger Konvertierungsfunktionen.

(Ziel 3.12: flexible Propagationssteuerung) Um eine Steuerbarkeit der Propagation zu erreichen, entwickeln wir in Ergänzung der Verwaltung versionierter Schemata auf Schemaebene einen Propagationsmechanismus auf Objektebene, der eine flexible und deklarative Spezifikation der Sichtbarkeit von Objekten durch verschiedene Schemaversionen mit Hilfe von Parametern gestattet, die die Sichtbarkeit von Objekten oder Objektzuständen auf Teilmengen der Versionen eines Schemas einschränken können. Dies wird nicht nur während der Entwicklung neuer

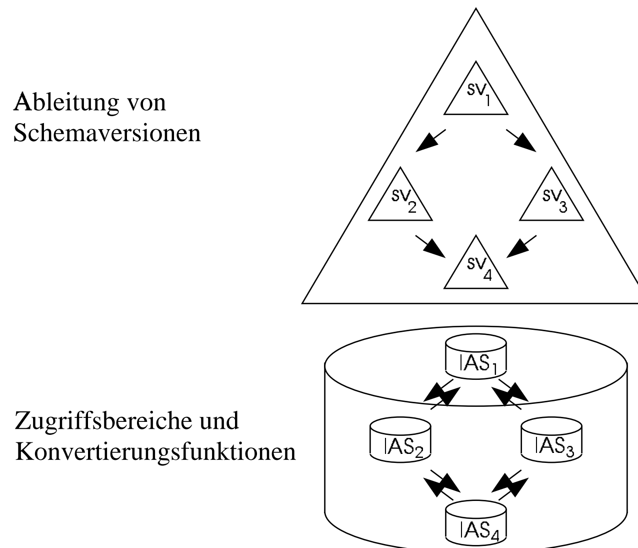


Abbildung 3.10: Abstrakte Darstellung des Schemaversionierungsansatzes.

Applikationen und Schemaversionen benötigt, sondern auch später noch, um gewisse generelle Restriktionen bzw. Konsistenzbedingungen garantieren zu können. Die Ausführung der Konvertierungsfunktionen kann durch sog. *Propagationsflags* flexibel gesteuert werden.

(Ziel 3.13: mehrfache, getrennte Datenwerte) Wir hatten bereits erläutert, daß wir die Objekte der Datenbank in mehreren Versionen ablegen, wobei für jede Schemaversion, in der ein Objekt sichtbar ist, genau eine zugehörige Objektversion angelegt wird. Auf dieser Grundlage kann ohne Mühe die Forderung erfüllt werden, daß die für die Applikationen verschiedener Schemaversionen sichtbaren Objektwerte nicht notwendigerweise immer denselben logischen Objektwert repräsentieren müssen. Mehrfache, getrennte Datenwerte können nämlich einfach durch Abschaltung der Propagation zwischen den Versionen eines Objektes erreicht werden.

(Ziel 3.14: semantische Beziehungen zwischen beliebigen Schemaausprägungen) Die in Abbildung 3.10 dargestellten Konvertierungsfunktionen bewerkstelligen die Transformation von Objekttypen entsprechend der Änderung einer Klasse von einer Schemaversion zu einem ihrer Nachfolger und wieder zurück. Die dabei beschreitbaren Propagationspfade der Objektebene verlaufen also stets entlang der Ableitungsbeziehungen auf Schemaebene und damit entspricht die Propagationsstruktur auf Objektebene, die sich als Summe der Propagationspfade aller Klassen ergibt, der Ableitungsstruktur auf Schemaebene. Diese, in Abbildung 3.10 leicht ersichtliche Feststellung bedeutet jedoch zunächst eine erhebliche Einschränkung der Propagationsmöglichkeiten, da die bisher erwähnten, herkömmlichen Konvertierungsfunktionen einer neuen Schemaversion demzufolge ausschließlich zu deren direkten Vorgängerschemaversionen definiert werden können. Um die genannte Einschränkung aufzuheben, führen wir zusätzlich einen weiteren Typ von Konvertierungsfunktionen ein, welche die bisherige Propagationsstruktur anreichern und damit die zwischen zwei beliebigen Versionen einer Klasse propagierbare Semantik verbessern.

3.3.4 Effizienz und Handhabung der Mechanismen

(Ziel 3.15: Effizienz) Der Schemaversionierungsansatz setzt nicht voraus, daß physikalische Kopien umfangreicher Schemaversionen angelegt werden. Speicherplatz kann durch eine einfache Form der Delta-Komprimierung gespart werden, indem ausgenutzt wird, daß sich oft nur sehr wenige Klassen von einer Schemaversion zur nächsten ändern.

Viel bedeutsamer als die effiziente Speicherung und Verarbeitung des Schemas ist jedoch die Erfüllung dieser Aufgaben auch für die Datenbank, da diese ungleich größer ist und häufiger zugegriffen wird. Aufwand für die Verwaltung der Datenbank ergibt sich in zwei verschiedenen Fällen. Zunächst muß bei der Ableitung einer neuen Schemaversion deren Zugriffsbereich mit Instanzen gefüllt werden, die den ggf. veränderten Typen der neuen Schemaversion entsprechen. Weiterhin müssen die sich durch spätere Änderungen einzelner Objekte ergebenden neuen Zustände in die Zugriffsbereiche anderer Schemaversionen propagiert werden. Während im ersten Fall die einmalige Propagation eines kompletten Zugriffsbereiches erforderlich ist, müssen im zweiten Fall zwar nur einzelne, durch eine Applikation veränderte Objekte propagiert werden, dies muß jedoch nach jeder Änderung wieder passieren und betrifft zudem mehrere Schemaversionen. In beiden Fällen würde bei der sofortigen Durchführung aus logischer Sicht erforderlicher Propagationsschritte ein erheblicher Aufwand entstehen. Im ersten Fall wird die gesamte Datenbank bei Ableitung einer neuen Schemaversion für eine ihrer Größe proportionale Totzeit blockiert, bis der Zugriffsbereich der neuen Schemaversion durch Propagation vollständig ermittelt ist. Im zweiten Fall müssen bei einem Zugriff durch eine Applikation nicht nur eine Objektversion sondern ggf. alle Versionen des zugegriffenen Objektes aktualisiert werden, was einen der Anzahl der Schemaversionen proportionalen Aufwand verursacht. In beiden Fällen erreichen wir eine erhebliche Verbesserung durch die sog. verzögerte Propagation. Dabei werden aus logischer Sicht sofort durchzuführende Propagationsschritte auf der physikalischen Ebene verzögert durchgeführt, nämlich erst dann, wenn tatsächlich von einer Applikation auf ein Objekt zugegriffen wird und die Zeit für einen Zugriff auf den Hintergrundspeicher ohnehin aufgewendet werden muß.[23]

(Ziel 3.16: geringer Spezifikationsaufwand für Schemaänderung, Konvertierungsfunktionen und Propagationssteuerung) Die Vereinfachung der Spezifikation durchzuführender Schemaänderungen und der damit verbundenen Objektpropagationen kann auf vielfältige Art und Weise erreicht werden. Zunächst muß sich das entsprechend Teilziel 3.1 zu realisierende Angebot an Schemaänderungsprimitiven nicht notwendigerweise auf das zur Erreichung einer vollständigen Schemaänderungstaxonomie erforderliche Minimum beschränken. Selbst wenn damit die Durchführung beliebiger Schemaänderungen konzeptionell bereits möglich ist, kann es durchaus sinnvoll sein, weitere Primitive für häufig benötigte oder sehr aufwendige Schemaänderungen anzubieten.

Ein Aspekt, der einen erheblichen Spezifikationsaufwand für den Schemaentwickler bedeuten kann, ist die Erstellung der Konvertierungsfunktionen. Diese werden normalerweise für zahlreiche Klassen und weiterhin sowohl in Vorwärts- als auch in Rückwärtsrichtung benötigt. Da in typischen Szenarien bei Schemaänderungen wohl jeweils nur ein kleiner Teil umfangreicher Schemaversionen verändert wird, fallen entsprechend viele Konvertierungsfunktionen sehr einfach aus. Sie realisieren dann nämlich die identische Abbildung. Daher kann der Spezifikationsaufwand durch die automatische Erzeugung von Defaultkonvertierungsfunktionen erheblich reduziert werden. Die Situation wird noch günstiger, wenn die automatisch erzeugten Konvertierungsfunktionen sogar einen Teil der Semantik der durchgeführten Schemaänderungen widerspiegeln. Dies ist dem Schemaversionierungsansatz folgend möglich, weil die verwalteten Ableitungsbeziehungen über die auf Schemaebene durchgeführten Änderungen Aufschluß geben und sich daraus entsprechend gute, d.h. den tatsächlichen Absichten des Schemaentwicklers möglichst nahe kommende Konvertierungsfunktionen auf Objektebene erstellen lassen.

[23] Da Zugriffe auf den Hintergrundspeicher bei der heutigen Rechnertechnologie mehrere Größenordnungen langsamer sind als im Hauptspeicher, ist die Zahl der Zugriffe auf den Hintergrundspeicher die kritische Größe bei der Geschwindigkeit eines Datenbanksystems und alle rein leistungsorientierten Verbesserungsbestrebungen gehen dahin, diese zu minimieren.

Weiterhin kann der Schemaentwickler durch Werkzeuge unterstützt werden, die ihm den Umgang mit dem Datenbankmanagementsystem erleichtern, indem sie ihm beispielsweise bei der Suche nach Klassen mit bestimmten Eigenschaften helfen.

(Ziel 3.17: Lokalität bei der Spezifikation neuer Schemazustände) Der vom Schemaentwickler zu betreibende Aufwand wird auch durch die angestrebte Lokalität bei der Spezifikation neuer Schemaversionen verringert. Während bei der Verwaltung einer unstrukturierten Menge von Schemazuständen eine kombinatorische Anzahl von Propagationswegen zu beschreiben ist, kann der Aufwand durch eine (Halb-) Ordnung der Schemazustände, d.h. der Schemaversionen, auf eine in der Zahl der Schemaversionen linear wachsende Menge reduziert werden. Auch hier kommt uns wieder die Verwaltung der Ableitungsbeziehungen zwischen den Schemaversionen zugute. Ist die Auswahl einer Quelle für die Integration einer Klasse erst einmal getroffen, so kann die dieser Auswahl wohl zugrunde liegende Ähnlichkeit zwischen gewählter Quellklassenversion und gewünschter Zielklassenversion auch für eine möglichst gute Propagation von Objekten genutzt werden.

(Ziel 3.18: Konsistenz des Datenbanksystems) Die aus unversionierten Datenbanksystemen bekannten Konsistenzbedingungen zwischen den Komponenten des Schemas treten in Systemen mit Schemaversionierung innerhalb von Schemaversionen auf (intra-Schemaversion). Darüberhinaus bestehen Abhängigkeiten zwischen verschiedenen Schemaversionen (inter-Schemaversion). Analog existieren neben den Konsistenzbedingungen zwischen Objekten und Schema im versionierten Fall auch Abhängigkeiten zwischen den Objekten einer Schemaversion (intra-Zugriffsbereich) und zwischen den Versionen eines Objektes, die in verschiedenen Schemaversionen zugreifbar sind (inter-Zugriffsbereich).

Auf Schemaebene wird die Konsistenzerhaltung dem internen Ansatz folgend durch das Angebot von Schemaänderungsprimitiven geleistet. Dabei realisiert jedes Primitiv gewisse im einzelnen noch zu beschreibende Regeln, die ggf. während der Ausführung des Primitives temporär auftretende Inkonsistenzen nach gewissen Strategien beheben und lehnt die Durchführung einer Schemaänderung ab, wenn keine solche Regel eine sinnvolle Wiederherstellung der Konsistenz ermöglicht. Dem externen Ansatz folgend kann eine Konsistenzprüfung erst zu dem Zeitpunkt geschehen, wenn die neue Schemaversion wieder in das System eingebracht wird. Naturgemäß besteht während der Durchführung externer Änderungen keine Möglichkeit ggf. auftretende Inkonsistenzen automatisch zu erkennen. Dies birgt die Gefahr, daß bereits in einer frühen Phase eine Konsistenzverletzung unerkannt bleibt und die Spezifikation sämtlicher auf der inkonsistenten Schemaversion aufbauenden Schemaänderungen nicht genutzt werden kann. Daher sollten externe Schemaänderungen nur mit besonderer Vorsicht durchgeführt und die sich ergebenden Schemaversionen möglichst früh wieder in das System eingebracht werden.

Auf der Objektebene wird die Konsistenzerhaltung durch die Vorgabe automatisch erzeugter Defaultkonvertierungsfunktionen erleichtert. Weiterhin kann ein Werkzeug die auf Schemaebene durchgeführten Änderungen auf ihre Konsequenzen für die Objekte hin untersuchen und ggf. Verbesserungsvorschläge machen. Diese betreffen insbesondere die Ableitungsstruktur und damit auch die sich daraus ergebenden Propagationspfade.

(Ziel 3.19: Transparenz für Applikationsentwickler) Versionierungskonzepte zur Unterstützung von Schemaevolution können grundsätzlich auf der Granularität der Klassen oder auf der ganzer Schemata eingesetzt werden. Wir verfolgen hier die letztere Möglichkeit, da sie zwischen Klassen bestehende Abhängigkeiten bereits berücksichtigt womit sich eine Konfigurationsverwaltung zwischen zusammenhängenden Klassenversionen erübrigt. Ein weiterer Vorteil ist, daß sich den Applikationsentwicklern eine Schemaversion genau wie ein unversioniertes Schema darstellt und damit die gewünschte Transparenz erreicht wird.

3.4 Zusammenfassung und Bewertung

Ausgehend von unseren Erfahrungen im GOODSTEP-Projekt haben wir in diesem Kapitel zunächst ein Szenario aus dem Umfeld des Softwareengineering beschrieben, um eine Vorstellung von den Systemen zu vermitteln, von denen wir in dieser Arbeit ausgehen. Die unterschiedlichen Beweggründe, die in solchen Systemen die Durchführung von Schemaänderungen jeglicher Art erfordern, dienen uns als Motivation für unsere Arbeit.

Im nächsten Schritt haben wir die hier verfolgten Ziele aus dem Bereich Schemaänderungs-Management detailliert vorgestellt. Diese verwenden wir als Maßstab für die Bewertung sowohl von in der Literatur beschriebenen Ansätzen als auch zur Validierung unserer eigenen Konzepte. Natürlich können wir hier nicht alle wünschenswerten oder gar alle denkbaren Anforderungen berücksichtigen. Insbesondere die grundlegenden Eigenschaften von Datenbanksystemen, auf die wir im vorangegangenen Kapitel kurz eingegangen waren, werden als gegeben angenommen. Weitere Leistungsmerkmale wie beispielsweise die Verteilung von Datenbanken in Rechnernetzen sind für unsere Betrachtungen hier nicht von Belang. Auch Aspekte, die in dem hier behandelten Umfeld ggf. von Interesse sein könnten, haben wir bewußt außer Acht gelassen. Dazu gehört etwa der große Bereich der Autorisierung [CFMS94]. Wie beschrieben gehen wir von mehreren Schemaentwicklern und Applikationsentwicklern aus, so daß hier durchaus Bedarf an Konzepten zur Trennung zwischen den Arbeitsbereichen verschiedener Personen besteht. Derartige Aspekte betrachten wir jedoch nur sehr rudimentär.

Abschließend haben wir ein grobes Lösungsmodell entworfen, das auf der Anwendung von Versionierungskonzepten auf Datenbankschemata basiert. Wir konnten skizzieren, wie sich damit die zuvor aufgestellten technischen Teilziele im einzelnen konzeptionell erreichen lassen. Damit haben wir die begründete Hoffnung, Probleme, die herkömmliche Systeme bei der Durchführung von Schemaänderungen haben, überwinden und ein leistungsfähiges Konzept entwickeln zu können. Nach einer Literaturanalyse im folgenden Kapitel werden wir die einzelnen Aspekte unseres Lösungsmodells näher betrachten und eine detaillierte Modellbildung vornehmen.

Kapitel 4

Bestehende Konzepte zur Erreichung der technischen Teilziele

Dieses Kapitel gibt einen Überblick über bestehende Konzepte zur Unterstützung der Schemaevolution. Als ein Ergebnis dieser Literaturanalyse ergibt sich eine Kategorisierung verschiedener Ansätze in vier grundsätzlich verschiedene Vorgehensweisen: isolierte Datenbanken, direkte Schemaevolution, Simulation der Schemaevolution mit Sichten und Einsatz von Versionierungskonzepten. Diese grundsätzlichen Kategorien beschreiben wir im folgenden jeweils zunächst allgemein bevor wir auf Besonderheiten einiger wichtiger Vertreter[24] des jeweiligen Konzeptes eingehen, diese bewerten und vergleichen. Als Maßstab verwenden wir dabei die technischen Teilziele aus Abschnitt 3.2. Die dort beschriebenen Anforderungen entstammen Veröffentlichungen zahlreicher Autoren, welche unterschiedliche Konzepte und Mechanismen zur Lösung der jeweils behandelten Probleme vorschlagen. Um eine einheitliche Beschreibung der Anforderungen verschiedener Quellen zu erreichen, hatten wir in Abschnitt 3.2 bewußt neutrale Formulierungen gewählt, die insbesondere so allgemein gehalten waren, daß sie keinen Lösungsweg implizieren.

Sofern die hier vorgestellten Konzepte die in den technischen Teilzielen formulierten Anforderungen erfüllen, instanziieren wir quasi die generische Beschreibung der Anforderungen mit Blick auf das jeweilige Modell.

4.1 Schemaänderungen ohne persistente Objekte

Das Problem der Durchführung von Schemaänderungen wird in erheblichem Umfang erleichtert, wenn dabei keine persistenten Objekte zu berücksichtigen sind, d.h. wenn das technische Teilziel 3.11 der vollständigen Propagation nicht erreicht werden muß. Der Verlust existierender Objekte ist im produktiven Einsatz sicher selten hinnehmbar. Jedoch kann es in einer frühen Entwicklungsphase durchaus vorkommen, daß einfach noch keine persistenten Objekte existieren.

Unter der genannten Vereinfachung treten dann im wesentlichen noch dieselben Probleme auf, die auch von der allgemeinen Softwaretechnik (ohne speziellen Datenbankbezug) zu lösen sind. Dazu zählen beispielsweise der Compilerbau und die Entwicklung von Software-Bibliotheken et-

[24] Unberücksichtigt blieben hier insbesondere die Datenbanksysteme F2 [AJFL93, AJEFL95], Iris [FBC+87, FAB+89], Postgres [SK91], Rufus [SS94b], TIGUKAT [ÖPB+93, PÖ95] und ZEITGEIST [FJL+88], sowie die in [AJ96, Bar91, Bel95, BM93b, HY95, MMW94, MZ92, MZ93, NR89, Osb89, Rod92b, RCR93, TYH+91, Wal91] beschriebenen Ansätze. Eine kommentierte Bibliographie zu Datenbanksystemen mit Unterstützung für Schemaevolution hat John Roddick [Rod92a] erstellt, eine Evaluierung kommerzieller Systeme findet sich in [RS99]. Auch [CM95] bespricht einige Referenzen im genannten Themengebiet und [JTTW89] faßt einige Beiträge eines Workshops zusammen.

wa für kaufmännische oder wissenschaftliche Berechnungen. Werden bei der Entwicklung und Implementierung objektorientierte Programmiersprachen eingesetzt, so sind dort ähnlich wie hier vorhandene Klassenhierarchien zu ändern und zu ergänzen, d.h. letztlich sind Schemaänderungen durchzuführen. Diese finden dort wie hier im Rahmen eines iterativen Softwareentwicklungsprozesses statt, wobei entsprechend verschiedener Modelle (beispielsweise das iterierte Phasenmodell oder das Spiralmodell) immer wieder Korrekturen und Ergänzungen an vorhandenen Schemata durchzuführen sind. Ergänzungen und Verfeinerungen eines bestehenden Entwurfs werden nach dem Top-Down-Entwurf durch schrittweise Spezialisierung erreicht, die in der Regel durch Hinzufügen von Unterklassen realisiert wird. Dabei wird insbesondere vom Mechanismus des späten Bindens in objektorientierten Systemen profitiert. Aus dessen Vorteilen ergibt sich nämlich die hier benutzte Möglichkeit, Applikationen auf der Basis gewisser Klassen zu entwickeln. Werden diese Klassen später durch Ergänzung um Unterklassen spezialisiert, so besteht keinerlei Notwendigkeit die sie benutzenden Applikationen anzupassen. Damit können vorhandene Applikationen unverändert weiterverwendet werden (Teilziel 3.9), wobei sie durch das dynamische Einbinden der Implementierung ergänzter Unterklassen trotzdem von deren Funktionalitäten Gebrauch machen.

Manchmal wird jedoch auch entsprechend dem Bottom-Up-Entwurf generalisierend vorgegangen, etwa wenn gemeinsame Eigenschaften mehrerer Klassen quasi ausgeklammert werden, indem eine neue, gemeinsame Oberklasse angelegt wird, von der die Eigenschaften dann geerbt werden. Diese neue Oberklasse kann dann ebenfalls als Ausgangspunkt für die Wiederverwendung dienen.

Unter dem Begriff der *Schemaevolution* wollen wir in dieser Arbeit allerdings nur den allgemeinen Fall verstehen, bei dem insbesondere persistente Objekte entsprechend Teilziel 3.11 zu berücksichtigen sind. Entsprechend zählen wir die Lösungsansätze für die hier kurz angerissene Aufgabenstellung nicht zu unseren vier grundsätzlichen Kategorien. Im weiteren Verlauf dieser Arbeit gehen wir nur noch auf den allgemeinen Fall ein.

4.2 Isolierte Datenbanken

4.2.1 Grundsätzliche Vorgehensweise

Jedes Datenbanksystem bietet Möglichkeiten zur Ein- und Ausgabe von Daten. Damit besteht grundsätzlich immer die Möglichkeit Änderungen an einem vorliegenden Schema s manuell in sechs Schritten vorzunehmen (siehe Abbildung 4.1). Dazu muß zunächst eine externe Schemaänderung entsprechend Teilziel 3.4 durchgeführt werden (Schritte (1) bis (3)), woraus ein die veränderten Anforderungen erfüllendes Datenbankschema s' resultiert. Daraufhin wird auf Objektebene der gesamte Inhalt einer existierenden Datenbank db zunächst ausgegeben und in irgendeiner Form in Dateien des Betriebssystems, außerhalb des Datenbanksystems abgelegt (4). Als nächstes müssen die außerhalb des DBMS gespeicherten Daten an das neue Schema angepaßt werden (5). Dies ist wohl der aufwendigste Schritt dieser Vorgehensweise, da dazu in der Regel spezielle Programme zu implementieren sind. Zuletzt sind eine neue Datenbank db' für s' zu erzeugen und die Daten aus der Datei in db' einzuladen (6). Als Resultat liegen schließlich zwei voneinander unabhängige Datenbanken db und db' mit Schemata s und s' vor.

4.2.2 Bewertung der grundsätzlichen Vorgehensweise

Die manuelle Erstellung einer neuen Datenbank nach jeder Schemaänderung erfüllt erwartungsgemäß nicht viele unserer technischen Teilziele. Natürlich kann bei der Erstellung eines komplett neuen Schemas s' jede beliebige Änderung im Vergleich zu s erreicht werden (Teilziel 3.2). Die Änderungen betreffen auch immer das gesamte Schema (Teilziel 3.3) und werden (ausschließlich)

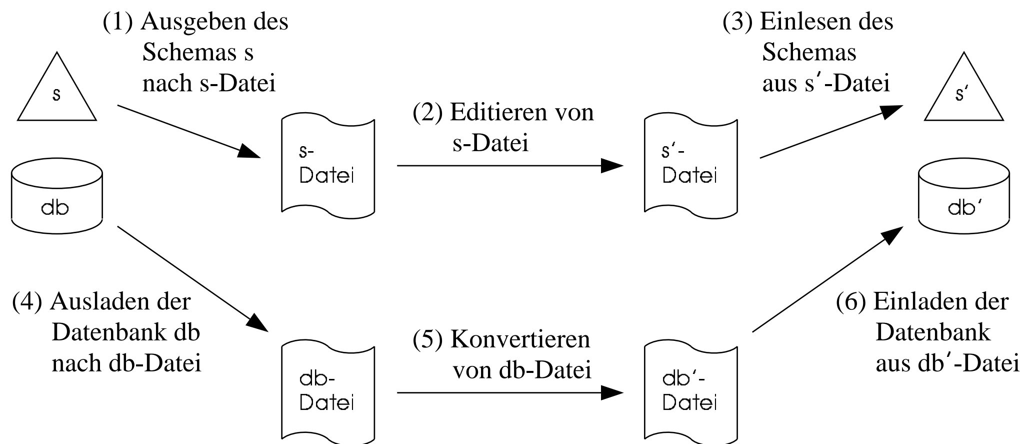


Abbildung 4.1: Vorgehensweise bei der Erstellung isolierter Datenbanken.

extern durchgeführt (Teilziel 3.4). Auch können bisherige Schemazustände extern aufbewahrt und sogar für die Erzeugung verschiedener Alternativen verwendet werden (Teilziele 3.6 und 3.7). Um diese jedoch für Applikationen nutzbar zu machen, müßten mehrere Instanzen des DBMS parallel betrieben werden, und zwar jeweils eine für jeden Schemazustand s und die dazugehörige Ausprägung db der Datenbank. Jede Applikation muß dann entsprechend ihren Anforderungen mit der für sie passenden Instanz des DBMS arbeiten.

Da die für die Umsetzung der ausgeladenen Datenbank benötigten Programme für jede Schemaänderung neu zu erstellen sind, kann hier jede beliebige Form der Umsetzung erreicht werden. Trotz des erheblichen Aufwandes (Teilziel 3.15), der hiermit verbunden ist, können die technischen Teilziele 3.12 (Kontrolle über die Propagation), 3.13 (mehrfache, getrennte Datenwerte) und 3.10 (Kooperation) nicht voll erreicht werden. Es besteht zwar die Möglichkeit durch mehrfaches Ausführen der beschriebenen Vorgehensweise des Ausladens, Konvertierens und Einladens der Daten immer wieder einen Abgleich zwischen den verschiedenen Datenbanken (auch in Rückwärtsrichtung) zu erreichen, dieser Abgleich müßte jedoch immer manuell angestoßen werden und kann weiterhin nur dann durchgeführt werden, wenn gerade keine Applikationen ausgeführt werden. Insbesondere ein automatisches Triggern nach jeder Datenbankänderung ist nicht realisierbar. Außerdem kann so keine Kooperation parallel ablaufender Applikationen verschiedener Zustände s und s' des Schemas stattfinden, da keine Synchronisation realisiert werden kann und ein späterer Abgleich der in verschiedenen Datenbanken db und db' parallel durchgeführten Änderungen i.Allg. nicht möglich ist.

Solange wie in Abschnitt 4.1 beschrieben noch keine Instanzen in der Datenbank db enthalten sind, ist nur Schritt (2) durchzuführen, wodurch sich der Aufwand für die beschriebene Vorgehensweise deutlich verringert und ein praktischer Einsatz denkbar wird. Jedoch bestehen weiterhin erhebliche Defizite bei der Verwaltung der verschiedenen Ausprägungen des Schemas, die komplett manuell durchgeführt werden muß. Sind an einem Schema im Vergleich zu dessen Größe nur geringfügige Änderungen notwendig, stellt eine erneute Spezifikation des gesamten Schemas s' allerdings im Vergleich zu einer Spezifikation der durchzuführenden Änderungen einen nicht zu vernachlässigenden Mehraufwand dar. Sobald eine populierte Datenbank vorliegt, wird der Arbeitsaufwand insbesondere für Schritt (5) so hoch sein, daß in der Praxis viele erwünschte Schemaänderungen gar nicht durchgeführt werden.

4.2.3 Konkrete Vertreter der Vorgehensweise

Wenn ein Datenbanksystem keinerlei Unterstützung für Schemaevolution anbietet, dann ist die beschriebene Vorgehensweise die einzige Möglichkeit, Änderungen am Schema durchführen zu können. Umgekehrt besteht diese manuelle Möglichkeit zur Durchführung einer Schemaänderung in jedem Datenbanksystem. Da dabei keinerlei Unterstützung der Schemaevolution angeboten wird, verzichten wir hier auf die Vorstellung konkreter Vertreter.

4.3 Direkte Schemaevolution

4.3.1 Grundsätzliche Vorgehensweise

Der traditionelle Ansatz zur Unterstützung der Schemaevolution besteht darin, an einem vorliegenden Schema Veränderungen direkt, d.h. an Ort und Stelle (engl. *in place*) vorzunehmen, wobei der alte Zustand wie bei einer Variablenzuweisung überschrieben wird und damit unwiederbringlich verloren geht. Wir nennen diese Vorgehensweise daher *direkte Schemaevolution* (siehe Abbildung 4.2).

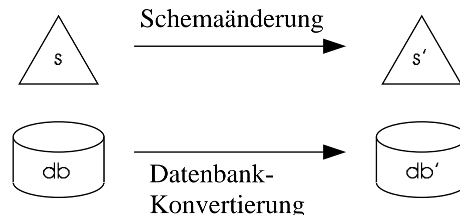


Abbildung 4.2: Der direkte Ansatz zur Schemaevolution.

Zur Spezifikation gewünschter Änderungen stellen Datenbanksysteme eine Menge sog. *Schemaänderungsprimitive* (engl. *schema update primitives*) zur Verfügung, die in einer *Taxonomie* gruppiert sind. Eine solche Taxonomie von Schemaänderungsprimitiven muß natürlich auf das darunterliegende Datenmodell abgestimmt sein, weswegen es keinen generell akzeptierten Satz von benötigten Schemaänderungsprimitiven gibt. Abschnitt 4.3.1.1 stellt eine Taxonomie für das objektorientierte Datenbankmanagementsystem ORION vor.

Durch die Anwendung eines solchen Schemaänderungsprimitives wird ein Schema s in ein modifiziertes Schema s' überführt. Damit ist die Änderung auf Schemaebene abgeschlossen.

Bei nichtleeren Datenbanken müssen gemäß dem technischen Teilziel 3.11 der vollständigen Propagation die Instanzen für Applikationen des veränderten Schemas s' zugreifbar gemacht werden. Die dazu notwendige Konvertierung existierender Objekte wird in der Regel durch klassenspezifische *Konvertierungsfunktionen* geleistet. Bei der reinen Transformation einer Datenbank aufgrund einer Schemaänderung behalten Objekte ihre Klassenzugehörigkeit (es sei denn ihre direkte Klasse wurde gelöscht). Eine einmalige Ausführung einer Konvertierungsfunktion transformiert ein als Parameter übergebenes Objekt dementsprechend von alten Typ seiner Klasse zu ihrem neuen Typ. Konvertierungsfunktionen können vom System vorgegeben (*Defaultkonvertierungsfunktionen*) oder vom Schemaentwickler (*benutzerdefinierte Konvertierungsfunktionen*) angegeben werden. Die Ausführung der Konvertierungsfunktion kann entweder für alle vorhandenen Objekte zum Zeitpunkt der Schemaänderung (*sofortige Propagation*) oder für jedes einzelne Objekt zum Zeitpunkt des ersten Zugriffs auf dieses Objekt (*verzögerte Propagation*) erfolgen.

Manche Systeme [DA99, ADHP00] unterstützen gleichzeitig die *Migration* von Objekten, d.h. die Objekte können ihre Klassenzugehörigkeit ändern. Dies wird durch sog. *Migrationsfunktionen* bewerkstelligt. Abgesehen von dem Fall, daß die direkte Klasse eines Objektes gelöscht

wird, hat die Objektmigration keinen direkten Bezug zur Schemaevolution. Sie wird jedoch mitunter zusätzlich angeboten, da technische Ähnlichkeiten zwischen Konvertierungs- und Migrationsfunktionen bestehen, und zwar sowohl bezüglich der Spezifikation als auch bezüglich der Ausführung. Das erleichtert die Implementierung eines zusätzlichen Migrationsmechanismus.

4.3.1.1 Angebotene Schemaänderungsprimitive

Die erste Taxonomie für Schemaänderungsprimitive [BCG+87, BKKK87] wurde für das OODBMS ORION vorgeschlagen, das wir in Abschnitt 4.5.3.7 noch näher vorstellen werden. Da diese Taxonomie die elementaren Schemaänderungsprimitive enthält, soll sie hier kurz aufgeführt werden.

1 Änderungen am Inhalt eines Knotens der Vererbungshierarchie, d.h. an einer Klasse

- 1.1 Änderungen einer Instanzvariablen, d.h. eines Attributes
  - 1.1.1 Hinzufügung eines neuen Attributes zu einer Klasse
  - 1.1.2 Löschung eines existierenden Attributes einer Klasse
  - 1.1.3 Änderung des Namens eines Attributes einer Klasse
  - 1.1.4 Änderung des Typs eines Attributes einer Klasse
  - 1.1.5 Änderung der Oberklasse eines Attributes, d.h. erben eines gleichnamigen Attributes von einer anderen Klasse
  - 1.1.6 Änderung des Defaultwertes eines Attributes einer Klasse
- 1.2 Änderungen einer Klassenvariablen, d.h. eines für alle Instanzen einer Klasse wertgleichen Attributes
  - 1.2.1 Hinzufügung einer Klassenvariablen
  - 1.2.2 Änderung einer Klassenvariablen
  - 1.2.3 Löschung einer Klassenvariablen
- 1.3 Änderungen an einer Methode
  - 1.3.1 Hinzufügung einer neuen Methode zu einer Klasse
  - 1.3.2 Löschung einer existierenden Methode einer Klasse
  - 1.3.3 Änderung des Namens einer Methode einer Klasse
  - 1.3.4 Änderung des Codes einer Methode einer Klasse
  - 1.3.5 Änderung der Oberklasse einer Methode, d.h. erben einer gleichnamigen Methode von einer anderen Klasse

2 Änderungen an den Kanten der Vererbungshierarchie

- 2.1 Hinzufügen einer Klasse s zu der Oberklassenliste einer Klasse c
- 2.2 Löschung einer Klasse s aus der Oberklassenliste einer Klasse c
- 2.3 Änderung der Ordnung in der Oberklassenliste einer Klasse c

3 Änderungen an der Knotenmenge der Vererbungshierarchie

- 3.1 Hinzufügung einer neuen Klasse
- 3.2 Löschung einer existierenden Klasse
- 3.3 Änderung des Namens einer Klasse

Nach der in [BKKK87] gegebenen Definition ist diese Taxonomie (entsprechend dem dortigen Theorem 2) *vollständig*, d.h. jedes beliebige Ausgangsschema s kann durch Anwendung der genannten Schemaänderungsprimitive in jedes beliebige Zielschema s' überführt werden. Diese Eigenschaft ist für uns jedoch nur von geringem Interesse, da der im Beweis dieser Eigenschaft beschrittene Weg über die Löschung aller Klassen von s und neu Anlegen aller Klassen aus s' führt. Diese theoretische Lösung ist bereits auf Schemaebene wenig praktikabel. Auf Objektebene gehen dabei sogar alle Instanzen verloren, weil das Schema zwischenzeitlich keinerlei benutzerdefinierte Klassen enthält.

Nach [AA93] lassen sich die Schemaänderungsprimitive der vorgestellten Taxonomie auch als Operationen einer Algebra darstellen. In [SGD93] werden insbesondere komplexe Änderungen von Typen in NO^2 (dort *value sets* genannt) klassifiziert und einzeln untersucht.

4.3.2 Bewertung der grundsätzlichen Vorgehensweise

Der Ansatz der direkten Schemaevolution bietet im Vergleich zu dem unabhängiger Datenbanken erhebliche Vorteile.

Auf Schemaebene kann i.Allg. von umfangreichen Schemata ausgegangen werden, an denen relativ zu ihrer Größe wenig verändert werden muß. Daher ist die Spezifikation der anzuwendenden Änderungsprimitive deutlich einfacher als eine komplette Spezifikation des gewünschten Zielschemas (technisches Teilziel 3.1). Weiterhin kann die Spezifikation des Zielschemas durch Konsistenzprüfungen nach jeder Anwendung eines Änderungsprimitives unterstützt werden, da der Schemaentwickler auf Fehler ggf. sofort aufmerksam gemacht werden kann. Schließlich können aufwendige Schemaänderungen bereits als Primitiv vorgegeben werden (technisches Teilziel 3.3). Beispielsweise das Umbenennen einer Klasse muß nämlich nicht nur in der Klasse selbst, sondern auch in ihren Unter- und Kundenklassen[25] geschehen. Ein vorgegebenes, einfach zu benutzendes Primitiv kann alle notwendigen Schritte korrekt ausführen.

Auf Objektebene kann durch Analyse der für die Transformation von Schema s in s' angewendeten Schemaänderungsprimitive auf die notwendige Datenbanktransformation geschlossen und eine entsprechende Menge von Defaultkonvertierungsfunktionen vorgeschlagen werden (Teilziel 3.16).

Jedoch bleiben einige Nachteile des internen Ansatzes ungelöst und es können sich sogar zusätzliche Probleme ergeben.

Zunächst können Schemaänderungen bei Fehlern nicht rückgängig gemacht werden (technisches Teilziel 3.5), weil der alte Datenbankzustand durch das Überschreiben unwiederbringlich verloren geht. Da dieses Risiko nicht eingegangen werden kann, muß vor jeder Schemaänderung eine Sicherheitskopie des Schemas und der gesamten Datenbank angelegt werden, was ein sehr platz- und zeitintensiver Prozeß ist.

Weiterhin kann das Schema nur in eine Richtung entwickelt werden. Da es immer nur einen Zustand des Schemas gibt, besteht keine Möglichkeit, Alternativen zu entwickeln und zu vergleichen oder verschiedene Teile eines Schemas separat und parallel zu verändern.

Das Problem der langwierigen Datenbanktransformation (technisches Teilziel 3.15) bleibt auch im direkten Ansatz zunächst ungelöst. Wenn eine Schemaänderung Auswirkungen auf die Objektebene hat, so wird i.Allg. die gesamte Datenbank während des Transformationsprozesses gesperrt.[26] Während der linear mit der Datenbankgröße steigenden Transformationszeit können somit keine Applikationen ausgeführt werden, auch solche nicht, die nur mit unveränderten

[25] Mit dem Begriff der *Kundenklassen* einer Klasse c sind hier diejenigen Klassen gemeint, die die Klasse c beispielsweise als Komponente nutzen.

[26] Hier existiert allerdings Raum für erhebliche Verbesserungen, insbesondere wenn nur sehr wenige Klassen verändert wurden.

Teilen des Schemas operieren. Weiterhin ist keine sofortige Fortsetzung der Schemaänderungsarbeiten möglich. Wenn die Auswirkungen einer Schemaänderung auf die Objektebene für den Schemaentwickler nicht offensichtlich sind, etwa wenn durch Hinzufügen von Vererbungskanten indirekt neue Attribute in Unterklassen ergänzt werden, so könnten ggf. resultierende längere Stillstandszeiten vom Schemaentwickler als unbegründet erachtet werden.

Ein neues Problem ergibt sich durch die vom Datenbanksystem fest vorgegebene Taxonomie von Schemaänderungsprimitiven. Auch wenn diese in dem Sinne als vollständig zu bezeichnen ist, daß jede gewünschte Schemaänderung durchgeführt werden kann, so werden sich mitunter Situationen ergeben, in denen Primitive benötigt werden, die applikationsspezifisch sind und damit vom Hersteller eines DBMS nicht vorhergesehen werden können. Somit kann sich die Festlegung auf eine bestimmte Menge von Schemaänderungsprimitiven als Einschränkung erweisen (Teilziel 3.4).

Bei der bisher beschriebenen Vorgabe einer zu benutzenden Taxonomie findet jede Schemaänderung unter der Kontrolle des DBMS statt, weshalb diese Vorgehensweise auch als *interner Ansatz* bezeichnet wird. Im Gegensatz dazu erlauben dem *externen Ansatz* folgende Systeme die beliebige Modifikation eines Schemas außerhalb der Kontrolle des DBMS. Dies kann z.B. durch Bearbeitung einer textuellen Schemabeschreibung in einem Texteditor geschehen. Bevor eine extern durchgeführte Schemaänderung wirksam werden kann, muß sie in Form einer neuen Schemabeschreibung s' (in der Regel als ODL-Datei) in das Datenbankmanagementsystem eingebracht werden. Das DBMS steht dann vor der Aufgabe, s und s' miteinander zu vergleichen, um die durchgeführten Veränderungen erkennen zu können. Die Aufgabenstellung ähnelt damit derjenigen der Schemaintegration, auf die wir in den Abschnitten 4.5.4 und 5.5.7 noch eingehen werden. Wir werden das dem externen Ansatz folgende System Tess in Abschnitt 4.3.4.4 näher vorstellen.

Da beim direkten Ansatz zur Schemaevolution das bestehende Schema s nach der Veränderung durch s' überschrieben ist, kann Teilziel 3.9 (Vermeidung manueller Applikationsanpassungen) nicht erfüllt werden. Daher müssen bei jeder Schemaänderung sämtliche existierenden Applikationen beendet, an das neue Schema s' angepaßt und neu gestartet werden, bevor sie wieder benutzt werden können. Insbesondere der manuell durchzuführende Schritt der Applikationsanpassung kann bei zahlreichen Applikationen sehr arbeitsaufwendig sein. Applikationen können in verschiedenen Programmiersprachen implementiert sein und ggf. liegt dem Datenbankbetreiber der Quellcode von Applikationen gar nicht vor, wenn beispielsweise eine Bibliothek von einem Drittanbieter eingekauft wurde. Wir werden in Abschnitt 4.3.4.3 ein System vorstellen, das aufgrund von Schemaänderungen notwendige Applikationsanpassungen automatisch durchführen kann.

4.3.3 Vergleich der Vorgehensweise mit der Schemaversionierung

Der Ansatz der direkten Schemaevolution unterscheidet sich wesentlich von dem der Versionierung. Dies soll hier an verschiedenen Aspekten belegt werden, bezüglich denen wir die beiden Ansätze gegenüberstellen. Die Differenzen werden bereits bei der Betrachtung konzeptioneller Aspekte deutlich. Zwar erlauben beide Ansätze die Durchführung von Schemaänderungen, sie beziehen ihre Motivation jedoch aus gänzlich unterschiedlichen Motiven und richten sich entsprechend anders aus.

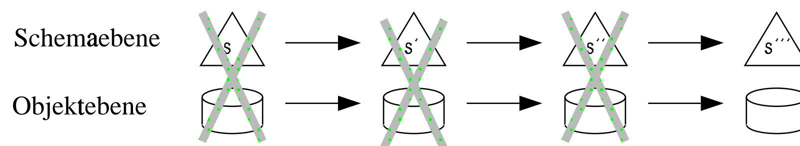
- Die direkte Schemaevolution betrachtet immer nur eine Aufgabe und demzufolge existiert zu jeder Zeit nur ein Schema, das für die gegebene Aufgabenstellung als korrekt empfunden wird. Schemaänderungen sind ausschließlich notwendig, um Korrekturen eines fehlerhaften Entwurfes oder Anpassung an (z.B. durch neue Umweltbedingungen) veränderte Aufgabenstellungen durchführen zu können.

- Die Schemaversionierung geht dahingegen von verschiedenen, gleichzeitig existierenden Bedürfnissen für die Speicherung und Änderung von Information aus. Schemaänderungen dienen damit nicht nur zur Korrektur bestehender Modelle, sondern auch zur Modellierung alternativer Bedürfnisse. Die vor der Änderung modellierte Struktur der Diskurswelt wird mit der Schemaänderung nicht hinfällig, sondern bleibt weiterhin gültig.

Aus den unterschiedlichen Motivationen leiten die beiden Ansätze dann folgerichtig verschiedene Zielsetzungen ab.

- Die direkte Schemaevolution sieht eine Schemaänderung als korrigierenden Eingriff, der eine Verbesserung bzw. Anpassung des Schemas an eine Veränderung der einen, gegebenen Aufgabenstellung erreicht. Daher kann der alte Zustand des Schemas bei der Änderung überschrieben werden (siehe Abbildung 4.3a). Die daraus resultierende Notwendigkeit der sofortigen Anpassung aller Applikationen wird als unumgänglich erachtet, da die Verbesserung natürlich auch in den Applikationen erreicht werden muß.

a) direkte Schemaevolution



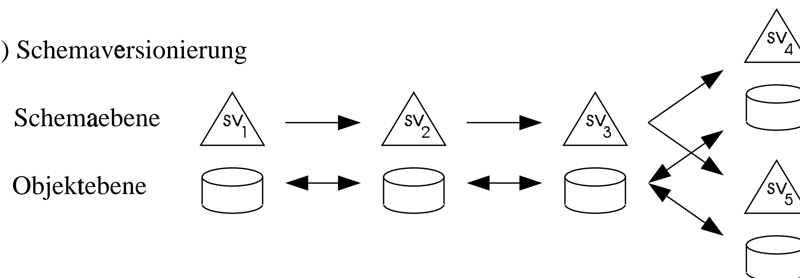


Abbildung 4.3: Vergleich der direkten Schemaevolution mit der Schemaversionierung.

- Die Schemaversionierung betrachtet Schemaversionen dagegen als verschiedene Schnittstellen zu derselben Datenbank und diese können demzufolge auch koexistieren. In Ergänzung der Sichtweise des direkten Ansatzes dienen Schemaänderungen nicht nur der Korrektur und Verbesserung eines gegebenen Modells, sondern auch der Erstellung alternativer Modellierungen (siehe Abbildung 4.3b). Demzufolge werden existierende Applikationen nicht notwendigerweise hinfällig und ihre Anpassung an ein alternatives Modell ist oftmals unerwünscht. Selbst wenn eine Schemaänderung tatsächlich nur eine Korrektur darstellt, so ist die Anpassung existierender Applikationen zumindest zeitaufwendig, wenn nicht sogar unmöglich, etwa weil die Quellen einer Applikation nicht vorliegen. In solchen Fällen wäre die Durchführung von Schemaänderungen bei Verwendung des direkten Ansatzes gar nicht möglich.

Ein dritter Aspekt ist die Behandlung von Änderungen auf der Betrachtungsebene des Schemas.

- Beim direkten Ansatz werden Änderungen unmittelbar an der vorliegenden Spezifikation des Schemas vorgenommen (engl. *update in place*), wobei der alte Zustand verloren geht.

- Bei der Schemaversionierung wird zur Erstellung eines alternativen Modells eine neue Version angelegt, die beliebig definiert werden kann und die von existierenden Versionen

desselben Schemas Gebrauch machen kann, wobei die existierenden Versionen allerdings stets unverändert bleiben.

Beim Aspekt der Abbildung von Schemaänderungen auf die Betrachtungsebene der Objekte schließlich werden die Konsequenzen der unterschiedlichen Zielsetzungen besonders deutlich. Eine Schemaänderung kann sich für Objekte auswirken als Hinzufügen von Eigenschaften, Löschen von Eigenschaften oder Ändern von Eigenschaften. Die in allen Fällen notwendige Anpassung existierender Objekte der Datenbank wird als Konvertierung bezeichnet.

- Dem Ansatz der direkten Schemaevolution folgend wird beim Hinzufügen von Eigenschaften zu Klassen der dafür notwendige Speicherplatz in den betroffenen Objekten ergänzt. Beim Löschen werden die Werte der gelöschten Attribute unwiederbringlich aus den Objekten entfernt.

  Die Konvertierung geschieht nur in Vorwärtsrichtung, d.h. von einem alten zu einem neuen, als Verbesserung empfundenen, Zustand.

  Logisch, d.h. aus der Sicht von Schema- und Applikationsentwicklern, ist die Konvertierung untrennbar mit der Schemaänderung verknüpft und geschieht immer sofort danach. Die physikalische Konvertierung kann durch einen sofortigen oder verzögerten Mechanismus implementiert werden, wobei letzterer den Vorteil bietet, daß damit Totzeiten während der Durchführung umfangreicher Konvertierungen vermieden werden. Dabei ist eine transparente, d.h. für die Entwickler nicht von der sofortigen Konvertierung unterscheidbare Vorgehensweise anzustreben.

  Für die Implementierung der verzögerten Datenbankkonvertierung muß auch beim direkten Ansatz intern eine Liste historischer Schemaversionen verwaltet werden [FMZ94b, FMZ+95b]. Im Gegensatz zum Versionierungsansatz stehen diese dem Schemaentwickler beispielsweise für die Untersuchung von Alternativen allerdings nicht zur Verfügung, obwohl der zusätzliche Aufwand für die Verwaltung eines versionierten Schemas trotzdem entsteht.

- Bei der Schemaversionierung werden Objekten grundsätzlich immer nur Eigenschaften hinzugefügt, *nie* geht etwas verloren. Damit ist es jederzeit möglich auf vorherige Zustände zurückzugehen, beispielsweise um sich als unvorteilhaft erwiesene Änderungen zu verwerfen. Die Schemaversionierung unterstützt also ein schrittweises Entwickeln und Testen von Applikationen und Schema.

  Die Konvertierung geschieht in beide Richtungen, d.h. sowohl von alten zu neuen Schemaversionen (zeitlich vorwärts) als auch von neuen zu alten Schemaversionen (rückwärts). Die Vorwärtskonvertierung erlaubt neuen Applikationen den Zugriff auf zuvor (vor der Erstellung der Schemaversion dieser Applikation) erstellte Objekte, analog erlaubt die Rückwärtskonvertierung alten Applikationen den Zugriff auf Objekte neuerer Applikationen.[27]

  Wie bei der direkten Schemaevolution erfolgt auch hier die logische Konvertierung veränderter Objekte immer sofort. Der Zeitpunkt der physikalischen Konvertierung hängt von der Speicherungsart und von der Implementierungsstrategie ab.

  - Wenn von jedem Objekt nur eine Version gespeichert ist, erfolgt die Konvertierung, sobald auf eine andere Version zugegriffen wird.

[27] Die Bezeichnungen *alte* und *neue Applikationen* beziehen sich in dieser Arbeit nicht auf die Reihenfolge der Erstellung der Applikationen, sondern auf die Erzeugung der Schemaversionen, auf denen diese Applikationen aufsetzen. Natürlich kann auch zu einem späteren Zeitpunkt noch eine Applikation für eine ältere Schemaversion entwickelt werden; diese bezeichnen wir hier jedoch trotzdem als *alte Applikation*, weil für diese Arbeit die Schemaversionen von Interesse sind.

- Wenn Objektversionen für mehrere Schemaversionen gespeichert werden, kann eine Konvertierung erforderlich sein
  * wenn eine neue Schemaversion angelegt wird oder
  * wenn ein Objekt (eine Objektversion) verändert wird. Dazu kann ein Mechanismus verwendet werden, der ähnlich wie die Trigger in aktiven Datenbanksystemen [WC95] funktioniert.

Wie bei der direkten Schemaevolution werden auch hier ggf. mehrere Konvertierungsschritte benötigt, um ein Objekt, das physikalisch entsprechend einer Schemaversion gespeichert ist und in einer anderen Schemaversion zugegriffen werden soll, umzusetzen, wenn die beiden Schemaversionen nicht direkt zueinander in Beziehung stehen.

4.3.4 Konkrete Vertreter der Vorgehensweise

4.3.4.1 Der Ansatz von Penney und Stein

GemStone [BMO+89, BOS91, CM84, MSOP86, TT93] ist ein objektorientiertes Datenbankmanagementsystem, das von Servio Logic Development Corporation auf der Basis der Programmiersprache Smalltalk [GR83] entwickelt wurde.

In [PS87] schlagen Jason Penney und Jacob Stein einen Ansatz zur Unterstützung von Schemaänderungen in GemStone vor. Daher nehmen wir das in [PS87] beschriebene Objektmodell auch als Grundlage unserer Betrachtungen.

Wie Smalltalk-80 beschränkt sich auch GemStone auf ein Objektmodell mit Einfachvererbung. Eine Klasse kann durch Zusicherungen (engl. *constraints*) spezifizieren, welche Attributwerte für ihr angehörende Objekte als konsistent betrachtet werden. Die Zusicherungen einer Klasse verletzende Attributwerte dürfen einem Objekt dieser Klasse nicht zugewiesen werden. GemStone verfügt über ein Autorisierungsmodell, in dem Objekte und Klassen von Benutzern besessen werden und Zugriffe entsprechend verschiedener Rechte kontrolliert werden. Eine Menge von sechs Invarianten beschreibt Anforderungen, denen ein Schema und seine Datenbank nach jeder Operation entsprechen müssen. Zur Vermeidung unauflösbarer Referenzen (engl. *dangling references*) ist kein explizites Löschen von Objekten möglich.

Die Unterstützung für Schemaevolution wurde von Penney und Stein mit dem Ziel entwickelt, ohne die Verwaltung verschiedener Versionen von Klassen auszukommen. Die in [PS87] beschriebenen Schemaänderungsprimitive erlauben die Durchführung von aus praktischen Erwägungen heraus als sinnvoll erachteten Schemaänderungen. Jedoch ist die angebotene Menge von Primitiven nicht vollständig, da beispielsweise keine Möglichkeit besteht, die Oberklasse einer Klasse beliebig zu wechseln.

Für die Behandlung der Auswirkungen von Schemaänderungen auf Objektebene werden zwei Alternativen verglichen. Die erste Möglichkeit besteht in der sofortigen Konvertierung der gesamten Datenbank nach jeder (ggf. aus mehreren Teilschritten bestehenden) Schemaänderung. Die zweite Alternative wäre die nachträgliche Simulation der Schemaänderung mit unveränderten Objekten. Dieses als *Screening* bezeichnete Verfahren führt jedoch bei jedem zukünftigen Objektzugriff zu einem Mehraufwand. Die Alternativen unterscheiden sich für Penney und Stein im wesentlichen durch den Zeitpunkt, zu dem die von der Schemaänderung verursachte Mehrarbeit geleistet wird („pay me now or pay me later"). In beiden Fällen wird eine Schemaänderung daher als sehr teure Operation eingestuft. Angesichts des langen Lebenszyklus einer Datenbank wird für GemStone die sofortige Konvertierung vorgeschlagen.

Neu hinzugefügte Attribute werden bei Objekten der betroffenen Klassen mit **nil** initialisiert. Wird ein Attribut einer Klasse gelöscht, so wird diese Modifikation nicht an Unterklassen wei-

tergegeben. Stattdessen wird so verfahren, als wäre das gelöschte Attribut in den Unterklassen lokal definiert. Damit verändert sich zwar die Beschreibung der Unterklassen; die unter Berücksichtigung der Vererbung resultierende Menge von Attributen einer Klasse bleibt damit jedoch selbst dann unverändert, wenn in einer Oberklasse ein Attribut, das zuvor geerbt wurde, gelöscht wird. Demzufolge kann sich die Anpassung von Objekten auf direkte Instanzen der veränderten Klasse beschränken.

Entsprechend der Invariante über die vollständige Vererbung (engl. *full inheritance invariant*) werden auch Zusicherungen an Unterklassen vererbt; sie dürfen dort jedoch höchstens verstärkt werden, so daß sich der Bereich zulässiger Objektwerte in Unterklassen verkleinert. Eine im Rahmen einer Schemaänderung durchgeführte Verschärfung der Zusicherungen einer Klasse wird allerdings nicht an Unterklassen propagiert. In [PS87] ist jedoch keine Aussage enthalten, wie der entstehende Konflikt zur Invariante über die vollständige Vererbung behoben wird. Attributwerte von Objekten, die die Zusicherungen einer Klasse nach einer Verschärfung nicht mehr erfüllen, werden durch **nil** ersetzt, denn **nil** ist stets ein zulässiger Wert.

Die im Rahmen des Autorisierungsmodells vergebenen Rechte auf Objekte und Klassen unterliegen in GemStone keinen Beschränkungen. Da sich wie beschrieben einige Veränderungen einer Klasse auch auf deren Objekte und Unterklassen auswirken, können hier Konflikte entstehen, wenn die Person, die eine Klassenänderung durchführt, nicht die notwendigen Rechte auf den indirekt betroffenen Objekten und Unterklassen hat. Eine angedeutete Lösung des Problems durch die Versionierung von Klassen wird allerdings bewußt nicht verfolgt.

Penney und Stein bemerken, daß die explizite Verwaltung von Klassenextensionen durch zusätzliche Datenstrukturen im Rahmen der Schemaevolution zusätzliche Vorteile erbrächte, da damit bei Klassenänderungen nicht die gesamte Datenbank nach Objekten der veränderten Klasse durchsucht werden muß.

Die verhaltensmäßige Konsistenz von Klassenänderungen wird nicht näher untersucht, aber als offener Punkt für zukünftige Forschungstätigkeiten betont.

Da der Ansatz von GemStone im Vergleich zu der zuvor betrachteten allgemeinen Vorgehensweise der direkten Schemaevolution keine für unsere Betrachtungen ausschlaggebenden Details enthält, können wir hier auf eine separate Bewertung verzichten.

4.3.4.2 Der Ansatz von Ferrandina und Zicari

In seinen frühen Arbeiten im Rahmen des ESSE-Projektes (Environment Supporting Schema Evolution) [CPLZ92d, CPLZ92b] untersucht Roberto Zicari bereits Schemaänderungen in objektorientierten Datenbanksystemen, insbesondere in O_2 [Zic89a, Zic89b, Zic91a, Zic91b, Zic92]. Dabei unterscheidet er strukturelle und verhaltensmäßige Konsistenz von Schemata und schlägt ein interaktives Werkzeug namens *Interactive Consistency Checker (ICC)* vor, das den Schemaentwickler bei der Sicherstellung der Konsistenz der von ihm erstellten Schemata unterstützen soll. Ein weiteres von der Gruppe um Zicari entwickeltes Werkzeug beschäftigt sich speziell mit dem Überschreiben von Attributen und Methodenparametern in Unterklassen. Wie sehr viele objektorientierte Programmiersprachen[28] verwenden auch viele Datenbanksysteme, u.a. O_2, das Modell des kovarianten Überschreibens [ES94]. Im Gegensatz zur Kontravarianz kann bei Verwendung der Kovarianz keine strenge Typprüfung durchgeführt werden, d.h. Typfehler zur Laufzeit lassen sich nicht generell ausschließen. Das Werkzeug *Type Data Flow (TDF)* [CPLZ92d, CPLZ92a, CPLZ92c, DZ91, HS95, HS97] analysiert die dynamischen Typen von Variablen in Applikationen von O_2 und weist ggf. auf potentielle Probleme hin. Da eine exakte Analyse nicht möglich ist, geht TDF pessimistisch vor. Dabei werden dem Applikationsentwickler

[28] Trellis/Owl [OHK87, SCB+86] stellt durch die Verwendung des kontravarianten Modells eine der äußerst seltenen Ausnahmen dar.

zumindest alle Programmabschnitte angezeigt, in denen zur Laufzeit Typfehler entstehen können, jedoch werden ggf. auch typsichere Abschnitte angezeigt, wenn TDF dort die Typsicherheit nicht nachweisen kann.

In seinen gemeinsamen Arbeiten mit Fabrizio Ferrandina und anderen untersucht Zicari später auch den Einfluß von Schemaänderungen auf persistente Objekte der Datenbank [FMZ94b, FMZ94a, FMZ+95b, FL96, Zic97]. Auf die dort vorgestellten Konzepte wollen wir hier näher eingehen.

Das objektorientierte Datenbankmanagementsystem O_2 [BDK92, Ban93, Dea91, LRV88, LRV90, VDDW+90] entstand im Rahmen eines fünfjährigen Projektes, das 1986 vom Forschungskonsortium Altaïr begonnen wurde. Seither wird O_2 [O2 96a, O2 96c, O2 96b] kommerziell weiterentwickelt und vermarktet und zwar zunächst von der 1991 gegründeten Firma O_2 Technology, die dann 1999 in Ardent Software überging.

Das Datenmodell von O_2 ist in verschiedenen Veröffentlichungen durchgängig und formal spezifiziert [BDK92, Dea91, LRV88, LRV90]. Es bietet zahlreiche, orthogonale Typkonstruktoren und erlaubt Mehrfachvererbung, wobei Konflikte durch Umbenennung von Attributen und Methoden beseitigt werden. Methoden können überschrieben und überladen werden, parameterlose Methoden können auch durch Attribute überschrieben werden. O_2 verwendet das Modell der Persistenz durch Erreichbarkeit. Die Anfragesprache von O_2 heißt OQL (früher O_2SQL) und wurde von der ODMG[29] als Standard übernommen [BF95, Cat96, CB98, CBB+00]. Applikationen können in den Programmiersprachen C++ und Java (vormals auch Smalltalk) implementiert und mit handelsüblichen Compilern übersetzt werden. Weiterhin steht mit O_2C (früher CO_2 genannt) eine eigene Sprache als Erweiterung von C zur Verfügung, die datenbankspezifische Aspekte besonders gut integriert.

Zahlreiche Arbeiten [FMZ94b, FMZ94a, FMZ+95b, FL96, Zic89a, Zic89b, Zic91a, Zic91b] machen Vorschläge zur Unterstützung der Schemaevolution in O_2. Die meisten davon bieten eine Taxonomie von elementaren, auf das O_2-Datenmodell zugeschnittenen Primitiven. In [Brè96] werden darüberhinaus komplexe Schemaänderungsoperationen[30] für O_2 beschrieben.

Wir wollen im folgenden etwas näher auf die in [FMZ94b, FMZ94a, FMZ+95b] beschriebenen Konzepte von Fabrizio Ferrandina, Thorsten Meyer, Roberto Zicari, Guy Ferran und Joëlle Madec eingehen, da diese teilweise in Version 4.6 des Produktes [O2 96b] verfügbar sind.

Entsprechend der verwendeten Schemaänderungsprimitive stellt das System Defaultkonvertierungsfunktionen zur Verfügung, die Umbenennungen und Typänderungen von Attributen handhaben und vom Schemaentwickler modifiziert werden können.

Die genannten Arbeiten konzentrieren sich auf die Verbesserung der Effizienz der Datenbanktransformation (technisches Teilziel 3.15). Einige einfache Änderungen des Schemas, wie z.B. das Hinzufügen eines neuen Attributes zu einer existierenden Klasse c, erfordern auf Objektebene eine Transformation der Datenbasis. Im Beispiel müssen sämtliche Objekte der Klasse c um einen (Default-) Wert für das neue Attribut erweitert werden. Wenn solche Datenbanktransformationen nicht nur logisch, sondern auch physikalisch sofort nach jeder Schemaänderung durchzuführen sind, ergeben sich die bereits in Abschnitt 4.3.2 beschriebenen Nachteile. Deshalb wird von den Autoren ein *verzögerter Transformationsmechanismus* vorgeschlagen, der die physikalische Datenstruktur von der logischen Sicht auf die Objekte entkoppelt. Damit müssen Transformationen auf der physikalischen Ebene nicht mehr notwendigerweise sofort nach jeder Schemaänderung durchgeführt werden. Stattdessen genügt es die logische und die physikalische

[29] Die *Object Database Management Group (ODMG)* ist ein loser Zusammenschluß führender Hersteller von OODBMS, die eine Standardisierung elementarer Eigenschaften ihrer Systeme zur Verbesserung der Austauschbarkeit zum Ziel haben.

[30] Philippe Brèche nennt die elementaren Primitive *low level primitives* (LLPs) und die komplexen Schemaänderungsoperationen *high level primitives* (HLPs).

Sicht auf ein Objekt durch Ausführung der Konvertierungsfunktionen erst dann abzugleichen, wenn eine Applikation des neuen Schemas s' tatsächlich auf dieses Objekt zugreift. Dabei sollte der verzögerte Mechanismus natürlich genau dieselben Objektwerte liefern wie eine sofortige Propagation. Diese Eigenschaft wird als *Zeitäquivalenz* (engl. *time-equivalence*) bezeichnet.

Zur Realisierung der verzögerten Datenbanktransformation werden in [FMZ94b, FMZ+95b] Datenstrukturen vorgeschlagen und Algorithmen sowohl für die sofortige als auch für die verzögerte Propagation angegeben. Dabei werden Objektwerte bei Schemaänderungen physikalisch nicht überschrieben, sondern der aus der Konvertierung resultierende, dem neuen Schemazustand entsprechende Objektwert wird an den bisherigen angehängt. Damit von jedem Objekt stets nur ein Wert logisch sichtbar ist, werden die anderen bei einem Zugriff durch eine Applikation ausgeblendet, weswegen diese Technik auch als *Screening* bezeichnet wird. In [FMZ95a, HVZ90] finden sich des weiteren Kostenanalysen und Laufzeitmessungen auf der Basis des OO7-Benchmarks [CDWN93]. In [FMZ+95b] werden auch die bereits in Abschnitt 4.3.1 erwähnten Migrationsfunktionen studiert.

Jedoch kann die Zeitäquivalenz durch die angegebenen Algorithmen nicht garantiert werden, d.h. es kann vorkommen, daß die verzögerte Propagation einen anderen Wert für ein Objekt liefert, als es die sofortige Propagation getan hätte. Dieses Problem entsteht jedoch nur bei der Verwendung sog. *komplexer Konvertierungsfunktionen*, die zur Ermittlung des neuen Objektwertes nicht nur den alten Wert des Objektes, sondern auch den alten Wert von referenzierten Objekten verwenden. Wird nun in einer späteren Schemaänderung ein von einer komplexen Konvertierungsfunktion benötigtes Attribut aus der Klasse eines referenzierten Objektes gelöscht, so kann es passieren, daß die beiden Schemaänderungen in veränderter Reihenfolge auf die physikalischen Datenstrukturen abgebildet werden, d.h. die Löschung in dem referenzierten Objekt wird ggf. vor der Ausführung der komplexen Konvertierungsfunktion durchgeführt. Diese kann dann aber nicht mehr auf den Wert des gelöschten Attributes des Komponentenobjektes zugreifen.

Zur Sicherstellung der Zeitäquivalenz bei der verzögerten Ausführung komplexer Konvertierungsfunktionen schlagen Edi Fontana und Yves Dannebouy [FD95] vor, ausnahmslos alle Objektzustände durch Verwendung eines Objektversionierungsmechanismus aufzubewahren. Dabei steht der erreichte Nutzen jedoch in keinem Verhältnis mehr zu dem gestiegenen Speicherplatzbedarf weswegen diese Möglichkeit in der Praxis wohl ausscheidet.

Da zahlreiche Schemaänderungen durchgeführt werden können, bevor ein Objekt aufgrund eines Zugriffs physikalisch konvertiert wird, muß eine zeitlich geordnete Liste aller historischen Schemazustände systemintern verwaltet werden. Der dafür sowieso zu treibende Mehraufwand wird allerdings nicht genutzt, um dem Schemaentwickler zusätzliche Möglichkeiten etwa der Analyse der Evolution des Schemas zur Verfügung zu stellen. Stattdessen sieht der Schemaentwickler genau wie die Applikationsentwickler immer nur einen Zustand des Schemas.

4.3.4.3 Anpassung der Applikationen mit Demeter

Die Gruppe um Karl Lieberherr beschäftigt sich mit der Programmierung in objektorientierten Sprachen. Dazu wurden Regeln für guten Programmierstil aufgestellt und als *Law of Demeter* [LHR88, LH89] veröffentlicht, die beispielsweise auch auf die Programmiersprache C++ angewendet werden können [Sak88a]. Darauf aufbauend wird ein Ansatz zur Schemaevolution vorgeschlagen, der sich von den anderen hier dargestellten Ansätzen wesentlich unterscheidet. Anstatt die Existenz verschiedener Ausprägungen eines Schemas und darauf basierender Applikationen hinzunehmen und die sich daraus auf der Objektebene ergebenden Notwendigkeiten der Propagation zu bewerkstelligen, wird in verschiedenen Arbeiten [BH93, Ber94, Ber97, Hür95, HS96] untersucht, wie sich Methoden von Klassen automatisch an ein verändertes Schema anpassen lassen. Dieser Idee zufolge werden sämtliche Applikationen eines Schemas bei dessen Änderung

sofort und ohne manuelles Hinzutun eines Programmierers angepaßt. Damit arbeiten alle Applikationen stets auf der neuesten Ausprägung des Schemas; alte Schemazustände müssen nicht aufbewahrt werden. Auf der Objektebene muß demzufolge nur noch eine einzige Konvertierung der Datenbank in Vorwärtsrichtung vorgenommen werden. Spätere Propagationen von Objekten sind weder in Vorwärts- noch in Rückwärtsrichtung notwendig.

Der Ansatz basiert auf einer abstrakten Schemabeschreibung und kann daher auf verschiedene objektorientierte Programmiersprachen angewendet werden. Jede gewünschte Schemaänderung kann durch Hintereinanderausführung elementarer Änderungsprimitive ausgeführt werden. Daher genügt die separate Untersuchung der nach Anwendung eines elementaren Primitives durchzuführenden Anpassungen in den Methoden des veränderten Schemas.

Die Darstellung eines Schemas erfolgt in einem *Klassengraph*[31] (engl. *class dictionary graph, CDG*), der aus jeweils zwei Typen von Knoten und Kanten besteht. Knoten stellen Klassen dar, deren Beziehungen untereinander durch Kanten dargestellt werden. *Konstruktionskanten* repräsentieren eine Komponenten- (**is_part_of**) bzw. Assoziationsbeziehung (**has_a**) und sind mit dem Namen der *Parts* beschriftet. Diese Parts entsprechen Attributen oder Methoden in konkreten Programmiersprachen. Die unbenannten *Alternativkanten* repräsentieren eine Vererbungsbeziehung (**kind_of**) zwischen Klassen. *Alternativknoten* repräsentieren innere Klassen der Vererbungsstruktur. Diese inneren Klassen sind abstrakt, können also nicht instanziiert werden. Daher muß von jedem Alternativknoten mindestens eine Alternativkante ausgehen. Diese Alternativkante endet bei einer Unterklasse des Alternativknotens. *Konstruktionsknoten* stellen instanziierbare Klassen dar, welche Blätter der Vererbungsstruktur sind. Diese dürfen demzufolge keine ausgehenden Alternativkanten besitzen.

Die Konsistenz eines Schemas wird durch Bedingungen an den CDG beschrieben und muß nach jeder Schemaänderung sichergestellt sein. Von den acht angebotenen Primitiven sind fünf semantikerhaltend (engl. *object equivalence relation*) und drei semantikerweiternd (engl. *extension relation*).

Die Anpassung der Methoden kann nach einer Schemaänderung automatisch vorgenommen werden; sie ist bei typisierten Sprachen allerdings aufwendiger als bei untypisierten. Wenn alle Applikationen in objektorientierten Sprachen implementiert wurden und im Quellcode vorliegen, so ist damit eine automatische Anpassung an ein verändertes Schema möglich. Jedoch bestehen einige Einschränkungen. Auf semantikreduzierende Schemaänderungen (Teilziel 3.1) wird nicht eingegangen, da die dadurch notwendig werdenden Methodenanpassungen i.Allg. nicht immer durchgeführt werden können und daher semantisches Wissen des Schemaentwicklers erfordern. Die Vorgabe, daß alle inneren Klassen der Vererbungsstruktur abstrakt sein müssen, stellt eine erhebliche Vereinfachung dar und macht zusätzliche Klassen notwendig. Die semantikerhaltenden Primitive haben keinerlei Einfluß auf die Objekte; die Reskalierung eines Attributes ist mit den vorgestellten Primitiven beispielsweise nicht möglich.

Wenn sämtliche Applikationen im Quellcode vorliegen, werden die technischen Teilziele 3.9 (Vermeidung manueller Applikationsanpassungen) und 3.10 (Kooperation) mit dem vorgestellten Ansatz voll erreicht, wenn auch nicht alle Schemaänderungen möglich sind (technisches Teilziel 3.2). Dazu wird eine vollständige Propagation (Teilziel 3.11) durchgeführt, die jedoch nicht steuerbar ist. Die Vorgehensweise ist für den Applikationsentwickler nicht transparent (technisches Teilziel 3.19), da er sich nach jeder Schemaänderung mit den vom System an seinen Applikationen automatisch durchgeführten Änderungen vertraut machen muß. Ein Protokoll der an den Methoden durchgeführten Änderung kann dabei zwar hilfreich sein; eine Umgewöhnung der Programmierer und eine manuelle Anpassung der Dokumentation sind aber sicher notwendig.

[31] Eine formale Definition des Klassengraphen findet sich in [Ber94].

Weitere Untersuchungen der Auswirkungen von Schemaänderungen auf existierende Applikationen wurden von William Griswold [Gri91] und von William Opdyke [Opd92] angestellt. Auch [LZHL97] verfolgt einen ähnlichen Weg wie Demeter.

4.3.4.4 Der externe Ansatz von Lerner und Habermann

Barbara Staudt Lerner und Nico Habermann schlagen Konzepte zur Durchführung externer Schemaänderungen ähnlich wie in Teilziel 3.4 vor. Die dabei zugrunde gelegte Ausgangslage unterscheidet sich etwas von der in dieser Arbeit beschriebenen Situation. Wir waren bisher davon ausgegangen, daß ein vollständiges Datenbanksystem vorliegt, das zur Beschreibung des Schemas eine eigene Schemadefinitionssprache anbietet. Diese kann von den Programmiersprachen, in denen die Applikationen implementiert werden, vollkommen unabhängig sein und sogar ein eigenes Objektmodell verwenden. Lerner und Habermann gehen jedoch von der Verwendung sog. *persistenter Programmiersprachen* aus. Dabei existiert keine strikte Trennung zwischen der Schemadefinition und der Applikationsentwicklung. Stattdessen beschreibt eine Applikation alle von ihr verwendeten Typen[32] gemeinsam mit der Applikationslogik in einer speziellen Programmiersprache. Diese stellt meist eine Erweiterung einer bekannten Sprache um Konzepte zur Handhabung persistenter Objekte dar und wird dann als persistente Programmiersprache bezeichnet. Dabei liegt normalerweise ein Datenbanksystem zugrunde, das die Implementierung von Applikationen nur in der persistenten Programmiersprache erlaubt und deren Datenmodell ist mit demjenigen des Datenbanksystems identisch, womit eine besonders gute Integration gelingt. Durch die Beschränkung auf eine einzige Programmiersprache entfällt nämlich die Notwendigkeit zur Einführung spezieller Regeln und Mechanismen, die die Abbildung zwischen dem Objektmodell des Datenbanksystems und dem der jeweils verwendeten Programmiersprache erledigen.[33] Beispiele für kommerzielle, objektorientierte Datenbanksysteme, die eine der gegebenen Definition entsprechende persistente Programmiersprache verwenden, sind ObjectStore [LLOW91, Obj96] und Poet [Poe95] für C++[34] sowie GemStone [BMO+89, BOS91, CM84, MSOP86, TT93] und Versant [Ver98] für Smalltalk. Weiterhin begann ORION [BCG+87, CK86, CK88, KBC+89] als persistente Form der Programmiersprache LISP.

Bei der Verwendung einer persistenten Programmiersprache geschieht die Beschreibung des Schemas also in der Applikation selbst. Dazu bietet die persistente Programmiersprache ein Typkonzept an, das orthogonal zur Persistenz ist, d.h. persistente und transiente Typen werden auf dieselbe Art und Weise beschrieben und es entscheidet sich erst zur Laufzeit, ob ein Objekt persistent oder transient ist und dies erfolgt insbesondere unabhängig von dem Typ, dem das Objekt angehört. Damit ist das Schema einer Applikation dem der Datenbank identisch (oder eine Teilmenge davon). Dieser Vorgehensweise folgend bietet eine persistente Programmiersprache keine Konzepte für die Durchführung von Schemaänderungen an. Das Schema einer Applikation muß stattdessen bei deren Übersetzung extrahiert und das Schema der Datenbank notwendigenfalls entsprechend erweitert oder modifiziert werden. Demzufolge ist die Ausgangslage prinzipiell dieselbe wie in Teilziel 3.4: Das Schema einer vorliegenden Datenbank muß an eine zweite, von außen hinzukommende Schemabeschreibung angepaßt werden, wobei keinerlei Information dar-

[32] Das von Lerner und Habermann zugrunde gelegte Datenmodell ist nicht objektorientiert. Das Datenbankschema enthält lediglich Typen, jedoch keine Klassen. Dadurch fehlt u.a. das Konzept der Vererbung, das sowohl bei den Schemaänderungsoperationen als auch bei der späteren Analyse vorteilhaft hätte berücksichtigt werden können. Auf der Instanzenebene sprechen wir hier jedoch trotzdem von Objekten und nicht von Datensätzen.

[33] Zum ODMG-Standard, der ein sprachunabhängiges Datenbanksystem mit eigenem Datenmodell definiert, gehört demzufolge auch die Beschreibung der Abbildungen (engl. *mapping*) zwischen den Objektmodellen der verwendbaren Programmiersprachen und dem der Datenbank. Dieses Mapping ist in [CB98, CBB+00] in separaten Kapiteln für die Programmiersprachen C++, Java und Smalltalk beschrieben.

[34] ObjectStore und Poet unterstützen zwar auch Java; diese Programmiersprache ist konzeptionell jedoch nur eine Teilmenge von C++ und demzufolge hier nicht als zweite Programmiersprache zu werten.

über vorliegt, wie die extern erzeugte Schemabeschreibung aus dem Schema der persistenten Datenbank hervorgegangen ist. Damit sind die in den Arbeiten von Lerner und Habermann vorgeschlagenen Konzepte direkt auf unsere Aufgabenstellung übertragbar, wenngleich ihnen kein objektorientiertes Datenmodell zugrunde liegt.

Basierend auf TransformGen [GKL94] stellen Barbara Staudt Lerner und Nico Habermann mit OTGen [LH90] und Tess (Type Evolution Software System) [Ler94, Ler96, Ler97, Ler00] Systeme vor, die insbesondere die externe Durchführung der im technischen Teilziel 3.3 erwähnten komplexen Schemaänderungen (von Lerner *compound type changes* genannt) gestatten. Dazu gehören die Ersetzung eines Referenztyps durch den Typ des referenzierten Typs (engl. *inline*), die Umkehrung von Inline, also die Erzeugung eines neuen Typs durch Zusammenfassung von Eigenschaften bestehender Typen (engl. *encapsulate*),[35] die Ersetzung mehrerer Typen durch einen neuen Typ, der die Eigenschaften der ursprünglichen Typen ähnlich einem Join vereinigt (engl. *merge*), das Kopieren (engl. *duplicate*) oder Verschieben (engl. *move*) eines Teils einer Typdefinition in eine andere Typdefinition (wobei auf Objektebene Attributwerte verglichen werden), die Umkehrung der Richtung eines Referenztyps, d.h. der ursprünglich referenzierte Typ referenziert dann den ursprünglich referenzierenden Typ (engl. *reverse link*) und die Hinzufügung einer Referenz zwischen zwei existierenden Typen (engl. *link addition*).

Grundlage der Analyse von TransformGen, OTGen und Tess sind jeweils die Beschreibungen zweier Zustände eines Schemas. Der ältere der beiden Zustände wurde dabei dem externen Ansatz folgend außerhalb der Kontrolle des Datenbanksystems in den neueren überführt und das Ziel der Analyse ist die Feststellung, wie dies geschah. Beim Vergleich nichtleerer Schemazustände gibt es stets mehrere Wege, den einen in den anderen zu überführen. Dabei kommen zumindest immer zwei Alternativen in Betracht. Die eine besteht darin, sämtliche Komponenten des Ausgangsschemas zu löschen und sämtliche Komponenten des Zielschemas neu anzulegen; die zweite Alternative basiert auf Umbenennungen und Typänderungen bestehender Komponenten. Insbesondere unter Berücksichtigung der oben genannten komplexen Schemaänderungsoperationen ergeben sich jedoch noch sehr viel mehr Alternativen. Während auf Schemaebene alle Alternativen das Ausgangsschema in dasselbe Zielschema überführen und damit als äquivalent betrachtet werden können, ergeben sich auf Objektebene jedoch erhebliche Unterschiede. Wenn das Verschieben oder Kopieren von Attributen erkannt wird, so können auch Attributwerte verschoben oder kopiert werden, so daß diese auch nach der Schemaänderung ihre Semantik behalten. Anderenfalls muß davon ausgegangen werden, daß die zusätzlichen Attribute durch Neuanlegen entstanden sind und demzufolge können diese bestenfalls mit Defaultwerten initialisiert werden, wobei die zuvor enthaltene Semantik vollkommen verloren geht. Daher ist die korrekte Erkennung der durchgeführten Schemaänderungsoperationen unbedingt erforderlich.

Neben der Beschreibung der genannten komplexen Schemaänderungsoperationen werden in [GKL94, Ler94, Ler96, Ler97, Ler00] die zur Analyse verwendeten Algorithmen grob vorgestellt. Es fehlt jedoch eine grundlegende und formale Analyse der Problematik. Insbesondere können Fälle auftreten, in denen auf verschiedene Art und Weise Semantik von den Objekten des Ausgangsschemas zu denen des Zielschemas übertragen wird, d.h. die Objekte sind je nach gewählter Alternative mit unterschiedlichen Werten belegt. Solche Konflikte werden nicht behandelt und aus der ungenauen und unvollständigen Beschreibung der Algorithmen sind keine näheren Rückschlüsse möglich. Insbesondere eine formale Auflösung von Konflikten der geschilderten Art wäre wünschenswert. Auch hätten bei Annahme eines objektorientierten Datenmodells Vererbungsstrukturen berücksichtigt werden können, die durch Analyse von Graphisomorphismen weitere Anhaltspunkte zur Bewertung durchgeführter Schemaänderungen und zur Auflösung der genannten Konflikte ergeben könnten.

[35] Die Operationen inline und encapsulate werden von Tresch [TS93c, Tre95] auch *Nestung* (engl. *nest*) und *Entnestung* (engl. *unnest*) genannt.

Durch die Möglichkeit den Grad der Automatisierung der Analyse einstellen und gewisse Äquivalenzen vorgeben zu können, stellen die Werkzeuge trotz der genannten Schwachpunkte sinnvolle Hilfen bei der Erkennung extern durchgeführter Schemaänderungen dar, wenngleich nie für die Korrektheit der Analyse Gewähr übernommen werden kann, da ein automatisches Werkzeug die benutzerdefinierte Semantik eines Schemas nicht erfassen kann. Die Qualität der automatischen Analyse ist den Autoren zufolge umso besser, je geringer die Unterschiede zwischen den zu vergleichenden Schemabeschreibungen ausfallen.

Auf die Ableitung von Konvertierungsfunktionen aus den erkannten Schemaänderungen und auf die Steuerung der Propagation gehen die Autoren nicht näher ein, so daß diesbezüglich keine Bewertung vorgenommen werden kann. Die vorgestellten Mechanismen sind sowohl in Systemen mit sofortiger Umsetzung der Objekte, als auch in solchen mit Screening oder Versionierung einsetzbar. Unterschiede zwischen diesen Varianten werden jedoch nur angedeutet.

4.4 Simulation von Schemaänderungen durch Sichten

4.4.1 Sichten in objektorientierten Datenbanksystemen

Das Konzept der *Sichten* (engl. *views*) wurde zunächst für das relationale Datenmodell [Cod70, Cod79] entwickelt und leistet dort gute Dienste im Bereich der Datenabstraktion. Es ist in der Praxis fester Bestandteil vieler relationaler Datenbanksysteme. Daher werden zahlreiche Vorschläge gemacht, wie man Sichten auf objektorientierte Datenmodelle übertragen kann [AB91, BK93, Ber92, GZ94, GPZ88, HZ90, MM91, MP96, Run92, dSAD94, SS89]. Im Gegensatz zu dem einheitlichen, relationalen Datenmodell existieren sehr viele Varianten des objektorientierten Datenmodells und für jede dieser Varianten wurden andere Sichtkonzepte vorgeschlagen. Diese uneinheitliche Ausgangslage ist wohl mit dafür verantwortlich, daß kaum ein kommerzielles objektorientiertes Datenbanksystem erhältlich ist, das Sichten unterstützt.

Der Sichtenmechanismus von Bertino [Ber92] berücksichtigt im Vergleich zu relationalen Systemen zusätzlich die Aspekte Vererbung und Objektidentifikatoren. In [GPZ88] werden Datenbankänderungen auf der Basis einer formalen Definition von Sichten und deren Konsistenz ohne Bezug zu einem speziellen Datenmodell untersucht. Bei *statischen Sichten* müssen Änderungen der Basisdatenbank (durch sog. *abstraction functions*) auf Sichtdatenbanken abgebildet werden. *Dynamische Sichten* erlauben zusätzlich direkte Änderungen der Sichtdatenbank, die dann (durch sog. *translator*) auf die Basisdatenbank abgebildet werden.

4.4.2 Grundsätzliche Vorgehensweise

Wenn ein Datenbanksystem einen Sichtenmechanismus [AB91, BK93, Ber92, GZ94, GPZ88, HZ90, MM91, Mot87, Run92, dSAD94, SS89] anbietet, so besteht eine besondere Möglichkeit zur Durchführung von Schemaänderungen. Diese besteht darin, sämtliche Schemaänderungen gar nicht am Basisschema selbst durchzuführen, sondern an Sichten, die auf diesem Basisschema definiert sind (siehe Abbildung 4.4). Wenn alle Applikationen auf Sichten aufsetzen und nicht direkt auf das darunterliegende Basisschema zurückgreifen, so können Schemaänderungen durch Veränderungen der Sichten gleichsam simuliert werden. Idealerweise ist dabei keinerlei Änderung des Basisschemas notwendig. Stattdessen werden nur Sichten modifiziert. Falls die alte Definition einer Sicht noch von bestehenden Applikationen benötigt wird, so wird anstelle einer Änderung einfach eine zusätzliche Sicht ergänzt. Diese kann direkt auf dem Basisschema oder auf einer bereits existierenden Sicht definiert werden.

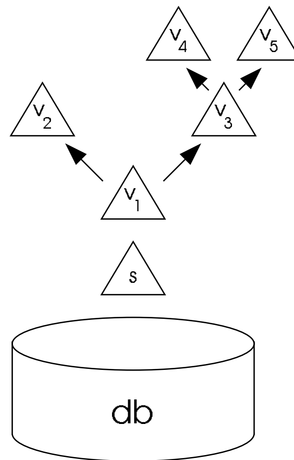


Abbildung 4.4: Simulation von Schemaänderungen durch Sichten.

4.4.3 Bewertung der grundsätzlichen Vorgehensweise

Die Verwendung von Sichten zur Simulation von Schemaänderungen entspricht nicht genau dem eigentlichen Sinn der zugrunde liegenden Konzepte. Nach Odberg [Odb95] können die unterschiedlichen Ansatzpunkte und Ziele etwa wie folgt beschrieben werden.

- Sichten beschäftigen sich damit, wie neue Abstraktionen von existierenden abgeleitet werden können und wie Erzeugung, Löschung und Veränderung von Instanzen der ersteren auf letztere abgebildet werden können.
- Mechanismen zur Unterstützung der Schemaevolution beschäftigen sich mit Modifikationen existierender Abstraktionen und damit wie persistente Instanzen einer Schemaausprägung konsistent durch andere Ausprägungen referenziert werden können.

Die Idee der Simulation von Schemaänderungen durch den Einsatz von Sichten verspricht allerdings viele Vorteile und wird in zahlreichen Arbeiten verfolgt, von denen wir einige im Anschluß an diesen Abschnitt vorstellen werden.

Sichtenmechanismen bieten sprachliche Möglichkeiten zur Spezifikation von Sichten an. Dabei kann sowohl auf das zugrunde liegende Basisschema als auch auf bereits existierende Sichten Bezug genommen werden. Die Spezifikation einer Sichtklasse geschieht jedoch oft nicht im Sinne einer Änderung einer bestehenden Klasse (eines Sichtschemas oder des Basisschemas), sondern wird im Widerspruch zu Teilziel 3.1 als Neuanlegen aufgefaßt, weswegen alle Eigenschaften einer Sicht bei deren Spezifikation aufgeführt werden müssen.

Selbst wenn keine expliziten Schemaänderungsprimitive durch eine Spezifikationssprache für Sichten angeboten werden, so sind doch einige neue Schemazustände durch eine komplette Beschreibung simulierbar, jedoch nicht alle (Teilziel 3.2). Sichten werden stets (direkt oder indirekt) von einem Basisschema abgeleitet, wobei die Modellierungsmächtigkeit dieses Basisschemas erhalten bleibt oder vermindert wird. Das Ausblenden einer Basisklasse bewirkt beispielsweise, daß die Objekte dieser Klasse durch das Sichtschema nicht mehr zugegriffen werden können. In Abhängigkeit von der Veränderung der Modellierungsmächtigkeit bei Definition einer Sicht sprechen wir von *kapazitätserhaltenden* bzw. von *kapazitätsvermindernden* Schemaänderungen. *Kapazitätserweiternde* Änderungen, die etwa Eigenschaften zu den Basisklassen hinzufügen, sind jedoch durch Sichtenmechanismen nicht erreichbar. Dazu gehört beispielsweise die Definition neuer Attribute, die nicht von bisherigen Attributen abgeleitet werden können, oder analog

das Anlegen neuer Klassen. Die bei der Schemaevolution angestrebte Unterstützung genereller Veränderungen an Klassen und Klassenhierarchien entsprechend Teilziel 3.2 wird damit also nicht erreicht. Zur Behebung dieses Mangels müssen Ansätze der hier besprochenen Kategorie zusätzliche Mechanismen entwickeln und mit dem Sichtenkonzept integrieren.

Beim reinen Sichtenkonzept werden alle Sichten gemeinsam verwaltet, d.h. es kann keine Gruppierung von Sichtklassen und demzufolge auch keine Zuordnung zu verschiedenen, gleichzeitig zu verwaltenden Ausprägungen eines Schemas vorgenommen werden. Änderungen finden damit also nicht auf Schemaebene statt (Teilziel 3.3), sondern auf Klassenebene. Um das Teilziel 3.3 zu erreichen, verwalten einige Konzepte separate *Sichtschemata*. Die Definition neuer Sichten erhält bisherige Zustände (Teilziel 3.5) und erlaubt damit prinzipiell auch die Entwicklung und die Integration von Alternativen (Teilziele 3.6 und 3.7); die Nutzbarkeit dieser Eigenschaften ist jedoch aufgrund der gemeinsamen Verwaltung aller Sichten stark eingeschränkt. Ähnliches gilt für die Verwaltung der Ableitungsbeziehungen zwischen verschiedenen Ausprägungen (Teilziel 3.8). Da sich jede Sichtklasse von einer weiteren Sichtklasse oder von einer Basisklasse ableitet, ist eine Ableitungsbeziehung inhärent gegeben; aufgrund der Vermischung aller Schemazustände ist dies allerdings wiederum nur begrenzt hilfreich.

Die Simulation eines Schemazustandes geschieht dadurch, daß eine bestehende Sicht modifiziert oder eine neue Sicht angelegt wird. Die Modifikation wird dabei nur dann angewendet, wenn der ursprüngliche Zustand der Sicht nicht mehr benötigt wird. Damit wird die Anpassung existierender Applikationen vermieden (Teilziel 3.9).

Auch die Kooperation zwischen Applikationen verschiedener Schemazustände (Teilziel 3.10) wird in natürlicher Weise durch den Sichtenmechanismus gewährleistet, da dieser letztendlich alle auf Sichten durchgeführten Zugriffe auf das darunterliegende, gemeinsame Basisschema abbildet. Auf diese Weise werden Änderungen der Datenbank sofort für die Applikationen aller Sichten realisiert, womit implizit eine vollständige Propagation (Teilziel 3.11) durchgeführt wird. Die Teilziele 3.9, 3.10 und 3.11 werden, wie bereits bemerkt, jedoch nur bei dem Verzicht auf kapazitätserweiternde Schemaänderungen erreicht. Weiterhin bestehen bei Sichten konzeptionell bedingte Einschränkungen bei Objektzugriffen, auf die wir weiter unten bei der Untersuchung der Transparenz des Simulationsansatzes näher eingehen werden.

Da für die Propagation der den Sichten eigene Mechanismus verwendet wird, bestehen hier wenige Möglichkeiten der Einflußnahme (Teilziel 3.12). Die Abbildung von Objekten des Basisschemas auf Objekte einer Sicht, die ungefähr der geforderten Vorwärtskonvertierung entspricht, wird aus der Sichtdefinition abgeleitet. Dabei kann die Form der Konvertierung nur indirekt und in unterschiedlichem Umfang beeinflußt werden. Das Analogon zur Rückwärtskonvertierung, die Abbildung von Objekten einer Sicht auf Objekte des Basisschemas ist aufgrund des Sichtenänderungsproblems (engl. *view update problem*) i.Allg. nicht durchführbar. Wir werden hierauf bei der Untersuchung der Transparenz des Simulationsansatzes näher eingehen.

Eine Verwaltung mehrfacher, getrennter Datenwerte, so wie sie von Teilziel 3.13 gefordert wird, ist mit dem Sichtenmechanismus nicht erreichbar. Jedem Objekt liegt stets nur ein Wert zugrunde, der bei Änderungen ersetzt wird, so daß der neue Objektzustand sofort auch durch alle Sichten gesehen wird. Dies ist insbesondere von der Erzeugungszeit der beteiligten Sichten unabhängig, d.h. eine Unterscheidung zwischen Vorwärts- und Rückwärtskonvertierung entsprechend des Alters oder der Ableitungsbeziehungen zwischen verschiedenen Sichten ist nicht möglich. Bei der Betrachtung der Objektpropagation ähnelt der Simulationsmechanismus einer Situation, in der das Basisschema die Wurzel der Ableitungsstruktur darstellt und alle Sichten direkt davon abgeleitet sind. Daher kann auch die Verwaltung semantischer Beziehungen zwischen Schemaversionen (Teilziel 3.14) nicht wie gewünscht durchgeführt werden.

Bei der Betrachtung der Effizienz (Teilziel 3.15) sind zwei sich widersprechende Aspekte zu berücksichtigen, nämlich Speicherplatzbedarf und Laufzeit. Ein Sichtenmechanismus speichert jedes Objekt nur genau einmal, unabhängig davon, wieviele Sichten auf dem Basisschema definiert sind; der Aufwand an Speicherplatz ist also minimal. Bei der Betrachtung des Laufzeitverhaltens ist zu beachten, daß alle Konvertierungen von Objekten vom Basisschema zu einem Sichtschema und umgekehrt jeweils erst bei einem Zugriff durch eine Applikation, also quasi verzögert, durchgeführt werden. Alle Objekte sind stets entsprechend dem Basisschema gespeichert. Da dieses jedoch nie verändert wird, sind keinerlei Transformationen der Datenbank notwendig. Dieser Vorteil wird allerdings dadurch abgeschwächt, daß keine kapazitätserweiternden Schemaänderungen durchführbar sind, es sei denn sie werden eben doch direkt auf dem Basisschema durchgeführt (siehe Abschnitt 4.4.7.2) oder es kommen zusätzliche Mechanismen zum Einsatz.

Wenn mehrfache Schemaänderungen durch die Definition von Sichten auf der Basis anderer Sichten realisiert werden, so könnten sich nach zahlreichen Schemaänderungen jedoch Effizienzprobleme ergeben, weil dann eine verkettete Ausführung mehrerer Abbildungen notwendig ist. Dies gilt insbesondere, wenn die einzelnen Abbildungen bereits aufwendig sind. Um dieses Problem zu beheben, ist der Einsatz materialisierter Sichten möglich. Dabei werden die Objekte nicht mehr bei jedem Zugriff durch eine Sicht in die benötigte Form konvertiert. Stattdessen werden die Ergebnisse der Konvertierung physikalisch gespeichert. Damit wird zwar einerseits die angestrebte Beschleunigung beim Zugriff erreicht, andererseits geht jedoch gleichzeitig der Vorteil des minimalen Speicherplatzbedarfes verloren. Hier gilt es also, zwischen den beiden Aspekten der Effizienz Laufzeit und Speicherplatzbedarf abzuwägen.

Wie bereits dargelegt, wird die Form der Konvertierung bei der Definition einer Sicht weitgehend implizit angegeben. Dies ist mit der Situation zu vergleichen, daß eine durchgeführte Schemaänderungsoperation automatisch eine entsprechende Konvertierung impliziert. Diese wird jedoch nicht explizit notiert und kann demzufolge auch nicht vom Schemaentwickler verändert werden.

Um eine Veränderung des Zustandes eines Objektes durch eine Applikation, die auf einer Sicht basiert, zu ermöglichen, wird eine Abbildung von der Sicht auf das Basisschema benötigt. Ein solches Analogon zu der von uns eingeführten Rückwärtskonvertierung ist jedoch in den meisten Systemen nicht vorgesehen. Aufgrund der genannten Einschränkungen kann sich der Spezifikationsaufwand (Teilziel 3.16) auf die Definition der benötigten Sichten beschränken und fällt damit vergleichsweise gering aus.

Entsprechend dem Teilziel 3.16 des geringen Spezifikationsaufwandes für Schemaänderungen sollten existierende Sichtklassen in neuen Sichtschemata mit oder ohne Veränderung verwendet werden können. Daher ist es sinnvoll, existierende Sichten gemeinsam zu verwalten, um dem Schemaentwickler einen Überblick zu geben und so die mehrmalige Implementierung derselben Sichtklassen zu vermeiden.

Durch die Möglichkeit Sichten nicht nur direkt auf dem Basisschema zu definieren, sondern auch indirekt auf anderen Sichten, wird das Teilziel 3.17 der Lokalität bei der Spezifikation neuer Schemazustände erreicht.

Die Konsistenz (Teilziel 3.18) des Schemas bleibt konzeptionsbedingt dadurch erhalten, daß am Basisschema keinerlei Änderungen vorgenommen werden. Neue Sichten werden nicht durch Übernahme und Veränderung bisheriger Modellierungen erreicht, sondern ausschließlich durch sukzessives Hinzufügen neuer Sichtklassen. Modifikationen oder Löschungen im eigentlichen Sinne werden dabei gar nicht benötigt, so daß die Notwendigkeit korrigierender Maßnahmen zunächst entfällt. Sollen jedoch kapazitätserweiternde Schemaänderungen ermöglicht werden, so sind, wie bereits bei der Bewertung bezüglich Teilziel 3.2 angedeutet, zusätzliche Mechanismen anzubieten womit die beschriebene konzeptionsinhärente Konsistenzerhaltung verloren geht und

ähnliche Maßnahmen zu ergreifen sind wie bei Ansätzen der direkten Schemaevolution oder der Schemaversionierung.

Die Forderung nach Transparenz für Applikationsentwickler (Teilziel 3.19) kann vom Konzept der Simulation von Schemaänderungen durch Sichten aus verschiedenen Gründen nicht voll erfüllt werden. Die Problematik der eingeschränkten Durchführbarkeit von Datenbankänderungen (engl. *view update problem*), die schon bei relationalen Datenbanksystemen auftritt, besteht natürlich auch hier. Durch Sichten berechnete Werte lassen sich i.Allg. nicht aktualisieren, insbesondere dann nicht, wenn sie von mehreren Werten der Datenbank abhängen, wie z.B. aggregierte Werte (Summen, Mittelwerte, etc.) oder wenn sie durch den Verbindungsoperator (engl. *join*) entstanden sind.

Jedoch kann selbst der Fall ausschließlich lesender Zugriffe problematisch sein, da bei mehrfachem Zugriff auf eine unveränderte Datenmenge dasselbe Ergebnis erwartet wird. OODBMS verwenden das fundamentale Konzept der Objektidentitäten, das es zum einen gestattet, ein Objekt auch nach einer Zustandsänderung noch an seiner unveränderten Identität wieder zu erkennen. Zum anderen erlaubt die Eindeutigkeit der Objektidentitäten die Unterscheidung zweier Objekte selbst dann noch, wenn sich diese in demselben Zustand befinden. Im Gegensatz dazu erzeugen einige Sichtenkonzepte neue Objektidentitäten, so daß neue Objekte implizit als Instanzen der Sichtklassen angelegt werden. Dabei werden sogar bei jeder Ausführung der Sichtanfrage neue Objektidentitäten generiert. Damit geht die oben genannte, elementare Grundeigenschaft der Unveränderlichkeit von Objektidentitäten verloren, was z.B. bei der Zugehörigkeit von Objekten zu Mengen oder bei Aggregatstrukturen (komplexen Objekten) zu Problemen führen kann.

Ein weiteres den Sichten grundsätzlich anhaftendes Problem, das nicht erst beim Einsatz des Sichtenkonzeptes für die Unterstützung der Schemaevolution auftritt, liegt in der Definition der Sichten begründet. Wird in der definierenden Sichtanfrage nämlich eine Selektion anhand eines Datenwertes vorgenommen, beispielsweise Jugendliche sind solche Personen, die unter 18 Jahre alt sind, so kann es passieren, daß ein Objekt nach der Änderung eines Datenwertes aus der Sicht verschwindet. Wenn das Alter eines vormals Jugendlichen auf 18 heraufgesetzt wird, dann verschwindet dieser nämlich aus der Sicht, da er das Selektionsprädikat nicht mehr erfüllt. Dieser Effekt tritt in Basisrelationen bzw. Basisklassen nicht auf und könnte den Anwender verwirren, wenn er das veränderte Objekt plötzlich nicht mehr sieht. Auch hier ergibt sich also eine Einschränkung der Transparenz des Sichtenkonzeptes.

Ein weiterer Aspekt der mangelnden Transparenz spiegelt sich in dem unterschiedlichen Verhalten von Sichtklassen und Basisklassen; wie bereits dargestellt sind bei Sichten keine Hinzufügungen von Eigenschaften möglich.

4.4.4 Vergleich der Vorgehensweise mit dem manuellen Ansatz

Weder der Einsatz von Sichten zur Simulation von Schemaänderungen noch der bereits in Abschnitt 4.2 beschriebene manuelle Ansatz erreichen die gesteckten Ziele. Die Propagation von Objekten wird den beiden Konzepten folgend zwar jeweils durchgeführt, die Möglichkeiten der Einflußnahme durch den Schemaentwickler sind jedoch in beiden Fällen sehr eingeschränkt. Interessanterweise verfolgen die beiden Propagationsmodelle dabei recht gegensätzliche Ziele.

Bei der manuellen Vorgehensweise mit Ausladen und Kopieren der Datenbank, manuellem Umsetzen und schließlich wieder Einladen der Kopie entstehen gänzlich unabhängige Datenbanken,

zwischen denen nach ihrer Erstellung keinerlei Propagation mehr erfolgt.[36] Demzufolge können keinerlei Konsistenzen zwischen den verschiedenen Datenbanken garantiert werden.

Dem Sichtenkonzept folgend wird nur eine einzige Datenbank verwaltet, die in ihrer Struktur dem Basisschema entspricht und in der alle Objekte abgelegt sind. Alle Zugriffe werden auf diese eine Datenbank abgebildet, unabhängig davon auf welcher Sicht, d.h. auf welchem Schemazustand, die zugreifende Applikation operiert. Damit wirkt sich jede Objektänderung sofort auf alle Applikationen aus, durch deren Sichten das geänderte Objekt zugreifbar ist.

Im Gegensatz zur Verwendung isolierter Datenbanken wird durch die vollständige Propagation des Sichtenkonzeptes eine enge Verbindung zwischen den Applikationen verschiedener Schemazustände realisiert. Die technischen Teilziele fordern hier insbesondere die Möglichkeit zur flexiblen Einstellbarkeit von Abstufungen zwischen den beiden dargestellten Extremen.

4.4.5 Vergleich der Vorgehensweise mit der direkten Schemaevolution

Die wichtigsten Unterschiede zwischen Sichten und Schemaevolution sind die folgenden:

- Bei Verwendung des Sichtenkonzeptes sind ohne Hinzunahme weiterer Konzepte keine kapazitätserweiternden Schemaänderungen möglich. Damit ist eine Sicht nur eine Teilmenge eines Gesamtmodells, das durch die Sichtdefinition unbeeinflußt bleibt. Die Schemaevolution beschäftigt sich jedoch mit generellen Veränderungen an Klassen und Klassenhierarchien.

- Bei Sichtenkonzepten werden entweder mehrere externe Sichten oder in das Basisschema integrierte Sichtklassen benutzt. Im letzteren Fall müssen alle Klassen in ein Schema passen, was unnatürlich sein kann. Der erstere Ansatz entspricht mehr der Schemaversionierung bei der nie Information verloren geht; das Sichtenkonzept spiegelt den Evolutionsansatz aber nicht besonders gut wieder: Die Basisklassen werden nie modifiziert und zu ihren Instanzen wird nie Information hinzugefügt oder davon entfernt; Information wird nur versteckt, aber nicht gelöscht, d.h. daß Sichtendefinitionen im Gegensatz zur Schemaevolution den Aspekt, daß etwas hinfällig wird, nicht berücksichtigen.

- Oft verhält sich eine Sichtklasse nicht genau wie eine Basisklasse, z.B. sind keine Hinzufügungen von Eigenschaften möglich. Bei der Schemaevolution besteht zwischen Klassen, die neu angelegt wurden, und Klassen, die von älteren Schemazuständen übernommen wurden, kein Unterschied. Beide haben genau dieselben Eigenschaften wie Klassen in Systemen ohne Schemaevolution.

- Sichtenmechanismen beschäftigen sich mit der Definition neuer Abstraktionen während Schemaevolution existierende Eigenschaften modifiziert. Schemaevolutionskonzepte behandeln Identität zwischen Klassen (beispielsweise durch Beibehaltung des Klassennamens in der neuen Schemaausprägung) auch bei deren Veränderung, wohingegen bei Sichten keine Identität bewahrt wird und sich die Namen der Sichtklassen von denen der Basisklassen unterscheiden müssen. Damit hat eine Änderung auch Auswirkungen bzw. Seiteneffekte auf von der geänderten Klasse abhängige Klassen; es besteht also keine Transparenz von Typänderungen.

  Außerdem entsteht das Problem der Platzierung der neuen Klassen in der Klassenhierarchie. Bei Schemaevolution ist es besonders wichtig, daß beliebige Änderungen vorgenommen werden können (siehe technisches Teilziel 3.2). Änderungen, die die Kapazität

[36] Wir gehen davon aus, daß die in Abschnitt 4.2 angedeutete Möglichkeit der manuellen Durchführung der Propagation aufgrund der konzeptionellen Schwächen und wegen des damit verbundenen enormen Aufwandes nicht in Frage kommt.

nicht erweitern, können mit Sichten zwar erreicht werden, aber selbst wenn kapazitätserweiternde Änderungen möglich sind, so bestehen dafür gewisse Einschränkungen (z.B. bei Bertino [Ber92] werden Änderungen von Attributen durch Entfernen und anschließendes neu Anlegen erreicht aber es kann keine Abhängigkeit zwischen den beiden Attributen (in beiden Richtungen) verwaltet werden, ähnliche Einschränkungen existieren bei Rundensteiner (siehe Abschnitt 4.4.7.2)). Für Schemaevolutionskonzepte ist die Verwaltung solcher und ähnlicher Abhängigkeiten elementar. Sichtenmechanismen können nur solche Änderungen handhaben, bei denen Klasseneigenschaften von den Eigenschaften der Basisklassen abgeleitet werden können.[37]

- Bei der Schemaevolution bleiben die fundamentalen Objektidentitäten auch in neuen Schemaausprägungen bewahrt, wobei sich ein Objekt in verschiedenen Schemaausprägungen verschieden verhalten kann, als Instanz verschiedener Ausprägungen seiner Klasse. Im Gegensatz dazu erzeugen einige Sichtenkonzepte neue Objektidentitäten, so daß neue Objekte implizit als Instanzen der Sichtklassen angelegt werden.

- Die Spezifikation der durchzuführenden Konvertierungen geschieht beim Simulationskonzept nur implizit und kann nicht vom Schemaentwickler modifiziert werden. Der Ansatz der Schemaevolution bietet durch die automatische Erzeugung von Defaultkonvertierungsfunktionen ebenfalls eine auf die durchgeführte Schemaänderung zugeschnittene Umsetzung an. Diese ist jedoch im Gegensatz zum Simulationskonzept explizit notiert und kann vom Schemaentwickler bei Bedarf verändert werden.

Zusammenfassend kann festgehalten werden, daß Schemaevolution und Sichten verschiedene Probleme auf verschiedene Weisen angehen und in vielen Bereichen eher komplementäre Leistungen erbringen. Durch eine geschickte Kombination beider Konzepte, wie sie insbesondere von Rundensteiner vorgeschlagen wird (siehe Abschnitt 4.4.7.2), lassen sich jedoch sehr gute Resultate erzielen.

4.4.6 Vergleich der Vorgehensweise mit der Schemaversionierung

Das Sichtenkonzept ist insbesondere für solche Schemaänderungen nützlich, die die Kapazität eines Schemas reduzieren oder erhalten. Um auch kapazitätserweiternde Schemaänderungen durchführen zu können, sind Erweiterungen notwendig, z.B. durch Kombination mit dem direkten Ansatz. In manchen Situationen sind jedoch einige Schemaänderungen aufgrund von Konsistenzbedingungen, die durch Benutzung des Sichtenkonzeptes zusätzlich zu beachten sind, nicht durchführbar. Weiterhin müssen Sichten der Zugriffszeit wegen materialisiert werden, wenn die dynamische Berechnung aller Transformationen zur Laufzeit zu zeitaufwendig wird.

Der Sichtenansatz weist sehr viele Ähnlichkeiten mit dem Versionierungsansatz auf. Eine gemeinsame Datenbank kann gleichzeitig über mehrere Schnittstellen zugegriffen werden und die Objekte müssen entsprechend hin- und herkonvertiert werden. Bei materialisierten Sichten enthalten sogar die physikalischen Datenstrukturen in beiden Fällen vergleichbare Informationen. Das Versionierungskonzept ist jedoch sehr viel klarer, z.B. weil Vorwärts- und Rückwärtskonvertierungsfunktionen hier explizit auftreten und eine Veränderung von systemgenerierten Defaultkonvertierungsfunktionen gestatten. Während die Konvertierung auf ähnlichen Wegen vonstatten geht, geschieht die Spezifikation der Konvertierungsfunktionen beim Sichtenkonzept nur implizit: Vorwärtskonvertierungsfunktionen werden aus den durchgeführten Schemaänderungen

[37] Dies stellt keinen Widerspruch zu dem oben gesagten dar. Während es dort um die Schemaebene geht, d.h. eine Sicht wird neu definiert, geht es hier um die Objektebene, d.h ein neues Objekt wird von existierenden Objekten abgeleitet.

abgeleitet und können nicht manuell verändert werden. Die in typischen Sichtendefinitionssprachen gegebenen Möglichkeiten zur Konvertierung von Objekten sind allerdings sehr beschränkt. Hier könnte die explizite Verwendung von durch den Schemaentwickler modifizierbaren Konvertierungsfunktionen die Flexibilität beträchtlich steigern. Um das Problem der Sichtenaktualisierung (engl. *view update problem*) zu lösen, bieten einige Ansätze als Option explizite Änderungsfunktionen (engl. *update functions*), mit deren Hilfe spezifiziert werden kann, wie sich Änderungen von Sichtenobjekten auf die darunterliegende Basisdatenbank auswirken sollen. Dies ist den Rückwärtskonvertierungsfunktionen des Versionierungsansatzes sehr ähnlich.

4.4.7 Konkrete Vertreter der Vorgehensweise

Wir stellen hier einige konkrete Ansätze näher vor, die die Benutzung von Sichten zur Simulation von Schemaänderungen vorschlagen. Weitere Vertreter dieser Vorgehensweise, auf die wir hier allerdings nicht näher eingehen, sind beispielsweise [Ber92, BFK95, Bra92, Bra93, Sch93] etc.

4.4.7.1 Der Ansatz von Tresch und Scholl

Markus Tresch und Marc Scholl verwenden den in [SLT91, ST94] eingeführten Sichtenmechanismus zur Entwicklung eines Evolutionskonzeptes [TS92, TS93c, Tre95]. Die Arbeiten beruhen auf dem objektorientierten Datenmodell COCOON [SLR+92, Tre95], das klar zwischen drei Objektebenen unterscheidet. Dies sind die Ebenen der Daten, des Schemas und des Metaschemas. Dabei ist besonders beachtenswert, daß auf allen drei Ebenen dieselben Operationen zur Manipulation der jeweiligen Objekte verwendet werden können. Somit läßt sich die Evolution eines Schemas durch die Anwendung von Operationen auf ein Schemaobjekt erreichen, genau so wie Änderungen des Datenbankzustandes durch Operationen auf Datenobjekten erzielt werden.

Durch die klare Trennung zwischen Typen, die Schnittstellen beschreiben, und Klassen, die Objektmengen (eines bestimmten Typs) verwalten, wird die Grundlage für die Einführung eines Sichtenmechanismus gelegt. Sichttypen müssen demzufolge in die Typhierarchie integriert werden, Sichtklassen entsprechend in die Klassenhierarchie. Um die Überfrachtung eines Schemas durch die Integration sämtlicher darauf definierter Sichten zu vermeiden, führen Tresch und Scholl ergänzend zu den Sichten ein zweites Konzept zur Datenabstraktion ein, indem sie sog. *Subschemata* erlauben. Ein Subschema extrahiert einen für eine Applikation bedeutungsvollen Teil des um Sichten erweiterten Gesamtschemas, so daß die Applikationsentwickler von einer übersichtlicheren Beschreibung der Datenbankstruktur ausgehen können. Die Subschemata sind insbesondere vollständig (engl. *closed*).

Das Ziel der Anwendung von Sichten zur Simulation von Schemaänderungen ist für Tresch und Scholl vornehmlich die Vermeidung möglicherweise aufwendiger Datenbankreorganisationen sowie die Demonstration der Brauchbarkeit ihres Sichtenkonzeptes für diese Aufgabe. Für eine Bewertung des Ansatzes bezüglich unserer technischen Teilziele finden sich jedoch nicht genügend Details.

4.4.7.2 Der Ansatz von Ra und Rundensteiner

Der von der Gruppe um Elke Rundensteiner entwickelte und implementierte Sichtenmechanismus *MultiView* [KR95, KR96b, KR96a, Run92, RKR+96] bietet zahlreiche Einsatzmöglichkeiten und dient insbesondere auch als Grundlage für Arbeiten im Bereich der Schemaevolution. Hier wurde das *Transparent Schema Evolution (TSE) System* [RR95, CTR96] entwickelt, um die Verwendbarkeit von MultiView zur Durchführung von Schemaänderungen zu zeigen.

MultiView verwaltet nicht nur einzelne Sichtklassen, sondern komplette Sichtschemata, die sich logisch, also aus der Perspektive der Applikationsentwickler, wie herkömmliche Schemata verhalten und von diesen nicht unterscheidbar sind.

Technisch, also aus der Perspektive der Schemaentwickler, unterscheidet MultiView zwischen drei Typen von Schemata.

- Jede Datenbank hat genau ein *Basisschema* (engl. *base schema*), das für den Zugriff auf die physikalisch gespeicherten Daten erforderlich ist und das dem Schema eines OODBMS ohne Sichten entspricht.
- Für eine Datenbank können beliebig viele *Sichtschemata* (engl. *view schemas*) definiert werden.
- Das *globale Schema* (engl. *global schema*) einer Datenbank enthält sowohl ihr Basisschema als auch sämtliche dafür definierten Sichtschemata.

Weiterhin unterscheidet MultiView drei Typen von Klassen.

- Die Klassen des Basisschemas heißen *Basisklassen* (engl. *base classes*).
- *Virtuelle Klassen* (engl. *virtual classes*)[38] werden auf einer bereits existierenden Datenbank durch eine objektorientierte Anfrage definiert und dem globalen Schema dynamisch hinzugefügt. Dabei können sämtliche Klassen des globalen Schemas verwendet werden. Die Objekte einer virtuellen Klasse sind durch eine Funktion bestimmt; sie werden aber typischerweise[39] nicht explizit gespeichert.
- Die zu einem Sichtschema gehörigen Klassen heißen *Sichtklassen* (engl. *view classes*). Eine Sichtklasse ist entweder eine Basisklasse oder eine virtuelle Klasse.

Das globale Schema ist zunächst, d.h. solange keine Sichten definiert wurden, mit dem Basisschema identisch. Die Spezifikation von Sichten geschieht in MultiView in drei unabhängigen Schritten:

1 Die Modifikation von Schema und Datenbank durch Definition virtueller Klassen mittels objektorientierter Anfragen,

2 die Erweiterung des globalen Schemas durch die Integration der virtuellen Klassen und

3 die Spezifikation von Sichtschemata durch Selektion von Klassen des erweiterten, globalen Schemas.

Für die Automatisierung der obigen Schritte 2 und 3 bietet Rundensteiner geeignete Algorithmen an. Diese berücksichtigen insbesondere die Vererbungsstruktur im globalen Schema und fügen dazu in Schritt 2 neu definierte, virtuelle Klassen an geeigneten Positionen ein und stellen in Schritt 3 die Vererbungsbeziehungen zwischen den Klassen eines Sichtschemas her. Wenn zwischen zwei in Vererbungsbeziehung stehenden Klassen eines Sichtschemas nicht zur Sicht gehörige Klassen des globalen Schemas liegen, so wird die Menge der durch die Unterklasse geerbten Eigenschaften durch entsprechende Anpassungen der Sichtklassen geeignet ergänzt.

[38] Virtuelle Klassen werden von Rundensteiner synonym auch als *derived classes* bezeichnet. Wir vermeiden hier allerdings die deutsche Übersetzung als *abgeleitete Klassen*, um Verwechslungen zu verhindern. Den Begriff der Ableitung verwenden wir in dieser Arbeit ausschließlich im Zusammenhang mit Versionen.

[39] Zur Verbesserung der Zugriffsgeschwindigkeit können virtuelle Klassen in MultiView allerdings materialisiert werden.

Beim Einsatz von MultiView zur Unterstützung von Schemaevolution konzentrieren sich Ra und Rundensteiner auf das bereits in Abschnitt 4.4.3 erwähnte Problem, daß zunächst keine kapazitätserweiternden Schemaänderungen mit Sichten möglich sind. Um diese Einschränkung aufzuheben, wird die Idee einer Kombination des Sichtenkonzeptes mit dem direktem Ansatz zur Durchführung von Schemaänderungen verfolgt. Demzufolge sind kapazitätserweiternde Schemaänderungen durch Ergänzungen direkt im Basisschema auszuführen, während kapazitätserhaltende und -vermindernde Änderungen durch die Definition von Sichten bewerkstelligt werden. Dabei ist jedoch zu beachten, daß durch Änderungen am Basisschema schon existierende Sichtschemata indirekt verändert werden könnten. Um dies zu vermeiden, nimmt TSE nach Änderungen des Basisschemas ggf. notwendig werdende Anpassungen existierender Sichtschemata automatisch vor. Die dazu eingesetzten Algorithmen sind in verschiedenen Veröffentlichungen detailliert erläutert.

Für die Durchführung von Änderungen eines zugrunde liegenden Basisschemas bietet TSE Schemaänderungsprimitive an, die in der Erzeugung von Sichtklassen resultieren, welche in die Vererbungsstruktur des Basisschemas integriert werden. Obwohl damit systemintern ein alle Sichten und das Basisschema umfassendes Gesamtschema entsteht, werden aus Benutzersicht verschiedene Schemaausprägungen verwaltet, auf denen verschiedene Applikationen aufsetzen und kooperieren können. Damit ist mit dem TSE-System auch die paralelle Durchführung unabhängiger Schemaänderungen durch verschiedene Schemaentwickler ohne die Gefahr wechselseitiger Beeinflussung durch Seiteneffekte möglich.

Im Vergleich zu dem in Abschnitt 4.4.2 beschriebenen, prinzipiellen Vorgehen bei der Anwendung von Sichten zur Simulation von Schemaänderungen haben die hier vorgestellten Konzepte erhebliche Vorteile.

Zur Durchführung von Schemaänderungen werden tatsächliche Primitive (Teilziel 3.1) ähnlich den in Abschnitt 4.3.1.1 vorgestellten angeboten. Deren Verwendung ist erheblich einfacher als die Erzeugung eines entsprechenden Schemas mit Hilfe einer Sichtendefinitionssprache. Abgesehen von wenigen, durch das Sichtenkonzept bedingten Einschränkungen sind mit den angebotenen Primitiven jederzeit sämtliche Schemaänderungen durchführbar (Teilziel 3.2) und zwar auf Schemaebene (Teilziel 3.3). Mit der in [CJR98] vorgeschlagenen, erweiterbaren Bibliothek benutzerdefinierbarer Schemaänderungen sind zwar keine externen Schemaänderungen entsprechend Teilziel 3.4 durchführbar, die Nachteile einer systemseitig fest vorgegebenen Schemaänderungstaxonomie werden damit jedoch in vergleichbarer Art und Weise gemildert.

Die Teilziele 3.9, 3.10 und 3.11 werden trotz der Berücksichtigung kapazitätserweiternder Schemaänderungen voll erreicht. Durch die Verwendung materialisierter Sichten unter Ausnutzung von optimierten Verfahren zur Propagation von Objektänderungen [KR96b] wird eine gute Effizienz (Teilziel 3.15) erzielt.

4.5 Einsatz von Versionen

Neben den Ansätzen der direkten Schemaevolution und der Simulation von Schemaänderungen durch Sichten besteht noch eine weitere Möglichkeit, nämlich die Verwendung von Versionen. Auf Ansätze, die diesem Bereich zuzuordnen sind, gehen wir im folgenden näher ein.[40]

[40] Die in [ALP91, BB95, RCR93] beschriebenen Ansätze zur Schemaversionierung bleiben hier unberücksichtigt.

4.5.1 Grundsätzliche Vorgehensweise

Der direkte Ansatz zur Schemaevolution führt Änderungen unmittelbar am vorliegenden Schema durch, wobei der vorherige Zustand jeweils überschrieben wird und damit verloren geht. Daraus ergibt sich die Notwendigkeit zur sofortigen Anpassung sämtlicher Applikationen nach jeder Schemaänderung. Der Aufwand für diesen Anpassungsprozeß, sofern dieser überhaupt durchführbar ist, wächst mit der Anzahl und dem Umfang vorhandener Applikationen, was in der Praxis oftmals dazu führt, daß wünschenswerte Schemaänderungen gar nicht durchgeführt werden. Insbesondere bei Systemen mit zahlreichen und umfangreichen Applikationen erscheint es deutlich einfacher, im Datenbanksystem eine Möglichkeit zur Anpassung der Daten zu spezifizieren, als alle Applikationen anzupassen. Auf der Grundlage dieser Idee ist ein Mechanismus zu entwerfen, der die Objekte der Datenbank in unterschiedlichen Formen präsentieren kann. Je nach der Erwartung, die eine Applikation an den Typ der zugegriffenen Objekte hat, sind diese entsprechend auszuliefern.

Die Vorstellung, ein Objekt sei quasi in verschiedenen Formen vorrätig, wovon jeweils eine passende ausgewählt werden könne, erinnert stark an das Konzept der *Versionen* (siehe Abschnitt 2.2). Als Versionen werden dabei Ausprägungen oder Zustände bezeichnet, in denen ein Objekt vorliegt. Ein Designobjekt kann beispielsweise in verschiedenen vorläufigen und in einer endgültigen Version vorliegen, ein Produkt kann in verschiedenen Versionen, etwa in unterschiedlichen Farben, verkauft werden, etc. Das Konzept der Versionierung von Objekten wurde bereits intensiv erforscht. Es ist in einigen Datenbanksystemen verfügbar und wird in verschiedenen Bereichen eingesetzt. Den im folgenden vorzustellenden Ansätzen liegt die Idee zugrunde, das Konzept der Versionen auf das Schema bzw. auf einzelne Klassen daraus anzuwenden. Wir hatten einzelne Aspekte dieses Ansatzes bereits bei der groben Vorstellung unseres Lösungsmodells in Abschnitt 3.3 angesprochen. Die Anwendung auf der Abstraktionsebene des Schemas geschieht zunächst ohne Besonderheiten im Vergleich zur Versionierung anderer komplexer Objekte. Klassen bzw. Schemata werden dabei als versionierte Instanzen des Metaschemas aufgefaßt, wobei jede Änderung im Anlegen einer neuen, zusätzlichen Version resultiert. Bei der Anwendung auf einzelne Klassen sprechen wir von *Klassenversionierung*. Hierbei entsteht jedoch die bereits in Teilziel 3.3 beschriebene Notwendigkeit der Verwaltung von Konfigurationen konsistenter Klassenversionen. Weiterhin sind Schemaänderungen, die mehrere Klassen betreffen,[41] nicht vernünftig darstellbar. Daher ziehen wir entsprechend Teilziel 3.3 den Ansatz der *Schemaversionierung* vor. Dabei wird das gesamte Schema als Entwurfsobjekt angesehen und als Ganzes versioniert. Die verschiedenen Zustände eines Schemas werden dabei *Schemaversionen* genannt. Die Entscheidung, ob zwei Zustände als Versionen desselben Schemas aufgefaßt werden können, ist ein rein semantisches Problem. Daher sollten dem Schemaentwickler diesbezüglich keinerlei Restriktionen aufgebürdet werden.

Wie wir bei der Vorstellung der Schemaevolutionskonzepte von O_2 in Abschnitt 4.3.4.2 bereits gesehen hatten, müssen auch beim direkten Ansatz der Schemaevolution alte Schemazustände aufgehoben und verwaltet werden, um eine verzögerte Konvertierung der Datenbank zu ermöglichen. Dabei wird die Aufbewahrung veralteter Schemazustände benötigt, um diesen Schemazuständen entsprechend gespeicherte, bisher nicht konvertierte Objekte auch nachträglich noch korrekt interpretieren und konvertieren zu können. Diese aufbewahrten Schemazustände entsprechen de facto den Schemaversionen, sie werden dort jedoch lediglich intern benutzt und stehen dem Anwender bzw. existierenden Applikationen nach einer Schemaänderung nicht mehr zur Verfügung, woraus sich das Problem der Anpassungen der Applikationen ergibt (Teilziel 3.9).

[41] Nicht nur komplexe Schemaänderungen, bei denen Attribute zwischen verschiedenen Klassen verschoben werden, wirken sich auf mehr als eine Klasse aus. Auch einfache Änderungen wie das Löschen eines Attributes betreffen ggf. zahlreiche Unterklassen.

Es genügt jedoch nicht, verschiedene Versionen eines Schemas gleichzeitig sichtbar zu machen. Auch die Objekte der Datenbank selbst müssen für Applikationen zugreifbar sein. Hier gehen Klassen- und Schemaversionierungskonzepte über die pure Anwendung bekannter Objektversionierungskonzepte auf Klassen bzw. Schemata hinaus. Die sich dabei auf Objektebene ergebenden Anforderungen werden durch die in diesem Abschnitt zu besprechenden Mechanismen erbracht.

4.5.2 Bewertung der grundsätzlichen Vorgehensweise

Grundsätzlich werden durch den Einsatz der Versionierung zur Unterstützung der Schemaevolution die folgenden Vorteile erreicht.

Das technische Teilziel 3.2 hatte gefordert, daß jeder beliebige Schemazustand in jeden gewünschten Schemazustand transformierbar sein muß. Wie inzwischen ausgeführt verstehen wir unter einem Schemazustand eine Schemaversion. Demzufolge bedeutet die Forderung, daß jede gegebene Schemaversion in jede gewünschte Schemaversion transformierbar sein muß, d.h. jede beliebige Schemaversion kann abgeleitet werden. Die Forderung bezieht sich jedoch nicht auf den Zustand des versionierten Schemas insgesamt und beinhaltet demzufolge nicht, daß der Schemaableitungsgraph in beliebiger Art und Weise modifizierbar sein muß. Insbesondere werden keine Primitive gefordert, die eine explizite Veränderung der Ableitungsbeziehung zwischen existierenden Schemaversionen erlauben oder die eine neue Schemaversion zwischen zwei existierenden Schemaversionen des Schemaableitungsgraphen integrieren können. Für die Erreichung der Teilziele 3.1 und 3.2 genügt es daher bereits, wenn ein Versionierungsmechanismus eine Taxonomie von Schemaänderungsprimitiven anbietet, die der in Abschnitt 4.3.1.1 vorgestellten im wesentlichen entspricht.

Da niemals tatsächlich Information entfernt wird, besteht nie die Gefahr, daß etwas verloren geht. Die Schemaversionierung bietet damit sehr viel Flexibilität, weil stets die Möglichkeit besteht, zu vorherigen Versionen zurückzukehren (Teilziel 3.5). Dies ist insbesondere in komplexen Anwendungsgebieten wichtig, in denen eine Entscheidung beim Entwurf oder bei der Veränderung eines Schemas nicht offensichtliche Konsequenzen haben kann. Deshalb wird bei der Entwicklung komplexer Applikationen oft eine Vorgehensweise gewählt, bei der verschiedene Alternativen ausprobiert und verglichen werden. Der Schemaversionierungsansatz unterstützt diese Vorgehensweise besonders gut.

Wir hatten bereits erwähnt, daß Schemaänderungen nicht immer korrigierende Eingriffe bedeuten, sondern oft auch auf verschiedene Auffassungen derselben Diskurswelt zurückzuführen sind. Demzufolge ist der Ansatz, verschiedene Versionen eines Schemas nebeneinander anzubieten, als sehr natürlich einzustufen (Teilziel 3.6).

Schemaversionen ermöglichen Änderungstransparenz und vermeiden damit die Notwendigkeit der Anpassung von Applikationen (Teilziel 3.9). Lediglich Applikationen, die die am Schema durchgeführten Änderungen benötigen, müssen angepaßt werden.

Natürlich existieren auch Bereiche, in denen der Ansatz der Schemaversionierung Nachteile mit sich bringen kann. Beispielsweise entsteht durch die Verwaltung mehrerer Versionen eines Schemas und der dazwischen bestehenden Verbindungen eine erhöhte Komplexität, die, sofern sie für den Schemaentwickler sichtbar ist, zu einer Herabsetzung der Verwaltbarkeit eines Datenbanksystems führen kann. Weiterhin kann die Notwendigkeit, die Objekte der Datenbank entsprechend verschiedener Spezifikationen liefern zu können, eine Beeinträchtigung der Effizienz bedeuten. Schließlich sind insbesondere in kleineren Systemen Situationen denkbar, in denen ausschließlich korrigierende Schemaänderungen durchgeführt werden, so daß man auf die hier vorgestellten Konzepte verzichten kann. Eine Abwägung zwischen dem Einsatz direkter Schemaänderungen

und dem von Versionen unter Berücksichtigung durchzuführender Schemaänderungen findet sich in [FL96] (siehe dazu auch Abschnitt 5.4.2).

4.5.3 Konkrete Vertreter der Vorgehensweise

Bevor wir im folgenden einige Vertreter zunächst von Klassen- und dann von Schemaversionierungsansätzen vorstellen, möchten wir kurz das System Goose (*Graphical Object-Oriented Schema Environment*) erwähnen. Die Arbeiten [MNK92, MNS94] zu diesem System setzen auf den Schemainvarianten von ORION [BCG+87] (siehe Abschnitt 4.5.3.7.1) auf. Sie beschreiben einen Schemamanager, der mehrere Versionen eines Schemas in einer gemeinsamen Klassenhierarchie verwaltet. Die Untersuchungen konzentrieren sich jedoch auf Änderungen an Methoden des Datenbankschemas und auf Konsequenzen für das Schema selbst. Auf die Abbildung von Schemaänderungen auf die Objekte wird nicht näher eingegangen.

Weitere Arbeiten, wie etwa [AHLPZ96, AHLPZ96, BBP96], betrachten wir hier nicht näher.

4.5.3.1 Der Datenbankversionierungsansatz von Cellary und Jomier

Die Gruppe um Geneviève Jomier verfolgt in zahlreichen Veröffentlichungen die Idee, die gesamte Datenbank als versionierte Instanz anzusehen. Die Grundlagen dieses als *Datenbankversionierung* bezeichneten Ansatzes finden sich in [CJ90, CJ92]. Eine versionierte Datenbank (engl. *multiversion database*) kann verschiedene Zustände der Diskurswelt modellieren und enthält dazu versionierte Objekte (engl. *multiversion objects*). Eine Datenbankversion beschreibt einen konsistenten Zustand der Diskurswelt und enthält von jedem Objekt der Datenbank maximal eine Version. Dabei entsteht allerdings ein Konfigurationsproblem, da beliebige Kombinationen von Versionen verschiedener Objekte i.Allg. keinen konsistenten Zustand beschreiben. Um aus der kombinatorischen Vielfalt möglicher Konfigurationen diejenigen herauszufiltern, die einen konsistenten Zustand beschreiben und die demzufolge als Datenbankversion bezeichnet werden, werden in der Regel explizite Verweise zwischen Objektversionen derselben Datenbankversion verwaltet. Der damit verbundene Verwaltungsaufwand verhindert allerdings den Einsatz in großen Datenbanken [CJ90, CJ92]. Im Gegensatz zu den herkömmlichen Modellen speichert der Datenbankversionierungsansatz nach Jomier nur die Zugehörigkeit einer Objektversion zu einer Menge von Datenbankversionen. Daraus kann die Konsistenz zweier Versionen verschiedener Objekte indirekt abgeleitet werden: Objektversionen beschreiben demzufolge nämlich genau dann einen konsistenten Zustand, wenn sie einer gemeinsamen Datenbankversion zugeordnet sind.

Neue Datenbankversionen können von existierenden abgeleitet werden, wobei ein Ableitungsbaum von Datenbankversionen (engl. *database version derivation tree*) entsteht. Aus Benutzersicht wird beim Ableiten einer Datenbankversion eine komplette Kopie der direkten Vorgängerdatenbankversion angelegt, die danach vollkommen unabhängig bearbeitet werden kann. Somit kann das Ableiten als Kopieren auf der logischen Ebene verstanden werden. Zur Verringerung der Laufzeit- und Speicherplatzanforderungen werden unveränderte Objektversionen auf physikalischer Ebene jedoch von verschiedenen Datenbankversionen gemeinsam genutzt. Eine Datenbankversion *dbv* wird durch einen hierarchischen Identifikator (engl. *database version stamp*) bezeichnet, der den Pfad von der Wurzel des Ableitungsbaumes zu *dbv* beinhaltet. Damit kann die gemeinsame Nutzung von Objektversionen durch mehrere Datenbankversionen transparent realisiert werden. In [CJ90, CJ92] wird weiterhin die Eignung des Datenbankversionierungsansatzes für nebenläufige Transaktionen und für komplexe Objekte untersucht.

In [MJ94] wird gezeigt, wie Datenbankversionen für die Modellierung alternativer Szenarien und zeitlich veränderlicher (temporaler) Szenarien, die beide typischerweise in geographischen

Informationssystemen (GIS) auftreten, eingesetzt werden können. In [MBJ96] wird dazu der Datenbankversionierungsmechanismus um ein verallgemeinertes Sichtenkonzept ergänzt.

In [DGJM96b, DFG+97, DGJM96a] werden Mechanismen zur Konsistenzsicherung von unversionierten (engl. *monoversion database*) auf versionierte Datenbanken übertragen, wobei eine Kategorisierung verschiedener Integritätsbedingungen vorgenommen wird. Die Konsistenz einer versionierten Datenbank bedingt zum einen die separate Konsistenz jeder ihrer Datenbankversionen analog der Konsistenz einer unversionierten Datenbank (engl. *intra dbv consistency*) und zum anderen die Konsistenz zwischen verschiedenen Datenbankversionen (engl. *inter dbv consistency* oder *mutual consistency*). Beziehungen zwischen temporalen Datenbanksystemen, Integritätsbedingungen und Datenbankversionen werden in [DGJM96b, DFG+97] untersucht.

Das Arbeitspapier [BWJ96] erweitert den Anwendungsbereich des Versionierungsansatzes von Objekten auf Schemata. Eine Formalisierung beschreibt versionierte Objekte mit logischen und physikalischen Objektversionen und analog dazu versionierte Schemata mit logischen und physikalischen Schemaversionen. Allerdings werden keine darüber hinausgehenden und für die Schemaversionierung spezifischen Aussagen gemacht. So bleibt beispielsweise offen, wie neue Schemaversionen erzeugt werden und welche Konsequenzen Schemaänderungen für existierende Objekte haben.

Zahlreiche andere Aspekte des Datenbankversionierungsansatzes werden in weiteren Veröffentlichungen untersucht: Orthogonalität der Datenbankversionierung zur Transaktionsverwaltung (engl. *concurrency control*) und Erhöhung der Nebenläufigkeit durch versionierte Objekte [CJ94], Formalisierung des Ansatzes [CJK91, GJ94], flexible Verwaltung von Graphen, die Zusammenhänge zwischen verschiedenen Versionen einer Datenbank repräsentieren [GJZ95], Verwendung in CAD/CIM-Systemen [RC96], etc.

Die hier vorgestellten Arbeiten konzentrieren sich auf den Versionierungsansatz für Datenbanken. Dieser wird in [BWJ96] zwar auch auf das Schema angewendet, das Schema wird dabei allerdings nicht anders behandelt als ein komplexes Objekt. Es finden sich keinerlei Konzepte zur Durchführung von Schemaänderungen oder zu deren Auswirkungen auf der Objektebene. Daher können wir hier keine Bewertung der vorgeschlagenen Konzepte entsprechend unserer technischen Teilziele durchführen.

4.5.3.2 Der Typversionierungsansatz von Skarra und Zdonik

Andrea Skarra und Stanley Zdonik [SZ86, SZ87, Zdo86] verfolgen einen Versionierungsansatz für Typen in ENCORE [BZ87, ZW88, ZM91]. Dabei handelt es sich um ein objektorientiertes DBMS, das zwischen der Intension einer Klasse (dort *type* genannt) und der Extension einer Klasse (engl. *class*) unterscheidet. Ein Typ beschreibt im ENCORE Objektmodell neben Attributen (engl. *properties*) und Methoden (engl. *operations*) auch Zusicherungen (engl. *constraints*), welche von den Werten der Attribute eingehalten werden müssen. Mit Hilfe der Zusicherungen können also insbesondere Wertebereiche für die Attribute spezifiziert werden. ENCORE erlaubt Mehrfachvererbung von Typen, wobei Obertypen ihre Eigenschaften (Attribute und Methoden) und ihre Zusicherungen an Untertypen weitergeben. Zusicherungen dürfen in Untertypen nur verstärkt werden, womit sich eine Inklusionsbeziehung der zugehörigen Klassen ergibt. Damit ist ein Objekt eines Typs T auch in den Klassen aller Obertypen von T enthalten und kann für deren Objekte substituiert werden. Es besteht also eine **is_a**-Beziehung zwischen den Klassen.

In [SZ86] wird eine Taxonomie möglicher Schemaänderungen angegeben, die auf das ENCORE Datenmodell zugeschnitten ist. Damit sind neben Hinzufügen und Löschen struktureller (Attribute) und verhaltensmäßiger (Methoden) Eigenschaften und Vererbungsbeziehungen auch Verstärkungen und Abschwächungen der Zusicherungen eines Typs möglich. Um Typevolution

auch in Anwesenheit persistenter Objekte transparent erreichen zu können (engl. *transparent type evolution*), wird eine Erweiterung des semantischen Datenmodells vorgenommen. Dabei soll insbesondere unter Vermeidung manueller Applikationsanpassungen (technisches Teilziel 3.9) die vollständige Propagation (technisches Teilziel 3.11) gewährleistet werden. Das heißt neue Applikationen können auch auf bereits existierende, persistente Objekte zugreifen und bestehende, alte Applikationen haben analog Zugriff auf von neuen Applikationen erzeugte Objekte. Dazu wird ein Versionierungsmechanismus auf die Typbeschreibungen angewendet. Dieser legt bei jeder Änderung eines Typs eine neue Version dieses Typs und all seiner Untertypen an (engl. *deep versioning*).

Zu jedem Typ eines ENCORE Schemas wird eine als *Version-Set* bezeichnete Menge all seiner verschiedenen Ausprägungen verwaltet, die entsprechend ihrer zeitlichen Entstehungsreihenfolge in einer linearen Sortierung angeordnet sind. Bei jeder Typänderung wird eine neue Version des Typs in seinem Version-Set ergänzt.

Jedes Objekt entspricht während seiner gesamten Existenz derselben, bei seiner Erzeugung festgelegten Version seines Typs. Zugriffe sollen jedoch durch jede Version des Typs möglich sein. Dazu bestehen grundsätzlich zwei Möglichkeiten. Zum einen kann ein Objekt dauerhaft in die benötigte Typversion konvertiert werden (engl. *coercion*) oder das Objekt emuliert gleichsam die Eigenschaften der benötigten Typversion. Skarra und Zdonik verfolgen aus mehreren Gründen die letztere Alternative. Sie sehen die Konvertierung sehr großer Datenbestände als zu langwierigen und teuren Prozeß an. Weiterhin besteht bei gewissen Typänderungen die Gefahr, daß Informationen bei der Konvertierung von Objekten unwiederbringlich verloren gehen, wenn in der neuen Typversion weniger Objektsemantik beschrieben werden kann. Auch ist Skarra und Zdonik zufolge keine rückwärtsgerichtete Konvertierung der Objekte möglich und eine Anpassung sämtlicher Applikationen an die jeweils neuesten Typversionen kommt aufgrund des immensen Aufwandes nicht in Frage.

Die vorgeschlagene Idee enthält zwei Konzepte, nämlich das des *Version-Set-Interface* und das des *error handling*. Das Ziel dabei ist, von der Schnittstelle der Typversion eines Objektes zu abstrahieren und das Objekt gleichsam nur noch als Instanz seines Typs zu sehen, ohne die Typversion berücksichtigen zu müssen. Das Version-Set-Interface stellt dabei die Schnittstelle des Typs insgesamt dar und enthält die Vereinigung der Attribute und Methoden aller Versionen des Typs. Durch das Version-Set-Interface kann ein Objekt durch alle Versionen seines Typs benutzt werden. Voraussetzung dazu ist allerdings, daß die Namen der Eigenschaften unveränderlich und eindeutig über alle Versionen eines Typs sind. Da ein Objekt als Instanz einer konkreten Typversion nicht die gesamte Schnittstelle des Version-Set-Interface seines Typs anbietet, können verschiedenartige Typfehler auftreten, die durch Error-Handler behandelt und korrigiert werden, bevor eine Applikation das Resultat ihres Zugriffes erhält (siehe Abbildung 4.5). Die Error-Handler haben also die Aufgabe, eine Typversion um genau die Funktionalität des Version-Set-Interface zu ergänzen, die von der Typversion eines persistenten Objektes nicht selbst erbracht wird. Jeder einzelne Handler ist dabei nur für eine spezielle Fehlersituation beim Zugriff auf eine spezielle Eigenschaft einer Typversion zuständig.

Um feststellen zu können, welche Arten von Fehlersituationen auftreten können, untersuchen wir im folgenden eine Situation, in der eine Applikation für eine Typversion T_i implementiert wurde, also Objekte mit der Schnittstelle von T_i erwartet, das zugegriffene Objekt jedoch tatsächlich in der Version T_j seines Typs vorliegt. Für die folgenden Betrachtungen können wir der Einfachheit halber einige Abstraktionen vornehmen. Das Hinzufügen oder Entfernen von Obertypen kann wie das lokale Hinzufügen oder Entfernen der von dem Obertyp geerbten Eigenschaften behandelt werden. Weiterhin kann der Aufruf einer Methode des Objektes wie ein schreibender Zugriff auf die Eingabeparameter und ein lesender Zugriff auf den Rückgabeparameter angesehen werden. Damit lassen sich Veränderungen des Wertebereiches von Methodenparametern wie

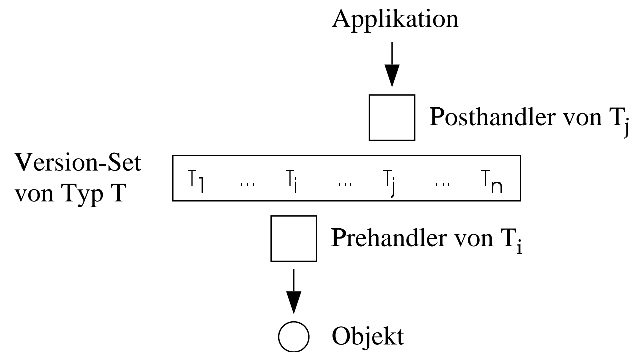


Abbildung 4.5: Die Typversionen eines Version-Sets und der Zugriff einer Applikation von T_j auf ein entsprechend T_i gespeichertes Objekt.

das Verstärken oder Abschwächen von Zusicherungen handhaben. Veränderungen des Codes von Methoden oder der Anzahl der übergebenen Parameter werden auf Hinzufügung und Löschung von Methoden zurückgeführt. Damit müssen wir nur noch zwei Fälle untersuchen, nämlich das Hinzufügen oder Entfernen von Eigenschaften und das Verstärken oder Abschwächen von Zusicherungen.

Wir gehen zunächst auf das Hinzufügen und Entfernen von Eigenschaften ein.

- Im Falle, daß von T_i zu T_j eine Eigenschaft entfernt wurde, entsteht ein Problem, wenn eine Applikation von T_i auf ein in T_j gespeichertes Objekt zugreift und dabei die in T_j entfernte Eigenschaft benötigt. Diese ist beim Objekt dann nämlich nicht vorhanden, so daß ein *Undefined_property-Error* signalisiert wird.
- Umgekehrt entsteht ein analoges Problem, wenn von T_i zu T_j eine Eigenschaft hinzugefügt wurde und eine Applikation, die von T_j auf ein Objekt von T_i zugreift, genau die in T_i noch nicht vorhandene Eigenschaft benötigt. Hier wird ebenfalls ein Undefined_property-Error signalisiert.

Beim Verstärken oder Abschwächen von Zusicherungen müssen wir zwischen lesenden und schreibenden Zugriffen unterscheiden.

- Im Falle, daß von T_i zu T_j eine Zusicherung verstärkt, ein Wertebereich also verkleinert wurde, kann ein Problem entstehen, wenn eine Applikation von T_i auf ein in T_j gespeichertes Objekt schreibend zugreifen möchte. Erfüllt der zu schreibende Wert nämlich die Zusicherung von T_i, nicht jedoch die von T_j, so liegt eigentlich ein zulässiger Zugriff der Applikation vor, den ein Objekt von T_j allerdings nicht bearbeiten kann (*writer's problem*). Hier wird ein *Unknown_value-Error* signalisiert. Greift umgekehrt eine Applikation von T_j auf ein Objekt von T_i lesend zu, so kann es passieren, daß der von Objekt gelieferte Wert die Zusicherung von T_i, nicht jedoch die von T_j erfüllt (*reader's problem*). Obwohl von dem Objekt ein zulässiger Wert geliefert wird, kann dieser von der Applikation von T_j nicht verarbeitet werden. Auch hier wird ein Unknown_value-Error signalisiert.
- Im Falle, daß von T_i zu T_j eine Zusicherung abgeschwächt, ein Wertebereich also vergrößert wurde, können analoge Probleme auftreten. Sowohl beim lesenden Zugriff einer Applikation von T_i auf ein Objekt von T_j als auch beim schreibenden Zugriff einer Applikation von T_j auf ein Objekt von T_i kann ein Wert außerhalb des erwarteten Wertebereiches liegen, wodurch es zu einem Unknown_value-Error kommt.

Der Fall, daß eine geforderte Eigenschaft gar nicht vorhanden ist (Undefined_property-Error) kann also bei Hinzufügen oder Entfernen von Eigenschaften auftreten, während Werte außerhalb

des zulässigen Wertebereiches (Unknown_value-Error) durch Verstärkung oder Abschwächung von Zusicherungen hervorgerufen werden können.

Zur Behandlung der potentiellen Fehlerfälle werden verschiedene Arten von Handlern eingesetzt. In [SZ87] werden diese in drei Dimensionen kategorisiert.

- Undefined_property-Handler und Unknown_value-Handler werden zur Behebung der entsprechenden Fehlerfälle benutzt.
- Zwischen der Behandlung lesender und schreibender Zugriffe wird durch Read- und Write-Handler unterschieden.
- Die Behandlung einer Fehlersituation kann zwei Schritte erfordern, wozu Pre- und Posthandler (in dieser Reihenfolge) eingesetzt werden.

Jeder in ENCORE definierbare Handler läßt sich in die obigen Dimensionen einordnen; allerdings ist nicht jede Kombination von Eigenschaften der verschiedenen Dimensionen sinnvoll und es können nicht alle erlaubten Arten von Handlern für dieselbe Eigenschaft spezifiziert werden. Ist beispielsweise eine Eigenschaft in einer Typversion nicht enthalten, so ist ein Undefined_property-Handler sinnvoll (ein Unknown_value-Handler jedoch nicht); für eine in einer Typversion vorhandene Eigenschaft macht nur ein Unknown_value-Handler (nicht jedoch ein Undefined_property-Handler) Sinn.

Prehandler werden eingesetzt, wenn eine durch eine Applikation benötigte Eigenschaft von einem referenzierten Objekt nicht erbracht wird oder wenn ein Objekt einen von einer Applikation übergebenen Wert für die entsprechende Eigenschaft als Unknown_value signalisiert. Ein Posthandler wird ausgeführt, wenn ein Objekt einen Wert liefert, für den die zugreifende Applikation einen Unknown_value-Error signalisiert.

Wenn eine Applikation des Typs T_j auf ein Objekt von Typ T_i zugreift, dann wird der Prehandler von T_i oder der Posthandler von T_j (oder beide) ausgeführt. Ein Prehandler einer Typversion T_i arbeitet also auf Objekten von T_i während ein Posthandler von Objekten aller anderen Versionen des Typs T gelieferte Werte auf ihre Eignung für Applikationen von T_i hin prüft und undefinierte Werte gleichsam ausfiltert.

Beim Hinzufügen von Eigenschaften zu einem Typ werden also Undefined_property-Handler für die bisherigen Versionen des Typs benötigt; beim Löschen von Eigenschaften werden nur Undefined_property-Handler für den neuen Typ benötigt. Bei Erweiterung eines Wertebereiches müssen Unknown_value-Posthandler bei älteren Typversionen, die die entsprechende Eigenschaft schon haben, ergänzt werden; bei Verkleinerung eines Wertebereiches werden entsprechend Unknown_value-Posthandler in der neuen Typversion benötigt.

In einer Typversion bearbeiten Unknown_value-Write-Prehandler und Unknown_value-Read-Posthandler denselben Wertebereich.

Handler können in ENCORE entlang der **is_a**-Beziehung zwischen Typen vererbt werden.

In dem vorgestellten Schemaevolutionsansatz wird die Typversion, der ein Objekt angehört, bei dessen Erzeugung statisch festgelegt. Damit wird eine dynamische Anpassung von Objekten an erweiterte Typen ausgeschlossen, denn für neu hinzugekommene Attribute kann in den Objekten kein zusätzlicher Speicherplatz verwaltet werden. Stattdessen werden im Version-Set-Interface des Typs enthaltene, von der festgelegten Typversion eines Objektes aber nicht erbrachte Eigenschaften durch attributspezifische Handler geleistet. Demzufolge können schreibende Zugriffe auf neue Attribute nur dann ausgeführt werden, wenn sie genau denjenigen Objektwert speichern, der von dem entsprechenden Lese-Handler sowieso geliefert würde. Ein Objekt bleibt demzufolge immer in einer Typversion verhaftet und es bestehen damit Unterschiede zwischen Objekten derselben Klasse.

Ein Effizienz-Problem könnte als weitere Konsequenz aus der statischen Zuordnung von Objekten zu Typversionen entstehen. Demzufolge können von Handlern gelieferte Werte wie bereits erwähnt nicht gespeichert, sondern müssen immer wieder neu berechnet werden. Da mit zunehmender Anzahl von Versionen eines Typs die Wahrscheinlichkeit sinkt, daß ein persistentes Objekt in der richtigen Objektversion vorliegt, müssen Handler bei fast jedem Zugriff ausgeführt werden, selbst wenn auf ein Objekt immer wieder entsprechend derselben Typversion zugegriffen wird.

Reskalierungen von Attributen, beispielsweise von Inch zu Zentimeter, sind aufgrund der Voraussetzung unveränderlicher und typweit eindeutiger Eigenschaftsnamen nicht möglich.

Eine erhebliche Einschränkung der Kooperation (Teilziel 3.10) zwischen Applikationen verschiedener Versionen eines Typs entsteht bei ENCORE dadurch, daß die physikalische Speicherrepräsentation eines Objektes zu dessen Erzeugungszeit festgelegt wird und danach nicht mehr verändert werden kann. Dies hat nämlich zur Folge, daß für Attribute, die nicht in der physikalischen Repräsentation eines Objektes enthalten sind, nur der Defaultwert geschrieben werden darf.

Der Handler-Ansatz von Skarra und Zdonik enthält einige Ähnlichkeiten zum Konzept der Simulation von Schemaänderungen durch Sichten. Zum einen ist ein Objekt immer in allen Versionen seines Typs sichtbar. Das Teilziel 3.11 der vollständigen Propagation wird somit erreicht. Es besteht jedoch keine Möglichkeit die Sichtbarkeit eines Objektes auf bestimmte Versionen seines Typs zu verringern (Teilziel 3.12) oder die Propagation nach ändernden Objektzugriffen zu beschränken. Zum anderen können sämtliche, durch verschiedene Versionen eines Typs sichtbaren Objektzustände auf der Grundlage eines einzigen, gespeicherten Objektwertes berechnet werden, d.h. eine Trennung zwischen verschiedenen Werten eines Objektes in verschiedenen Versionen seines Typs ist nicht möglich (Teilziel 3.13).

Da Handler nicht zwischen aufeinanderfolgenden Typversionen sondern immer relativ zum Version-Set-Interface spezifiziert werden, sind sehr viele Handler notwendig (Teilziel 3.16). Wenn das Version-Set-Interface durch spätere Typänderungen erweitert wird, sind für ältere Typversionen sogar die Redefinition bestehender und die Definition neuer Handler notwendig. Dies widerspricht dem technischen Teilziel lokaler Schemaänderungen (Teilziel 3.17).

AVANCE [BB88, BH89] (vormals OPAL genannt) verwendet zur Ermöglichung von Typänderungen einen ähnlichen Ansatz wie ENCORE. Dieser sieht Handler für die Behandlung der Ausnahmen *SpecificationMismatch* und *RepresentationMismatch* vor.

In [Zdo90] hebt Zdonik die bisherige Einschränkung eines Objektes auf nur eine physikalische Repräsentation auf. In Erweiterung von [SZ86, SZ87] wird nun für jedes Objekt für jede Version seines Typs eine eigene physikalische Repräsentation abgelegt, die *Objektversion* genannt wird. Damit können neu zu einem Typ hinzugefügte Attribute nicht nur wie bisher Defaultwerte lesen und schreiben, sondern verfügen über separaten Speicherplatz, dem beliebige Werte zugewiesen werden können. Für die beschriebene Erweiterung wird eine verzögerte Vorgehensweise vorgeschlagen, d.h. die Belegung von Speicherplatz und die Propagation der Objektwerte geschieht erst bei einem tatsächlichen Zugriff auf eine Objektversion.

4.5.3.3 Der Klassenversionierungsansatz von Monk und Sommerville

Simon Monk und Ian Sommerville [MS92, Mon93, MS93] verfolgen den Klassenversionierungsansatz, um den Schemaevolutionsprozeß zu unterstützen. Das in der Lisp-Erweiterung *Common Lisp Object System* (CLOS) prototypisch implementierte Datenbanksystem CLOSQL kann mehrere Versionen einer Klasse verwalten, die allerdings in einer der Entstehungszeit entsprechenden, linearen Sortierung angeordnet sein müssen. Für die Änderung eines Schemas wird eine Taxono-

mie von Primitiven angeboten, die weitestgehend der in [BKKK87] vorgeschlagenen entspricht. Applikationen und Anfragen arbeiten implizit stets mit der neuesten Version einer Klasse. Soll mit einer älteren Klassenversion gearbeitet werden, so ist deren Versionsnummer explizit anzugeben.

Objekte können zu jedem Zeitpunkt in jeder Version einer Klasse erzeugt werden und sind immer entsprechend dem Typ einer Version ihrer Klasse physikalisch gespeichert. Wird auf ein Objekt zugegriffen, das nicht in der geforderten Klassenversion gespeichert ist, so wird eine Konvertierung mit Hilfe sog. *Update-* und *Backdate-Methoden* vorgenommen. Diese Methoden konvertieren ein Objekt zwischen zwei Versionen seiner Klasse, die direkt nacheinander entstanden sind. Konvertierungen zwischen beliebigen Versionen einer Klasse werden durch automatische Hintereinanderausführung der entsprechenden Methoden erreicht (siehe Abbildung 4.6). Jede Update- und Backdate-Methode setzt sich aus *Update-* bzw. *Backdate-Funktionen* zusammen, welche jeweils den Wert eines Attributes der Zielklassenversion spezifizieren.

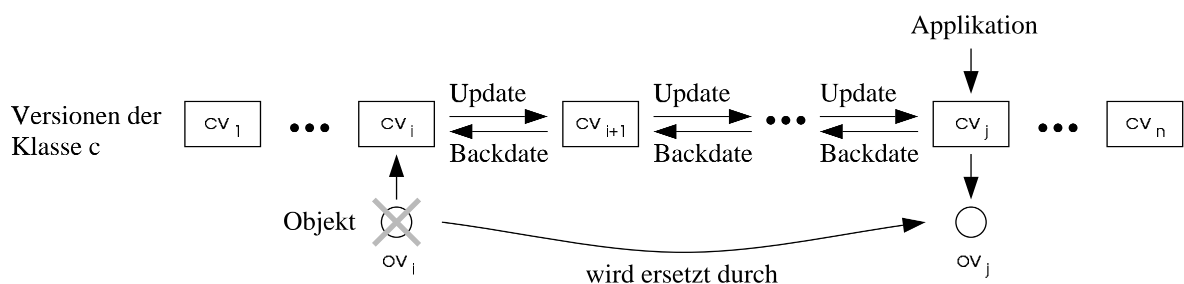


Abbildung 4.6: Die Update- und Backdate-Methoden beim Zugriff einer Applikation von Klassenversion cv_j auf ein entsprechend cv_i gespeichertes Objekt.

Damit werden in CLOSQL sowohl *Vorwärts-* als auch *Rückwärtskompatibilität* der Applikationen erreicht. Darunter verstehen Monk und Sommerville, daß neue Applikationen mit Objekten arbeiten können, die entsprechend älterer Klassenversionen erzeugt wurden (Vorwärtskompatibilität) und umgekehrt (Rückwärtskompatibilität). Für die Durchführung von Schemaänderungen wird eine graphische Benutzeroberfläche angeboten, über die auch die Generierung von Defaultkonvertierungsfunktionen durch CLOSQL ausgelöst werden kann.

Konvertierungen werden in CLOSQL verzögert durchgeführt, d.h. erst dann, wenn eine Applikation oder eine Anfrage auf ein Objekt zugreift. Ein Objekt ist dabei immer entsprechend derjenigen Version seiner Klasse gespeichert, durch die der letzte Zugriff auf das Objekt erfolgte. Andere Versionen des Objektes werden nicht gespeichert. Wird durch eine Klassenversion auf ein Objekt zugegriffen, die nicht mit der Version, in der das Objekt momentan gespeichert ist, übereinstimmt, so wird die alte Objektrepräsentation grundsätzlich durch die neue ersetzt. Dadurch könnte es passieren, daß Attributwerte durch Hin- und Herkonvertierung verloren gehen. Gegeben seien zwei verschiedene Versionen cv_i und cv_j einer Klasse c und ein in cv_i erzeugtes Objekt o. Wenn vor dem nächsten Zugriff auf o durch cv_i ein Zugriff durch cv_j erfolgt, so wird das Objekt hin- und herkonvertiert. Dabei ist es gleichgültig, ob cv_j neuer als cv_i ist (d.h. das Objekt wird erst vorwärts und dann rückwärts konvertiert) oder älter (d.h. das Objekt wird erst rückwärts und dann vorwärts konvertiert). Wenn cv_j ein Attribut von cv_i fehlt, ginge der Wert, den das Objekt o in diesem Attribut hatte, bei der Hin- und Herkonvertierung verloren. CLOSQL vermeidet solche Verluste, indem der Wert des in cv_j fehlenden Attributes systemintern zwischengespeichert und bei der Rückkonvertierung nach cv_i wiederhergestellt wird. Dabei kann es jedoch zu Inkonsistenzen kommen, wenn sich die konvertierten Attributwerte geändert haben und semantisch nicht mehr zu dem systemintern automatisch zwischengespeicherten Wert passen. Abbildung 4.7 zeigt ein Beispiel einer solchen Situation. Die zusätzliche Angabe der alten Postleitzahl wurde bei der Änderung von Version cv_i zu cv_j der Klasse `Person` entfernt. Das

dargestellte Objekt sei in cv_i angelegt und nach cv_j propagiert worden. Wenn nun jedoch die in der Abbildung angegebene Adressänderung von Frankfurt nach München durch eine Applikation von cv_j geschieht, so wird das Attribut `PLZ_alt` bei der Ausführung der Backdate-Methode nicht gesetzt und demzufolge bleibt der zwischengespeicherte Wert erhalten. Damit entsteht in dem Objekt in cv_i eine Inkonsistenz zwischen den beiden Postleitzahlen 80331 (`PLZ`) und 6000 (`PLZ_alt`).[42]

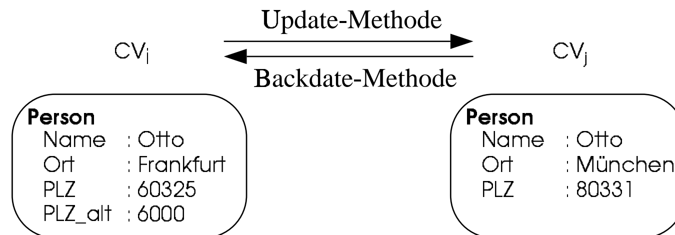


Abbildung 4.7: Probleme mit der Wiederherstellung zwischengespeicherter Attributwerte bei CLOSQL.

Bei der Entwicklung von CLOSQL wurde besonderer Wert auf die Beseitigung von Schwachpunkten in ENCORE und AVANCE (siehe Abschnitt 4.5.3.2) gelegt. Zum einen wird die dynamische Speicherung eines Objektes in CLOSQL entsprechend verschiedener Klassenversionen herausgestellt. Durch die statische Zuordnung eines Objektes zu einer Version seiner Klasse bei seiner Erzeugung, sind in ENCORE und AVANCE keine ändernden Schreibzugriffe auf Attribute möglich, welche nicht in der gespeicherten Repräsentation des Objektes enthalten sind. Im Gegensatz zu dieser *Emulation* anderer Klassenversionen führt CLOSQL eine echte *Konvertierung* durch, bei der im Falle hinzugefügter Attribute auch zusätzlicher Speicherplatz zur Verfügung gestellt wird, so daß auch auf diese neuen Attribute ohne Einschränkung schreibend zugegriffen werden kann. Zum anderen heben Monk und Sommerville die Restriktion auf, daß gleichnamige Attribute in verschiedenen Klassenversionen dieselbe Semantik tragen müssen. Durch die Verwendung der Konvertierungsfunktionen wird so beispielsweise die Reskalierung eines Attributes namens `Größe` von Inch in Zentimeter möglich, ohne daß dabei wie in ENCORE und AVANCE eine Umbenennung des Attributes erforderlich würde.

Ein weiterer Unterschied im Vergleich zu ENCORE und AVANCE ist die bereits erwähnte Zwischenspeicherung von Attributwerten, die in einer Zielklassenversion nicht enthalten sind. Davon abgesehen wird jedoch auch in CLOSQL jedes Objekt nur entsprechend einer einzigen Klassenversion gespeichert. Eine weitergehende Verbesserung besteht in der Möglichkeit Instanzen in CLOSQL durch Konvertierungsfunktionen dynamisch erzeugen und löschen zu können. Damit eröffnet sich die Möglichkeit für komplexe Änderungen, bei denen Objekte ihre Klassenzugehörigkeit ändern, miteinander verschmelzen oder sich in mehrere Teilobjekte aufspalten.

Die dem Klassenversionierungskonzept anhaftenden Nachteile (Teilziel 3.3), die bereits zuvor erörtert wurden, treten natürlich auch bei CLOSQL auf. Bemerkenswerterweise wird in [Mon93] kurz die Möglichkeit angedeutet, Mengen von Klassen gemeinsam zu versionieren, was eine deutliche Annäherung an das noch zu besprechende Konzept der Schemaversionierung darstellt. Trotz der Abhängigkeit einer Unterklasse von ihren Oberklassen wird in CLOSQL nur eine *flache Versionierung* (engl. *shallow versioning*) durchgeführt, d.h. daß bei Erzeugung einer neuen Version einer Klasse c nicht automatisch auch neue Versionen der Unterklassen von c angelegt

[42] Das sehr einfach gehaltene Beispiel der Abbildung 4.7 basiert auf einer konstruierten, klasseninternen (engl. *intra class*) Konsistenzanforderung für Personen in cv_i. Zum einen ließe sich das dargestellte Problem mit einer Normalisierung des Schemas, d.h. durch eine Unterscheidung zweier Klassen `Person` und `Adresse` nicht beheben; es träte dann in gleicher Form in der Klasse `Adresse` auf. Zum zweiten können analoge Situationen sogar bei klassenübergreifenden (engl. *inter class*) Konsistenzbedingungen eintreten. Aus Gründen der Einfachheit haben wir uns in Abbildung 4.7 jedoch auf ein Minimalbeispiel beschränkt.

werden. Die Beschränkung auf die lineare Versionierung ohne die Möglichkeit zur Ableitung alternativer Klassenversionen (Teilziel 3.6) wird nicht motiviert.

Einige der Konzepte von Monk und Sommerville lassen sich nur schwer beurteilen, da eine präzise Beschreibung, beispielsweise der Verwaltung der zwischengespeicherten Attributwerte fehlt. Außerdem muß mit Inkonsistenzen gerechnet werden, wenn eine bereits existierende Objektversion *ov* durch eine Änderung in einer anderen Klassenversion mittels Update- oder Backdate-Funktionen aktualisiert wird, diese jedoch nicht allen Attribute neue Werte zuweisen. Dann reflektieren einige Attribute von *ov* noch den alten Objektzustand, während andere durch die Konvertierung einen neueren Objektzustand darstellen. Konsequenzen von Objekterzeugungen und -löschungen durch Konvertierungsfunktionen insbesondere im Zusammenhang mit deren verzögerter Ausführung werden nicht berücksichtigt und die Auswirkung von Vererbungs- und Komponentenbeziehungen zwischen Klassen auf deren Versionierung sind nicht ausreichend untersucht.

Trotz der dargestellten Verbesserungen gegenüber den als Ausgangspunkt dienenden Systemen ENCORE und AVANCE, bleiben auch in CLOSQL einige unserer technischen Teilziele unerreicht. Ohnehin konzentriert sich die prototypische Implementierung auf die Schemaevolutionskonzepte und verzichtet auf die Realisierung eines vollständigen Datenbanksystems.

4.5.3.4 Der Schemaversionierungsansatz von Clamen

Steward Clamen stellt in [Cla92, Cla94] einen Ansatz vor, der ebenso wie ENCORE die Vermeidung von Applikationsanpassungen nach Schemaänderungen (technisches Teilziel 3.9) anstrebt. Dazu können von jeder Klasse mehrere Versionen angelegt werden. Ein von einer Applikation benutztes Schema enthält von jeder Klasse höchstens eine Version; genauere Angaben zum Verhältnis zwischen Klassenversionen und Schemaversionen werden allerdings nicht gemacht.

In der Datenbank wird ein Objekt durch die disjunkte Vereinigung einer oder mehrerer sog. *Facetten* (engl. *facets*) repräsentiert. Diese Facetten entsprechen Objektversionen; verschiedene Facetten eines Objektes entsprechen den Typen verschiedener Versionen seiner Klasse. Aus logischer Sicht wird für jede Version der Klasse eines Objektes genau eine passende Facette des Objektes angelegt und nur auf diese Facette kann durch eine Applikation der entsprechenden Klassenversion zugegriffen werden. Die Propagation von Werten zwischen verschiedenen Facetten eines Objektes geschieht mit Hilfe attributspezifischer Abbildungen. Diese von Clamen als *Ableitungsfunktionen* (engl. *derivation functions*) bezeichneten Abbildungen entsprechen also den Update- und Backdate-Funktionen von CLOSQL.

Bezüglich der Beziehungen zwischen Attributen verschiedener Facetten eines Objektes unterscheidet Clamen vier Attributkategorien:

- Ein *gemeinsames* (engl. *shared*) Attribut repräsentiert in verschiedenen Facetten dieselbe Attributsemantik und hat demzufolge denselben Wert.
- Der Wert eines *abgeleiteten* (engl. *derived*) Attributes kann direkt aus den Attributwerten einer anderen Facette berechnet werden.
- Ein *abhängiges* (engl. *dependent*) Attribut hängt zwar von den Attributen anderer Facetten ab, sein Wert kann jedoch nicht ausschließlich aus deren Werten berechnet werden.
- Ein *unabhängiges* (engl. *independent*) Attribut kann nicht durch Wertänderungen der Attribute anderer Facetten beeinflußt werden.

Nach [Cla92] besteht eine Erweiterung dieses Ansatzes gegenüber ENCORE in der Unterstützung abhängiger Attribute.

Durch die Kategorisierung der Attribute können die bei der Änderung eines Attributwertes in einer Facette zur Propagation in eine andere Facette notwendigen Schritte ermittelt werden.

- Für ein gemeinsames Attribut der beiden Facetten muß nur der veränderte Wert kopiert werden, d.h. die Ableitungsfunktion ist die Identität. Der Schemaentwickler muß lediglich spezifizieren, welche Attribute zueinander *synonym* sind.

- Für abgeleitete Attribute ist eine Ableitungsfunktion anzugeben, die nach der Wertänderung auszuführen ist und die den Wert des abgeleiteten Attributes aus dem Wert der Attribute einer anderen Facette berechnet.

- Für abhängige Attribute ist eine Ableitungsfunktion anzugeben, die den Wert des abgeleiteten Attributes aus dem Wert der Attribute aller Facetten des Objektes berechnet.

- Für unabhängige Attribute ist keine Ermittlung des Attributwertes möglich.

Das beschriebene Versionierungsmodell mit Facetten würde bei einer direkten Übertragung auf eine Implementierung einen erheblichen Aufwand an Platz zur Speicherung der Facetten und an Zeit zur sofortigen Neuberechnung aller Facetten bedeuten. Daher schlägt Clamen auf der physikalischen Ebene einige Konzepte vor, die zur Steigerung der Effizienz seines Ansatzes beitragen, die jedoch auf der bisher beschriebenen logischen Ebene transparent bleiben.

- Speicherplatz für eine Facette muß erst dann belegt werden, wenn auf diese Facette das erste mal zugegriffen wird. Auch die Ausführung der Ableitungsfunktionen kann jeweils bis zu einem tatsächlich durchgeführten Zugriff einer Applikation verzögert werden. Der aktuelle Wert einer Facette wird dann aufgrund der neuesten Facette ermittelt.

- Gemeinsame Attribute können an demselben Speicherplatz abgelegt werden. Damit wird nicht nur Speicherplatz gespart sondern auch das Laufzeitverhalten verbessert, da das Kopieren der Attributwerte eingespart wird.

- Wenn für eine Ableitungsfunktion auch die Umkehrfunktion bekannt ist, so genügt es bei voneinander abgeleiteten Attributen den Wert eines dieser Attribute zu speichern. Hier bestehen verschiedene Alternativen zwischen den Extremen minimalen Speicherplatzverbrauchs und minimaler Zugriffszeit. Die von ENCORE realisierte Variante stellt laut Clamen das erstgenannte Extrem im Spektrum möglicher Alternativen dar.

In [Cla94] postuliert Clamen die Anwendbarkeit seines Schemaevolutionsmechanismus in verteilten OODBMS. Wenn diese nämlich *heterogen* strukturiert sind, d.h. wenn jeder Knoten des Netzwerkes ein eigenes Datenbankschema haben kann, dann entsteht das Problem der Integration solcher Schemata. Zu dessen Lösung müssen die beteiligten Teilschemata eine Evolution auf ein globales Schema hin durchlaufen und dazu sind die präsentierten Mechanismen einsetzbar. Auf Objektebene bestehen weiterhin Ähnlichkeiten zwischen ableitbaren Facetten einerseits und den in verteilten DBMS eingesetzten Replikaten von Objekten andererseits.

Clamen stellt seine Konzepte in [Cla92, Cla94] nur beispielhaft vor, wobei er sich der Einfachheit halber stets auf nur zwei Versionen einer Klasse beschränkt. Aus den Darstellungen geht nicht hervor, welche Ableitungsfunktionen bei drei und mehr Versionen einer Klasse benötigt werden und wie die Kategorisierung der Attribute zu verallgemeinern ist. Ein abhängiges Attribut könnte bei Hinzufügen einer neuer Klassenversion ableitbar werden und die Unabhängigkeit eines Attributes kann mit neuen Klassenversionen verloren gehen.

4.5.3.5 Der Schemaversionierungsansatz von Labib und Saunders

Gamal Labib und Dave Saunders [LS93, LS94] stellen einen Ansatz zur Schemaevolution vor, der auf der Versionierung des Schemas beruht. Da sich in der Regel von einer Schemaversion zur nächsten nur relativ wenige Klassen ändern, sind die Klassen die eigentlichen Objekte der Versionierung, d.h. jede Schemaversion enthält (höchstens) eine Version jeder Klasse. Eine von einer Schemaänderung nicht betroffene Klasse kann in derselben Version in mehreren Schemaversionen enthalten sein.

Ein Hauptaugenmerk der Autoren liegt auf der Beseitigung von Schwachstellen der Systeme ENCORE (siehe Abschnitt 4.5.3.2) und CLOSQL (siehe Abschnitt 4.5.3.3). An dem von ENCORE benutzten Handler-Konzept wird kritisiert, daß Attribute umbenannt werden müssen, wenn ihre Semantik verändert werden soll, d.h. wenn Wertebereiche von Attributen vergrößert oder verkleinert oder Attributwerte reskaliert werden sollen. Die Kritik an CLOSQL richtet sich auf dessen Beschränkung der Ableitungsstruktur von Klassenversionen auf eine lineare Liste, weswegen immer nur die neueste Version modifiziert werden kann.

Die genannten Schwachpunkte sollen durch eine Kombination der in den beiden Systemen verwendeten Mechanismen beseitigt werden. Dazu werden Version-Sets und Handler wie in ENCORE verwendet; in Erweiterung der Originalarbeiten von Skarra und Zdonik erlauben Labib und Saunders jedoch mehrere Version-Sets pro Klasse. Da entsprechend dem technischen Teilziel 3.11 der vollständigen Propagation jedes Objekt einer Klasse in jeder Version dieser Klasse zugreifbar bleiben soll, werden Abbildungen zwischen Klassenversionen desselben Version-Set (intra-version-set) und zwischen verschiedenen Version-Sets (inter-version-set) einer Klasse benötigt. Abbildungen der ersten Kategorie werden wie in ENCORE durch entsprechende Handler realisiert. Für Abbildungen der zweiten Kategorie werden *Forward-* und *Backward-Transformer* verwendet, die im wesentlichen den Update- und Backdate-Methoden von CLOSQL entsprechen.

Zwischen Versionen einer Klasse mit übereinstimmender Attributsemantik kann eine dem Handler-Konzept folgende Bearbeitung von Ausnahmefällen vorgenommen werden. Eine neu entstandene Klassenversion wird dementsprechend einem passenden Version-Set der jeweiligen Klasse hinzugefügt. Die zwischen den Versionen eines Version-Set verwalteten Verbindungen entsprechen damit nicht wie sonst bei Versionierungskonzepten üblich der Ableitungsbeziehung (**is_derived_from**), sondern stellen eher eine Ähnlichkeit bezüglich der verwendeten Attributsemantik dar. Eine genauere Beschreibung dieses Aspektes fehlt allerdings in [LS93, LS94]. Stimmt die Attributsemantik einer neuen Klassenversion jedoch mit keiner existierenden Version der Klasse überein, so wird ein neues Version-Set für die Klasse angelegt. Ähnlichkeiten zwischen verschiedenen Version-Sets einer Klasse werden in einem sog. *Version-Set-Graph* dargestellt (siehe Abbildung 4.8).

Um entscheiden zu können, ob eine neue Klassenversion zu einem der bestehenden Version-Sets hinzugefügt werden kann oder ob ein neues Version-Set für die Klasse angelegt werden muß, wird in [LS93] eine Taxonomie vorgestellt, die die verwendbaren Schemaänderungsprimitive in zwei disjunkte Kategorien einteilt. Während die erste Kategorie diejenigen Primitive enthält, die die bestehende Attributsemantik unverändert lassen, gehören der zweiten Kategorie in ENCORE nicht verwendbare, semantikverändernde Primitive an, mit denen beispielsweise Wertebereiche von Attributen verändert oder Attributwerte reskaliert werden können. Werden bei der Ableitung einer neuen Klassenversion cv' von cv ausschließlich Primitive der ersten Kategorie verwendet, so kann cv' in den Version-Set von cv integriert werden, ansonsten muß ein neuer Version-Set angelegt werden. Während in ersteren Fall wie bei ENCORE ggf. Handler zu den Klassenversionen des erweiterten Version-Set hinzugefügt werden müssen, wird im zweiten Fall die Definition eines Forward- und eines Backward-Transformers zwischen den Version-Sets von cv und cv' notwendig.

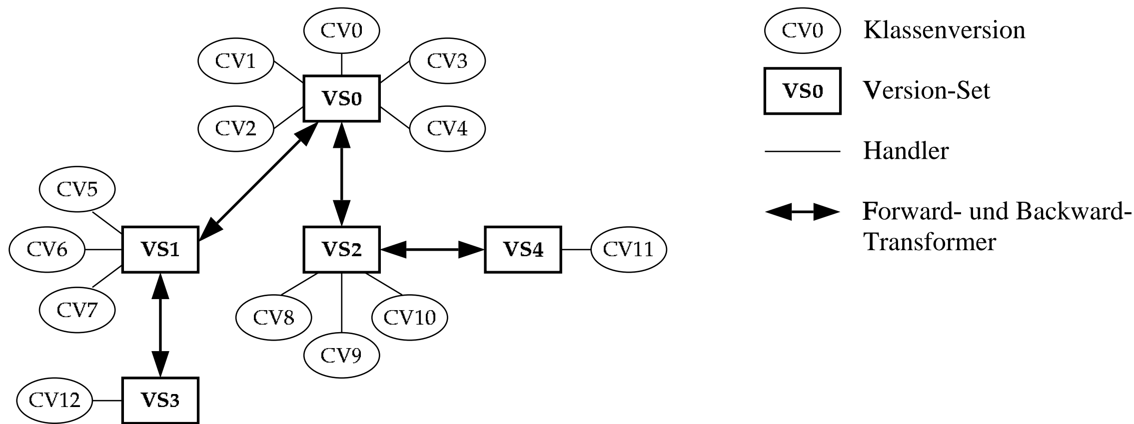


Abbildung 4.8: Ein Version-Set-Graph mit Handlern und Transformern.

Wenn bei der Ableitung einer neuen Klassenversion nur Methoden verändert werden, so hat dies keinen Einfluß auf die Objekte. Daher kann die neue Klassenversion in einem solchen Fall in denselben Version-Set integriert werden, wie die Vorgängerversion.

Wird auf ein entsprechend der Klassenversion cv gespeichertes Objekt durch eine Klassenversion cv' zugegriffen, so sind i.Allg. drei Schritte notwendig, nämlich die Anwendung von Handlern im Version-Set von cv, die Anwendung von Transformern entlang des Pfades vom Version-Set von cv zum Version-Set von cv' durch den Version-Set-Graph und die Anwendung von Handlern im Version-Set von cv' (siehe Abbildung 4.8). Die Anwendung von zwischen voneinander abgeleiteten Version-Sets definierten Transformern ist natürlich nur dann notwendig, wenn cv und cv' verschiedenen Version-Sets angehören. Dabei kann es passieren, daß durch die Hintereinanderausführung der Transformer auf dem Pfad vom Version-Set von cv zum Version-Set von cv' Attributsemantik verloren geht. Dies passiert bei Attributen von cv und cv', die nicht in allen auf dem Pfad durch den Version-Set-Graph gelegenen Version-Sets enthalten sind. Zur Behebung dieser Problematik empfehlen die Autoren, daß der Schemaentwickler in jedem Version-Set eine Klassenversion anlegt, die alle Attribute der Klasse beinhaltet.

Die Autoren schlagen auch ein Konzept zur *Objekt-Migration* vor. Darunter verstehen sie allerdings nicht wie üblich den Wechsel eines Objektes zu einer anderen Klasse, sondern zu einer anderen Version derselben Klasse. Damit verbunden ist die Änderung des physikalischen Speicherformates der migrierten Objekte, was aus drei Gründen vorteilhaft sein kann. Zum einen kann ein Objekt in einer Klassenversion mit zusätzlichen Attributen oder vergrößerten Wertebereichen mehr Semantik speichern und ist nicht auf von Handlern gelieferte Defaultwerte beschränkt. Zum anderen können nicht mehr von Applikationen benötigte Klassenversionen gelöscht werden, wenn die Möglichkeit besteht, ihre Objekte in andere Versionen der Klasse zu migrieren. Schließlich kann es zur Steigerung der Zugriffsgeschwindigkeit auf ein Objekt angebracht sein, dieses in eine Klassenversion zu migrieren, von der es besonders häufig verwendet wird.

Die von Labib und Saunders vorgeschlagene Verbindung der Konzepte von ENCORE und CLOSQL hebt einige der Schwachpunkte der separaten Ansätze auf. Im Vergleich zu ENCORE wurde die verfügbare Schemaänderungstaxonomie um Primitive ergänzt, mit denen Veränderungen der Semantik existierender Attribute möglich sind. Solche Veränderungen beziehen sich jedoch nur teilweise auf das Schema, so daß sie hier nicht als Erweiterung der Taxonomie der Schemaänderungsprimitive (Teilziel 3.1) gewertet werden. Die Einschränkung von CLOSQL auf eine lineare Liste von Versionen wurde durch eine Hierarchie von Klassenversionen aufgehoben. Aus [LS93, LS94] geht jedoch nicht hervor, wie eine neue Version einer Klasse mit existierenden Klassenversionen auf Ähnlichkeiten der Attributsemantik hin verglichen wird und wie demzufolge

die Beziehungen zwischen den Versionen einer Klasse bzw. eines Version-Sets und zwischen den Version-Sets selbst aussehen.

Die Länge des gesamten Transformationspfades, d.h. die Anzahl der für die Transformation eines Objektes auszuführenden Handler und Transformer verringert sich gegenüber CLOSQL im Mittel dadurch, daß mehrere Versionen in einem Version-Set zusammengefaßt sind und der Pfad durch einen Version-Set durch einen einzigen Transformer traversiert werden kann. Dadurch wird eine Verbesserung der Effizienz (Teilziel 3.15) gegenüber ENCORE erreicht. Diese verbessert sich bei Labib und Saunders weiterhin durch die Möglichkeit Objekte zu migrieren. Dadurch können Objekte physikalisch entsprechend derjenigen Klassenversion abgelegt werden, durch die sie am häufigsten zugegriffen werden, womit sich die Zahl der notwendigen Konvertierungen verringert.

Bei mehreren Version-Sets einer Klasse kann auf dem Transformationspfad allerdings Semantik verloren gehen, was nach Labib und Saunders durch manuelles Anlegen einer zusätzlichen Klassenversion pro Version-Set vermieden werden soll. Diese als *intermediate class version* bezeichnete Klassenversion soll alle Attribute der Klasse enthalten, womit allerdings weitere Nachteile insbesondere bezüglich des Spezifikationsaufwandes (Teilziel 3.16) verbunden sind. Zum einen wird dadurch der Vorteil einer verringerten Anzahl zu definierender Handler wieder relativiert. Um Verluste an Semantik bei der Propagation zu vermeiden muß entsprechend der Empfehlung von Labib und Saunders weiterhin sichergestellt werden, daß jedes Version-Set eine Klassenversion enthält, die alle Attribute der Klasse beinhaltet. Sofern das nicht ohnehin der Fall ist, muß eine weitere Klassenversion in dem jeweiligen Version-Set zusätzlich spezifiziert werden. Schließlich kann die in verschiedenen Version-Sets unterschiedliche Attributsemantik nicht durch eine der existierenden Klassenversionen allein erfaßt werden. Daher müssen die zusätzlichen Klassenversionen Hilfsattribute mit modifizierten Namen enthalten. Damit geht zumindest für den Schemaentwickler, der die Hilfsattribute definieren muß, gegenüber ENCORE der Vorteil verloren, daß keine Umbenennungen von Attributen notwendig werden.

Viele allgemeine Methodiken für die objektorientierte Modellierung empfehlen, nur abstrakte Klassen als Oberklassen zu verwenden; Systeme wie z.B. Demeter (siehe Abschnitt 4.3.4.3) erzwingen diese Form der Programmierdisziplin sogar. Auch das von Labib und Saunders verwendete Objektmodell läßt die Speicherung von Objekten nur in Blättern der Klassenhierarchie zu. Damit erleichtert sich die Realisierung von Schemaänderungsprimitiven zur Modifikation innerer Klassen erheblich, da diese dann keine direkten Objekte enthalten.

Von den vorgeschlagenen Mechanismen scheint es keine Implementierung zu geben und die Erläuterungen sind stellenweise wenig formal. Bei der Beschreibung des Version-Set-Graph wird nicht auf die Problematik mehrdeutiger Transformationspfade eingegangen, was dadurch begründet sein könnte, daß es sich tatsächlich um einen Version-Set-Baum handelt.

4.5.3.6 Der Ansatz von Odberg

Erik Odberg verfolgt in seinen Arbeiten [Odb92, Odb94b, Odb94a, Odb95] primär das Ziel der Typänderungstransparenz (engl. *type change transparency*). Er definiert diese wie in [SZ86, SZ87] (siehe Abschnitt 4.5.3.2), was unserem Ziel 3.9 der Vermeidung von Applikationsanpassungen entspricht.

4.5.3.6.1 Das Datenmodell von Odberg

Das von Odberg in [Odb92] verwendete Datenmodell basiert auf der Unterscheidung zwischen

Typen und Klassen.[43] Deren Definition weicht jedoch etwas von der üblichen Definition ab und soll daher hier zunächst kurz vorgestellt werden.

Ein *Typ* beschreibt nach [Odb92] eine Menge extern zugreifbarer Methoden, die von einer Instanz des Typs (d.h. von einem Objekt) als Schnittstelle angeboten wird. Mittels Subtypisierung kann ein Typ Schnittstellen anderer Typen erben. Die Subtypisierung ist *strikt* (Methoden können in Untertypen nur ergänzt, nicht jedoch entfernt werden) und wird explizit spezifiziert. Damit wird implizit auch die Substituierbarkeit zwischen Instanzen verschiedener Typen festgelegt. Jeder Typ hat eine *Extension*, die alle Instanzen dieses Typs (und seiner Untertypen) beinhaltet.

Eine *Klasse* implementiert einen Typ, indem sie seine Methoden und Attribute zur Repräsentation seiner Instanzen implementiert. Eine Subklassenbeziehung erlaubt die Vererbung der Eigenschaften (Attribute und Methoden) zwischen Klassen, wobei Unterklassen geerbte Eigenschaften nicht nur umbenennen und reimplementieren sondern auch löschen dürfen. Jede Klasse bestimmt explizit einen oder mehrere Typen, die sie implementiert, indem sie zumindest alle Methoden dieser Typen implementiert. Ein Typ kann durch mehrere Klassen implementiert werden. Zwischen Klassen- und Typhierarchie muß kein Zusammenhang bestehen.

Abstrakte Typen werden durch keine Klassen implementiert und können demzufolge nicht instanziiert werden. Sie können jedoch an andere Typen vererben. *Abstrakte Klassen* implementieren keinen Typ, können jedoch an andere Klassen vererben.

Ein Schema besteht nach [Odb92] aus einer Menge von Typen und einer Menge von Klassen. Applikationen sehen Objekte als Instanzen von Typen, entsprechend sind Objektreferenzen typisiert. Die Klasse eines Objektes wird lediglich bei seiner Erzeugung angegeben.

4.5.3.6.2 Schemaversionierung nach Odberg

Das Hauptziel von Odberg ist die Vermeidung von Applikationsanpassungen nach Schemaänderungen. Daher wird eine Versionierung auf Schemaebene durchgeführt, die Änderungen an Typen und Klassen widerspiegelt. Jede Applikation ist statisch an eine Schemaversion gebunden, die die benutzbaren Typ- und Klassenversionen spezifiziert, und bleibt daher von Schemaänderungen unbetroffen. Für jede Applikation sollen alle Objekte zugreifbar sein (engl. *transparent object dereferencing*), selbst wenn die Klasse eines Objektes nicht in der Schemaversion der Applikation enthalten ist. Eine solche Klasse muß dann zur Laufzeit dynamisch an die jeweilige Applikation gebunden werden.

Sowohl strukturelle Änderungen durch Erzeugung neuer Klassen oder Klassenversionen als auch verhaltensmäßige, also Typen betreffende Änderungen können Inkompatibilitäten zwischen Objekten und Applikationen zur Folge haben. Zur Erreichung der Transparenz von Schemaänderungen muß jedes Objekt alle Versionen seiner Typen unterstützen, d.h. jede Methode, die in einer beliebigen Version eines Typs definiert ist, muß von jedem Objekt dieses Typs angeboten werden. Dazu wird für jeden Typ implizit die Vereinigungsmenge aller seiner Versionen (engl. *union set type version, USTV*) gebildet und automatisch verwaltet. Der USTV-Typ eines Typs ist Untertyp jeder Version dieses Typs.

Entsprechend wird für jeden Typ eine abstrakte USTV-Klasse verwaltet, die Defaultimplementierungen für alle Methoden des USTV-Typs anbietet, jedoch keine Attribute enthält. Die USTV-Klasse eines Typs ist Oberklasse aller Klassen, die diesen Typ implementieren und all derer Klassenversionen. Die Defaultimplementierungen der Methoden in einer USTV-Klasse können in deren Unterklassen überschrieben werden.

---

[43] Dasselbe Verhältnis zwischen Typen und Klassen wird beispielsweise in POOL-I [AL90] verwendet. Auch das Objektmodell der ODMG [Cat96, CB98, CBB+00] macht eine ähnliche Unterscheidung.

Das Hinzufügen von Attributen zu einer Klasse würde die Belegung zusätzlichen Speicherplatzes für existierende Objekte erfordern und wird in [Odb92] nicht unterstützt, d.h. es gibt nur eine (bezüglich der Attribute unveränderliche) Objektversion je Objekt. Die in [Odb92] beschriebene Idee der USTV-Typen und -Klassen erfordert viele sehr umfangreiche Klassen, da jede Klassenversion alle Methoden anderer Versionen derselben Klasse ebenfalls implementieren muß, wenn nicht die Defaultimplementierung der USTV-Klasse verwendet werden soll. Damit kann weiterhin die Ergänzung von Klassen bereits in Benutzung befindlicher Schemaversionen notwendig werden, wenn von diesen Klassen in einer neuen Schemaversion neue Versionen angelegt werden, die bisher nicht vorhandene Methoden implementieren. Die in [Odb92] eingeführte Trennung zwischen Klassen und Typen und die Konzepte der USTV-Typen und -Klassen werden in den späteren Arbeiten [Odb94b, Odb94a, Odb95] allerdings nicht weiter verfolgt. Dort wird stattdessen eine Erweiterung des C++-Objektmodells vorgenommen.

In [Odb94b] wird ein Rollenmodell vorgestellt, um dynamische Typänderungen von Objekten während ihres Lebenszyklus zulassen zu können. Analog dieser Evolutionsunterstützung auf Objektebene befaßt sich [Odb94a] mit Evolution auf Schemaebene und stellt Konzepte für Schemaänderungsmanagement (engl. *schema modification management, SMM*) vor. Die Dissertation [Odb95] faßt beide Ergebnisse schließlich zusammen und zeigt Ähnlichkeiten zwischen evolutionärer und der sog. *multi-perspectived* Natur von Objekten und evolutionärer und *multi-perspectived* Natur von Klassen auf. Die von Odberg für Objekte und Schemata vorgeschlagenen Evolutionskonzepte werden in den beiden folgenden Abschnitten näher vorgestellt.

4.5.3.6.3 Das Rollenmodell von Odberg

Ähnlich wie die Schemaversionierung zur Unterstützung dynamischer Änderungen an Schemata verwendet werden kann, eignet sich das in zahlreichen Publikationen [Ber92, LL95, LL98, Per89, RS91, Wie90, WdJ91, WCL97] behandelte Konzept der *Rollen* zur Beschreibung sich zu ihrer Lebenszeit dynamisch ändernder Objekte. In [Odb94b] führt Odberg ein Rollenmodell ein, das sich weitgehend an den genannten früheren Veröffentlichungen orientiert. Danach sind Rollen Instanzen von Rollenklassen, die gemeinsam mit herkömmlichen Klassen in einer Subklassenbeziehung stehen. Ein Objekt kann zu jeder Zeit eine beliebige Menge von Rollen spielen, die verschiedene seiner Aspekte beschreiben (Objekte sind *multi-perspectived*). Die Rollen werden dynamisch, d.h. zur Lebenszeit des Objektes, angenommen oder abgelegt, indem entsprechende Rollen-Konstruktoren und -Destruktoren (analog zu Objekt-Konstruktoren und -Destruktoren) aufgerufen werden. Dabei kann und muß jedoch jede Rolle separat identifiziert werden und ein Objekt kann zu jedem Zeitpunkt nur eine Rolle jeder Rollenklasse spielen.

Neben der Festlegung der Intension einer Klasse, d.h. der möglichen Zustände, die ein Objekt der Klasse annehmen kann, behandelt Odberg den Aspekt der Kategorisierung. Dabei wird der extensionale Charakter einer Klasse hervorgehoben und es wird untersucht, welchen Klassen ein existierendes Objekt angehören kann. Diese Kategorisierung geschieht dynamisch in Abhängigkeit von dem aktuellen Zustand eines Objektes und den Rollen, die es gerade spielt. Die dazu in [Odb94b, Odb95] eingeführten *kategorisierenden Klassen* (engl. *category classes*) stellen eine Erweiterung herkömmlicher Klassen dar, bei denen die Zugehörigkeit von Objekten durch Angabe von je einem Prädikat eingeschränkt oder ermöglicht werden kann. Das Prädikat kann den Zustand eines Objektes, seine Rollen und seine Beziehungen zu anderen Objekten berücksichtigen. Die Kategorisierung eines Objektes kann dabei manuell oder automatisch (oder kombiniert) geschehen. *Manuell* kategorisierende Klassen erlauben das Hinzufügen oder Entfernen von Rollen bei Objekten nur dann, wenn das entsprechende Prädikat erfüllt ist. *Automatisch* kategorisierende Klassen fügen ihren Objekten sofort neue Rollen hinzu oder entfernen diese wieder, wenn das entsprechende Prädikat erfüllt bzw. verletzt wird. Wenn ein zwischenzeitlich verletztes Prädikat erneut wahr wird, so wird die zuvor durch den Automatismus entfernte Rolle wieder sichtbar

und befindet sich noch in demselben Zustand wie vor ihrer Entfernung.[44] Die Extension einer kategorisierenden Klasse besteht aus Objekten einer anderen Klasse *candidate class*, die potentielle Kandidaten enthält. Herkömmliche Klassen können als Spezialfall der kategorisierenden Klassen, nämlich gerade solche mit leerem Prädikat (TRUE), angesehen werden. Mit automatisch kategorisierenden Klassen können disjunkte/überlappende und komplette/nicht komplette Partitionen einer Klasse gebildet werden.

Die Semantik bezüglich der Vererbung zwischen Rollenklassen wird jedoch nicht vollständig beschrieben. Wenn ein Objekt eine Rolle spielt, dann treten vermutlich auch alle Rollen der direkten Oberklassen der Rollenklasse auf. Bei Konflikten zwischen verschiedenen direkten Oberrollenklassen wird eine Rolle wohl nur einmal gespielt, wenn sie jeweils von einer gemeinsamen Oberrollenklasse geerbt wird. Es ist nur maximal eine Rolle pro Rollenklasse und Objekt zu einer Zeit möglich und bei vielen verschiedenen Datenbankänderungen muß die Methode `MakePersist` jeweils explizit aufgerufen werden, um die Persistenz der Änderungen zu erreichen.

4.5.3.6.4 Schemaänderungsmanagement nach Odberg

In [Odb94a] befaßt sich Odberg mit der Verwaltung sich ändernder Schemata und den Auswirkungen von Schemaänderungen auf existierende Objekte. Dabei verwendet er den Ansatz der Schemaversionierung und berücksichtigt dabei auch, daß semantische Abhängigkeiten zwischen verschiedenen Schemaversionen nicht nur entlang der Ableitungsbeziehung existieren können.

Für das Schemaänderungsmanagement werden im Gegensatz zu [Odb92] nur noch einfache Klassen verwendet; die Konzepte der Typen und der kategorisierenden Klassen finden dabei keine Berücksichtigung mehr.

Der direkte Schemaevolutionsansatz geht nach Odberg stets davon aus, daß Schemaänderungen korrigierender Natur sind (GemStone [PS87], ORION [BKKK87]). Im Gegensatz dazu werden frühere Spezifikationen beim Schemaversionierungsansatz (AVANCE [BB88, BH89], ENCORE[45] [SZ86, SZ87], CLOSQL [MS92], Clamens Arbeit [Cla92], Bratsbergs Arbeit [Bra93]) nicht obsolet. Existierende Applikationen sind damit von Veränderungen des Schemas nicht betroffen und können mit der Datenbank wie vor der Änderung weiterarbeiten. Damit wird die Änderungstransparenz (engl. *change transparency*) auf Schemaebene erreicht; ein Hauptproblem bleibt dabei zunächst, wie sich ein Objekt mit mehreren Spezifikationen verhält.

Laut Odberg besteht eine Ähnlichkeit zwischen Schemaevolution und Schemaversionierung darin, daß Konvertierungsfunktionen (von Odberg *coercion functions* genannt) angegeben werden, die die Umsetzung eines Objektes zwischen Ausprägungen seines Typs beschreiben. Während diese Konvertierung im direkten Ansatz nur in eine Richtung erfolgt, geht sie bei der Versionierung in beide Richtungen. Durch transitive Anwendung von Konvertierungsfunktionen kann sich ein Objekt entsprechend jeder Version seiner Klasse verhalten, unabhängig von der Klassenversion, in der das Objekt erzeugt wurde. Bei ENCORE (siehe Abschnitt 4.5.3.2) werden Konvertierungsfunktionen im Gegensatz zu diesem allgemeinen Modell der Schemaversionierung zwischen jeder Klassenversion (bzw. Typversion) und der Vereinigung der Eigenschaften aller Versionen einer Klasse (eines Typs) definiert.

Odberg bemängelt an den bestehenden Ansätzen insbesondere die folgenden Schwachstellen.

[44] Odberg schlägt als Konzept für eine prototypische Implementierung einen verzögerten Mechanismus vor, der die Zugehörigkeit eines Objektes zu einer Klasse erst dann prüft, wenn auf das Objekt als Instanz dieser Klasse zugegriffen wird. Damit wird die in [Odb95] beschriebene Erhaltung des Zustandes zwischenzeitlich unsichtbarer Rollen schon implementierungstechnisch bedingt. Die Alternative einer Reinitialisierung von Rollen bei ihrer erneuten Annahme ist gar nicht möglich, da eine temporäre Unsichtbarkeit mit einem verzögerten Mechanismus nicht festgestellt werden kann.

[45] Das Konzept der USTV-Typen und -Klassen ist in gewissem Maße den von ENCORE (siehe Abschnitt 4.5.3.2) verwendeten Typversionen, welche sämtliche Attribute aller Versionen eines Typs umfassen, vergleichbar. Odberg erweitert die Idee von Skarra und Zdonik um die Integration des entstehenden Gesamttyps in eine Typhierarchie.

- Eine Versionierung auf Klassenebene ist problematisch bei komplexen Änderungen, die gleichzeitig mehrere Klassen betreffen. Dazu gehören beispielsweise das Verschieben von Attributen oder die Restrukturierung der Vererbungshierarchie (siehe Teilziel 3.3).
- Eine Klassenhierarchie beschreibt nicht nur Eigenschaften von Objekten sondern auch ein konzeptuelles Modell. Mit jeder Klasse wird implizit eine Klassifikationssemantik angegeben, die bestimmt, welche Objekte als der Klasse angehörig angesehen werden können. Damit sind in verschiedenen Kontexten verschiedene Klassifikationen möglich und ein Objekt kann Mitglied verschiedener Klassen sein.
- Konvertierungsfunktionen können nur zwischen zusammenhängenden, also direkt voneinander abgeleiteten Schemaversionen definiert werden. In Fällen, in denen alte Eigenschaften oder ganze Klassen, die zwischenzeitlich gelöscht waren, später wieder auftauchen, kann hier kein semantischer Zusammenhang mehr hergestellt werden und auf Objektebene kann Attributinformation verloren gehen. Stattdessen sollte eine globale Sicht auf alle Schemaversionen erlaubt werden, mit der semantische Verbindungen in beliebiger Richtung unbeschränkt etabliert werden können (siehe Teilziel 3.14).

Applikationen sind auch bei Odberg an eine Schemaversion gebunden, die den Kontext für Objektzugriffe definiert (von Odberg *schema version context* genannt). Diese Schemaversion kann zwar geändert werden; dies erfordert jedoch eine entsprechende Anpassung und Recompilierung der Applikation.

Die von Odberg verwaltete Schemaableitungsstruktur ist ein Baum, d.h. eine neue Schemaversion kann von jeder existierenden Schemaversion abgeleitet werden. Dazu sollte der Schemaentwickler diejenige Schemaversion als Ausgangspunkt auswählen, die die größte Ähnlichkeit zur gewünschten Zielschemaversion besitzt, so daß möglichst wenige Änderungen zur Erreichung der gewünschten Schemaversion notwendig werden. Die Ableitung realisiert soweit nur eine Möglichkeit zur Etablierung einer Wiederbenutzungs-Beziehung (engl. *reuse relationship*). Auch Odberg präferiert die in Teilziel 3.1 geforderte, relative Spezifikation neuer Schemaversionen im Vergleich zu existierenden entsprechend dem internen Ansatz zur Schemaevolution gegenüber dem externen Ansatz (von Odberg *copy-edit-commit* genannt). Denn nur damit sind semantische Abhängigkeiten zwischen verschiedenen Schemaversionen sicher und vollständig erkennbar. Spezielle Fähigkeiten des DBMS sind notwendig, um diese Abhängigkeiten spezifizieren und verwalten zu können.

Im Gegensatz zu der in Teilziel 3.2 vorgeschlagenen, integrierten Schemabeschreibungs- und -änderungssprache führt Odberg zwei getrennte Sprachen ein, die er als *data definition language (DDL)* und als *change specification language (CSL)* bezeichnet. Die Ableitung neuer Schemaversionen geschieht mittels der CSL, die Operationen zur Auswahl der Quellschemaversion und zur Durchführung notwendiger Änderungen (Hinzufügen, Löschen, Ändern und Verschieben von Klassen) beinhaltet. Ein semantischer Zusammenhang zwischen Objekten in verschiedenen Schemaversionen kann mit der CSL-Anweisung **mov** hergestellt werden. Dies geschieht jedoch anders als bei den Propagationsmechanismen der meisten anderen Ansätze auf Attributebene, was bezüglich der Konsistenz zwischen verschiedenen Attributen eines Objektes ähnliche Probleme wie bei den Update- und Backdate-Funktionen von Monk und Sommerville (siehe Abschnitt 4.5.3.3) verursachen kann. Mit **add class** kann solch ein Zusammenhang aber auch auf Klassenebene hergestellt werden.

Unter dem Begriff der Schemaversionskompatibilität (engl. *schema version compatibility*) werden alle Aspekte zusammengefaßt, die der Überwindung von Unterschieden und Inkompatibilitäten zwischen Komponenten verschiedener Versionen desselben Schemas dienen. Hierbei werden drei Dimensionen der Kompatibilität unterschieden, auf die wir hier etwas näher eingehen werden, nämlich die Kompatibilität der Klassifikation, der Schnittstelle und der Repräsentation.

Die erste Dimension betrifft die Kompatibilität der Klassifikation. Nach Odberg abstrahiert das Konzept der Klassen von zwei streng zu unterscheidenden Aspekten, nämlich von dem der Klassenzugehörigkeit und von dem der Klasseneigenschaften. Objekte klassifizieren sich entsprechend gewisser Gemeinsamkeiten als zu der Extension einer gemeinsamen Klasse gehörig. Dies hat jedoch nur implizit mit dem zweiten Aspekt zu tun, nach dem Objekte derselben Klasse gemeinsame Eigenschaften haben und sich ähnlich verhalten. Die Zuordnung eines Objektes zu einer oder mehreren Klassen, d.h. die Festlegung derjenigen Klassenextensionen, in denen das Objekt vertreten ist, wird als *Klassifikation* bezeichnet und die hier beschriebene Dimension beschäftigt sich ausschließlich mit diesem Aspekt. Wir betrachten im folgenden ein Objekt o, das in einer Klasse c einer Schemaversion sv_x angelegt wurde. Im Rahmen der Schemaversionierung kann das Objekt o in verschiedenen Schemaversionen $sv_y \neq sv_x$ sichtbar sein und demzufolge muß seine Klassifikation im Gegensatz zu unversionierten Systemen über Schemaversionsgrenzen hinweg definiert werden. Nach Odberg existieren fest vorgegebene, *implizite Klassifikationsbeziehungen*, die vom Schemaentwickler um *explizite Klassifikationsbeziehungen* ergänzt werden können. Innerhalb der erzeugenden Schemaversion sv_x ist das Objekt o, wie in unversionierten Systemen üblich, in den Extensionen der Klasse c und denen ihrer Oberklassen enthalten. Diese werden von Odberg als *explizite Klassen* (engl. *explicit classes*) des Objektes bezeichnet. Damit ist ein Objekt explizit in allen Versionen seiner expliziten Klassen enthalten. Weiterhin ist o in einer Schemaversion $sv_y \neq sv_x$ aufgrund impliziter Klassifikationsbeziehungen auch in allen Oberklassen seiner expliziten Klassen entsprechend der Vererbungshierarchie von sv_y enthalten, auch wenn o in sv_x nicht in diesen Klassen enthalten ist oder wenn diese Klassen gar nicht zu sv_x gehören.

Ausgehend von den expliziten und impliziten Klassen eines Objektes kann die Menge derjenigen Klassenversionen, in denen das Objekt tatsächlich sichtbar ist, durch die Angabe expliziter Klassifikationsbeziehungen modifiziert werden. Damit kann spezifiziert werden, daß ein Objekt in einer Schemaversion sv_y in bestimmten Klassen enthalten sein soll (engl. *propagation relationships*), in denen es allein durch die impliziten Klassifikationsbeziehungen nicht enthalten wäre oder daß es in bestimmten Klassen nicht enthalten sein soll (engl. *inhibition relationships*) in denen es durch die impliziten Klassifikationsbeziehungen enthalten wäre. Dieser explizite Einschluß in oder Ausschluß aus Klassenextensionen von sv_y kann auch vom Zustand des Objektes abhängig spezifiziert werden (engl. *conditional classification relationships*), ähnlich einer Selektion. Die Klassifikationsbeziehungen können sowohl vorwärts (d.h. von älteren zu neueren Schemaversionen) als auch rückwärts angegeben werden und ihre Anwendung pflanzt sich auch transitiv fort.

Das läßt sich bei Odberg relativ einfach realisieren, weil dort jedes Objekt nur in einer Version vorliegt und diese einfach nur verschiedenen Klassenextensionen (also einfachen Mengen) zugeordnet sind. Über die Struktur eines Objektes wird dadurch keine Aussage gemacht und es wird hierfür keine Konvertierungsfunktion angegeben bzw. benötigt. Nachteil ist, daß sehr viele, möglicherweise nicht vorhergesehene „Konvertierungen“ (das sind hier eigentlich nur andere Interpretationen einer einzigen Objektversion) durchgeführt werden mit eventuell nicht gewollten Effekten und daß sich verschiedene dieser Beziehungen widersprechen können und eine sehr komplizierte (auch für den Schemaentwickler schwer zu durchschauende) Konfliktauflösung notwendig wird, die zur Laufzeit merkwürdige Effekte produzieren könnte, über die zumindest in [Odb94a] jedoch nichts ausgesagt wird.

Die zweite Dimension betrifft die Kompatibilität der Schnittstelle. Während die Kompatibilität der Klassifikation den ersten Aspekt einer Klasse behandelte, nämlich den der Zugehörigkeit von Objekten zu Klassenextensionen, betrifft die Kompatibilität der Schnittstelle den zweiten Aspekt einer Klasse, nämlich den der Klasseneigenschaften. Applikationen verschiedener Schemaversionen haben verschiedene Anforderungen an den Typ eines zugegriffenen Objektes. Die Kompatibilität der Schnittstelle muß gewährleisten, daß sich ein Objekt als Instanz von Klassen,

denen es in allen Schemaversionen angehört, konsistent verhält. Dies ist eine notwendige Voraussetzung, um statisch garantieren zu können, daß ein Objekt alle Anforderungen als Instanz einer bestimmten Klasse beantworten kann. Unter diesem Gesichtspunkt ist Typsicherheit auch über Schemaänderungen hinweg gegeben. Dies wird durch Verwendung des in Abschnitt 4.5.3.6.3 vorgestellten Rollenkonzeptes und durch spezielle *Klassenversionsrollen* (engl. *class version roles*) erreicht.

Die dritte Dimension betrifft die Kompatibilität der Repräsentation. Sie beschäftigt sich damit, wie ein Objekt alle Informationen speichern und verwalten kann, die es als Instanz von Klassen verschiedener Schemaversionen haben muß. Die Kompatibilität der Repräsentation steht in engem Zusammenhang mit den Attributen als informationsspeichernden Elementen und muß sicherstellen, daß deren Werte entsprechend aller impliziter und expliziter Abhängigkeiten zwischen verschiedenen Schemaversionen verwaltet werden. Im Gegensatz zu den repräsentierenden Attributen werden Methoden als Interpretation von Daten aufgefaßt.

Die Versionen, in denen eine Klasse in verschiedenen Schemaversionen vorliegt, sind Ausprägungen derselben Abstraktion von Eigenschaften der Diskurswelt. Demzufolge wird erwartet, daß sich die durch verschiedene Versionen einer Klasse vorgegebenen Repräsentationen überschneiden und sich somit eine Klassifikation von Attributen entsprechend Clamen (siehe Abschnitt 4.5.3.4) in gemeinsame, abgeleitete, abhängige und unabhängige Attribute vornehmen läßt. Weiterhin sind Repräsentationsabhängigkeiten nicht nur zwischen Versionen derselben Klasse sondern auch zwischen verschiedenen Klassen möglich. Das fundamentale Problem ist dabei die Fähigkeit gemeinsame Daten für Applikationen verschiedener Schemaversionen bereitszustellen (Teilziel 3.10), wobei Anforderungen späterer Schemaversionen bei der Erzeugung eines Objektes noch nicht berücksichtigt werden können. Daraus ergibt sich die Notwendigkeit zu späteren Anpassungen in der Datenbank vorhandener Objekte.

Die Repräsentation eines Objektes ist durch die Menge seiner Attribute gegeben und kann sich aus mehreren Rollen zusammen setzen. Jede dieser Rollen kann von mehreren Schemaversionen gemeinsam genutzt werden (engl. *shared representation, SR*).

Die Konvertierung zwischen verschiedenen Repräsentationen geschieht mittels sehr flexibler Konvertierungsfunktionen (engl. *mutation propagation functions, MPF*). Diese können beliebige Gruppen von Attributen (eventuell sogar aus verschiedenen Klassen, jedoch nicht aus verschiedenen Objekten) sowohl vorwärts als auch rückwärts propagieren. Nicht ableitbare Attribute werden dabei mit dem Wert **nil** besetzt. Möglicherweise auftretende, mehrdeutige Konvertierungspfade werden nicht behandelt oder beschrieben.

Die von Odberg verwendete Schemabeschreibungssprache (engl. *data definition language, DDL*) und seine Schemaänderungssprache (engl. *change specification language, CSL*) sind sich so ähnlich, daß man vorteilhaft auf diese Trennung verzichtet hätte (Teilziel 3.2).

Einen weiteren Nachteil stellt Odbergs Einschränkung der Ableitungsbeziehung auf einen Baum dar. Diese muß er nachträglich durch die Einführung des Gleichheitsoperators seiner CSL ausgleichen, der die Spezifikation semantischer Querbeziehungen zwischen beliebigen Schemaversionen erlaubt. Diese Querbeziehungen sind allerdings aus dem Ableitungsbaum auch nicht ersichtlich.

4.5.3.7 Der Schemaversionierungsansatz von Kim und Chou

ORION [BCG+87, BKK88, KWG+87, KBG+88, KBC+89, KGBW90, KBGW91] ist ein objektorientiertes Datenbankmanagementsystem, das im Rahmen des Advanced Computer Architecture Programmes im Object-Oriented and Distributed Systems Laboratory der Microelectronics and Computer Technology Corporation (MCC) seit 1985 entwickelt wurde. ORION wurde in Common LISP implementiert und auf den Einsatz in den sog. fortschrittli-

chen Anwendungsgebieten für Datenbanksysteme ausgerichtet. Dazu bietet ORION Unterstützung insbesondere für nebenläufige Transaktionen [KLMP84, GK88], komplexe Objekte [KCB87, KBC+87, KBG89], Objektversionierung [CK86, CK88], Schemaevolution und Schemaversionierung [BKKK87, KC88] und für Multimedia-Objekte [WKL86, WK87]. Heute wird ORION unter dem Namen ITASCA [IBE95] von IBEX Corporation weiterentwickelt und kommerziell vertrieben.[46]

Das Objektmodell von ORION [BCG+87] beinhaltet die elementaren Eigenschaften objektorientierter Systeme, wie Objekte mit Attributen (engl. *instance variables*) und Methoden, Nachrichten, die zur Kommunikation zwischen Objekten dienen und die Ausführung entsprechender Methoden auslösen, sowie Klassen und Vererbung (**is_a**-Beziehung). In ORION kann eine Klasse mehrere direkte Oberklassen besitzen (Mehrfachvererbung). Treten dabei Vererbungskonflikte auf, so wird die speziellste der in Konflikt stehenden Eigenschaften vererbt, falls eine solche existiert. Anderenfalls wird die Eigenschaft von der ersten direkten Oberklasse geerbt, die die betreffende Eigenschaft aufweist. Dazu wird in jeder Klasse der Menge ihrer direkten Oberklassen eine benutzerdefinierbare und -veränderbare Ordnung aufgeprägt. Neben den genannten, in objektorientierten Systemen üblicherweise anzutreffenden Konzepten unterstützt ORION noch weitere. Dazu zählen die Möglichkeiten Defaultwerte für Attribute (engl. *default-valued variables*) vorzugeben und *Klassenattribute* (engl. *shared-value variables*) anzulegen. Der Wert eines Klassenattributes stimmt bei allen Objekten einer Klasse überein, z.B. fliegen alle Objekte der Klasse `Flugzeug` durch das `Medium` Luft, so daß man `Medium` als Klassenattribut von `Flugzeug` deklarieren könnte. Weiterhin können komplexe Objekte beschrieben werden, wobei nach [KBG89] zwischen abhängigen (engl. *dependent*) und unabhängigen (engl. *independent*) sowie zwischen exklusiv (engl. *exclusive*) und nicht-exklusiv (engl. *shared*) zugeordneten Komponentenobjekten unterschieden werden kann. Die älteren ORION-Veröffentlichungen besprechen allerdings nur die Variante existentiell abhängiger, exklusiv zugeordneter Komponentenobjekte.

Für die Beschreibung von Schemaevolutions- und Schemaversionierungsansatz in ORION wird eine Menge von *Invarianten* vorgegeben. Diese beschreiben Bedingungen, die ein konsistentes Schema erfüllen muß. Durch die Ausführung eines der bereits in Abschnitt 4.3.1.1 (siehe Seite 61) vorgestellten Schemaänderungsprimitive werden möglicherweise eine oder mehrere der Invarianten verletzt und es bestehen i.Allg. verschiedene Möglichkeiten, die Konsistenz des Schemas wiederherzustellen. Daher wird in [BKKK87] neben den Invarianten eine Menge von Regeln angegeben, die festlegen, auf welche Art und Weise ggf. verletzte Invarianten wieder sichergestellt werden. Im folgenden werden wir diese Invarianten und Regeln näher vorstellen.

4.5.3.7.1 Die Invarianten der Schemaevolution von ORION

Wir führen hier zunächst die in [BKKK87] angegebenen Invarianten auf, die ein Schema dem ORION-Objektmodell entsprechend zu jeder Zeit erfüllen muß.

Invariante 4.1 {**Klassenvererbungsgraph** (engl. ***Class Lattice Invariant***)}
Die Klassenvererbungsstruktur wird durch einen zusammenhängenden, gerichteten, azyklischen Graphen mit eindeutig benannten Knoten und numerierten Kanten dargestellt. Die Nummerierung aller an einem Knoten eingehenden Kanten ist eindeutig. Der Klassenvererbungsgraph hat genau eine Wurzel, nämlich die systemdefinierte Klasse **Object**.

[46] Wenn man ein Schema als komplexes Objekt betrachtet, so kann Itasca mittels seiner Unterstützung für Objektversionierung auch verschiedene Versionen des Schemas verwalten. Dabei ausgeführte Schemaänderungen haben jedoch keinerlei Einfluß auf die Instanzenebene, d.h. auf die persistenten Datenbankobjekte. Ein Schemaversionierungsmechanismus mit Objektpropagation so wie er in dieser Arbeit oder in [KC88] vorgestellt wird, steht damit in dem kommerziellen System nicht zur Verfügung [Heu97, IBE95].

Invariante 4.2 {**eindeutige Namen** (engl. ***Distinct Name Invariant***)}
Alle (geerbten oder lokal definierten) Attribute einer Klasse haben eindeutige Namen. Entsprechendes gilt für alle Methoden einer Klasse.

Invariante 4.3 {**eindeutige Herkunft** (engl. ***Distinct Identity (Origin) Invariant***)}
Alle Attribute und Methoden einer Klasse haben eine eindeutige Herkunft (Identität).

Invariante 4.4 {**vollständige Vererbung** (engl. ***Full Inheritance Invariant***)}
Eine Klasse erbt sämtliche Attribute und Methoden all ihrer Oberklassen, es sei denn Invariante 4.2 oder 4.3 würden dadurch verletzt. Das heißt, von zwei gleichnamigen Attributen verschiedener Herkunft in verschiedenen direkten Oberklassen muß mindestens eines geerbt werden. Wenn jedoch zwei Attribute verschiedener direkter Oberklassen derselben Herkunft sind, so muß nur eines davon geerbt werden.

Invariante 4.5 {**Typkompatibilität**[47] (engl. ***Domain Compatibility Invariant***)}
Wenn ein Attribut a' einer Klasse c' von einem Attribut a einer Oberklasse c von c' geerbt wird, dann muß der Wertebereich von a' derselbe sein wie der von a oder eine Unterklasse davon. Desweiteren muß der (Default-) Wert eines (Klassen-) Attributes dem Typ dieses Attributes entstammen.

4.5.3.7.2 Die Regeln der Schemaevolution von ORION

Die Regeln beschreiben, wie ORION verfährt, wenn während der Durchführung eines Schemaänderungsprimitives temporäre Inkonsistenzen auftreten. Die Gültigkeit der Invarianten wird nach Möglichkeit durch korrigierende Maßnahmen wieder hergestellt. Ist dies nicht möglich, so muß die Durchführung der Schemaänderung abgelehnt werden.

Die folgenden drei Regeln zur Auflösung von Vererbungskonflikten (engl. *Default Conflict Resolution Rules*) dienen der deterministischen Auflösung von Namens- und Identitätskonflikten und stellen damit die Invarianten 4.2 und 4.3 sicher.

Regel 4.1
Wenn ein in einer Klasse c definiertes Attribut denselben Namen wie ein Attribut einer Oberklasse von c hat, so wird das lokal definierte Attribut verwendet. Entsprechendes gilt für Methoden.

Regel 4.2
Wenn zwei oder mehr Oberklassen einer Klasse c gleichnamige Attribute verschiedener Herkunft besitzen, so wird das Attribut der ersten (entsprechend der Nummerierung der eingehenden Kanten des Knotens c im Klassenvererbungsgraph) der miteinander in Konflikt stehenden Oberklassen geerbt. Entsprechendes gilt für Methoden.

Regel 4.3
Wenn zwei oder mehr Oberklassen einer Klasse c Attribute derselben Herkunft besitzen, so wird das Attribut mit dem speziellsten Typ geerbt, falls ein solches existiert. Ansonsten wird das Attribut der ersten der miteinander in Konflikt stehenden Oberklassen geerbt.

Eine zweite Gruppe von Regeln betrifft die Änderung und Weitergabe geerbter Eigenschaften (engl. *Property Propagation and Change Rules*).

[47] Den in der Originalliteratur zu ORION verwendeten, englischsprachigen Begriff *domain* haben wir in dieser Arbeit stets mit *Typ* übersetzt, da unsere Verwendung des Typbegriffs dem gemeinten sehr nahe kommt. Den Begriff der Domäne benutzen wir synonym zu *Wertebereich*.

Regel 4.4
Wenn ein Merkmal (z.B. der Defaultwert) eines Attributes oder einer Methode in einer Klasse c verändert wird, so wird diese Veränderung an jede Unterklasse von c weitergegeben, die die veränderte Eigenschaft zuvor von c erbte, es sei denn diese Unterklasse hat das betreffende Merkmal der Eigenschaft selbst redefiniert.

Regel 4.5
Eine neu hinzugefügte oder umbenannte Eigenschaft wird nur an diejenigen Unterklassen weitergegeben, in denen dadurch keine weiteren Konflikte entstehen. Eine Unterklasse, an die eine hinzugefügte oder umbenannte Eigenschaft nicht weitergegeben wird, gibt die Veränderung auch nicht an ihre eigenen Unterklassen weiter. Im Konfliktfall hat Regel 4.5 Priorität gegenüber Regel 4.2.

Regel 4.6 {**Veränderung von Typen** (engl. ***Domain Change Rule***)}
Der Typ eines Attributes kann nur generalisiert werden. Desweiteren kann der Typ eines geerbten Attributes nicht weiter generalisiert werden als der Typ des Originalattributes.

Die folgenden vier Regeln zu Veränderungen des Klassenvererbungsgraphen (engl. *DAG Manipulation Rules*) stellen sicher, daß sich beim Hinzufügen oder Entfernen von Kanten oder Knoten im Klassenvererbungsgraphen keine drastischen Veränderungen ergeben.

Regel 4.7 {**Hinzufügen einer Vererbungskante** (engl. ***Edge Addition Rule***)}
Wird eine Klasse c Oberklasse einer Klasse c', so wird sie als letzte Klasse in die (aufgrund der Nummerierung der eingehenden Vererbungskanten) geordnete Menge der Oberklassen von c' aufgenommen. Daher können möglicherweise entstehende Namenskonflikte ignoriert werden. Konflikte bezüglich der Herkunft von Attributen müssen jedoch mit Regel 4.3 aufgelöst werden.

Zur Sicherstellung eines zusammenhängenden Klassenvererbungsgraphen (Invariante 4.1) wird die folgende Regel benötigt.

Regel 4.8 {**Entfernen einer Vererbungskante** (engl. ***Edge Removal Rule***)}
Wenn die Klasse c die einzige direkte Oberklasse einer Klasse c' ist und die Vererbungskante von c zu c' soll entfernt werden, so werden alle direkten Oberklassen von c zu direkten Oberklassen von c' und zwar in derselben Ordnung wie in der Klasse c.

Eine Konsequenz aus Regel 4.8 lautet, daß eine Vererbungskante zur Wurzelklasse **Object** des Klassenvererbungsgraphen nicht entfernt werden kann.

Regel 4.9 {**Hinzufügen eines Knotens** (engl. ***Node Addition Rule***)}
Wenn bei der Erzeugung einer neuen Klasse c keine direkte Oberklasse angegeben wird, so wird die Klasse **Object** *(einzige) Oberklasse von c.*

Das Löschen einer Klasse geschieht in drei Schritten: Zuerst werden alle Vererbungskanten zu Unterklassen entfernt, dann die zu Oberklassen und zuletzt wird die Klasse selbst gelöscht. Die folgende Regel dient der Sicherstellung der Invariante 4.1.

Regel 4.10 {**Entfernen eines Knotens** (engl. ***Node Removal Rule***)}
Wenn die zu löschende Klasse die einzige Oberklasse einer Klasse c ist, dann wird Regel 4.8 auf die Klasse c angewendet. Systemdefinierte Klassen wie **Object** *können nicht gelöscht werden.*

Schließlich werden Regeln zur Behandlung komplexer Objekte (engl. *Composite Object Rules*) festgelegt.

Regel 4.11 {**existentielle (has_a) Referenz** (engl. ***Composite Link Rule***)}
Die Eigenschaft, daß ein durch ein bestimmtes Attribut referenziertes Objekt von dem referenzierenden Objekt existentiell abhängt (engl. composite link property)*, kann nachträglich aus- aber nicht eingeschaltet werden.*

Diese Einschränkung liegt darin begründet, daß beim Einschalten die Einhaltung der Eigenschaft bei allen in der Datenbank befindlichen Objekten nachgeprüft werden müßte.

Regel 4.12
Wenn die Eigenschaft, daß ein durch ein bestimmtes Attribut referenziertes Objekt von dem referenzierenden Objekt existentiell abhängt (engl. composite link property)*, nachträglich ausgeschaltet wird, dann bleibt die Referenz zwar bestehen, das referenzierte Objekt wird bei späterem Löschen des referenzierenden Objektes allerdings nicht mehr automatisch entfernt.*

4.5.3.7.3 Die Regeln der Schemaversionierung von ORION

In [KC88] schlagen Won Kim und Hong-Tai Chou einen Schemaversionierungsmechanismus für ORION vor. Die Schemaableitungsstruktur ist ein Baum, der zwischen *transienten Schemaversionen* und *Arbeitsschemaversionen* (engl. *working schema versions*) unterscheidet. Jeder Zugriffsbereich (engl. *access scope*) besteht aus einem *direkten* und einem *geerbten* Teil. Die folgenden Regeln beschreiben ergänzend zu den bereits genannten die Propagation von Instanzen von ihrer *Erzeugerschemaversion* (engl. *creator schema version*) in deren Nachfolgerschemaversionen (engl. *descendant schema versions*) und den Zugriff von Applikationen auf diese Objekte. Weiterhin werden Datenstrukturen und Algorithmen für eine verzögerte Propagation vorgeschlagen.

Regel 4.13 {**Zustand einer Schemaversion** (engl. ***Schema-Version Capability Rule***)}
Eine Schemaversion ist eine transiente Schemaversion *oder eine* Arbeitsschemaversion. *Eine transiente Schemaversion kann geändert, gelöscht oder zu einer Arbeitsschemaversion hochgestuft werden. Eine Arbeitsschemaversion kann nicht verändert werden; sie kann jedoch gelöscht oder zu einer transienten Schemaversion zurückgestuft werden, falls sie keine Nachfolgerschemaversionen besitzt.*

Regel 4.14 {**Ableitung einer Schemaversion** (engl. ***Schema-Version Derivation Rule***)}
Von jeder existierenden Schemaversion können jederzeit beliebig viele neue Schemaversionen abgeleitet werden, wodurch ein Schemaversionsableitungsbaum (engl. schema-version derivation hierarchy) *entsteht. Neu abgeleitete Schemaversionen sind zunächst transient. Wird eine neue Schemaversion von einer transienten Schemaversion abgeleitet, so wird letztere automatisch zu einer Arbeitsschemaversion hochgestuft.*

Regel 4.15 {**Löschen einer Schemaversion** (engl. ***Schema-Version Deletion Rule***)}
Eine Schemaversion kann genau dann gelöscht werden, wenn sie ein Blatt des Schemaversionsableitungsbaumes ist. Wenn eine Schemaversion gelöscht wird, so wird ihr direkter Zugriffsbereich ebenfalls gelöscht.

Regel 4.16 {**Vererbung von Zugriffsbereichen** (engl. ***Access-Scope Inheritance Rule***)}
Wurde eine Schemaversion sv' von sv abgeleitet, so erbt sv' den Zugriffsbereich der Schemaversion sv per Default. Der Schemaentwickler kann die Vererbung jedoch blockieren. Desweiteren wird der Zugriffsbereich einer Schemaversion sv per Default unveränderbar, sobald eine von sv abgeleitete Schemaversion den Zugriffsbereich erbt. Wiederum kann der Schemaentwickler die Standardregelung umgehen und den Zugriffsbereich von sv unter sv veränderbar belassen.

Regel 4.17 {**Zugriffsbereiche** (engl. ***Access-Scope Rule***)}
Der Zugriffsbereich (engl. access scope) *einer Schemaversion sv ist die Menge aller in sv sichtbaren, d.h. lesbaren oder veränderbaren Objekte und setzt sich genau aus den lokal definierten Objekten und den Objekten der von Vorgängerschemaversionen geerbten Zugriffsbereiche zusammen.*

Regel 4.18 {**Veränderung direkter Zugriffsbereiche** (engl. ***Direct-Access-Scope Update Rule***)}
Der Zugriffsbereich einer Schemaversion sv ist unveränderbar unter sv, falls eine Schemaversion sv′ von sv abgeleitet wurde, es sei denn die Vererbung des Zugriffsbereiches von sv wurde in sv′ blockiert oder sv′ hat den Zugriffsbereich von sv unter sv veränderbar belassen.

Regel 4.19 {**Veränderung geerbter Zugriffsbereiche** (engl. ***Inherited-Access-Scope Update Rule***)}
Alle geerbten Objekte des Zugriffsbereiches einer Schemaversion sv können verändert oder gelöscht werden. Solche Änderungen oder Löschungen sind jedoch nur in sv sichtbar und in Nachfolgerschemaversionen von sv, die dessen Zugriffsbereich erben.

Einschränkungen des Wertebereiches eines Attributes sind nicht möglich, da kein Mechanismus angeboten wird, mit dem man Attributwerte existierender Objekte „abschneiden“könnte. Erweiterungen des Wertebereiches sind möglich, da sie keinerlei Einfluß auf existierende Instanzen haben, und daher ohne Rücksicht auf existierende Objekte durchgeführt werden können.

Auf einige Schwachpunkte des Objekt- und Schemaevolutionsmodells von ORION gehen wir im folgenden näher ein.

- Eingeschränkte Substituierbarkeit bei Mehrfachvererbung

  Das Vererbungsmodell von ORION unterstützt Mehrfachvererbung und verfolgt zur Auflösung von Vererbungskonflikten eine zweistufige Strategie. Seien $c_1, \ldots, c_n$ direkte Oberklassen der betrachteten Klasse c, die jeweils ein Attribut namens a enthalten, wobei der Typ von $c_i.a$ ($1 \leq i \leq n$) mit T_i bezeichnet werde. Im ersten Schritt prüft ORION, ob es ein $T \in \{T_1, \ldots, T_n\}$ gibt, das einen Untertyp aller T_i ($i = 1, \ldots, n$) hat. In diesem Falle wird dieses (eindeutig bestimmte) T_i als Typ für $c.a$ verwandt. Existiert jedoch kein solches T_i, so übernimmt ORION den Typ aus der ersten[48] Oberklasse von c, die das Attribut a enthält, also T_1. Dabei ist es möglich, daß ein Objekt der Klasse c dann nicht mehr Objekte aller Oberklassen von c substituieren kann. Ein Beispiel für eine solche Situation ist in Abbildung 4.9a dargestellt. Geerbte Attribute wie a in c_4 werden in den Abbildungen in runden Klammern aufgeführt. In der Klasse c_4 entsteht ein Vererbungskonflikt, der nicht mit der ersten Stufe der ORION-Strategie aufgelöst werden kann, da weder c' einen Untertyp von c'' hat noch umgekehrt. Daher wird (entsprechend Regel 4.3) der zweiten Stufe der ORION-Strategie folgend der Typ c' des Attributes a in der ersten Oberklasse c_2 von c_4 verwandt. Nun kann ein Objekt der Klasse c_4 ein Objekt der Oberklasse c_3 allerdings nicht mehr substituieren.

  Zu vergleichbaren Situationen kann es durch Veränderung der Vererbungsbeziehung einzelner Eigenschaften (Primitive 1.1.5 und 1.2.5 aus Abschnitt 4.3.1.1) oder ganzer Klassen (Primitiv 2.3) kommen.

- Beliebige Veränderbarkeit von Attributtypen

  Nach Regel 4.6 über die Veränderung von Attributtypen ist bei direkten Schemaänderungen in ORION lediglich die Generalisierung eines Attributtyps zulässig. Bei indirekt

[48] Wie bereits erwähnt, verwaltet ORION eine Ordnung zwischen den Oberklassen einer Klasse.

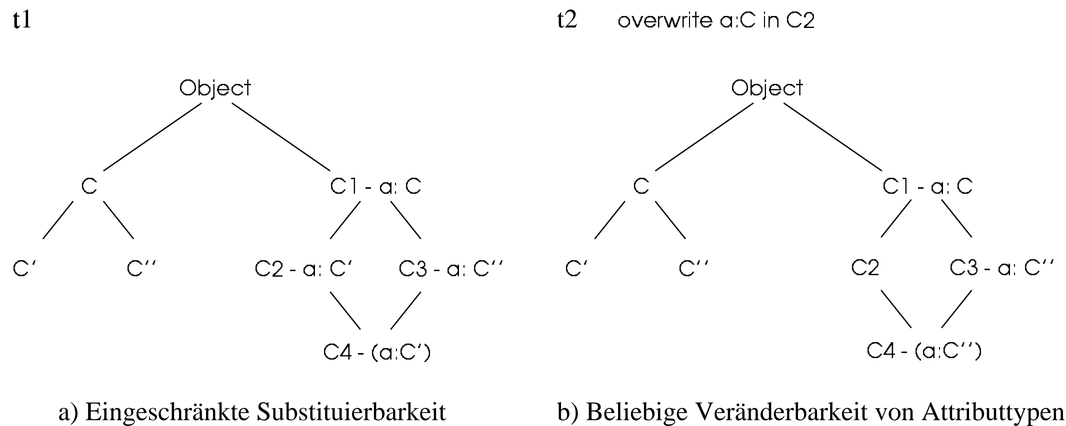


Abbildung 4.9: (a) Ein Objekt der Klasse c_4 kann ein Objekt der Oberklasse c_3 nicht substituieren, da c' keine Unterklasse von c'' ist. (b) Beliebige Veränderbarkeit von Attributtypen.

mitveränderten Attributen in Unterklassen ist dies jedoch nicht mehr gewährleistet. Wenn in der in Abbildung 4.9a dargestellten Situation zum Zeitpunkt t_2 die Überschreibung von Attribut a in der Klasse c_2 entfernt wird[49] (siehe Abbildung 4.9b), so ändert sich der Typ von a in Klasse c_4 von c' zu c'', obwohl c'' keine Generalisierung von c' darstellt.

- Schemabeschreibungssprache unter Schemaänderungsprimitiven nicht abgeschlossen

  Ein erheblicher Schwachpunkt des Schemaevolutionsmodells von ORION liegt in der Tatsache begründet, daß die Menge der durch die Schemabeschreibungssprache darstellbaren Schemata unter den definierten Schemaänderungsprimitiven nicht abgeschlossen[50] ist. Es können also durch Anwendung von Änderungsprimitiven Schemata entstehen, die sich nicht in der reinen Schemabeschreibungssprache, d.h. ohne Verwendung von Schemaänderungsprimitiven, ausdrücken lassen. Soll das zu einem bestimmten Zeitpunkt t_2 von ORION verwendete Schema dokumentiert werden, beispielsweise um einem Applikationsentwickler Auskunft über die in der Datenbank zur Verfügung stehenden Klassen und deren Eigenschaften zu geben, so genügt es i.Allg. nicht einen Ausdruck der Schemabeschreibungssprache anzugeben, so wie das in einem abgeschlossenen System möglich ist. Stattdessen muß der Zustand des Schemas zu einem früheren Zeitpunkt $t_1 < t_2$ zusammen mit einer Liste der zwischen t_1 und t_2 durchgeführten Schemaänderungen angegeben werden.

  Eine solch problematische Situation kann sich ergeben, wenn in einer Klasse c eine Eigenschaft (ein Attribut oder eine Methode) hinzugefügt oder umbenannt wird. Diese Änderung wird dann nämlich laut Regel 4.5 nur zu denjenigen Unterklassen von c propagiert, in denen dadurch keine Konflikte entstehen.

  Abbildung 4.10 zeigt zwei Beispiele. Die Attribute x, y und z werden in den Klassen c_1 und c_2 lokal definiert und in c_3 geerbt. Attribute, die eine Klasse (im Beispiel c_3) erbt, sind in runden Klammern dargestellt und mit der Nummer der Herkunftsklasse indiziert. In Abbildung 4.10a erbt c_3 also zunächst Attribut y von Klasse c_2. Nach der Hinzufügung eines gleichnamigen Attributes y zu c_1 würde die Spezifikation c_3 **superclasses** c_1, c_2 implizieren, daß Attribut y in c_3 von c_1 geerbt sei (in Abbildung 4.10a daher als y_1 dargestellt). In diesem Fall kann das zum Zeitpunkt t_2 aktuelle Schema jedoch durch Umsortieren der Oberklassen von c_3 noch korrekt angegeben werden (c_3 **superclasses** c_2, c_1). Dies ist in

[49] Dieses Entfernen der Überschreibung des Attributtyps von a in der Klasse c_2 stellt in ORION eine nach Regel 4.6 zulässige Verallgemeinerung dar.

[50] Hier ist der mathematische Begriff der Abgeschlossenheit gemeint, wie er in der Definition abelscher Gruppen verwendet wird.

der zweiten Situation (Abbildung 4.10b) nicht mehr möglich. Hier kann eine Beschreibung des Schemas zum Zeitpunkt t_2 nur dadurch erfolgen, daß das Schema zum Zeitpunkt t_1 zusammen mit der inzwischen durchgeführten Schemaänderung angegeben wird. Hierbei ist zu beachten, daß die Angabe des Schemas zum Zeitpunkt t_2 auch durch Hinzufügung einer lokalen Redefinition von Attributtypen in c_3 i.Allg. nicht möglich ist. Das ist nämlich nur dann erlaubt, wenn c' Unterklasse von c ist. Auch ein Verzicht auf die Vererbungsbeziehung und eine vollständige Definition aller Attribute von c_3 in c_3 selbst ist nicht möglich, da dann die **is_a**-Beziehung zwischen Objekten von c_3 einerseits und c_1 und c_2 andererseits nicht mehr gilt.

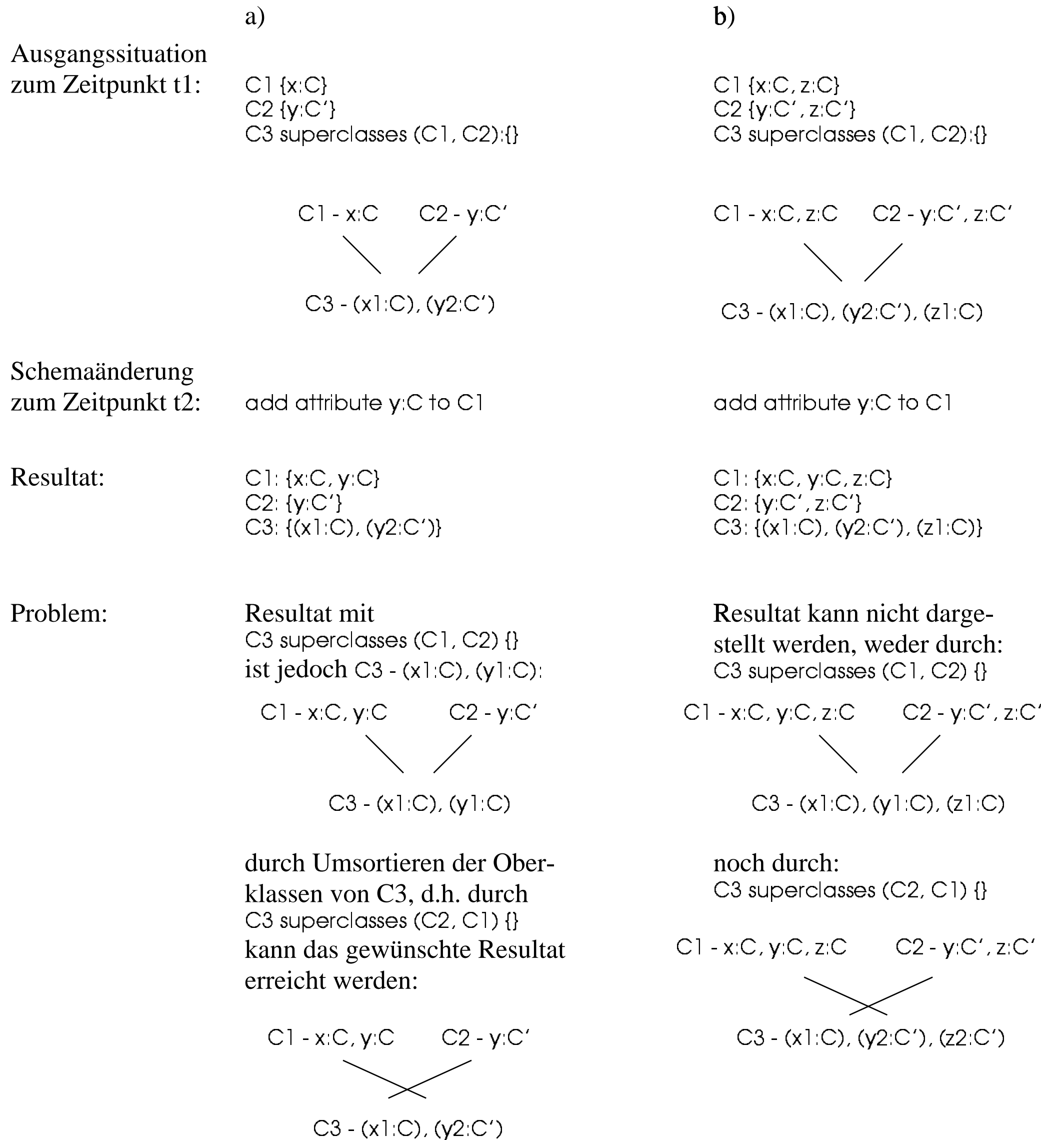


Abbildung 4.10: In Fall (a) kann das nach der Hinzufügung von Attribut y in C_1 resultierende Schema allein durch eine erzeugende Schemabeschreibungssprache dargestellt werden; bei Fall (b) ist das nicht möglich.

In dem in Abbildung 4.10b dargestellten Beispiel kommt es weiterhin zu dem bereits besprochenen Problem, daß Instanzen von c_3 Instanzen von c_1 i.Allg. nicht substituieren können. Das ist nur noch dann der Fall, wenn $c_2.y$ einem Untertyp von $c_1.y$ entstammt.

In [BKKK87] wird die Vollständigkeit und Korrektheit der Schemaänderungsprimitive von ORION anhand sog. *Eigenschaftsvererbungsgraphen* (engl. *property inheritance graph, PIG*) nachgewiesen. Die oben aufgeführten Schwachpunkte treten jedoch trotzdem auf, weil die PIGs wichtige Merkmale von Klassenvererbungsgraphen gar nicht abbilden und demzufolge auch keine vollständige Untersuchung zulassen. Zum einen abstrahieren die PIGs sowohl von den Identitäten als auch von den Typen der Eigenschaften (Attribute und Methoden). Damit werden diesbezügliche Widersprüche verdeckt. Zum anderen können Vererbungsbeziehungen in den PIGs nur klassenweise spezifiziert werden, obwohl die Schemaänderungsprimitive 1.1.5 und 1.2.5 Änderungen der Vererbungsbeziehungen auch für einzelne Eigenschaften erlauben.

Die bereits in Abschnitt 4.3.1.1 aufgeführte Taxonomie war die erste veröffentlichte Aufstellung dieser Art und erfüllt unsere diesbezüglichen Anforderungen (Teilziel 3.1) voll. Das hier vorgestellte Konzept der Schemaversionierung aus [KC88] erlaubt insbesondere die Ableitung alternativer Schemaversionen (Teilziel 3.6). Die Integration von Komponenten verschiedener Schemaversionen (Teilziel 3.7) wird jedoch weder auf der Schema- noch auf der Instanzenebene untersucht.

Die Kooperation (Teilziel 3.10) zwischen Applikationen verschiedener Schemaversionen ist etwas eingeschränkt. Attributwerte des indirekten Zugriffsbereiches können zwar modifiziert werden; solche Änderungen werden jedoch stets durch spätere Änderungen desselben Objektes in Vorgängerschemaversionen überschrieben. Dies kann allerdings nur dann passieren, wenn der Zugriffsbereich der Vorgängerschemaversion veränderbar belassen wurde.

Der Schemaversionierungsmechanismus von ORION scheint davon auszugehen, daß alle Applikationen früher oder später an eine neue Schemaversion angepaßt werden, denn es wird nur eine unvollständige Propagation (Teilziel 3.11) in Vorwärtsrichtung (d.h. von alten zu neuen Schemaversionen) unterstützt.

Die Flexibilität bei der Steuerung der Propagation (Teilziel 3.12) wird rudimentär dadurch erreicht, daß die Propagation von Objekten unterbunden und die Durchführbarkeit von Objektänderungen blockiert werden kann. Dabei kann jedoch nur entweder der gesamte Zugriffsbereich einer direkten Vorgängerschemaversion (engl. *parent schema version*) propagiert werden, oder gar nichts.

Es wird kurz die Idee erwähnt, die Propagation von Instanzen auf eine zusammenhängende Folge von Schemaversionen zu beschränken. Dabei zu erwartende Inkonsistenzen zwischen Objekten werden jedoch nicht untersucht.

4.5.4 Integration von Schemata

Insbesondere bei der Verschmelzung organisatorischer Einheiten entsteht oft die Aufgabe, bestehende Informationssysteme zu integrieren. Diese können demselben oder verschiedenen Datenmodellen entsprechen und demzufolge werden Arbeiten in sehr verschiedenen Bereichen der Integration durchgeführt.[51]

Da sich die von den zu integrierenden Schemata modellierten Diskurswelten normalerweise überlappen, sind vor der Integration zunächst auch Änderungen an den einzelnen Schemata erforderlich. Neben diesem Zusammenhang zwischen den Bereichen der Schemaevolution und der Schemaintegration besteht jedoch noch eine weitergehende Verwandtschaft, wenn der Ansatz der Schemaversionierung zur Durchführung von Schemaänderungen verwendet wird. Die zu integrierenden Schemata können dann nämlich als Versionen desselben Schemas aufgefaßt werden. Diese wurden dem allgemeinen Problem der Schemaintegration [Bre90, TS93b, TS93a] folgend komplett unabhängig voneinander erstellt. Alternative Versionen desselben Schemas sind zwar

[51] Breitbart [Bre90] gibt einen Überblick über Konzepte zur Integration von Datenbanken.

von einer gemeinsamen Vorgängerschemaversion abgeleitet, über die semantischen Ähnlichkeiten zu dieser werden allerdings keinerlei Voraussetzungen gemacht. Damit besteht ein zweiter Zusammenhang zwischen der Integration von Schemata (desselben zugrunde liegenden Datenmodells) und der Integration von Versionen desselben Schemas entsprechend dem technischen Teilziel 3.7.

Ähnlich wie bei der externen Durchführung von Schemaänderungen (siehe Teilziel 3.4) besteht bei der Integration unabhängiger, aber potentiell sich überlappender Schemata die Notwendigkeit, semantische Beziehungen auf den Ebenen verschiedener Schemakomponenten zu erkennen und ggf. durch den Schemaentwickler verifizieren zu lassen. Dazu können insbesondere Namens- und Typvergleiche sowie Prüfungen möglicher Isomorphismen zwischen (Teilen verschiedener) Vererbungsstrukturen eingesetzt werden.

Bei der Vorstellung unseres Ansatzes der Schemaversionierung werden wir den Vergleich zwischen Schemaintegration und Schemaevolution durch Versionierung konkreter fortführen und auf bestehende Unterschiede hinweisen.

4.6 Zusammenfassung und Bewertung

Die im vorangegangenen Kapitel beschriebenen technischen Teilziele stellen nur relativ unpräzise formulierte Rahmenbedingungen dar, denen ein Konzept zur Unterstützung der Schemaevolution ungefähr entsprechen sollte. Demzufolge gestaltet sich die Messung und Bewertung von Systemen oder Konzepten der Literatur mit unseren Maßstäben als schwierig und stellenweise sogar ungerecht. Wir sind in diesem Kapitel jedoch auf zahlreiche Systeme näher eingegangen und haben die von ihnen eingesetzten Mechanismen unter Berücksichtigung ihrer eigenen Zielsetzungen vorgestellt. Trotz der genannten Einschränkungen einer tabellarischen Bewertung individueller Systeme versuchen wir abschließend einen kurzen Überblick über die Ergebnisse unserer Analysen zu geben (siehe Abbildung 4.11). Wir möchten nochmals betonen, daß die teilweise recht deutlich von unseren Anforderungen abweichenden Ziele der vorgestellten Systeme in der tabellarischen Darstellung keinerlei Berücksichtigung finden konnten.

In Abbildung 4.11 verwenden wir fünf verschiedene Grade der Erfüllung eines technischen Teilzieles. Diese reichen von *vollständig erfüllt* bis zu *nicht erfüllt* und werden durch die Symbole ++, +, +/−, − und − − dargestellt. Die technischen Teilziele sind jedoch nicht vollkommen orthogonal. Beispielsweise wird der Spezifikationsaufwand (Teilziel 3.16) für ein System, das keine Kooperation (Teilziel 3.10) erlaubt, deutlich geringer ausfallen, ohne daß dies als besonders positive Eigenschaft zu bewerten wäre. Wir setzen Bewertungen in der Tabelle daher in runde Klammern, wenn sie nur aufgrund der unvollständigen Erfüllung anderer Teilziele so gut ausfallen. Von einzelnen Ansätzen nicht berücksichtigte Aspekte bleiben unbewertet.

Grundsätzliche Vorgehensweisen (in der Tabelle der Kürze wegen *Prinzipien* genannt) werden jeweils so bewertet, wie sie in den obigen Abschnitten vorgestellt wurden. Es kann daher durchaus vorkommen, daß ein konkreter Vertreter eines Prinzips durch Hinzunahme spezieller Erweiterungen mehr technische Teilziele erfüllt, als das verwendete Prinzip allein.

Zusammenfassend läßt sich feststellen, daß wir die in der Literatur beschriebenen Ansätze vier grundsätzlichen Vorgehensweisen zuordnen konnten. Die damit erreichte Kategorisierung verdeutlicht Gemeinsamkeiten und Unterschiede bezüglich der grundlegenden Konzepte und gestattet gleichzeitig eine Gruppierung in Bezug auf die Erreichung unserer technischen Teilziele. Die manuelle Durchführung von Schemaänderungen ohne Unterstützung des Datenbanksystems gestaltet sich außerordentlich schwierig und aufwendig. Da kommerziellen Systemen die Schemaevolution unterstützende Mechanismen weitestgehend fehlen, muß in der Praxis häufig auf die Durchführung gewünschter Schemaänderungen verzichtet werden. Erwartungsgemäß weisen

Nr.	technisches Teilziel	Prinzip manuell	Prinzip direkt	Zicari (O2)	Lieberherr (Demeter)	Lerner, Habermann (Tess)	Prinzip Sichten	Rundensteiner (TSE)	Prinzip Versionen	Skarra, Zdonik (ENCORE)	Monk, Sommerville (CLOSQL)	Labib, Saunders	Odberg	Kim, Chou (ORION)
3.1	Angebot an Schemaänderungsprimitiven	−−	++	++	+	−−	+/−	++		++	++	++	++	++
3.2	Ausführbarkeit aller Schemaänderungen	(+)	++	++	(++)	+	(++)	+		++	++	++	++	++
3.3	Änderungen auf Schemaebene	+	+	+	+	+	+/−	++		+	−−	−−	++	++
3.4	externe Schemaänderungen	++	+/−	−	−−	++	−−	++		−−	−−	−−	−−	−−
3.5	Erhaltung der Schemazustände	+/−	−−	−−	−−	−−	+	+		+	+	+	++	+
3.6	alternative Schemaversionen	−	−−	−−	−−	−−	++	++		−−	−−	+	++	+
3.7	Integration von Schemaversionen	−−	−−	−−	−−	−−	+	+		−−	−−	−−	+	−−
3.8	Verwaltung der Ableitungsbeziehungen	−−	−−	−−	−−	−−	+	+		−−	−	−	+	+
3.9	Vermeidung von Applikationsanpassungen	−−	−−	−−	(++)		(++)	++		++	++	++	++	+
3.10	Kooperation	−−	−−	−−	(++)		(++)	++		+/−	+	+	++	+/−
3.11	vollständige Propagation	−	(+)	(+)	(++)		(++)	++		+	++	++	++	−
3.12	flexible Propagationssteuerung	(+)	+/−	+/−	−−		−	−		−	−	−	−	−
3.13	mehrfache, getrennte Datenwerte	(+)	−−	−−	−−		−−	−−		−−	−	−	−	−
3.14	semantische Beziehungen	−−	−−	−−	−−		+/−	+/−		−	−−	−−	+	+/−
3.15	Effizienz	−−	−	(+)	++		+/−	+		(+)	+	+	+	+
3.16	geringer Spezifikationsaufwand	−−	(+/−)	(+)	(++)		(+)	(+)		−	+	+	+/−	+
3.17	Lokalität bei der Spezifikation	−	(++)	(++)	(++)		+	+		+/−	(+)	(+)	+/−	+
3.18	Konsistenz	+/−	+	+	++		++	++		(+)	+	+	+	−
3.19	Transparenz für Applikationsentwickler	−−	−−	−−	−		++	++		+	+	+	+	(++)
	formale Beschreibung	−−		+	+	−−				−	−	−	−	++

Abbildung 4.11: Tabellarische Zusammenfassung der Bewertung vorgestellter Mechanismen

Konzepte der direkten Schemaevolution dann Defizite auf, wenn es um die Bewahrung bisheriger Schemazustände und die Weiterverwendung existierender Applikationen geht. Dagegen können Simulationsmechanismen auf der Basis von Sichten mit derlei rückwärtsgerichteten Aspekten sehr gut umgehen. Sie sind jedoch als Konsequenz des verwendeten Sichtenkonzeptes zunächst auf kapazitätserhaltende und -vermindernde Schemaänderungen beschränkt und bedürfen erheblicher Ergänzungen, um zu einer allgemein verwendbaren Grundlage zu werden. Schließlich haben wir einige Arbeiten vorgefunden, die Versionierungskonzepte auf Klassen- oder Schemaebene einsetzen. Obwohl dabei keine konzeptionellen Einschränkungen etwa bezüglich der verwendbaren Schemaänderungen oder bei rückwärtsgerichteten Aspekten bestehen, erfüllt keiner der untersuchten Ansätze die Gesamtheit unserer technischen Teilziele zufriedenstellend. Weiterhin muß bemerkt werden, daß kein kommerzielles Datenbanksystem Versionierungskonzepte in der hier vorgestellten Art und Weise auf Klassen- oder Schemaebene einsetzt. Daher gehen wir die Aufgabe einer detaillierten Modellbildung und einer Realisierung in den folgenden Kapiteln an.

Kapitel 5

Schemaversionierung auf Schemaebene

Wir verfolgen in dieser Arbeit das Schemaversionierungskonzept [KC88, Lau96a], da es bei Schemaevolutionsprozessen bessere Unterstützung bieten kann, als beispielsweise der direkte Ansatz [BKKK87, FMZ+95b, LH90], der dem Benutzer stets nur eine Schemaversion anbietet, auf der Änderungen nur unmittelbar ausgeführt und damit nicht mehr rückgängig gemacht werden können, obwohl solche Systeme intern oftmals trotzdem eine Verwaltung mehrerer Versionen des Schemas benötigten. Zu den Vorteilen des Schemaversionierungskonzeptes zählen zum einen diejenigen, die bereits von der Anwendung der Versionierungsidee auf Objektebene bekannt sind, und die ihren Nutzen für reale Systeme in der Praxis unter Beweis gestellt haben [AN91, CJ90, CK86, DL88, Kat90, TOC93]. Dazu gehören die Unterstützung von Entwurfsprozessen durch Verwaltung von zeitlich nacheinander entwickelten historischen Versionen sowie die Möglichkeit alternative Versionen gleichzeitig zu entwickeln, zu testen und miteinander zu vergleichen. Zum anderen ergeben sich weitere in dieser Arbeit vorzustellende Vorteile, die sich aus der Existenz von Objekten in der Datenbank ergeben, welche von dem versionierten Entwurfsobjekt, also dem Schema, abhängen.

Dieses Kapitel beschreibt das Konzept der Schemaversionierung auf der Abstraktionsebene des Schemas. Nach einer kurzen Zusammenfassung grundlegender Aspekte der Schemaversionierung führen wir das hier verwendete COAST-Objektmodell zunächst für unversionierte Schemata formal ein. Durch eine Menge sog. Schemainvarianten legen wir die Anforderungen fest, die in dem Modell an ein konsistentes Schema gestellt werden und die jederzeit erfüllt sein müssen. Daraufhin erweitern wir das Objektmodell und die Menge der Invarianten auf versionierte Schemata. Diese Betrachtungen behandeln die Komponenten eines versionierten Schemas, seine Konsistenz, die Ableitung neuer Schemaversionen und die Anbindung von Applikationen. Der Kern dieses Kapitels ist die Entwicklung der Schemabeschreibungs- und -änderungssprache. In diesem Arbeitsschritt gehen wir detailliert auf die einzelnen Primitive und die Auswirkungen deren Anwendung auf Schemaebene ein. Die Instanzen der Objektebene werden wir in Kapitel 6 behandeln.

5.1 Grundlegende Aspekte der Schemaversionierung

Wir möchten hier die wesentlichen Aspekte der Bewertung der Schemaversionierung zusammenfassend rekapitulieren.

- Die Schemaversionierung ermöglicht Änderungstransparenz (technisches Teilziel 3.9) und vermeidet damit teure oder unmögliche Änderungen und Recompilierung existierender Applikationen. Obwohl jede Datenbank die Nutzung gemeinsamer Daten verschiedener Anwender zum Ziel hat, kann es sein, daß gewisse Schemaänderungen nicht von allen Nutzern erwünscht sind. Beim direkten Ansatz muß eine Lösung stets den Anforderungen aller Applikationen gleichzeitig gerecht werden, die dann entsprechend angepaßt und recompiliert werden müssen. Im Gegensatz dazu müssen bei der Schemaversionierung nur diejenigen Applikationen angepaßt werden, für die eine solche Änderung wünschenswert ist. Der Ansatz der Schemaversionierung ist damit flexibler, da die Entscheidung ob überhaupt eine Anpassung durchgeführt werden soll und wenn ja wann für jede Applikation individuell getroffen werden kann.

- Schemaänderungen sind nicht immer korrigierender Natur, sondern können auch neue Sichtweisen auf existierende Datenbankstrukturen etablieren. Verschiedene Benutzer können unterschiedliche Anforderungen an die Speicherung und Verwaltung von Informationen haben, d.h. eine einzige Spezifikation ist nicht für alle Anwender und für alle Anforderungen ideal. Weiterhin sind neue Schemaspezifikationen nicht notwendigerweise in allen Belangen besser als alte. Der Schemaversionierung liegt der Gedanke zugrunde, daß menschliche Wahrnehmungen derselben realen Welt oder Diskurswelt verschiedene Abstraktionen und Organisationsformen ergeben können.

- Im Gegensatz zum direkten Ansatz, wo bei Schemaänderungen grundsätzlich Information verloren geht, wird im Schemaversionierungsansatz keinerlei Information gelöscht. Damit wird stets die Möglichkeit bewahrt, Fehler zu korrigieren, indem auf vorherige Zustände zurückgesetzt wird. Der Versionierungsansatz ist flexibler, da er neue Schemata emulieren kann, ohne daß irreversible Veränderungen entstehen. Beim direkten Ansatz besteht keine Möglichkeit, zu einem vorherigen Zustand zurückzukehren, wenn sich eine durchgeführte Schemaänderung als unvorteilhaft erweist. Damit entstehen Risiken bei der Durchführung einer Schemaänderung, die sich insbesondere im Hinblick auf den experimentellen Charakter des Schemadesigns für komplexe Anwendungsgebiete nachteilig auswirken. Im Gegensatz dazu kann beim Versionierungsansatz stets problemlos zu vorherigen Zuständen zurückgekehrt werden (Teilziel 3.5). Dies gilt insbesondere auch auf der Objektebene, wenn das technische Teilziel 3.12 der Propagationskontrolle erfüllt ist.

Die gleichzeitige Verwaltung mehrerer Versionen eines Schemas stellt natürlich zusätzliche Anforderungen an das Datenbanksystem und den Schemaentwickler. Diese sind jedoch in Bezug auf die erzielten Vorteile zu bewerten.

- Das Konzept der Schemaversionierung erscheint zunächst komplexer als der direkte Ansatz, da mehrere Schemaversionen parallel zu verwalten sind. Ansätze wie der von Ferrandina und Zicari (siehe Abschnitt 4.3.4.2), die die direkte Schemaevolution mit einem verzögerten Mechanismus zur Objektkonvertierung realisieren, müssen jedoch ebenfalls mehrere Versionen eines Schemas verwalten. Dies geschieht für den Anwender transparent, womit die Komplexität, die der Schemaentwickler sieht, zwar geringer ist, die vom System zu bewältigende Komplexität ist der beim Schemaversionierungsansatz allerdings vergleichbar, ohne daß dem Anwender die Möglichkeit gegeben würde, Vorteile aus der sowieso vorhandenen, internen Verwaltung mehrerer Schemaversionen zu ziehen.

- Die Sicherung der Konsistenz zwischen den durch verschiedene Schemaversionen sichtbaren Datenbanken kann gewiß Nachteile bezüglich der Effizienz mit sich bringen. Diese ergeben sich allerdings zwangsläufig aus der gewonnenen Flexibilität und sind daher im Vergleich zu den Konsequenzen zu sehen, die sich aus der Undurchführbarkeit notwendiger Anpassungen ergeben würden.

- Manche Schemaänderungen mögen durchaus rein korrigierender Natur sein, so daß mitunter auf die Erhaltung einer bestehenden Schemaversion verzichtet werden könnte. Diesbezügliche Verbesserungsmöglichkeiten lassen sich allerdings ohne weiteres in den Versionierungsmechanismus integrieren [FL96]. Auf diesen Aspekt werden wir in Abschnitt 5.4.2 näher eingehen.

5.2 Formale Definition des COAST-Objektmodells für unversionierte Schemata

Für den in dieser Arbeit entwickelten Ansatz zur Unterstützung von Schemaevolution durch Versionierung benötigen wir kein spezielles Objektmodell. Unser Objektmodell ist denen anderer Systeme, wie z.B. dem von O_2 [BDK92, Dea91] sehr ähnlich, so daß sich unsere Konzepte leicht auf andere Objektmodelle übertragen lassen. Entsprechend ist unsere formale Beschreibung beispielsweise der allgemeinen von Vossen [Vos94] oder speziellen, wie der von O_2 [LRV88, LRV90] ähnlich. Wir führen an dieser Stelle jedoch trotz der genannten Ähnlichkeiten ein eigenes formales Modell ein, um dieses dann in nächsten Abschnitt homogen auf versionierte Schemata erweitern zu können.

In den Abschnitten 5.2.1 bis 5.2.3 geben wir formale Definitionen für einige elementare Komponenten des COAST-Objektmodells auf denen wir im folgenden aufbauen. Abbildung 5.1 zeigt die wesentlichen Komponenten des Objektmodells und ihre Beziehungen zueinander.

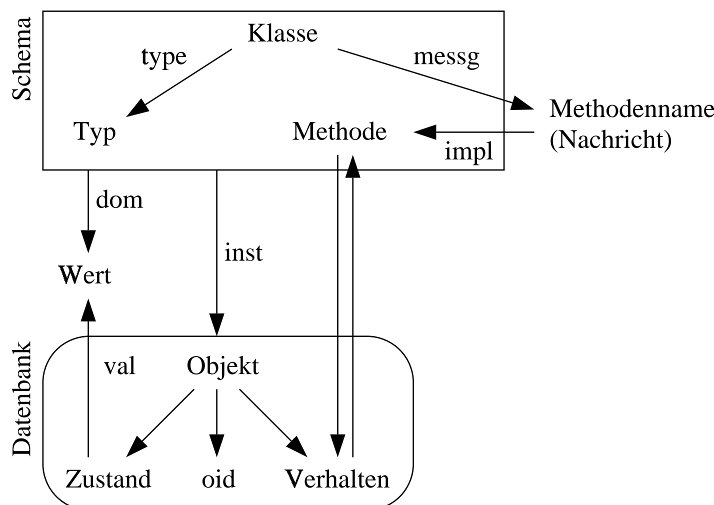


Abbildung 5.1: Die Komponenten des COAST-Objektmodells für unversionierte Schemata.

Die Abschnitte 5.2.4.1 und 5.2.4.2 definieren die Konsistenz eines COAST-Schemas auf der Basis einer Menge von Konsistenzbedingungen. Die das Schema betreffenden Konsistenzbedingungen sind teilweise an die Schemainvarianten des OODBMS ORION angelehnt, welche wir in Abschnitt 4.5.3.7 noch genauer besprechen werden. Daher bezeichnen auch wir unsere Konsistenzbedingungen als Invarianten.

5.2.1 Identifikatoren, Namen, atomare Wertebereiche, Werte und Objekte

Als Grundlage unseres formalen Modells führen wir zunächst Identifikatoren und Namen ein.

Definition 5.1 {Identifikatoren und Namen}
Die folgenden Mengen seien paarweise disjunkt:

- *Die Menge* oId *der Objektidentifikatoren.*
- *Die Menge* sId *der Schemaidentifikatoren.*
- *Die Menge* $svId$ *der Schemaversionsidentifikatoren, die stets die Konstante* sv_0 *enthält.*[52]
- *Die Menge* cId *der Klassenidentifikatoren, die stets die Konstante* c_0 *enthält.*
- *Die Menge* aId *der Identifikatoren struktureller Objekteigenschaften (Attribute).*
- *Die Menge* mId *der Identifikatoren verhaltensmäßiger Objekteigenschaften (Methoden).*
- *Die Menge* $dbId$ *der Datenbankidentifikatoren.*
- *Die Menge* NS (engl. name space) *der vom Benutzer verwendbaren Namen, welche die Vereinigung der folgenden disjunkten Mengen ist:*
  - *Die Menge* $sNames$ *von Schemanamen.*
  - *Die Menge* $svNames$ *von Schemaversionsnamen, die stets die Konstante* **RootSV** *enthält.*[52]
  - *Die Menge* $cNames$ *von Klassennamen, die stets die Konstante* **Object** *enthält.*
  - *Die Menge* $tNames$ *von Typnamen.*
  - *Die Menge* $aNames$ *von Attributnamen.*
  - *Die Menge* $mNames$ *von Methodennamen.*
  - *Die Menge* $dbNames$ *von Datenbanknamen.*

Die systemdefinierten Konstanten **nil**, **none**, **unit** *und* **any** *seien in keiner der genannten Mengen enthalten. Die Menge der Objekteigenschaften* pId *sei die Vereinigung struktureller und verhaltensmäßiger Objekteigenschaften:* $pId := aId \cup mId$.

Definition 5.2 {atomare Wertebereiche}
Die folgenden atomaren Wertebereiche seien gegeben:

- *Der Wertebereich der Wahrheitswerte* dom_bool.
- *Der Wertebereich der ganzen Zahlen* dom_int.
- *Der Wertebereich der rationalen Zahlen* dom_real.
- *Der Wertebereich der Einzelzeichen* dom_char.
- *Der Wertebereich der Zeichenketten* dom_string.
- *Der Wertebereich der Datumsangaben* dom_date.

Der Wertebereich D *sei die Vereinigung aller definierten Wertebereiche* ($D := dom_bool \cup dom_int \cup dom_real \cup dom_char \cup dom_string \cup dom_date$).

[52] Der Einfachheit halber berücksichtigen wir hier bereits die Mengen $svId$ und $svNames$, obwohl wir diese erst für die Erweiterung des Modells auf versionierte Schemata in Abschnitt 5.3.1 benötigen werden.

Ein *Wert* ist eine Instanz, d.h. eine Ausprägung, eines *Typs*. Ein Typ entspricht damit einem Wertebereich. Wir unterscheiden *atomare Werte* (z.B. 1, 2, 3, „a“, etc.), Objektreferenzen (*oids*) und *komplexe* (d.h. zusammengesetzte) *Werte*. Gültige *atomare Typen* sind entsprechend **Boolean**, **Integer**, **Real**, **Char**, **String** und **Date**.[53] *Referenztypen* beziehen sich auf die Klassen des Datenbankschemas und werden durch Angabe des Klassennamens ($cname \in cNames$) spezifiziert. *Komplexe Typen* können mit Typkonstruktoren wie **tuple** (Tupel), **set** (Einfachmenge), **bag** (Mehrfachmenge), **array** (Feld), etc. aus atomaren Typen, Referenztypen oder aus komplexen Typen zusammengesetzt werden.

Definition 5.3 $\left\{\textbf{Wert, } v \in V^{oId}\right\}$

Wir definieren die Menge $V^{oId'}$ *von* Werten über $oId' \subseteq oId$ *als* $V^{oId'} := \bigcup_{i \in \mathbb{N}_0} V_i^{oId'}$. *Die* $V_i^{oId'}$ *sind dabei wie folgt rekursiv definiert:*

- $V_0^{oId'} := D \cup \{\textbf{nil}\} \cup oId'$
- *Seien* $m, n \in \mathbb{N}_0$, $\{a_1, \ldots, a_m\} \subseteq aId$ *und* $\{v_1, \ldots, v_m\} \subseteq \bigcup_{i=1}^{n} V_i^{oId'}$ *Werte, aber* $\{v_1, \ldots, v_m\} \not\subseteq \bigcup_{i=1}^{n-1} V_i^{oId'}$. *Ein Element von* $V_{n+1}^{oId'}$ *(mit m Komponenten) kann dann eines der folgenden sein:*
  - *eine Abbildung*

    $$m_1 = \begin{cases} aId & \rightarrow \bigcup_{i=1}^{n} V_i^{oId'} \\ m_1(a_j) & := v_j \qquad\qquad j \in \{1, \ldots, m\} \end{cases}$$

    die als Tupelwert bezeichnet und als $[a_1 : v_1, \ldots, a_m : v_m]$ *notiert wird.* $[\,]$ *sei der leere Tupelwert.*
  - *eine Teilmenge* $m_2 = \{v_1, \ldots, v_m\} \subseteq \bigcup_{i=1}^{n} V_i^{oId'}$ *die als Mengenwert bezeichnet und als* $\{v_1, \ldots, v_m\}$ *notiert wird.* $\{\}$ *sei der leere Mengenwert.*
  - *eine Abbildung*

    $$m_3 = \begin{cases} \{1, \ldots, m\} & \rightarrow \bigcup_{i=1}^{n} V_i^{oId'} \\ m_3(j) & := v_j \qquad\qquad j \in \{1, \ldots, m\} \end{cases}$$

    die als Listenwert bezeichnet und als $\langle v_1, \ldots, v_m \rangle$ *notiert wird.* $\langle\rangle$ *sei der leere Listenwert.*

Ein Element v von $V_n^{oId'}$ *heißt ein Wert über* oId' *mit der Dimension* $\dim(v) := n$. *(Die Dimension eines Wertes ist wohldefiniert, da die* $V_i^{oId'}$ *disjunkt sind.) Ein Wert v heißt* atomar *wenn* $\dim(v) = 0$, *ansonsten* komplex $(\dim(v) > 0)$. *Ein Element von* $V^{oId'}$ *heißt ein* Wert über oId'. *Ein Element von* V^{oId} *heißt ein* Wert.

Hierbei ist zu beachten, daß bei einem komplexen Wert der Dimension $n+1$ nicht alle Komponenten der Dimension n sein müssen (dann hätten wir in Definition 5.3 $v_1, \ldots, v_m \in V_n^{oId'}$ gefordert). Es genügt, wenn mindestens eine Komponente der Dimension n enthalten ist.

[53] Wir verzichten hier auf den Versuch, eine möglichst vollständige Definition aller atomaren Typen zu geben. Es ist offensichtlich, wie Definition 5.2 ergänzt werden müßte, um als Grundlage für weitere atomare Typen wie Langzahl (**LongInt**), Uhrzeit (**Time**), Zeichenketten bestimmter Länge oder andere dienen zu können.

Das folgende Beispiel zeigt zwei Werte (mit den Dimensionen 2 und 1):

```
[   family_name = "Kennedy",
    first_name  = "John F.",
    birthday    = [ day     = 29,
                    month   = 5,
                    year    = 1917],
    company     = "USA"]

[   name    = "United States of America",
    boss    = "JFK"]
```

Zwei Werte.

Definition 5.4 $\{$**Objekt,** $o\}$
Ein Objekt *o ist ein Paar $o = (oid, v)$ mit $oid \in oId$ und $v \in V^{oId}$.*

Zur Vereinfachung unserer Darstellungen definieren wir $oid(o) := oid$ und $v(o) := v$.

Den Begriff der *Instanz* verwenden wir synonym zu Objekt.

5.2.2 Typen, Wertebereiche von Typen und Typhierarchien

Da die atomaren Wertebereiche disjunkt sind, können wir den Typ eines Objektes o aus seinem Wert $v(o)$ erkennen.

Definition 5.5 $\left\{\textbf{Typ, } t \in T^{cId}\right\}$
Wir definieren die Menge *$T^{cId'}$* von Typen über *$cId' \subseteq cId$ als $T^{cId'} := \bigcup_{i \in \mathbb{N}_0} T_i^{cId'}$. ($T^{cId'}$ ist eine Obermenge von $tNames$.) Gegeben sei eine Abbildung $\boldsymbol{def}$ von $tNames$ nach $T^{cId'}$, die in dem Sinne azyklisch sein muß, daß $\forall tname \in tNames\ \exists n \in \mathbb{N}, t \in T^{cId'} \backslash tNames : \boldsymbol{def}^n(tname) = t$. Wir definieren die Abbildung type als die transitive Hülle von $\boldsymbol{def}$ in dem Sinne, daß type von $T^{cId'}$ nach $T^{cId'} \backslash tNames$ abbildet.*

$$type = \left\{ \begin{array}{ccc} T & \rightarrow & T^{cId'} \backslash tNames \\ type(t) & := & \left\{ \begin{array}{ll} t & falls\ t \in T^{cId'} \backslash tNames \\ type(\boldsymbol{def}(t)) & falls\ t \in tNames \end{array} \right. \end{array} \right.$$

Die T_i sind dabei wie folgt rekursiv definiert:

- $\{\textbf{bool}, \textbf{int}, \textbf{real}, \textbf{char}, \textbf{string}, \textbf{date}, \textbf{any}\} \cup cId' \subseteq T_0^{cId'}$
- *$tname \in T_n^{cId'}$ genau dann, wenn $tname \in tNames \wedge type(tname) \in T_n^{cId'}$*
- *Seien $m, n \in \mathbb{N}_0$, $\{a_1, \ldots, a_m\} \subseteq aNames$, $t \in T_n^{cId'}$ und $\{t_1, \ldots, t_m\} \subseteq \bigcup_{i=1}^{n} T_i^{cId'}$. Jedoch gelte $\{t_1, \ldots, t_m\} \not\subseteq \bigcup_{i=1}^{n-1} T_i^{cId'}$. Dann kann ein Element von $T_{n+1}^{cId'}$ eines der folgenden sein:*
  - *ein Tupeltyp, notiert als $[a_1 : t_1, \ldots, a_m : t_m]$. $a_1, \ldots, a_m$ werden* Attribute *des Tupeltyps genannt.*
  - *ein Mengentyp, notiert als $\{t\}$.*
  - *ein Listentyp, notiert als $\langle t \rangle$.*

Ein Element t von $T_n^{cId'}$ wird Typ der Dimension $\dim(t) := n$ *genannt. (Die Dimension ist wohldefiniert, weil die Funktion **def** existiert und weil die $T_i^{cId'}$ disjunkt sind.) Ein Element $t \in T^{cId}$ heißt* Typ. *Ein Typ t heißt* atomar, *wenn* $\dim(t) = 0$, *ansonsten* komplex *($\dim(t) > 0$).*

Ähnlich wie bei komplexen Werten ist auch hier zu beachten, daß bei einem komplexen Typ der Dimension $n+1$ nicht alle Komponenten der Dimension n sein müssen (dann hätten wir in Definition 5.5 $t_1, \ldots, t_m \in T_n^{cId'}$ gefordert). Es genügt, wenn mindestens eine Komponente der Dimension n enthalten ist.

Ein benannter Typ ist ein Paar aus Name und Typ, wobei der Name als abkürzende Schreibweise für einen existierenden Typ benutzt werden kann. Die Funktion ***def*** bildet den Namen eines benannten Typs auf den diesem Namen zugeordneten Typ ***def***$(tname)$ ab, welcher allerdings wiederum der Name eines benannten Typs sein kann. Daher definieren wir die Abbildung *type*, die die Abbildung ***def*** n-mal anwendet, so daß ***def***$^n(tname) \in T^{cId'} \backslash tNames$, d.h. ***def***$^n(tname)$ ist tatsächlich ein Typ (und nicht mehr der Name eines benannten Typs). Die Forderung nach Zyklenfreiheit wird benötigt, um zyklische Definitionen zu verbieten, wie z.B. ***def***$(tname_1) := tname_2$ und ***def***$(tname_2) := tname1$, was nicht erlaubt werden kann.

Das folgende Beispiel zeigt die Typen der beiden Klassen `Person_class` und `Company_class` (`Person_class, Company_class` $\in cId$) (mit den Dimensionen 2 und 1). Dies sind allerdings nicht die Typen der beiden, im vorangegangenen Beispiel gezeigten Werte.

```
Person_type = [ name : [family_name : string,
                        first_name  : string],
                birthday   : date,
                spouse     : Person_class,
                company    : Company_class]

Company_type = [   name  : string,
                   boss  : Person_class]
```

Zwei Typen.

Hierbei ist zu beachten, daß ein Attribut eines Typs aus cId (wie z.B. **spouse** und **boss** im Beispiel) nur eine Objektreferenz darstellt (im Gegensatz zu Attributen, die andere Typen darstellen (wie z.B. **Company_type**)). Daher sind die Dimensionen der Typen **Person_type** und **Company_type** unabhängig vom Typ der referenzierten Klasse **Person_class**. Eine Definition der Dimension, welche die Typen referenzierter Klassen betrachtet, wäre nicht sinnvoll, da mit Objektreferenzen auch zyklische Strukturen gebildet werden können (d.h. zwei Objekte können sich gegenseitig referenzieren), wobei die Dimension in diesen Fällen undefiniert wäre, was ihren Nutzen stark einschränkt. Die zyklische Verschachtelung von Typen ist hingegen nicht erlaubt (siehe die Forderung nach der Existenz der Funktion ***def*** in Definition 5.5), was ein notwendiges Kriterium für die Wohldefiniertheit unserer Definition 5.5 der Dimension eines Typs darstellt.

Definition 5.6 $\left\{\right.$**Wertebereich** $dom^{cId'}(t)$ **eines Typs** t **über** $cId' \subseteq cId\left.\right\}$
Wir definieren den Wertebereich $dom^{cId'}(t)$ eines Typs $t \in T^{cId'}$ *wie folgt:*

- $dom^{cId'}(\mathbf{bool}) := dom_bool$
- $dom^{cId'}(\mathbf{int}) := dom_int$
- $dom^{cId'}(\mathbf{real}) := dom_real$

- $dom^{cId'}(\mathbf{char}) := dom_char$
- $dom^{cId'}(\mathbf{string}) := dom_string$
- $dom^{cId'}(\mathbf{date}) := dom_date$
- $dom^{cId'}(\mathbf{none}) := \{\mathbf{nil}\}$
- $dom^{cId'}(c) := oId \cup dom^{cId'}(\mathbf{none})$ *für jedes* $c \in cId'$
- $dom^{cId'}([a_1 : t_1, \ldots, a_m : t_m]) := \{[a_1 : t_1, \ldots, a_m : t_m] \mid (\forall i, 1 \leq i \leq m)\ v_i \in dom^{cId'}(t_i)\}$
- $dom^{cId'}(\{t\}) := \{\{v_1, \ldots, v_n\} \mid (\forall i, 1 \leq i \leq m)\ v_i \in dom^{cId'}(t)\}$
- $dom^{cId'}(\langle t \rangle) := \{\langle v_1, \ldots, v_n \rangle \mid (\forall i, 1 \leq i \leq m)\ v_i \in dom^{cId'}(t)\}$
- $dom^{cId'}(tname) := dom^{cId'}(type(tname))$ *für* $tname \in tNames$
- $dom^{cId'}(\mathbf{any}) := \{\mathbf{unit}\}$

Der Wertebereich $dom^{cId'}(\mathbf{none})$ enthält nur den für alle Referenztypen gültigen Nullwert **nil**. Wir haben **none** eingeführt, damit auch der Typ von **nil** einen Namen hat, so wie der Typ von **unit** den Namen **any** trägt.

Der Wertebereich $dom^{cId'}(c)$ einer Klasse c ist die Menge aller Objektidentifikatoren (vereinigt mit der Konstanten **nil**). Dies bedeutet jedoch nur eine obere Schranke. In Definition 5.20 einer Datenbank werden wir zusätzlich einschränkend fordern, daß nur solche Objektidentifikatoren zugewiesen werden, die auch zu Objekten einer Klasse aus cId' gehören (und nicht zu einer beliebigen Klasse).

Der Typ **any** enthält keinerlei Information und sein Wertebereich $dom^{cId'}(\mathbf{any})$ kann daher durch eine einelementige Menge repräsentiert werden. Wir haben hierfür die Konstante **unit** gewählt, die einen Wert hat, der in keinem der atomaren Wertebereiche aus Definition 5.2 enthalten ist. Da der Wertebereich des Typs **any** nur ein Element enthält, benötigt eine Variable dieses Typs keinen Speicherplatz. Jeder Wert jedes anderen Typs kann einer Variablen vom Typ **any** zugewiesen werden, aber dabei geht jegliche Information verloren.

Die Situation hier ist mit der des leeren Tupeltyps ([]) vergleichbar. Dieser ist der gemeinsame Obertyp aller Tupeltypen, aber eine Variable dieses Typs kann keinerlei Information speichern. Hat eine Klasse c den leeren Tupeltyp, so können Unterklassen von c nichtleere Tupeltypen haben und jede Variable vom Typ c kann auf Instanzen dieser Unterklassen verweisen. Konvertiert man jedoch den Inhalt eines Objektes einer solchen Klasse zum leeren Tupeltyp, so geht dabei die gesamte darin enthaltene Information verloren.

Wenn ein Wert oder eine Variable eines Typs einer Variablen eines anderen Typs zugewiesen werden kann, so sprechen wir von *Typkompatibilität.* Diese ist jedoch keine symmetrische Beziehung zwischen Typen, weswegen wir genauer *Ober-* und *Untertypen* unterscheiden, die in einer Typhierarchie angeordnet sind.

Definition 5.7 {Typhierarchie mit Ober- und Untertypbeziehung, $\leq_t$}
Wir definieren die Ober- und Untertypbeziehung (engl. subtyping relationship) $\leq_t \subseteq (T^{cId'})^2$ *wie folgt:*

- **bool** $\leq_t$ **int** $\leq_t$ **real**
- **char** $\leq_t$ **string**

- $t \leq_t$ **any** *für jedes* $t \in T$
- $t \leq_t t$ *für jedes* $t \in T$
- $[a_1 : t_1^1, \ldots, a_{m+n} : t_{m+n}^1] \leq_t [a_1 : t_1^2, \ldots, a_m : t_m^2]$
  $m, n \in \mathbb{N}_0$, $\{a_1, \ldots, a_{m+n}\} \subseteq aNames$, $t_1^1, \ldots, t_{m+n}^1, t_1^2, \ldots, t_m^2 \in T$
  für $t_i^1 \leq_t t_i^2$, $i = 1, \ldots, m$
- $\{t_1\} \leq_t \{t_2\}$, *für* $t_1 \leq_t t_2$
- $\langle t_1 \rangle \leq_t \langle t_2 \rangle$, *für* $t_1 \leq_t t_2$
- $t_1 \leq_t t_2$, *für* $type(t_1) \leq_t type(t_2)$
- $c_1 \leq_t c_2$, *für* $c_1, c_2 \in cId'$ *genau dann, wenn* $c_1 \leq_c c_2$.[54]

Ein Typ T_u *heißt* Obertyp *eines Typs* T_v *und* T_v *entsprechend* Untertyp *von* T_u *genau dann, wenn* $T_v \leq_t T_u$.

Bei Tupeltypen muß jedes Attribut a von T_u (beschrieben als $T_u.a$) auch in T_v enthalten sein und der Typ von $T_u.a$ muß ein Obertyp des Typs von $T_v.a$ sein. Bei Mengentypen muß der Elementtyp von T_u ein Obertyp des Elementtyps von T_v sein.[55] Sind T_u und T_v Referenztypen (d.h. ihr Wertebereich ist die Extension einer Klasse), dann muß die von T_u referenzierte Klasse eine Oberklasse der von T_v referenzierten Klasse sein.

Damit die Untertypbeziehung tatsächlich verwendet werden kann, muß definiert sein, wie ein Wert eines Typs t' bei der Programmausführung als Wert eines Obertyps t von t' verwendet wird. Dazu muß die Konvertierung von t' nach t für jedes $t \geq_t t'$ definiert sein. Diese Definition muß konsistent sein in dem Sinne, daß das Ergebnis der Konvertierung unabhängig von dem Pfad und den dabei entstehenden Zwischenschritten sein muß, wenn in der Typhierarchie mehrere Typen zwischen t' und t existieren. Im Falle von Tupeltypen wird die Konvertierung durch die Projektion geleistet, in allen übrigen Fällen sollte die natürliche Art der Konvertierung offensichtlich sein.

5.2.3 Klassen, Vererbung und (strukturelle und verhaltensmäßige) Schemata

Definition 5.8 $\{$**Klasse**, $c\}$
Eine Klasse c *ist ein 4-Tupel* $c = (cid, cname, type, ms)$ *mit den folgenden Komponenten:*

- $cid \in cId$ *ist der Identifikator der Klasse.*
- $cname \in cNames$ *ist der benutzerdefinierte Name der Klasse.*
- $type \in T$ *ist der Typ der Klasse.*[56]

[54] Die hier verwendete Klassenhierarchie $\leq_c$ wird in Definition 5.9 eingeführt werden. Invariante 5.7 wird sicherstellen, daß der Typ einer Unterklasse c_1 ein Untertyp des Typs einer Oberklasse c_2 ist ($type(c_1) \leq_t type(c_2)$).

[55] Wollen wir in Erweiterung unseres Objektmodells neben Einfach- auch Mehrfachmengen zulassen, so müßten wir hier zusätzlich fordern, daß T_u ein Mehrfachmengentyp und T_v ein beliebiger Mengentyp (Einfach- oder Mehrfachmengentyp) ist oder daß T_u und T_v Einfachmengentypen sind.

[56] Hier sind also beliebige Typen verwendbar. Die meisten der existierenden Systeme schränken hier allerdings auf Tupeltypen ein, da diese in der Praxis wohl am häufigsten vorkommenden. Wir werden stellenweise auch von dieser Einschränkung ausgehen.

- *$ms \subseteq mId$ ist die Menge der Methoden, die in dieser Klasse definiert wurden. Die Namen der Methoden müssen innerhalb einer Klasse nicht eindeutig sein, da wir Überschreiben und Überladen von Methoden zulassen möchten.*

Zur Vereinfachung unserer Darstellungen definieren wir $cid(c) := cid$, $cname(c) := cname$, $type(c) := type$ und $ms(c) := ms$.

Die Klasse **Object** *ist definiert als* $(c_0, \textbf{Object}, \textbf{any}, \emptyset)$. *Wir werden die Klasse c_0 im folgenden auch durch ihren Namen* **Object** *identifizieren.*

Das COAST-Objektmodell unterstützt Einfach- und Mehrfachvererbung. Bereits bei der Verwendung der Einfachvererbung sind einige Bedingungen einzuhalten, so daß gewünschte Eigenschaften des Verhältnisses von Ober- zu Unterklasse garantiert werden können. Weitere Bedingungen sind bei der Verwendung der Mehrfachvererbung zu beachten, damit die dort ggf. auftretenden Vererbungskonflikte aufgelöst werden können. Wir gehen auf die beiden genannten Gruppen von Bedingungen im folgenden separat ein.

Im COAST-Objektmodell erfüllt das Verhältnis zwischen einer Klasse c und einer Unterklasse c' von c stets zwei Eigenschaften: Zum einen ist die Unterklasse c' stets eine Spezialisierung (oder identisch mit) ihrer Oberklasse, zum anderen besteht eine Teilmengenbeziehung zwischen den Extensionen von c und c'. Die erste Eigenschaft besagt, daß eine Unterklasse stets nur eine speziellere Beschreibung ihrer Objekte zuläßt als die Oberklasse. Daher werden die Möglichkeiten zur Veränderung einer geerbten Attributmenge (falls die Oberklasse einen Tupeltyp hat) wie folgt eingeschränkt: Attribute, die nicht von einer Oberklasse geerbt werden, können in c hinzugefügt werden, geerbte Attribute können jedoch nicht entfernt werden. Weiterhin kann der Typ eines geerbten Attributes zwar spezialisiert (d.h. ein Typ T_u wird durch einen Untertyp $T_v \leq_t T_u$ ersetzt), nicht aber generalisiert werden. Mit diesen Einschränkungen kann ein Objekt der Unterklasse c' ein Objekt der Klasse c substituieren, d.h. ein Objekt der Unterklasse kann überall dort verwendet werden, wo ein Objekt der Oberklasse erwartet wird. Dazu ist es lediglich notwendig, die Werte ggf. in der Unterklasse hinzugefügter Attribute „abzuschneiden“. Damit ist auch die zweite oben genannte Eigenschaft garantiert. Da jedes Objekt auch als Objekt der Oberklassen seiner direkten Klasse verwendet werden kann, ist die Extension einer Klasse stets eine Teilmenge der Extensionen ihrer Oberklassen. Insbesondere umfaßt die Wurzelklasse **Object** alle Objekte der Datenbank.

Die dargestellte Kombination der beiden Eigenschaften wird auch als *is_a-Semantik* bezeichnet und heißt in der Klassifikation von Wegner [Weg87] *strikte Vererbung*. Im Allgemeinen müssen die beiden genannten Eigenschaften der Vererbungsbeziehung nicht zusammenfallen (siehe z.B. [Heu97]). Wir beschränken uns in dieser Arbeit allerdings auf dieses einfache Modell, da es weit verbreitet ist und unser Hauptaugenmerk nicht auf der Entwicklung eines besonders leistungsfähigen Objektmodells liegt.

Zur Darstellung der Vererbungsbeziehung zwischen den Klassen eines Schemas definieren wir den *Klassenvererbungsgraphen*.

Definition 5.9 {Klassenvererbungsgraph, CIG}
Ein Klassenvererbungsgraph *CIG über einer Menge von Klassen $cids$* (engl. class inheritance graph) *(kurz auch* Vererbungsgraph *oder* Klassenhierarchie *genannt) ist ein verwurzelter, gerichteter, azyklischer und zusammenhängender Graph, dessen Knoten die Klassen aus cids repräsentieren und dessen Kanten die Vererbungsbeziehung zwischen diesen Klassen widerspiegeln. Eine Kante von Klasse c_v nach Klasse c_u ($c_u, c_v \in cids$) bedeute dabei, daß c_v von c_u erbt.*

Formal wird ein Klassenvererbungsgraph dargestellt als ein Paar $CIG = (cids, <_c^1)$ mit den folgenden Komponenten:

- *$cids \subseteq cId$ ist die Menge der Identifikatoren derjenigen Klassen, die zu dem Klassenvererbungsgraphen gehören.*
- *$\leq_c \subseteq cids^2$ ist eine Ordnungsrelation auf cids entsprechend Definition 2.3.*

Dabei müssen die folgenden Bedingungen erfüllt sein:

- $c_0 \in cids$
- *Die systemdefinierte Klasse c_0 ist die Wurzel des Klassenvererbungsgraphen.*
  Formal: $\forall c \in cids : c \leq_c c_0$

Entsprechend Definition 2.4 verwenden wir auch die Symbole $<_c$, $\leq_c$, $\leq_c^1$, $>_c$, $\geq_c$, $>_c^1$ und $\geq_c^1$ zur Darstellung der Vererbungsbeziehung.

Die zweite der obigen Bedingungen garantiert den Zusammenhang des Klassenvererbungsgraphen.

Definition 5.10 $\left\{ \begin{array}{l} \textbf{(direkte) Ober- und Unterklassen,} \\ pred(c), dpred(c), succ(c), dsucc(c) \end{array} \right\}$

Wenn in einem Klassenvererbungsgraphen $CIG = (cids, <_c^1)$ $c_v \leq_c c_u$ ($c_v <_c^1 c_u$) gilt, dann heißt c_u (direkte) Oberklasse *von c_v und c_v heißt* (direkte) Unterklasse *von c_u. Ferner führen wir die folgenden abkürzenden Schreibweisen für Teilmengen von cids ein:*

$$
\begin{array}{rcl}
pred(c) & := & \{c' \in cids | c \leq_c c'\} \\
dpred(c) & := & \{c' \in cids | c <_c^1 c'\} \\
succ(c) & := & \{c' \in cids | c' \leq_c c\} \\
dsucc(c) & := & \{c' \in cids | c' <_c^1 c\}
\end{array}
$$

Ein Schema ist grob gesagt eine Menge von Klassenintensionen mit den dazwischen bestehenden Beziehungen. Wir definieren den strukturellen und den verhaltensmäßigen Teil eines Schemas zunächst getrennt und fügen die beiden Teile dann zusammen.

Definition 5.11 $\{$**strukturelles Schema,** $s_{struct}\}$

Ein strukturelles Schema *ist ein 6-Tupel $s_{struct} = (sid, sname, cids, <_c^1, types, type)$ mit den folgenden Komponenten:*

- *$sid \in sId$ ist der eindeutige Identifikator des Schemas.*
- *$sname \in sNames$ ist der benutzerdefinierte Name des Schemas.*
- *$cids \subseteq cId$ ist eine Menge von Klassenidentifikatoren.*
- *$<_c^1 \subseteq cids^2$ ist eine Klassenrelation.*
- *types ist eine Menge von Typen[57] über T^{cids} ($types \subseteq T^{cids}$).*
- *type ist eine totale Funktion von der Menge der Klassenidentifikatoren cids in die Menge der Typen types ($type : cids \rightarrow types$).*

Vorbereitend für die Beschreibung von Methoden definieren wir zunächst deren Signaturen.

[57] Nach Definition 5.5 schließt die Menge *types* ggf. im Schema definierte Typnamen ein.

Definition 5.12 $\left\{\textbf{Methodensignatur über } cId' \subseteq cId,\ sig \in Sig^{cId'}\right\}$
Wir definieren die Menge $Sig^{cId'}$ der Methodensignaturen über $cId' \subseteq cId$ *(kurz auch* Signaturen *genannt) als* $Sig^{cId'} := \bigcup_{n \in \mathbb{N}_0} Sig_n^{cId'}$ *mit* $Sig_n^{cId'} := cId' \times (T^{cId'})^{n-1}$.

Ein Element von $Sig^{cId'}$ *heißt* Signatur über cId'. *Ein Element von* Sig_n *heißt* Signatur der Stelligkeit n. *Ein Element von* $Sig := Sig^{cId}$ *heißt* Signatur *und wird notiert als* $cid : t_1 \times \cdots \times t_n \to t_{n+1}$ *mit* $cid \in cId$ *und* $t_1, \ldots, t_{n+1} \in T^{cId}$.

Für Methoden ohne expliziten Rückgabewert nehmen wir $t_{n+1} =$ **any** *an und liefern bei Bedarf den Wert* **unit** *an das aufrufende Objekt zurück.*

Der Aufruf einer Methode m bei einem Objekt o mit Zuweisung ihres Rückgabewertes an eine Variable a ($a := o.m()$) darf natürlich nur dann erfolgen, wenn m tatsächlich explizit einen Rückgabewert liefert. Für andere Methoden nehmen wir nach Definition 5.12 zwar implizit einen Rückgabewert an; dadurch entsteht hier jedoch kein Problem. Der Rückgabewert ist nämlich **unit** und kann daher nur Variablen vom Typ **any** zugewiesen werden. Hat a also einen von *any* verschiedenen Typ und liefert m jedoch keinen Rückgabewert, so wird die Unzulässigkeit von Aufrufen der Art $a := o.m()$ aufgrund des entstehenden Typfehlers (**unit** $\notin dom(a)$) erkannt, und zwar statisch, d.h. bereits zur Übersetzungszeit.

Definition 5.13 {**Methode,** m}
Eine Methode m *ist ein 4-Tupel* $m = (mid, mname, sig, code)$ *mit den folgenden Komponenten:*

- $mid \in mId$ *ist der Identifikator der Methode.*
- $mname \in mNames$ *ist der benutzerdefinierte Name der Methode.*
- $sig \in Sig$ *ist die Signatur der Methode.*
- $code$ *ist der Code der Methode, also die Methodenimplementierung.*

Zur Vereinfachung unserer Darstellungen definieren wir $mid(m) := mid$, $mname(m) := mname$, $sig(m) := sig$ *und* $code(m) := code$.

Damit können wir nun auch den verhaltensmäßigen Teil des Schemas beschreiben.

Definition 5.14 {**verhaltensmäßiges Schema,** s_{behav}}
Ein verhaltensmäßiges Schema *ist ein 7-Tupel* $s_{behav} = (sid, sname, cids, M, mnames, messg, impl)$ *mit den folgenden Komponenten:*

- $sid \in sId$ *ist der eindeutige Identifikator des Schemas.*
- $sname \in sNames$ *ist der benutzerdefinierte Name des Schemas.*
- $cids \subseteq cId$ *ist eine Menge von Klassenidentifikatoren.*
- M *ist eine Menge von Methoden.*
- $mnames \subseteq mNames$ *ist eine Menge von Methodennamen,*[58] *wobei jedem* $mname \in mnames$ *eine nichtleere Menge von Signaturen über cids zugeordnet ist:* $sig(mname) = \{sig_1, \ldots, sig_l\}$, $l \geq 1$, *jedes* sig_h $(1 \leq h \leq l)$ *hat die Form* $sig_h : cid \times t_1 \times \cdots \times t_p \to t$ *für* $cid \in cids$ *und* $t_1, \ldots, t_p, t \in T$. *(Jede Signatur hat die Empfängerklasse der Nachricht als erste Komponente.)*

[58] Die Methodennamen werden in [Vos94] als *Nachrichtennamen* bezeichnet.

- *messg ist eine Abbildung von der Menge der Klassenidentifikatoren in die Potenzmenge der Methodennamen* ($messg : cids \rightarrow 2^{mnames}$)*, so daß für jedes* $c \in cids$ *und für jedes* $mname \in messg(c)$ *eine Signatur* $s \in sig(mname)$ *existiert mit* $sig[1] = c$*.*

- *impl ist eine partielle Funktion mit* $impl : \{(mname, sig, c) | mname \in messg(c), sig \in Sig^c\} \rightarrow M$*.*

Die Abbildung *messg* ordnet also jeder Klasse die Menge der Methodennamen zu, die ihre Objekte möglicherweise, d.h. in Abhängigkeit von der konkreten Signatur eines Aufrufs,[59] ausführen können.

Die Abbildung *impl* ordnet einem Methodenaufruf die auszuführende Methode zu. Deren Auswahl hängt ab von dem Namen der aufgerufenen Methode (*mname*), von den übergebenen Parametern und dem erwarteten Rückgabetyp[60] (spezifiziert als Signatur[61] *sig* des Aufrufs) und von der Klasse c des Objektes, bei dem der Aufruf stattfindet. Hier ist zu beachten, daß Definition 5.13 noch keine Zuordnung von Methoden zu Klassen vorgenommen hat.

Schließlich können wir ausgehend von den beiden Teilen die Definition eines Schemas geben.

Definition 5.15 {Schema}
Ein Schema ist ein 10-Tupel $s = (sid, sname, cids, <^1_c, types, type, M, mnames, messg, impl)$*, das die beiden folgenden Bedingungen erfüllt:*

- *Das 6-Tupel* $s_{struct} = (sid, sname, cids, <^1_c, types, type)$ *ist ein strukturelles Schema nach Definition 5.11.*

- *Das 7-Tupel* $s_{behav} = (sid, sname, cids, M, mnames, messg, impl)$ *ist ein verhaltensmäßiges Schema nach Definition 5.14.*

Zur Vereinfachung unserer Darstellungen definieren wir $sid(s) := sid$*,* $sname(s) := sname$ *und* $classes(s) := cids$ *als Symbol für die Menge aller Klassen eines Schemas s.*

Vorbereitend führen wir den Begriff der Herkunft eines Attributes bzw. einer Methode ein.

Definition 5.16 {Herkunft eines Merkmals einer Klasse c}
Die Herkunft eines (strukturellen oder verhaltensmäßigen) Merkmals (*aid* bzw. *mid*) einer Klasse *c ist diejenige Oberklasse von c, in der das betreffende Attribut (aid) bzw. die Methode (mid) lokal definiert (und nicht geerbt) wurde. Bei Methoden ist die Herkunft signatur-spezifisch, d.h. überladene Methoden einer Klasse können verschiedener Herkunft sein.*

5.2.4 Der Konsistenzbegriff für unversionierte Schemata

Nachdem wir im vorangegangenen Abschnitt definiert haben, was wir unter einem Schema und unter seinen strukturellen und verhaltensmäßigen Komponenten verstehen, leiten wir nun den

[59] Der Name *messg* deutet darauf hin, daß der Aufruf einer Methode eines Objektes o hier als das Senden einer Nachricht (engl. *message*) an o verstanden wird, worauf o mit der Ausführung der passenden Methode reagiert.

[60] Wenn der Rückgabewert einer Methode durch einen Aufruf der Art $a := o.m(\ldots)$ direkt einer Variablen a zugewiesen wird, kann deren Typ als Spezifikation eines erwarteten Typs verstanden werden und einen Einfluß auf die Auswahl der beim Objekt o auszuführenden Methode haben. Dies ist allerdings nur in sehr wenigen objektorientierten Modellen der Fall. Wir haben den Rückgabetyp hier der Vollständigkeit halber aufgenommen und weil wir so die bereits eingeführte Signatur zur formalen Spezifikation heranziehen können.

[61] Auch [Vos94] verwendet eine Abbildung *impl*, die einer bei einem Objekt eintreffenden Nachricht die entsprechend auszuführende Methode zuordnen soll. Dort fehlt allerding die Berücksichtigung der Parameter, d.h. der Signatur der Nachricht.

Konsistenzbegriff für unversionierte Schemata her. Zur Vorbereitung für diesen geben wir eine Menge sog. *Schemainvarianten* an. Diese sind ähnlich den Schleifeninvarianten bei der formalen Programmverifikation zu sehen. Sie müssen eingehalten werden, damit sich das System, hier speziell das Schema einer Datenbank, in einem konsistenten Zustand befindet. Entsprechend ihrer Aussage haben wir eine Zuordnung der Invarianten zu den Bereichen Eindeutigkeit, Vollständigkeit, Vererbungsbeziehungen und Verhalten vorgenommen und gehen auf diese im folgenden nacheinander ein.

5.2.4.1 Schemainvarianten

5.2.4.1.1 Invarianten bezüglich der Eindeutigkeit

Die Invarianten bezüglich der Eindeutigkeit der Schemakomponenten stellen sicher, daß diese unverwechselbar angesprochen werden können. Dabei betreffen die Namen mehr die Sicht des Benutzers eines Datenbanksystems während die Identitäten intern Verwendung finden.

Invariante 5.1 {eindeutige Namen, vergleiche ORION-Invariante 4.2}
Sämtliche innerhalb gewisser Gültigkeitsräume vergebene Namen müssen eindeutig sein. Dies gilt insbesondere für die Namen der Klassen innerhalb eines Schemas und für die (geerbten oder lokal definierten) Attribute innerhalb einer Klasse.[62] *In dieser Arbeit reservierte Namen, die insbesondere als Schlüsselworte der Schemabeschreibungssprache (siehe Abschnitt 5.5) Verwendung finden, dürfen nicht für benutzerdefinierte Schemakomponenten vergeben werden.*

Für die Klassenidentifikatoren und Klassennamen gilt:

$$\forall c_1, c_2 \in cids : cname(c_1) = cname(c_2) \Rightarrow c_1 = c_2$$

Bei der Eindeutigkeit der Eigenschaften einer Klasse muß berücksichtigt werden, daß diese geerbt oder überschrieben sein können.

Invariante 5.2 {eindeutige Herkunft, vergleiche ORION-Invariante 4.3}
Alle Attribute und Methoden einer Klasse haben eine eindeutige Herkunft (Identität).

Die in Invariante 5.2 geforderte Eindeutigkeit der Identität bedeutet, daß sich auch die Identifikatoren (*mid*) überschriebener und überladener Methoden unterscheiden.

Ohne die Forderung nach einer eindeutigen Herkunft jeder Methode (mitunter *seed* genannt) hätten wir Definition 5.13 um eine zusätzliche Komponente ergänzen müssen, um ggf. entstehende Mehrdeutigkeiten explizit auflösen zu können.

5.2.4.1.2 Invariante bezüglich der Vollständigkeit

Die Invariante der Vollständigkeit garantiert, daß alle benötigten Komponenten eines Schemas vorhanden sind. Dabei unterscheiden wir die für die Beschreibung eines Datenbankzustandes notwendige strukturelle Vollständigkeit von der für die Ausführung von Methoden benötigten verhaltensmäßigen Vollständigkeit.

Invariante 5.3 {Vollständigkeit}
Ein Schema ist vollständig, wenn es die folgenden Anforderungen erfüllt.

- *strukturelle Vollständigkeit:*
  - *Das Schema enthält die Wurzelklasse c_0 namens* **Object**.

[62] Da wir uns in dieser Arbeit stets mit nur einem Schema und einer Datenbank befassen werden, können wir hier darauf verzichten, auch für diese explizit eindeutige Namen zu fordern.

  - *Alle von Klassen im Rahmen von Vererbungs- oder Kompositionsbeziehungen als Oberklassen oder als Komponentenklassen referenzierten Klassen existieren.*
  - *Die Abbildung **def** muß wohldefiniert sein, d.h. für jeden Typnamen $tname$ muß ein entsprechender Typ $type(tname)$ im Schema existieren.*

    Formal: $\forall tname \in tnames \exists t \in types : type(tname) = t$
  - *Alle Typen, die für die Spezifikation von Attributen verwendet werden, sind in T definiert.*

    Formal: $\forall c \in cids : type(c) \in T^{cids}$ *und weiterhin:* $\forall t \in types : t \in T^{cids}$

- *verhaltensmäßige Vollständigkeit:*
  - *bezüglich Methodensignaturen*

    Alle Typen und Klassen, die von Ein- oder Rückgabeparametern einer Methode verwendet werden, existieren.
  - *bezüglich Methodenimplementierungen*

    Alle von Methodenimplementierungen referenzierten Klassen sowie benutzte Klasseneigenschaften (Attribute und Methoden) und Typen existieren.

5.2.4.1.3 Invarianten bezüglich der Vererbung

Die in diesem Abschnitt vorzustellenden Invarianten garantieren eine wohldefinierte Semantik der Vererbungsbeziehung zwischen verschiedenen Klassen eines Schemas. Dabei sind insbesondere zweierlei Konflikte zu berücksichtigen. Zum einen können sich Überschneidungen zwischen den lokal definierten und den von ihren Oberklassen geerbten Eigenschaften einer Klasse ergeben (Überschreibungen), zum anderen können sich Diskrepanzen zwischen den Eigenschaften mehrerer direkter Oberklassen ergeben (typischer Fall eines Konfliktes bei Mehrfachvererbung).

Invariante 5.4 {Klassenvererbungsgraph}
Die Vererbungsbeziehung zwischen den Klassen eines Schemas kann durch einen Klassenvererbungsgraphen über diesen Klassen nach Definition 5.9 dargestellt werden.

Die folgende Invariante ist eigentlich nicht notwendig, sie macht den Vererbungsgraphen allerdings übersichtlicher und verändert de facto nichts an der Semantik des Vererbungsgraphen.

Invariante 5.5 {minimaler Klassenvererbungsgraph}
Keine Klasse ist gleichzeitig direkte und echte indirekte Oberklasse einer anderen Klasse.

Formal: $\nexists c_1, c_2 \in cids : (c_1 <_c^1 c_2 \wedge \exists c \in cids : c_1 <_c c <_c c_2)$

Die Klassenvererbungsstruktur ist damit in dem Sinne minimal, daß nicht gleichzeitig eine direkte Kante und ein indirekter Pfad von einem Knoten c_1 zu einem Knoten c_2 existieren dürfen. Die Vererbungsbeziehung über die direkte Kante wäre in einem solchen Fall überflüssig.

Invariante 5.6 {vollständige Vererbung, vergleiche ORION-Invariante 4.4}
Eine Klasse erbt sämtliche Attribute und Methoden all ihrer Oberklassen, es sei denn Invariante 5.1 oder 5.2 würden dadurch verletzt. Wenn zwei Attribute verschiedener direkter Oberklassen derselben Herkunft sind, so muß nur eines davon geerbt werden.

Die folgende Invariante 5.7 erlaubt die kovariante Redefinition von Attributen.[63] Damit erhöht sich einerseits die Flexibilität des Systems, andererseits geht jedoch die strenge statische Typisierung von Attributen verloren. Daher wird diese Form des Überschreibens von den meisten

[63] Der Verlust der Typsicherheit durch die kovariante Redefinition von Attributen wurde bereits von William Cook [Coo89] bei seiner Arbeit mit der Programmiersprache Eiffel [Mey92] und von Luca Cardelli [Car84] untersucht.

Sprachen nicht unterstützt. Alternativ wird, u.a. von C++, das Prinzip der Subobjekt-basierten Vererbung (engl. *subobject-based inheritance*) verwendet. Dabei wird ein Objekt einer Unterklasse als zusammengesetzte Instanz betrachtet, deren geerbte Eigenschaften ein eigenes (Sub-) Objekt (ähnlich Komponentenobjekten) bilden, das konsequenterweise auch als solches verwendet werden kann. So kann z.B. der Konstruktor einer Oberklasse aufgerufen werden, der dann auch nur das betreffende Subobjekt initialisiert. Diese Betrachtungsweise hat folgende Konsequenzen:

- Die Redefinition von Attributtypen ist nicht möglich.
- Von verschiedenen Herkunftsklassen (engl. *origin classes*) geerbte Attribute (oder Methoden) müssen getrennt bleiben, womit die geerbten, miteinander in Konflikt stehenden Namen bedeutungslos werden.

Invariante 5.7 $\left\{\begin{array}{l}\textbf{strukturelle Typkonformität der Klassenvererbung,}\\ \textbf{vergleiche ORION-Invariante Typkompatibilität 4.5}\end{array}\right\}$

Die Klassenhierarchie $(cids, <_c^1)$ *eines Schemas heißt* strukturell typkonform, *wenn sie konsistent mit der Ober- und Untertypbeziehung* $(types, <_t)$ *nach Definition 5.7 ist. Wenn ein Attribut* a' *einer Klasse* c' *von einem Attribut* a *einer Oberklasse* c *von* c' *geerbt wird, dann muß der Typ von* a' *demzufolge ein Untertyp des Typs von* a *sein. Formal:* $\forall c_1, c_2 \in cids : c_1 <_c^1 c_2 \Rightarrow type(c_1) \leq_t type(c_2)$

Weiterhin dürfen in einer Klasse keine Überschreibungen von Attributen vorgenommen werden, die nicht geerbt werden, sondern lokal angelegt sind.

Schließlich muß der (Default-) Wert eines (Klassen-) Attributes dem Typ dieses Attributes entstammen.

Hierbei ist zu beachten, daß die Untertypbeziehung reflexiv ist. Invariante 5.7 erlaubt demzufolge, daß ein Attribut einer Klasse c in einer Unterklasse c' von c mit genau demselben Typ redefiniert wird, den c' ohnehin erben würde. Eine solche *überflüssige* Redefinition hat keinerlei Auswirkung auf die Klassenstruktur, ist jedoch trotzdem zulässig, weil Schemaänderungen, die sich aus mehreren Primitiven zusammensetzen, diese erfordern können.

Man beachte im folgenden, daß ein Attribut a im COAST-Modell durch eine *pid* identifiziert wird und geerbte bzw. redefinierte Attribute dieselbe *pid* besitzen. Benutzer eines Datenbanksystems können die Komponenten des Schemas allerdings nur über deren Namen ansprechen. Daher verwenden wir im folgenden den Namen *aname* eines Attributes a. Dieser ist dann allerdings nicht eindeutig und erlaubt i.Allg. keine Rückschlüsse auf mögliche Vererbungsbeziehungen zwischen gleichnamigen Attributen.

Neben der beschriebenen Substitutionsproblematik, die schon bei der Einfachvererbung auftritt, muß bei der Mehrfachvererbung eine Strategie angewendet werden, wie potentielle Widersprüche zwischen den von verschiedenen direkten Oberklassen geerbten Attributmengen aufgelöst werden. Konflikte können immer dann entstehen, wenn verschiedene direkte Oberklassen $c_1, \ldots, c_n$ einer Klasse c gleichnamige Eigenschaften besitzen. Wir gehen hier nur auf strukturelle Eigenschaften, d.h. auf Attribute ein. Wie schwerwiegend das mit einem Vererbungskonflikt eines Attributes a verbundene Problem ist, hängt von verschiedenen Fragen ab:

- Existiert eine gemeinsame Oberklasse c_x von $c_1, \ldots, c_n$, die das Attribut a definiert, d.h. haben die Attribute einen gemeinsamen Ursprung (eine gemeinsame Herkunft), oder wurde das Attribut verschiedenen Klassen unabhängig voneinander hinzugefügt?
- Wenn ein solches c_x existiert, blieb der Typ von a in den $c_1, \ldots, c_n$ im Vergleich zu c_x unverändert, oder wurde a in mindestens einem c_i spezialisiert?

- Hat das Attribut a in $c_1, \ldots, c_n$ denselben Typ, oder wurden dem Attribut in verschiedenen Klassen unterschiedliche Typen zugeordnet?

- Wenn a in $c_1, \ldots, c_n$ verschiedene Typen hat, existiert ein gemeinsamer Untertyp dieser verschiedenen Typen, oder nicht? Wenn ja, ist dieser gemeinsame Untertyp der Typ eines der $c_i.a$ oder existiert ein solcher Typ an anderer Stelle im Schema oder müßte er erst (automatisch oder manuell) definiert werden?

Entsprechend der genannten Fragen kann ein Objektmodell in mehr oder weniger Fällen das Anlegen der Klasse c als direkte Unterklasse von $c_1, \ldots, c_n$ erlauben. Einen weiteren Freiheitsgrad hat ein Objektmodell bei der Wahl einer Strategie zur Behebung noch zulässiger Konflikte.

Wir stellen hier einige Alternativen zur Behandlung von Konflikten vor. Dabei beginnen wir mit der restriktivsten Handhabung und führen dann schrittweise immer weitergehende Möglichkeiten ein, bis wir schließlich bei einer sehr flexiblen Variante ankommen. Dabei ist jedoch zu beachten, daß die Spezifikation des Modells und insbesondere der später darauf anzuwendenden Schemaänderungsprimitive umso aufwendiger und komplizierter werden, je flexibler das Modell der Mehrfachvererbung ist. Diese Komplexität beschränkt sich dabei nicht nur auf die Implementierung, sondern wird auch dem Schemaentwickler sichtbar, so daß wir hier zwischen Flexibilität einerseits und Verständlichkeit andererseits abwägen müssen. Wir betrachten im folgenden den Fall eines Attributes a, daß eine Klasse c von mehreren direkten Oberklassen $c_1, \ldots, c_n \in dpred(c)$ $(n > 1)$ erbt.

- Die restriktivste, wenngleich einfachste Form der Behandlung von Vererbungskonflikten ist deren generelle Vermeidung durch die Beschränkung auf maximal eine direkte Oberklasse je Klasse. Dies ist das Modell der Einfachvererbung und stellt eine erhebliche Einschränkung der Modellierungsmächtigkeit eines Objektmodells dar. Zum einen existieren zahlreiche Fälle, in denen die Mehrfachvererbung für einzelne Klassen sinnvoll eingesetzt werden kann, wie etwa für Kombinationen aus Studenten und Angestellten (HiWi), aus schwimmenden und rollenden Fahrzeugen (Amphibienfahrzeug) oder aus Dokumententeilen, die gleichzeitig in verschiedenen Medien erscheinen. Zum anderen wird die Mehrfachvererbung häufig eingesetzt, um vielen, höchst unterschiedlichen Klassen eine gemeinsame, weitere Eigenschaft hinzuzufügen, indem diese generelle Eigenschaft von einer separaten Klasse (engl. *mix-in*) realisiert und dann von den diese Eigenschaft benötigenden Klassen neben ihren sonstigen Oberklassen, mit denen sie in einem engen semantischen Zusammenhang stehen, geerbt wird. Zu den generellen Eigenschaften, die oft auf dem beschriebenen Wege realisiert werden, gehören beispielsweise die Möglichkeiten, ein Objekt einer Klasse persistent zu machen, es zu versionieren, es graphisch darstellen oder es in einer verteilten Umgebung einsetzen zu können.

- Eine konzeptionelle Erweiterung im Vergleich zu dem zuvor beschriebenen generellen Verbot der Mehrfachvererbung besteht darin, diese zumindest in denjenigen Fällen zuzulassen, in denen tatsächlich keine Vererbungskonflikte auftreten. Damit werden Konflikte nicht mehr grundsätzlich verhindert, sondern ihr Auftreten wird in jedem Einzelfall separat geprüft. Dies kann für die Vererbung der zuvor beschriebenen generellen Eigenschaften bereits ausreichen. Für die spezifischen Anwendungen wird dies jedoch häufig nicht der Fall sein, da die direkten Oberklassen einer mehrfach erbenden Klasse c in der Regel eine gemeinsame Oberklasse mit Eigenschaften besitzen, die c dann indirekt über mehrere direkte Oberklassen erbt. Ein Beispiel hierfür stellt der in einem Angestelltenverhältnis stehende Student dar. Die mehrfach erbende Klasse `HiWi` erbt von ihren direkten Oberklassen `Student` und `Employee`, welche beide (direkt oder indirekt) Attribute wie `name`

oder `date_of_birth` von ihrer gemeinsamen Oberklasse[64] `Person` erben. Dieser spezielle Fall eines Vererbungskonfliktes wird *wiederholte Vererbung* (engl. *repeated inheritance* oder auch *fork-join inheritance*) genannt und zeichnet sich dadurch aus, daß ein und dasselbe Attribut auf verschiedenen Pfaden vererbt wird.

- Der nächste Schritt besteht darin, die Fälle der wiederholten Vererbung zuzulassen. Diese können nämlich sehr einfach aufgelöst werden, wenn man unterstellt, daß es sich aufgrund der eindeutigen Herkunft des den Konflikt auslösenden Attributes immer um genau dieselbe Eigenschaft handelt und diese demzufolge auch nur einmal in der mehrfach erbenden Klasse c enthalten sein soll. Diese Annahme ist etwa in dem oben gegebenen Beispiel des in einem Angestelltenverhältnis stehenden Studenten sicher sinnvoll. Es handelt sich dabei nur um eine Person, die demzufolge auch nur einen Namen und ein Geburtsdatum hat und nicht zwei.

  Ein offenes Problem ist nun die Bestimmung des Typs des wiederholt geerbten Attributes. Hier ist die restriktivste Möglichkeit auch wieder die einfachste. Sie besteht darin, wiederholt geerbte Attribute nur dann zuzulassen, wenn deren Typ in keiner der beteiligten Klassen spezialisiert wurde. Damit hat das Attribut in allen (direkten und indirekten) Oberklassen der mehrfach erbenden Klasse denselben Typ und dieser kann dann sehr einfach auch für das wiederholt geerbte Attribut der Klasse c verwendet werden.

- Eine weitere Verallgemeinerung wird erreicht, wenn man sich damit begnügt, daß ein mehrfach geerbtes Attribut in den direkten Oberklassen $c_1, \ldots, c_n$ von c denselben Typ haben muß. Dieser kann dann zwar von dem Typ des Attributes in seiner Ursprungsklasse, d.h. in der gemeinsamen Oberklasse von $c_1, \ldots, c_n$ abweichen; da der Attributtyp jedoch in sämtlichen in Konflikt stehenden direkten Oberklassen derselbe ist, kann dieser wiederum sehr unkompliziert auch für das Attribut in der Klasse c verwendet werden.

- Soll weiterhin auch der Fall verschiedener Typen des in Konflikt stehenden Attributes in den direkten Oberklassen von c erlaubt sein, so kann die explizite Redefinition dessen Typs in der Klasse c gefordert werden. Dieser Typ muß dann Invariante 5.7 folgend spezieller sein als die Typen des betrachteten Attributes in den direkten Oberklassen von c.

- Ohne die Forderung nach einer expliziten Redefinition in Konflikt stehender Attribute kann man auskommen, wenn man aufgrund der Typen von Attribut a in $c_1, \ldots, c_n$ automatisch einen Typ bestimmen kann, der zumindest genauso speziell ist wie jeder dieser Typen und der dann als Typ für $c.a$ verwendet wird. Dabei kann gefordert werden, daß dieser Typ sich unter den Typen von $c_1.a, \ldots, c_n.a$ befinden muß, oder, noch allgemeiner, daß er nur irgendwo im Schema definiert sein muß. Manche Objektmodelle definieren analog zu der Wurzelklasse **Objekt** der Vererbungshierarchie einen Blatttyp **bot** der Typhierarchie, der Untertyp aller Typen ist. In solchen Modellen ist zumindest die Existenz des bei der Auflösung von Vererbungskonflikten in der hier beschriebenen Form gesuchten Typs gesichert. Mit der Feststellung, daß in jedem Typ eine Form eines Nullwertes (wir hatten diesen in Definition 5.3 **nil** genannt) zulässig ist, enthält die Domäne des künstlich definierten Blatttyps zumindest einen Wert. Die Unterscheidung zwischen der Zahl 0, einer Null-Referenz, einem leeren String, einer leeren Menge, etc. geht dabei allerdings verloren. Über den Nullwert hinaus muß jeder Typ, und damit auch **bot**, einen zweiten Wert enthalten, der einen undefinierten Zustand darstellt. Dieser Wert **undefined** muß sich von jedem normalen Wert einer Domäne unterscheiden.

[64] Es sei an dieser Stelle explizit darauf hingewiesen, daß die direkten Oberklassen von c sehr wohl mehrere gemeinsame, speziellste Oberklassen besitzen können. Wir beziehen uns hier aber stets auf diejenige gemeinsame, speziellste Oberklasse, von der das betrachtete Attribut geerbt wird und diese muß eindeutig sein. Ansonsten wäre die gemachte Forderung nach (genau) einer (und damit eindeutig bestimmten) gemeinsamen Oberklasse nicht erfüllt.

- Ein weiterer Schritt ist nun, auf die bisher stets vorausgesetzte Existenz einer Ursprungsklasse, von der die $c_1, \ldots, c_n$ das den Konflikt verursachende Attribut a erben, zu verzichten. Dabei tritt allerdings die Frage auf, ob es sich bei den $c_i.a$ ($i \in \{1, \ldots, n\}$) denn überhaupt noch, wie bisher angenommen, um dasselbe Attribut handelt oder ob Attribute verschiedener Semantik ohne besondere Absicht denselben Namen erhalten haben. Während im ersten Fall wie oben beschrieben verfahren werden kann, müßten im zweiten Fall eigentlich mehrere Attribute a geerbt und für die Herstellung der Namenseindeutigkeit umbenannt werden. Da eine solche Umbenennung bei Auftreten eines Vererbungskonfliktes allerdings auch in den dann verschiedenen Herkunftsklassen der verschiedenen Attribute gleichen Namens vorgenommen werden kann, kann konzeptionell auf diese Form der Ausnahmebehandlung verzichtet werden.[65]

- Auf die Konsequenzen der von ORION verfolgten Strategie, den direkten Oberklassen von c eine benutzerdefinierte Ordnung aufzuprägen und im Falle nicht auflösbarer Konflikte für $c.a$ einfach den Typ von a in der entsprechend der Ordnung ersten Oberklasse von c, welche das Attribut a enthält, zu verwenden, waren wir bereits am Ende von Abschnitt 4.5.3.7.3 eingegangen. Dabei hatten wir insbesondere festgestellt, daß dabei Invariante 5.7 (bzw. die ORION-Invariante Typkompatibilität 4.5) verletzt wird und damit die Substituierbarkeit von Objekten durch Objekte von Unterklassen verloren geht.

- In C++ wird so verfahren, daß in verschiedenen Klassen eines Objektes trotz der zwischen ihnen bestehenden Vererbungsbeziehungen unter demselben Namen verschiedene Attribute angesprochen werden. Dies mag im Umfeld der Programmiersprachen sinnvoll erscheinen, ist es jedoch für Datenbanksysteme nicht.

Wir gehen in der folgenden Invariante davon aus, daß jedem Attribut eine Identität, die sog. PropertyID (*pid*), zugeordnet wird. Diese bestimmt die Herkunft des Attributes, also die Klasse, in der das Attribut lokal angelegt (und nicht geerbt) wurde.

Invariante 5.8 {Mehrfachvererbung}
Geerbte oder überschriebene Attribute einer Klasse c, also genau diejenigen Attribute von c, die denselben Namen haben wie ein Attribut einer direkten Oberklasse von c, haben auch dieselbe PropertyID (pid) wie in dieser Oberklasse. Daraus ergibt sich insbesondere, daß alle gleichnamigen Attribute in direkten Oberklassen einer Klasse c dieselbe pid haben.

Weiterhin gilt umgekehrt, daß alle Attribute mit derselben pid auch denselben Namen haben.

5.2.4.1.4 Invarianten bezüglich des Verhaltens

Die beiden hier abschließend zu besprechenden Invarianten regeln das Überladen und das Überschreiben von Methoden.

Wir präzisieren zunächst die in Definition 5.14 gemachte Forderung nach der Existenz einer Funktion *impl*. Diese Funktion hat die Aufgabe, bei Eintreffen einer Nachricht, d.h. beim Aufruf einer Methode, aus den passenden Methodenimplementierungen genau eine auszuwählen, die dann auszuführen ist. Aufgrund des Überladens von Methoden kann es nämlich sein, daß mehrere der vorhandenen Implementierungen passen.

[65] In der Praxis kann die Möglichkeit, ein Attribut beim Auftreten eines Vererbungskonfliktes erst in der den Konflikt verursachenden, mehrfach erbenden Klasse c durch Umbenennung zu beseitigen, jedoch durchaus sinnvoll sein. Dies ist nämlich dann der Fall, wenn die Ursprungsklassen der verschiedenen a_i in von dritter Seite bezogenen Klassenbibliotheken enthalten sind, die zwar durch Erzeugung von Unterklassen spezialisiert aber nicht selbst verändert werden können. Wir verzichten hier auf die dargestellte Möglichkeit, da wir die Konsequenz, daß nämlich dasselbe Attribut in verschiedenen Klassen verschiedene Namen haben kann, als sehr unschöne Eigenschaft eines Objektmodells ansehen.

Invariante 5.9 $\left\{\text{Überladen von Methoden}\right\}$
Gegeben sei die Menge $M_n = \{m_1, \ldots, m_{q_n}\}$ *aller Methoden einer Klasse* c *mit Namen* $mname$ *und mit* n *Eingabeparametern. Die Signatur* $sig(m_i)$ *einer Methode* $m_i \in M_n$ *sei gegeben durch* $cid : t_1^i \times \cdots \times t_n^i \rightarrow t_{n+1}^i$.

Das Überladen von Methoden ist erlaubt, d.h. eine Klasse kann mehrere Methoden mit demselben Namen enthalten, wenn in jeder Situation maximal eine passende Methode $m_k \in M_n$ *gefunden werden kann. Dazu muß mindestens eine der folgenden Bedingungen erfüllt sein:*

- *Es existiert nur höchstens eine Methode namens* $mname$ *mit der passenden Parameteranzahl. Formal:* $|M_n| \leq 1 \; \forall n \in \mathbb{N}$. *Dies bedeutet, zwei gleichnamige Methoden derselben Klasse unterscheiden sich stets durch die Anzahl ihrer Eingabeparameter. In diesem Fall kann die zu verwendende Methode bereits statisch (d.h. zur Übersetzungszeit) ermittelt und gebunden werden.*

- *Existieren mehrere Methoden mit* n *Eingabeparametern* ($|M_n| > 1$), *so betrachten wir eine beliebige Kombination von* n *Parametertypen* $t_1^p, \ldots, t_n^p$. *Damit schränken wir die Kandidatenauswahl auf die Menge der anwendbaren Methoden* $M_n^p \subseteq M_n$ *ein:* $M_n^p := \{m_i \in M_n | t_j^p \leq_t t_j^i (\forall j \in \{1, \ldots, n\})\}$.

  Gilt $t_j^p \leq_t t_j^i$ *für ein* $m \in M_n$ *und ein* $j \in \{1, \ldots, n\}$ *nicht, so ist* m *nicht auf die angenommene Kombination von Parametertypen anwendbar* ($m \notin M_n^p$), *weil beim* j*-ten Parameter zur Laufzeit ein Typfehler auftreten könnte.*

  Für jede anwendbare Methode m_k *gilt, daß die Typen ihrer Parameter Obertypen der betrachteten Typen* $t_1^p, \ldots, t_n^p$ *sind und* m_k *somit ohne Typfehler auf solche Parameter angewendet werden kann.*

  Es muß nun gelten:

$$\begin{aligned} &\exists m_k \in M_n^p : \\ &\forall m_l \in M_n^p, l \neq k : \quad ((\exists i \in \{1, \ldots, n\} : t_i^l \neq_t t_i^k) && (1) \\ &\qquad\qquad \wedge \; (\forall i \in \{1, \ldots, n\} : t_i^k \leq_t t_i^l)) && (2) \end{aligned}$$

  - *Zeile (1) besagt, daß sich die Signatur von* m_k *zumindest bezüglich des Typs eines Parameters von den Signaturen sämtlicher anderer Methoden aus* M_n *unterscheiden muß.*
  - *Zeile (2) besagt, daß jede von* m_k *verschiedene, anwendbare Methode* $m_l \in M_n^p$ *nicht besser (und wegen (1) sogar echt schlechter) zu den betrachteten Parametertypen* $t_1^p, \ldots, t_n^p$ *paßt.*

  Die Parameter der Methoden besitzen also unterschiedliche Typen, so daß beim dynamischen Binden zur Laufzeit eindeutig eine der Methoden ausgewählt werden kann.[66] *Es sei hier explizit darauf hingewiesen, daß die Typen des Rückgabewertes von Methoden nicht zu deren Unterscheidung im obigen Sinne herangezogen werden.*

  In diesem Fall wird eine Nachricht mit Parametern der statischen Typen[67] $t_1^p, \ldots, t_n^p$ *an die Methode* $m_k \in M_n$ *gebunden, da keine Methode* $m_l \in M_n$ *existiert, die besser* paßt. *Dies*

[66] Im Englischen werden die Begriffe *multiple dispatching* oder *multi-methods* verwendet, um auszudrücken, daß die Auswahl einer dynamisch zu bindenden Methode nicht nur vom Empfänger einer Nachricht, sondern auch von anderen Objekten abhängt.

[67] Mit *dynamischen Parametertypen* sind hier die dynamischen Typen (siehe Abschnitt 2.1.5) der als Parameter übergebenen Variablen gemeint.

würde nämlich voraussetzen, daß deren sämtliche Typen mindestens so speziell sind wie die von m_k, einer davon jedoch sogar spezieller ist und daß die Methode trotzdem noch anwendbar ist.

Zur Problematik des Überladens von Methoden sind an dieser Stelle noch zwei Hinweise angebracht:

- In Programmiersprachen genügt es, die oben beschriebene Möglichkeit einer eindeutigen Zuordnung einer Methode nur für diejenigen Kombinationen von Parametertypen zu fordern, die in dem zu übersetzenden Programm auch tatsächlich vorliegen. Im Gegensatz dazu ist bei Datenbankschemata die Forderung wie oben geschehen für jede beliebige Kombination von Parametertypen zu erheben, da noch gar nicht bekannt ist, auf welche Art und Weise unabhängig erstellte Applikationen auf die angebotenen Methoden zugreifen werden.

- Die Prüfung der oben angegebenen Bedingungen für überladene Methoden kann durch die Mehrfachvererbung an unerwarteten Stellen notwendig werden. Seien beispielsweise drei nicht durch Vererbung zueinander in Beziehung stehende Klassen C, X und Y gegeben, und dabei eine Methode m in C überladen mit den Signaturen (X, Y) und (Y, X). In dieser Situation sind die obigen Bedingungen offenbar erfüllt. Wird nun jedoch eine Klasse Z als gemeinsame Unterklasse von X und Y erstellt, dann ist die zweite der obigen Bedingungen für die Parametersignatur (Z, Z) verletzt, d.h. es entsteht ein Fehler in der Klasse C.

Eine Klasse kann zwei Methoden derselben Herkunft und Signatur auf verschiedenen Pfaden erben. In solchen Fällen ist es erforderlich, daß die Klasse die Methode lokal überschreibt.

Nachdem wir in der vorausgegangenen Invariante für die Möglichkeit einer eindeutigen Auswahl unter überladenen Methoden gesorgt haben, schränken wir das Überschreiben von Methoden nun so ein, daß zur Laufzeit keine Typfehler mehr auftreten können (strenge Typisierung). Für das Überschreiben von Methoden bestehen grundsätzlich drei verschiedene Varianten:

- Das *Verbot jeglichen Überschreibens von Methoden mit Veränderung der Signatur* bedeutet die größte Einschränkung der Modellierungsmächtigkeit. Diese, in [ES94] als *strong nonvariance* bezeichnete Variante ist jedoch die einfachste Lösung und garantiert dazu strenge Typsicherheit, so daß sie in der Programmiersprache C++ und in Systemen wie ONTOS [Ont91] und ODE [AG89] Verwendung findet. Die etwas allgemeinere Variante der einfachen *nonvariance* erlaubt wenigstens die Spezialisierung des Rückgabetyps.

- Beim *kontravarianten Überschreiben* dürfen die Typen von Eingabeparametern einer Methode in Unterklassen nur verallgemeinert und der Typ des Ergebnisses nur spezialisiert werden. So können weder bei der Übergabe von Parametern beim Methodenaufruf noch bei der Rückgabe eines Ergebnisses Typkonflikte auftreten, weil Methoden in Unterklassen grob gesagt mehr Eingaben akzeptieren und höchstens eine Teilmenge der erwarteten Ergebnisse liefern.

- Beim *kovarianten Überschreiben* dürfen die Typen sowohl der Eingabeparameter als auch die der Ergebnisse in Unterklassen nur spezialisiert werden. Diese Variante wird als die natürlichste angesehen und daher beispielsweise sowohl in der Programmiersprache Eiffel [Mey92] als auch im Datenbanksystem O_2 (siehe Abschnitt 4.3.4.2) verwendet. Auf zur Laufzeit ggf. auftretende Typfehler können entsprechende Analysewerkzeuge aufmerksam machen.

Da wir uns in dieser Arbeit auf strukturelle Aspekte konzentrieren, verwenden wir hier das Modell der Kontravarianz. Außerdem verbieten wir der Einfachheit halber Überschreibungen von Methoden durch Attribute und umgekehrt.

Invariante 5.10 $\left\{ \begin{array}{l} \textbf{verhaltensmäßige Typkonformität der Klassenvererbung,} \\ \textbf{Überschreiben von Methoden} \end{array} \right\}$

Das Überschreiben von Methoden ist erlaubt, d.h. eine Klasse kann eine geerbte Methode durch eine eigene ersetzen, wenn die beiden folgenden Bedingungen erfüllt sind.

- *Es besteht Typkompatibilität zwischen überschriebenen Methoden entsprechend dem Modell der Kontravarianz:*

  $\forall c, c' \in cids, c' \leq_c c$ *und* $sig, sig' \in sig(m)$ *mit* $m \in mnames$ *und* $sig : c \times t_1 \times \cdots \times t_n \rightarrow t$, $sig' : c' \times t'_1 \times \cdots \times t'_n \rightarrow t'$ *gilt:* $t_i \leq_t t'_i$ *für alle* i $(1 \leq i \leq n)$ *und* $t' \leq_t t$

- *Methoden dürfen nicht durch Attribute überschrieben werden und Attribute dürfen nicht durch Methoden überschrieben werden; beide Fälle schließen insbesondere Methoden ohne Parameter ein.*

Wie zahlreiche andere, so basiert auch die formale Definition unseres Objektmodells auf der Einführung von Identifikatoren für die verschiedenen Komponenten des Modells (siehe Definition 5.1). Diese Identifikatoren sind für den Benutzer jedoch nicht sichtbar, so daß an der Benutzerschnittstelle etwas aufwendigere syntaktische Konstrukte angeboten werden müssen. Die Identifikation beispielsweise einer Objektversion kann formal durch einen Identifikator *ovid* erfolgen, während der Benutzer typischerweise Paare aus Objekt- und (objektbezogenem) Versionsidentifikator (oid, vid) verwendet. Ähnliche Betrachtungen sind für die Identifikation überschriebener Attribute und Methoden angebracht. Im Falle von Attributen genügt es stets, ihren Namen ergänzend durch den Namen einer beliebigen Oberklasse, in der der Attributname eindeutig ist, zu qualifizieren. Im Falle von Methoden stellt Markku Sakkinen [Sak92] fest, daß die herkömmliche Art der Qualifikation in C++ und einigen anderen Sprachen spätes Binden verhindert, eine Konsequenz die typischerweise nicht erwünscht ist. Daher schlägt er für C++ die folgende Syntax vor: $obj \rightarrow A..m()$. Dabei soll die speziellste, in der dynamischen Klasse von *obj* vorhandene Überschreibung der Methode $A :: m$ zur Ausführung kommen oder, falls keine Überschreibung vorhanden ist, $A :: m$ selbst.

Wenn sowohl Überschreiben als auch Überladen von Methoden erlaubt ist, dann sind auch Konstellationen, die beide Formen des Polymorphismus kombinieren, zu untersuchen. Seien fünf Klassen $C <_c B <_c A$ und $L <_c K$ gegeben und K definiere eine Methode $K.m(B)$ mit einem Parameter vom Referenztyp B. Nun ist zu spezifizieren, ob in der Unterklasse L gleichnamige Methoden $L.m(A)$, $L.m(B)$ und $L.m(C)$ definiert werden dürfen und wie dann das Binden einer Methode geschehen soll. Die Methode $L.m(A)$ könnte sowohl als Überschreibung von $K.m(B)$ als auch als Überladung von $L.m(B)$ und $L.m(C)$ angesehen werden. Jedoch muß beispielsweise im Falle einer Nachricht m an ein Objekt der Klasse L mit Parameter vom Referenztyp B entschieden werden, ob $K.m(B)$ oder $L.m(A)$ aufgerufen werden soll. Für solche Situationen könnte eine generelle Entscheidung getroffen werden, aber da keine der beiden Alternativen natürlicher oder sinnvoller ist, schlagen wir vor, die Entscheidung im Einzelfall dem Anwender zu überlassen.

5.2.4.2 Konsistenz unversionierter Schemata

Wir sind nun in der Lage, die strukturelle und verhaltensmäßige Konsistenz eines Schemas auf der Basis der im vorangegangenen Abschnitt 5.2.4.1 eingeführten Invarianten zu definieren.

Definition 5.17 {strukturell konsistentes (bzw. wohldefiniertes) Schema}
Ein strukturelles Schema $s_{struct} = (sid, sname, cids, <_c^1, types, type)$ *heißt* konsistent *(oder auch* strukturell konsistent *oder* wohldefiniert*)* (engl. conceptually consistent, well-formed)*, wenn die folgenden Bedingungen erfüllt sind:*

- *Eindeutigkeit der Namen und Identitäten der strukturellen Schemakomponenten entsprechend der Invarianten 5.1 und 5.2*
- *strukturelle Vollständigkeit des Schemas nach Invariante 5.3*
- *Invarianten bezüglich der Vererbung*
  - *Klassenvererbungsgraph nach Invariante 5.4*
  - *Minimalität des Klassenvererbungsgraphen nach Invariante 5.5*
  - *vollständige Vererbung nach Invariante 5.6*
  - *strukturelle Typkonformität nach Invariante 5.7*
  - *Mehrfachvererbung nach Invariante 5.8*

Definition 5.18 {verhaltensmäßig konsistentes Schema}
Ein verhaltensmäßiges Schema $s_{behav} = (sid, sname, cids, M, mnames, messg, impl)$ *heißt* konsistent *(oder auch* verhaltensmäßig konsistent*)* (engl. behaviourally consistent)*, wenn die folgenden Bedingungen erfüllt sind:*

- *Eindeutigkeit der Namen und Herkunft der verhaltensmäßigen Schemakomponenten (Methoden) entsprechend der Invarianten 5.1 und 5.2;*

  Die Eindeutigkeit der Namen der Methoden einer Klasse kann durch erlaubte Überladungen entsprechend Invariante 5.9 eingeschränkt sein.
- *verhaltensmäßige Vollständigkeit des Schemas nach Invariante 5.3*
- *verhaltensmäßige Typkonformität nach Invariante 5.10*
- *Methoden können in Unterklassen nur hinzugefügt werden.*
  Formal: $\forall c, c' \in cids, c \leq_c c' : messg(c') \subseteq messg(c)$
- *Für alle Nachrichten* $m \in messg(c)$ *an Objekte einer Klasse* c *existiert eine Oberklasse* $c' \in cids$ *von* c *(*$c \leq_c c'$*), die eine Implementierung* $impl(m, sig, c')$ *von* m *definiert.*

Der Konsistenzbegriff für unversionierte Schemata ergibt sich damit direkt aus den beiden vorangegangenen Definitionen.

Definition 5.19 {konsistentes Schema}
Ein Schema $s = (sid, sname, cids, <_c^1, types, type, M, mnames, messg, impl)$ *heißt* konsistent (engl. consistent)*, wenn die beiden folgenden Bedingungen erfüllt sind:*

- *Das strukturelle Schema* $s_{struct} = (sid, sname, cids, <_c^1, types, type)$ *ist konsistent nach Definition 5.17.*
- *Das verhaltensmäßige Schema* $s_{behav} = (sid, sname, cids, M, mnames, messg, impl)$ *ist konsistent nach Definition 5.18.*

5.2.5 Datenbanken

Eine Datenbank ist grob gesagt eine Menge von Klassenextensionen. Sie enthält gekapselte Objekte, deren Werte nur von Methoden der direkten Klasse des Objektes und von deren Unterklassen zugegriffen werden dürfen.

Definition 5.20 {Datenbank}
Eine (Objekt-) Datenbank (engl. object database) *db ist ein 7-Tupel* $db = (dbid, dbname, schemaid, oids, inst, val, named_variables)$ *mit den folgenden Komponenten:*

- $dbid \in dbId$ *ist der Identifikator der Datenbank.*
- $dbname \in dbNames$ *ist der benutzerdefinierte Name der Datenbank.*
- $sid \in sId$ *ist der Identifikator des Schemas, dem die Datenbank entspricht.*
- $oids \subseteq oId$ *ist die Menge der Identifikatoren der in der Datenbank enthaltenen Objekte.*
- $inst : cId \rightarrow 2^{oids}$ *ist eine totale Funktion, die jeder Klasse die Menge ihrer Objekte zuordnet und die die folgenden Bedingungen erfüllt:*
  - *Wenn zwei Klassen* $c, c' \in cId$ *keine gemeinsame Unterklasse besitzen,*[68] *dann haben ihre Extensionen keine gemeinsamen Objekte* $(inst(c) \cap inst(c') = \emptyset)$.
  - *Wenn eine Klasse c eine Unterklasse von c' ist* $(c \leq_c c')$, *dann ist ihre Extension eine Teilmenge der Extension von c'* $(inst(c) \subseteq inst(c'))$.
    (Seien $inst_1, inst_2 \in 2^{oids}$, *dann gelte die Teilmengenbeziehung* $inst_1 \subseteq inst_2$, *wenn* $\forall o \in inst_1 : o \in inst_2$.*)*
- $val : oids \rightarrow V^{oids}$ *ist eine Funktion, die jedem Objekt einen Wert des Typs seiner Klasse zuordnet. Formal:* $\forall c \in cids, \forall o \in inst(c) : val(o) \in dom(type(c))$
- $named_variables$ *ist eine Menge benannter Variablen. Diese ordnen Objekten der Datenbank textuelle Namen zu, die bei späteren Starts von Applikationen als Einstiegspunkte in die Datenbank dienen können.*

Zur Vereinfachung unserer Darstellungen definieren wir $oids(db) := oids$.

Hierbei ist zu beachten, daß durch die obige Definition der Funktion *val* das bei Definition 5.6 offen gebliebene Problem gelöst wird, nach dem wir bisher den Wertebereich einer Klasse lediglich als die Menge aller Objekte definieren konnten.

Definition 5.21 {Datenbank mit referenzieller Integrität}
Wir sagen eine Datenbank habe die referenzielle Integrität, *wenn alle Referenzen auf existierende Objekte verweisen, d.h. es kommen keine offenen Referenzen* (engl. dangling references) *vor.*
Formal: $\forall o \in oids(db) : val(o) \in V^{oids(db)}$

5.3 Erweiterung des COAST-Objektmodells auf versionierte Schemata

Nachdem wir im vorangegangenen Abschnitt 5.2 die grundlegenden Aspekte des COAST-Objektmodells formal eingeführt haben, wenden wir uns nun der Erweiterung dieses Modells auf

[68] Dies schließt insbesondere den Fall aus, daß c und c' (direkte oder indirekte) Ober- oder Unterklassen voneinander sind. Es gilt also $c \not\leq_c c' \wedge c' \not\leq_c c$.

versionierte Schemata zu. Dabei gehen wir auf Versionen von Schemata und von Klassen ein und ergänzen den bisherigen Konsistenzbegriff schließlich um Aspekte der Versionierung.

Abbildung 5.2 zeigt die wesentlichen Komponenten des um Schemaversionierung erweiterten Objektmodells (vergleiche Abbildung 5.1).

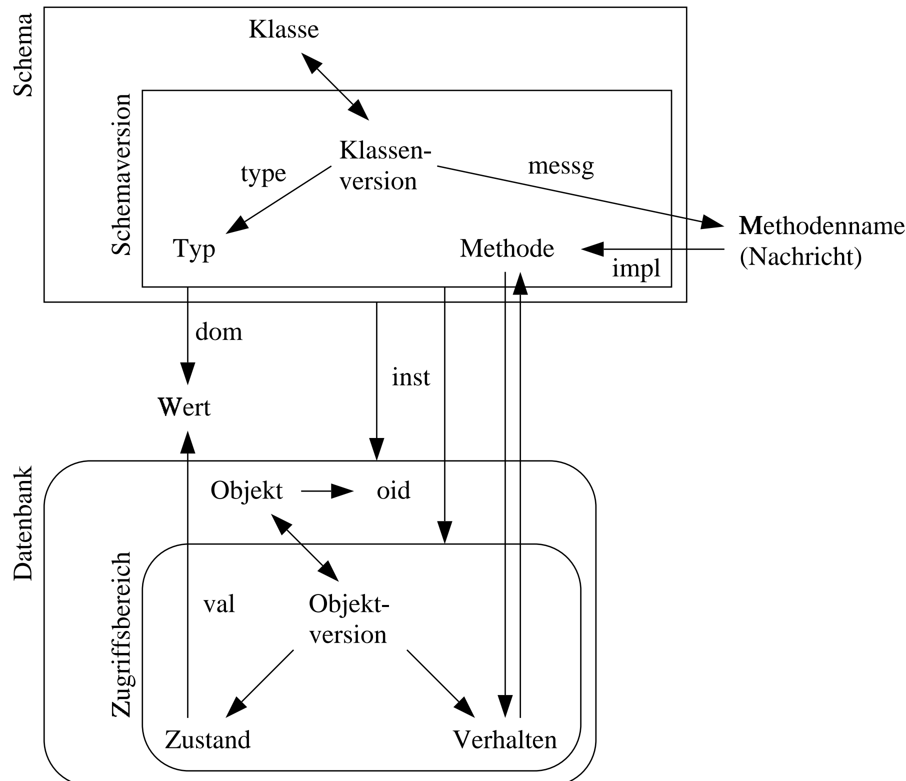


Abbildung 5.2: Die Komponenten des COAST-Objektmodells für versionierte Schemata.

5.3.1 Schemaversionen

Die grundlegende Idee des Schemaversionierungsansatzes ist die Bereitstellung mehrerer, gleichzeitig von verschiedenen Applikationen benutzbarer Schnittstellen zu einer gemeinsamen Datenbank. Diese Schnittstellen werden als *Schemaversionen* bezeichnet.

Eine Schemaversion stellt eine für eine bestimmte Benutzergruppe konsistente und komplette Definition einer Diskurswelt dar; sie ist also nicht global komplett. Jede Schemaversion hat einen eindeutigen Identifikator und einen Namen, der später z.B. bei der Integration von Klassen in neu abzuleitende Schemaversionen benötigt werden wird. Eine neue Schemaversion wird durch Ableitung von existierenden Versionen desselben Schemas angelegt. Nach Beendigung anschließender Schemaänderungen resultiert eine neue Schemaversion. Dabei kann jede beliebige Schemaänderung durchgeführt werden, solange eine Benutzergruppe damit eine sinnvolle Semantik verbindet und die Auswirkungen dieser Schemaänderung auf Datenbankebene spezifizieren kann.

Eine Schemaversion kann eine unter mehreren parallel entwickelten Entwurfsalternativen oder ein Zwischenergebnis einer sequentiellen Entwicklung darstellen.

In Erfüllung des technischen Teilzieles 3.19 der Transparenz entspricht eine Schemaversion aus der Sicht eines Applikationsentwicklers dem kompletten Datenbankschema eines herkömmlichen

Systems ohne Versionierung des Schemas. Aufgrund dieser Analogie können sämtliche bisher gegebenen Definitionen für unversionierte Schemata[69] auf Schemaversionen übertragen werden.

5.3.1.1 Spezifikation von Schemaversionen

Für die Spezifikation von Schemata wird eine Schemabeschreibungssprache (engl. *object definition language, ODL*) verwendet, die speziell auf versionierte Schemata zugeschnitten ist. Diese ODL bietet eine Menge sog. *Schemaänderungsprimitive* an, mit denen neue Schemaversionen erzeugt und modifiziert werden können. Die Schemaänderungen werden durch einen (oder mehrere) ausgewählte Schemaentwickler[70] spezifiziert. Zunächst soll hier auf grundsätzliche Aspekte der Spezifikation von Schemaversionen eingegangen werden. Eine genauere Beschreibung der Schemaänderungsprimitive der ODL folgt in Abschnitt 5.5.

Damit ein versioniertes Schema konsistent genannt werden kann, müssen zumindest all seine Schemaversionen entsprechend Definition 5.19 als konsistent gelten.[71] Der hier verwendete Konsistenzbegriff enthält jedoch keinerlei für einen bestimmten Anwendungsbereich eines OODBMS spezifischen Aspekte. Demzufolge stellt nicht jedes konsistente Schema notwendigerweise eine für einen Anwendungsbereich als sinnvoll oder natürlich geltende Modellierung dar. Die genannten Adjektive beschreiben die Qualität einer Modellierung aus der Sicht eines Spezialisten und entstammen einer Abstraktionsebene überhalb der des Konsistenzbegriffes. Der Konsistenzbegriff allein genügt damit also nicht als Gütekriterium für eine Modellierung, d.h. für ein Schema (bzw. eine Version davon).

Durch die Ausführung eines einzelnen Schemaänderungsprimitives kann lediglich eine elementare, quasi atomare Veränderung an einem Schema vorgenommen werden.[72] Diese garantiert zwar, daß ein konsistentes Schema stets auf ein konsistentes Schema abgebildet wird, aus der Sicht des Anwenders entsteht damit jedoch nur in den seltensten Fällen ein semantisch als sinnvoll oder natürlich zu bezeichnendes Modell der betrachteten Diskurswelt.

Aus der applikationsunabhängigen Konsistenzdefinition und der Atomizität der durch die Schemaänderungsprimitive bewirkbaren Modifikationen ergibt sich, daß der Anwender normalerweise mehrere dieser Schemaänderungen zusammenfassen und gemeinsam ausführen will, ohne daß nach Anwendung jedes einzelnen Primitives eine neue Schemaversion entsteht. Eine solche Zusammenfassung einer Liste von Primitiven beschreibt eine Schemaänderung auf der Abstraktionsebene der Anwender und bei ihrer Ausführung entsteht eine Schemaversion, die dann nicht nur konsistent sondern aus der Sicht zumindest einer Benutzergruppe auch sinnvoll, natürlich und vollständig ist.

Eine Zusammenfassung mehrerer Schemaänderungsprimitive modifiziert i.Allg. mehrere Komponenten einer Schemaversion. Entsprechend ist die Granularität einer Änderung eine komplette Schemaversion und nicht nur eine Klasse (siehe Teilziel 3.3), d.h. jede Schemaänderung resultiert in einer kompletten, neuen Schemaversion und nicht nur in neuen Versionen einer oder mehrerer einzelner Klassen. Dies liegt weiterhin darin begründet, daß einige der Schemaänderungsprimitive bereits alleine mehrere Klassen modifizieren und Schemaänderungen demzufolge nicht isoliert einer Klasse zugeordnet werden können. Schließlich vermeidet die Versionierung auf der Ebene des Schemas das bei der Beschreibung des technischen Teilzieles 3.3 bereits dargestellte

[69] Dies sind insbesondere die Definitionen von Klasse (5.8), Klassenvererbungsgraph (5.9), Ober- und Unterklasse (5.10), Schema (5.15) und Schemakonsistenz (5.19).

[70] Wir verwenden analog zu dem Begriff des Applikationsentwicklers die Bezeichnung *Schemaentwickler*, die synonym zu *Schemadesigner* verstanden wird.

[71] Wie bereits erwähnt entspricht ein unversioniertes Schema konzeptionell einer einzelnen Version eines versionierten Schemas, womit Definition 5.19 auch als Kriterium für die Konsistenz einer Schemaversion Anwendung finden kann.

[72] Dies wird bereits durch die Namensgebung „Schemaänderungs*primitiv*“ impliziert.

Konfigurationsproblem. Im Gegensatz dazu muß jeder Applikationsentwickler beim Klassenversionierungsansatz (siehe Abschnitt 4.5.3.3), selbst sicherstellen, daß er nur Klassenversionen benutzt, die untereinander konsistent sind. Dieses Konfigurationsproblem wird bei der Versionierung auf Schemaebene dadurch gelöst, daß die Konfigurationen (d.h. die Schemaversionen) dem OODBMS bereits zum Zeitpunkt der Schemaänderung bekannt gemacht werden, womit eine sofortige und automatische Konsistenzprüfung durchgeführt werden kann, von der alle diese Schemaversion benutzenden Applikationen profitieren.

Definition 5.22 {Schemaversion, sv}
Eine Schemaversion *sv repräsentiert eine bestimmte Ausprägung eines Datenbankschemas s, d.h. sie enthält einen kompletten Klassenvererbungsgraphen mit einer Wurzelklasse c_0 namens* **Object**.

Eine Schemaversion *sv ist ein Tripel $sv = (us, state, dt)$ mit den folgenden Komponenten:*

- *$us = (svid, svname, \ldots)$ ist ein unversioniertes Schema nach Definition 5.15, wobei jedoch der Identifikator aus der Menge $svId$ stammt*[73] *($svid \in svId$) und nicht aus sId und der Name analog aus $svNames$*[73] *($svname \in svNames$) und nicht aus $sNames$.*
- *$state \in \{frozen, unfrozen\}$ ist der Zustand der Schemaversion.*[74]
- *dt ist die Ableitungszeit* (engl. derivation time) *der Schemaversion.*

Zur Vereinfachung unserer Darstellungen definieren wir $svid(sv) := sid(us)$, $svname(sv) := sname(us)$, $state(sv) := state$ und $dt(sv) := dt$.

Die Schemaversion **RootSV** *ist definiert als $(sv_0, \textbf{RootSV}, frozen, 0)$. Wir werden die Schemaversion sv_0 im folgenden auch durch ihren Namen* **RootSV** *identifizieren.*

Da wir uns in dieser Arbeit fast ausschließlich mit einem einzigen Schema s befassen, verzichten wir zu Gunsten einer einfacheren Darstellung darauf, alle benutzten Symbole mit einem zusätzlichen Index s zur Identifikation des betroffenen Schemas zu versehen.

Im Gegensatz zum direkten Ansatz zur Unterstützung der Schemaevolution, wo Änderungen direkt an dem betroffenen Datenbankschema durchgeführt werden und somit dessen vorheriger Zustand verloren geht, erzeugen wir bei Ausführung einer Liste von Schemaänderungen jeweils eine neue Version des Schemas. So können im Laufe der Zeit zahlreiche Versionen eines Schemas entstehen, die gemeinsam in einem Container verwaltet werden. Wir werden für diesen Container in Definition 5.25 die Bezeichnung $sv(s)$ einführen.

Der Container $sv(s)$ enthält zu jedem Zeitpunkt sämtliche von einem Schema s vorliegenden Versionen. Darunter befindet sich insbesondere die systemdefinierte Schemaversion sv_0 namens **RootSV**.

5.3.1.2 Relative Spezifikation von Schemaversionen

Ein wichtiges Merkmal unseres Ansatzes ist, daß neue Schemaversionen nicht von Grund auf neu, d.h. absolut spezifiziert werden, sondern stets relativ zu bereits existierenden Schemaversionen (Teilziel 3.1). Wir sagen, eine neue Schemaversion werde von existierenden Versionen desselben Schemas *abgeleitet*. Dabei können, im Rahmen der neuen Schemaversion, existierende Klassen modifiziert oder gelöscht und neue Klassen hinzugefügt werden. Andere Schemaversionen bleiben von diesen Änderungen allerdings unberührt.

[73] Die Mengen $svId$ und $svNames$ hatten wir im Vorgriff bereits in Definition 5.1 eingeführt.
[74] Auf die Bedeutung der beiden Zustände werden wir in Abschnitt 5.5.2.4 näher eingehen.

Ein neu erzeugtes Schema enthält zunächst nur eine einzige, systemdefinierte Schemaversion, die den Identifikator sv_0 trägt und von der (zumindest) die erste benutzerdefinierte Schemaversion sv_1 abgeleitet wird. Da sv_0 keine für das Einsatzgebiet des OODBMS spezifischen, semantisch bedeutungsvollen Klassen enthält, ist die Ableitung einer Schemaversion von sv_0 vom Spezifikationsaufwand her mit einer Neudefinition gleichzusetzen. Aus konzeptioneller Sicht besteht jedoch kein Unterschied zwischen der Ableitung der ersten benutzerdefinierten Schemaversion sv_1 und späteren Spezifikationen von $sv_2, sv_3, \ldots$ Damit liegt der hier verwendeten Schemaversionierung nur ein einziges, durchgängig verwendetes Konzept zugrunde. Auf den Prozeß der Ableitung einer neuen Schemaversion werden wir in Abschnitt 5.4.1 genauer eingehen.

5.3.1.2.1 Die Schemaversionsableitungsbeziehung

Die ODL Spezifikation etabliert eine Schemaversionsableitungsbeziehung zwischen den Versionen eines Schemas. Entlang der Schemaversionsableitungsbeziehung werden Spezifikationen wiederverwendet (engl. *software reuse*). Weiterhin wird eine semantische Verbindung zwischen Schemaversionen etabliert, falls eine Klasse in zwei (oder mehr) Schemaversionen enthalten ist (d.h. sie wurde in einer Schemaversion erzeugt und in den anderen durch Integration eingebunden). Für Klassen und Eigenschaften, die in mehreren Versionen eines Schemas enthalten sind, wird sicher gestellt, daß Klassenextensionen und die Auswertung von Objekteigenschaften (desselben Objektes) durch Applikationen verschiedener Schemaversionen synchronisiert werden. Darauf werden wir in Kapitel 6 näher eingehen. Im Gegensatz zur direkten Schemaevolution wird durch diese Bindung die Vorgängerschemaversion nicht hinfällig.

Ein Konzept, das die relative Spezifikation neuer Schemaversionen erlaubt, ist der Beschränkung auf absolute Spezifikationen, z.B. durch den externen Ansatz,[75] vorzuziehen, da letzterer oftmals die gewünschte Semantik nicht darstellen kann. Die Verwendung des externen Ansatzes kann für die isolierte Definition des Zustandes einer Schemaversion ausreichend sein, jedoch muß für die Verwaltung mehrerer auf einer gemeinsamen Datenbasis aufsetzender Schemaversionen betrachtet werden, wie Schemaversionen voneinander abweichen oder übereinstimmen. Schemaversionen sind nicht unabhängig voneinander, sondern sie spezifizieren verschiedene Auffassungen bzw. Sichtweisen derselben Diskurswelt. Um eine Datenbank, die durch verschiedene Versionen eines Schemas (d.h. durch verschiedene Schemaversionskontexte) zugegriffen wird, in einem konsistenten Zustand zu halten, müssen Abhängigkeiten zwischen verschiedenen Schemaversionen erkannt, ausgedrückt und verwaltet werden können (Teilziel 3.8). Das ist unmöglich, wenn Schemaversionen absolut spezifiziert werden (siehe Beispiel in Abbildung 5.11). Daher müssen, um alle Aspekte einer Änderung in einem gemeinsamen Formalismus ausdrücken zu können, sowohl der Inhalt als auch die semantischen Abhängigkeiten in einer relativen Art und Weise spezifiziert werden. Da eine absolute Spezifikation entsprechend dem externen Ansatz der Schemaevolution (Teilziel 3.4) mitunter natürlicher sein mag, werden wir in Abschnitt 7.1.2.2 vorstellen, wie ein Werkzeug den Schemaentwickler bei der Spezifikation der semantischen Beziehungen zwischen absolut spezifizierten Schemaversionen unterstützen kann. Dabei ist das Werkzeug allerdings auf die Hilfe des Schemaentwicklers angewiesen.

Wie bereits beschrieben, werden neue Versionen eines Schemas s in der Regel nicht von Grund auf neu definiert. Stattdessen soll eine Möglichkeit angeboten werden, neue Versionen von s durch Veränderung bereits existierender Versionen aus $sv(s)$ *abzuleiten*. Diese Ableitungsbeziehung (**is_derived_from**) birgt für den Versionierungsansatz grundlegende Semantik über die Beziehung der Versionen eines Schemas untereinander. Diese ist sowohl auf Schema- als auch auf Objektebene von entscheidender Bedeutung. Auf Schemaebene wird die Ableitungsbeziehung benötigt, um die Evolution des Schemas und die dabei durchgeführten Veränderungen

[75] Der externe Ansatz für die Durchführung von Schemaänderungen wird von Odberg treffend als *copy-edit-commit-Prozeß* bezeichnet.

darzustellen. Dies entspricht im wesentlichen der Motivation für die Verwaltung der Ableitungsbeziehungen zwischen Versionen eines Objektes wie sie in Abschnitt 2.2 dargestellt wurde. Auf Objektebene werden die Beziehungen zwischen den Klassen verschiedener Versionen eines Schemas benötigt, um die Konvertierung der Objekte durchführen zu können. Zur Verwaltung der Ableitungsbeziehung zwischen den Versionen eines Schemas führen wir im folgenden den Schemaableitungsgraphen ein.

5.3.1.2.2 Die Schemaversionsableitungshierarchie

Ein weiteres Merkmal des hier vorgestellten Ansatzes ist, daß neue Schemaversionen von beliebigen existierenden Schemaversionen abgeleitet werden können. Dabei können insbesondere Teile verschiedener Versionen in eine neue Schemaversion integriert werden, so daß die graphische Darstellung der Ableitungsbeziehung zwischen den Versionen eines Schema i.Allg. einen gerichteten, azyklischen Graphen (DAG), den sog. *Schemaableitungsgraphen* ergibt. Dabei gibt es keine eindeutige *aktuelle* Schemaversion, wie das bei der Einschränkung der Ableitungsbeziehung auf eine lineare Anordnung der Schemaversionen der Fall wäre. Stattdessen können mehrere Blattknoten im Schemaableitungsgraphen existieren. Jede Schemaversion im Ableitungsgraphen ist zu jeder Zeit gültig und kann sowohl für die Ableitung neuer Schemaversionen als auch als Basis für die Implementierung neuer Applikationen dienen. Das steht im Gegensatz zum Grundgedanken der direkten Schemaevolution und auch demjenigen einiger Schemaversionierungsansätze, wo neuere Schemaversionen stets als besser angesehen werden und zum Teil neue Applikationen nur für die jeweils neueste Schemaversion implementiert werden dürfen.

In einigen Fällen kann eine neue Schemaversion als Korrektur bestehender Schemaversionen angesehen werden, womit sich die Anpassung existierender Applikationen an die neue Schemaversion empfiehlt. Es gibt in unserem Modell jedoch keinerlei Beschränkungen, die eine solche Anpassung erzwingen würden (Teilziel 3.9).

Odberg (siehe Abschnitt 4.5.3.6) erlaubt lediglich einen Schemaversions-Ableitungsbaum und geht damit bei der Ableitung einer neuen Schemaversion von genau einer Basisschemaversion aus, die zunächst komplett übernommen und dann verändert wird. Der Schemaversions-Ableitungsbaum kann dann jedoch mit **incorporate**-Beziehungen angereichert werden. Damit sind dann allerdings zwei verschiedene Typen von Beziehungen zu berücksichtigen. Weiterhin wird die Spezifikation damit asymmetrisch, da die sog. Basisschemaversion (engl. *base schema version*) anderen Schemaversionen, von denen über die **incorporate**-Beziehung integriert wird, vorgezogen wird.

Abbildung 5.3 stellt einen Schemaableitungsgraphen abstrahiert dar (eine detailliertere Darstellung eines Schemaableitungsgraphen befindet sich in Abbildung 5.10).

In der graphischen Darstellung werden Schemaversionen durch Dreiecke repräsentiert, die innen jeweils mit ihrem Identifikator beschriftet sind. Die Schemaversionsableitungsbeziehung von x nach y wird durch einen Pfeil (in dieser Richtung) wiedergegeben. Die Wurzel des durch die Schemaversionsableitungsbeziehung definierten Graphen ist die Basisschemaversion sv_0.

Da die Identifikatoren der Schemaversionen in Erzeugungsreihenfolge vergeben werden, ist in Abbildung 5.3 z.B. an sv_4 und an sv_7 zu erkennen, daß neue Schemaversionen nicht immer von Blattknoten, insbesondere nicht notwendigerweise von der jeweils neuesten Schemaversion abgeleitet werden müssen. Auch noch zu jedem späteren Zeitpunkt können neuere Schemaversionen von beliebigen Schemaversionen abgeleitet werden.

Definition 5.23 {**Schemaableitungsgraph,** SDG}
Ein Schemaableitungsgraph SDG über einer Menge von Schemaversionen *svids* (engl. schema derivation graph) *(kurz auch* Ableitungsgraph *genannt) ist ein verwurzelter, gerichteter, azyklischer und zusammenhängender Graph, dessen Knoten die Schemaversionen aus svids re-*

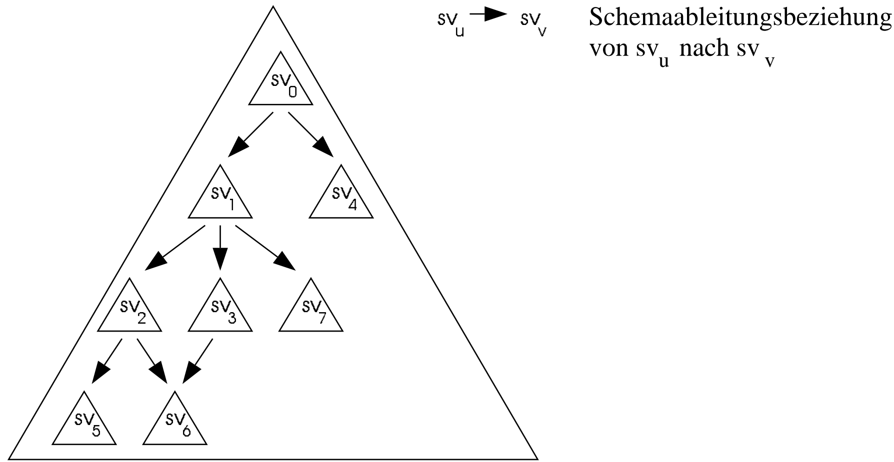


Abbildung 5.3: Beispiel eines Schemaversionsableitungsgraphen eines Schemas s mit sieben benutzerdefinierten Schemaversionen $sv_1, \ldots, sv_7$.

präsentieren und dessen Kanten die Ableitungsbeziehung zwischen diesen Schemaversionen widerspiegeln. Eine Kante von Schemaversion sv_v nach Schemaversion sv_u ($sv_u, sv_v \in svids$) bedeute dabei, daß sv_v von sv_u abgeleitet wurde.

Formal wird ein Schemaableitungsgraph dargestellt als ein Paar $SDG = (svids, <^1_{sv})$ mit den folgenden Komponenten:

- *$svids \subseteq svId$ ist die Menge der Identifikatoren derjenigen Schemaversionen, die zu dem Schemaableitungsgraphen gehören.*
- *$\leq_{sv} \subseteq svids^2$ ist eine Ordnungsrelation auf svids entsprechend Definition 2.3.*

Dabei müssen die folgenden Bedingungen erfüllt sein:

- *$sv_0 \in svids$*
- *Die systemdefinierte Schemaversion sv_0 ist die Wurzel des Schemaableitungsgraphen. Formal: $\forall sv \in svids : sv \leq_{sv} sv_0$*

Entsprechend Definition 2.4 verwenden wir auch die Symbole $<_{sv}$, $\leq_{sv}$, $\leq^1_{sv}$, $>_{sv}$, $\geq_{sv}$, $>^1_{sv}$ und $\geq^1_{sv}$ zur Darstellung der Ableitungsbeziehung.

Die zweite der obigen Bedingungen garantiert den Zusammenhang des Schemaableitungsgraphen.

Definition 5.24 $\left\{ \begin{array}{l} \textbf{(direkte) Vorgänger- und Nachfolgerschemaversionen,} \\ pred(sv),\ dpred(sv),\ succ(sv),\ dsucc(sv) \end{array} \right\}$

Wenn in einem Schemaableitungsgraphen $SDG = (svids, <^1_{sv})$ $sv_v \leq_{sv} sv_u$ ($sv_v <^1_{sv} sv_u$) gilt, dann heißt sv_u (direkte) Vorgängerschemaversion *von sv_v und sv_v heißt* (direkte) Nachfolgerschemaversion *von sv_u. Ferner führen wir die folgenden abkürzenden Schreibweisen für Teilmengen von svids ein:*

$$\begin{aligned} pred(sv) &:= \{sv' \in svids | sv \leq_{sv} sv'\} \\ dpred(sv) &:= \{sv' \in svids | sv <^1_{sv} sv'\} \\ succ(sv) &:= \{sv' \in svids | sv' \leq_{sv} sv\} \\ dsucc(sv) &:= \{sv' \in svids | sv' <^1_{sv} sv\} \end{aligned}$$

Definition 5.25 {versioniertes Schema, s}
Ein versioniertes Datenbankschema s ist ein 4-Tupel $s = (sid, sname, svids, <_{sv}^1)$ mit den folgenden Komponenten:

- *$sid \in sId$ ist der Identifikator des Schemas.*
- *$sname \in sNames$ ist der benutzerdefinierte Name des Schemas.*
- *$svids = \{sv_0, \ldots, sv_n\}$ sind die Identifikatoren von Schemaversionen nach Definition 5.22.*
- *$SDG = (svids, <_{sv}^1)$ ist ein Schemaableitungsgraph über svids nach Definition 5.23.*

Zur Vereinfachung unserer Darstellungen definieren wir $sid(s) := sid$, $sname(s) := sname$ und $sv(s) := svids$ als Symbol für die Menge aller $n+1$ Versionen eines Schemas s. Analog zu dem in Definition 5.15 eingeführten $classes(s)$ verwenden wir von nun an auch $classes(sv)$.

5.3.2 Klassenversionen

Eine neue Schemaversion wird durch eine Liste von Schemaänderungen erzeugt, die auf bestehende Schemaversionen angewendet wird. Dabei können Klassen hinzugefügt, gelöscht oder modifiziert werden.

Eine Klasse c einer existierenden Schemaversion sv_u kann benutzt werden, wenn eine neue Schemaversion sv_v abgeleitet werden soll. Beide Schemaversionen enthalten dann dieselbe Klasse c. Jedoch kann die Definition der Klasse c während der Ableitung von sv_v auch verändert werden, d.h. eine Klasse c kann in verschiedenen Schemaversionen in verschiedenen Ausprägungen vorliegen. Daher führen wir nun auch den Begriff der *Klassenversion* ein.

Eine Modifikation einer Klasse erzeugt eine veränderte Version dieser Klasse, d.h. es entstehen *Klassenversionen*. Eine veränderte Klassenversion entsteht,

- wenn neue Eigenschaften (Attribute oder Methoden) zu einer Klasse hinzugefügt werden oder wenn existierende Eigenschaften entfernt oder modifiziert werden oder
- wenn sich die Menge der (direkten) Oberklassen einer Klasse ändert oder
- wenn eine neue Version einer (möglicherweise indirekten) Oberklasse definiert wird (d.h. bei Änderung einer Klassenversion entstehen auch neue Versionen sämtlicher Unterklassen der modifizierten Klasse).

In allen anderen Fällen finden keine Veränderungen an Klassenversionen statt. Jedoch wird aus logischer Sicht trotzdem eine neue Klassenversion angelegt, wenn eine Klasse von sv_u beim Ableiten einer neuen Schemaversion sv_v verwendet wird.[76] Damit entsteht durch die Integration eine neue, mit der bisherigen Klassenversion identische Klassenversion in sv_v. Dies erleichtert uns den formalen Umgang mit Klassenversionen, da so jede Schemaversion ihre eigenen Klassenversionen besitzt und wir diese ohne weitere Umwege direkt identifizieren können (siehe Definition 5.26).

Eine neue Klassenversion kann nur im Kontext einer Schemaversion definiert werden. Einerseits können Versionen einer Klasse in verschiedenen Schemaversionen gleich (aber nicht identisch) sein, sofern diese Klasse (und ihre Oberklassen) zwischen den Ableitungen dieser Schemaversionen unverändert blieb(en). Andererseits können zwei verschiedene Versionen derselben Klasse genau dieselben Eigenschaften enthalten; dies kann z.B. dadurch verursacht sein, daß Attribute

[76] In der technischen Realisierung wird die Identität zweier Klassenversionen allerdings zur Verringerung des Platzbedarfes ausgenutzt.

entlang der Vererbungshierarchie verschoben wurden. Die Klassenversionen haben dann dieselben Eigenschaften,[77] sie unterscheiden sich jedoch dadurch, daß eine andere Teilmenge ihrer Eigenschaften lokal definiert (bzw. geerbt) ist. Schließlich ergibt jede Veränderung der Eigenschaften einer Klasse, also auch eine Änderung der Implementierung einer ihrer Methoden, eine neue Version der Klasse (und damit auch ihrer Unterklassen).

Eine aus Schemaversion sv_u in sv_v integrierte Klasse c bleibt allerdings unverändert, das heißt in sv_u und sv_v liegen identische Versionen der Klasse c vor, wenn sich lediglich von dieser Klasse referenzierte Klassen ändern. Somit kann sich eine unveränderte Klasse aufgrund veränderter Komponenten in einer neuen Schemaversion anders verhalten als zuvor.

Die Bedeutung einer Klassenversion liegt in der Festlegung einer Menge bestimmter Eigenschaften und deren Implementierungen. Eine Klassenversion beschreibt die Erwartungen an das Verhalten ihrer Objekte in der Schemaversion, in der die Klassenversion enthalten ist. Weiterhin beschreibt eine Klassenversion wie ein Objekt als Instanz dieser Klassenversion erzeugt wird. Klassenversionen werden jedoch nie explizit identifiziert, sondern stets indirekt über die Schemaversion und die Klasse, denen die Klassenversion angehört. Demzufolge setzt sich der Identifikator einer Klassenversion als ein Paar aus den Identifikatoren von Schemaversion und Klasse zusammen (siehe Definition 5.26). Weiterhin treten Klassenversionen nicht explizit in Erscheinung, um die Transparenz für die Applikationsentwickler aufrecht zu erhalten.

Die Erzeugung einer neuen Version einer Klasse kann Auswirkungen auf die Klienten dieser Klasse in der neuen Schemaversion haben. Diese müssen ggf. modifiziert werden, beispielsweise weil referenzierte Eigenschaften in der neuen Klassenversion nicht mehr enthalten sind. Damit müssen typischerweise die Methodenimplementierungen angepaßt werden, womit letztendlich doch (indirekt) auch neue Versionen der referenzierenden Klassen entstehen. Unterklassen erben Eigenschaften und demzufolge muß bei Änderung einer Oberklasse auch hier eine neue Klassenversion angelegt werden.

Eine Schemaversion kann damit auch als konsistente und explizit definierte Konfiguration von Klassenversionen betrachtet werden.

Definition 5.26 {Klassenversion, $sv.c$}
Eine Ausprägung einer Klasse c, die in einer bestimmten Version sv eines Datenbankschemas s vorliegt, heißt Klassenversion. *Sie wird formal als sv.c notiert.*

Aufgrund der Analogie zwischen einer unversionierten Klasse (siehe Definition 5.8) und einer Klassenversion muß in den bisher gegebenen Definitionen für unversionierte Schemata[69] der Begriff der Klasse durch den der Klassenversion ersetzt werden, um diese Definitionen auf Schemaversionen zu übertragen. Der neue Begriff der versionierten Klasse dient als Container zusammengehöriger Klassenversionen. Formal ist eine versionierte Klasse im Vergleich zu einer unversionierten zu einem Klassenidentifikator (cid) degeneriert.

Definition 5.27 {versionierte Klasse, c}
Eine versionierte Klasse *c ist eine nichtleere Menge von Klassenversionen* $sv_1.c, \ldots, sv_n.c$. *Sie wird durch den Identifikator cid der Klasse repräsentiert.*

Mit $cv(c)$ bezeichnen wir die Menge aller Versionen einer versionierten Klasse c.

5.3.2.1 Die Klassenableitungsbeziehung

Wenn eine Klassenversion $sv_v.c$ verändert oder unverändert aus einer Klassenversion $sv_u.c$ hervorging und sv_u eine direkte Vorgängerschemaversion von sv_v ist ($sv_v <^1_{sv} sv_u$), dann sagen wir

[77] Hier wird nur die Vereinigung lokal definierter und geerbter Eigenschaften betrachtet.

$sv_v.c$ sei von $sv_u.c$ bzw. von sv_u *abgeleitet*. Damit existieren zwei Wege wie eine Klasse c Bestandteil einer Schemaversion sv werden kann: Entweder die Klasse c wurde in sv neu definiert, womit $sv.c$ die erste Version dieser Klasse ist, oder sie wurde von einer bereits existierenden Version der Klasse c abgeleitet.

Definition 5.28 {**Erzeugerschemaversion einer Klasse,** $csv(c)$}
Die Erzeugerschemaversion $csv(c)$ einer Klasse c (engl. creator schema version) *ist die eindeutig bestimmte Version eines Schemas s, in der die Klasse c erzeugt wurde. In allen anderen Versionen von s, die die Klasse c enthalten, wurde diese direkt oder indirekt von* $csv(c).c$ *abgeleitet.*

Im Schemaableitungsgraphen kann eine Schemaversion mehrere direkte Vorgängerschemaversionen haben, von denen Schemakomponenten übernommen wurden. Eine einzelne Version einer Klasse betrachten wir jedoch als atomare Einheit und eine Klassenversion kann somit nur von einer anderen Version derselben Klasse abgeleitet werden. Darüber hinaus kann jede Schemaversion nur eine Version jeder Klasse enthalten. Eine Klassenversion kann jedoch von mehreren Nachfolgerschemaversionen integriert werden. Damit entsteht, wenn man die Betrachtung des Schemaableitungsgraphen auf die Ableitungsbeziehungen zwischen den Versionen einer einzelnen Klasse beschränkt, eine Baumstruktur, der sog. *Klassenableitungsbaum* (siehe Abbildung 5.4).

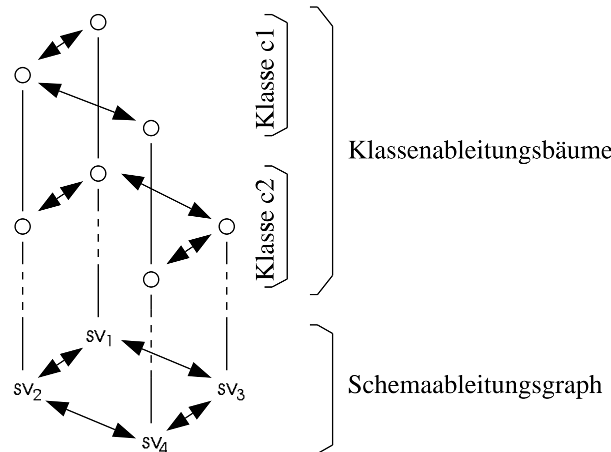


Abbildung 5.4: Der Ableitungsgraph eines Schemas s ergibt sich aus der Summe der Ableitungsbäume der Klassen von s.

Definition 5.29 {**Schemaversionen einer Klasse,** $sv(c)$}
Wir bezeichnen alle Versionen eines Schemas s, in denen eine Klasse c definiert ist, als Schemaversionen dieser Klasse. *Formal:* $sv(c) := \{sv \in sv(s) | \exists sv.c\}$.

Definition 5.30 {**Klassenableitungsbaum,** CDT}
Ein Klassenableitungsbaum CDT über einer Menge von Klassenversionen $cvids$ (engl. class derivation tree) *ist ein gerichteter Baum, dessen Knoten die Klassenversionen aus cvids repräsentieren und dessen Kanten die Ableitungsbeziehung zwischen diesen Klassenversionen widerspiegeln. Eine Kante von Klassenversion* cv_v *nach Klassenversion* cv_u ($cv_u, cv_v \in cvids$) *bedeute dabei, daß* cv_v *von* cv_u *abgeleitet wurde.*

Formal wird ein Klassenableitungsbaum dargestellt als ein Paar $CDT = (cvids, <^1_{sv,c})$ *mit den folgenden Komponenten:*

- $cvids \subseteq cvId$ *ist die Menge der Identifikatoren derjenigen Klassenversionen, die zu dem Klassenableitungsbaum gehören.*

- *Es handelt sich bei den cvids ausschließlich um Versionen derselben Klasse c, d.h.* $cvids = \{sv_1.c, \ldots, sv_n.c\}$.
- $<^1_{sv,c} \subseteq cvids^2$ ($\leq_{sv,c} \subseteq cvids^2$) *ist ein gerichteter Baum (eine Ordnungsrelation) auf cvids entsprechend Definition 2.21 (Definition 2.3).*

Dabei müssen die folgenden Bedingungen erfüllt sein:

- $csv(c) \in cvids$
- *Die Erzeugerschemaversion einer Klasse ist die Wurzel ihres Klassenableitungsbaumes. Formal:* $\forall cv \in cvids : cv \leq_{sv,c} csv(c)$

Da die Klasse **Object** *systemdefiniert ist, müssen wir die Ableitungsbeziehungen für sie explizit festlegen. Wir definieren daher, daß jede Version der Klasse* **Object** *direkt von* $sv_0.c_0$ *(***Object***) abgeleitet sei, d.h.* $\forall sv \in sv(s) - \{sv_0\} : sv.cv_0 <^1_{sv,c} sv_0.c_0$.

Entsprechend Definition 2.4 verwenden wir auch die Symbole $<_{sv,c}$, $\leq_{sv,c}$, $\leq^1_{sv,c}$, $>_{sv,c}$, $\geq_{sv,c}$, $>^1_{sv,c}$ *und* $\geq^1_{sv,c}$ *zur Darstellung der Ableitungsbeziehung.*

Definition 5.31 $\left\{\right.$ **(direkte) Vorgänger- und Nachfolgerklassenversionen,** $pred(sv.c)$, $dpred(sv.c)$, $succ(sv.c)$, $dsucc(sv.c)$ $\left.\right\}$

Wenn in einem Klassenableitungsbaum $CDT = (cvids, <^1_{sv,c})$ $sv_v.c <_{sv,c} sv_u.c$ $(sv_v.c <^1_{sv,c} sv_u.c)$ *gilt, dann heißt* $sv_u.c$ (direkte) Vorgängerklassenversion *von* $sv_v.c$ *und* $sv_v.c$ *heißt* (direkte) Nachfolgerklassenversion *von* $sv_u.c$. *Ferner führen wir die folgenden abkürzenden Schreibweisen für Teilmengen von cvids ein:*

$$\begin{aligned}
pred(sv.c) &:= \{sv'.c \in cvids | sv.c \leq_{sv,c} sv'.c\} \\
dpred(sv.c) &:= \{sv'.c \in cvids | sv.c <^1_{sv,c} sv'.c\} \\
succ(sv.c) &:= \{sv'.c \in cvids | sv'.c \leq_{sv,c} sv.c\} \\
dsucc(sv.c) &:= \{sv'.c \in cvids | sv'.c <^1_{sv,c} sv.c\}
\end{aligned}$$

Damit haben wir $dpred(sv.c)$ aus Gründen der Einheitlichkeit ebenfalls als Menge definiert, obwohl diese niemals mehr als ein Element enthält.

Die in Definition 5.23 gemachte Aussage über die Vorgänger- und Nachfolgerbeziehung im Schemaableitungsgraphen läßt sich damit wie folgt präzisieren: Eine Schemaversion sv_v ist genau dann eine direkte Nachfolgerversion einer Schemaversion $sv_u \neq sv_0$, wenn sv_v mindestens eine von **Object** verschiedene Klasse enthält, die von sv_v abgeleitet wurde. Wenn eine Schemaversion sv demzufolge keine Vorgängerschemaversion besitzt, dann soll sv_0 die einzige Vorgängerschemaversion von sv sein. Damit wird erreicht, daß der Schemaableitungsgraph zusammenhängend ist und sv_0 als Wurzel besitzt.

$$(sv_v <^1_{sv} sv_u) :\Leftrightarrow ((\nexists c \in (classes(sv_v) \cap classes(sv_u)) - \{\textbf{Object}\} : sv_v.c <^1_{sv,c} sv_u.c) \Rightarrow (u = 0))$$

Damit kann der Schemaableitungsgraph als „Summe“ der Klassenableitungsbäume aller Klassen angesehen werden, denn es gilt (mit Ausnahme der Klasse **Object**) $<^1_{sv,c} \subseteq <^1_{sv}$. Genauer gilt:

$$(\exists c : sv_v.c <^1_{SV,c} sv_u.c) \Rightarrow (sv_v <^1_{sv} sv_u)$$

Lemma 5.1 (Zusammenhang von Klassenableitungsbäumen)
Jeder Klassenableitungsbaum ist zusammenhängend, d.h. wenn $sv_v.c$ existiert, dann ist sv_v die Wurzel des Ableitungsbaumes von Klasse c ($csv(c) = sv_v$) oder es existiert ebenso eine Klassenversion $sv_u.c$ mit $sv_v <^1_{sv} sv_u$ und $sv_v.c <^1_{sv,c} sv_u.c$.

Beweisskizze:

- Eine Version der Klasse *c* kann nur dann in einer Schemaversion *sv* existieren, wenn sie in *sv* erzeugt oder von einer Vorgängerschemaversion von *sv* abgeleitet wurde.
- Nach einer Ableitung kann die Klasse aus der Vorgängerschemaversion nicht mehr gelöscht werden, da diese dann gefroren ist.
- In anderen Schemaversionen kann zwar eine Klasse mit demselben Namen wie *c* definiert werden, dies wäre aber trotzdem nicht dieselbe Klasse.

□

5.3.3 Der Konsistenzbegriff für versionierte Schemata

Wir führen nun die Erweiterung des Konsistenzbegriffes aus Definition 5.19 auf versionierte Schemata durch. Analog zu unserer Vorgehensweise in Abschnitt 5.2.4 stellen wir hier vorbereitend Invarianten für versionierte Schemata auf.

5.3.3.1 Schemainvarianten

Die beiden folgenden Invarianten garantieren die Eindeutigkeit der in Definition 5.26 eingeführten Notation *sv.c*.

Invariante 5.11 {Eindeutigkeit der Schemaversionen}
Jede Schemaversion $sv \in sv(s)$ besitzt innerhalb des Schemas s einen eindeutigen (und nicht vom System verwendeten) Namen und eine eindeutige Identität.

Invariante 5.12 {Eindeutigkeit der Klassenversionen}
Jede Version sv eines Schemas s enthält höchstens eine Version jeder Klasse von s.[78]

Die systemdefinierte Schemaversion sv_0 dient uns als Wurzel des Schemaableitungsgraphen und muß daher in jedem Schema enthalten sein. Sie ermöglicht uns, jede benutzerdefinierte Schemaversion konzeptionell als von bestehenden Schemaversionen abgeleitet zu betrachten, so daß wir nicht explizit zwischen dem Erstellen und dem Ableiten neuer Schemaversionen unterscheiden müssen.

Invariante 5.13 {Vollständigkeit des Schemas}
Das Schema s enthält die Wurzelschemaversion sv_0. Formal: $sv_0 \in sv(s)$

Ähnlich unserer Vorgehensweise bei Klassenvererbungsgraphen (siehe Invariante 5.5) fordern wir zu Gunsten der Übersichtlichkeit auch für Schemaableitungsgraphen eine Minimalitätseigenschaft.

[78] Dies ist eine Eindeutigkeitsforderung analog zu den Invarianten 5.1 und 5.2 für Klassen in unversionierten Schemata.

Invariante 5.14 {minimaler Schemaableitungsgraph}
Keine Version eines Schemas s ist gleichzeitig direkte Nachfolgerschemaversion der Wurzelschemaversion sv_0 *und einer anderen Schemaversion* sv'.
Formal: $\nexists sv \in sv(s) : (sv <^1_{sv} sv_0 \wedge \exists sv' \in sv(s) - sv_0 : sv <_{sv} sv')$

Die Schemaableitungsstruktur ist damit in dem Sinne minimal, daß eine Schemaversion sv*, die von einer Schemaversion* $sv' \neq sv_0$ *abgeleitet wurde, nicht gleichzeitig eine direkte Kante zu* sv_0 *besitzt. Die Ableitungsbeziehung zur Wurzelschemaversion* sv_0 *wäre in einem solchen Fall überflüssig, da von dieser (abgesehen von der systemdefinierten Wurzelklasse der Vererbung* **Object***) nichts integriert wurde.*

Man beachte, daß die hier für den Schemaableitungsgraphen ausgedrückte Einschränkung schwächer ist, als die in Invariante 5.5 für den Klassenvererbungsgraphen formulierte (siehe Abbildung 5.5). Dies liegt darin begründet, daß hier die dort verbotene Situation eines direkten und eines indirekten Weges (via eines Knotens z) von einem Knoten y zu demselben Vorgängerknoten x durchaus Sinn macht. Sie bedeutet nämlich, daß die von Knoten y repräsentierte Schemaversion Klassen sowohl aus x als auch aus z integriert hat. Dies ist streng zu unterscheiden von der Situation, daß nur Klassen von z (und nicht von x) integriert wurden. Selbst wenn die aus x integrierte Klasse c in z ebenfalls integriert wurde (was nicht der Fall sein muß), so kann sie dort verändert worden sein, so daß ein erheblicher Unterschied zwischen der Integration der Klassenversionen $x.c$ und $y.c$ besteht.

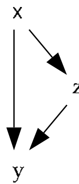


Abbildung 5.5: Ein Graph, der als Klassenvererbungsgraph (x, y und z seien Klassen einer Schemaversion mit den Vererbungsbeziehungen dazwischen) wegen Invariante 5.5 nicht erlaubt wäre, der jedoch trotz Invariante 5.14 ein gültiger Schemaableitungsgraph (x, y und z seien Versionen eines Schemas mit den Ableitungsbeziehungen dazwischen) ist.

5.3.3.2 Konsistenz versionierter Schemata

Aufgrund der vorgenommenen Erweiterung von unversionierten auf versionierte Schemata und der damit verbundenen Ergänzung um die Ableitungsbeziehungen zwischen Schemaversionen können wir einige der in Abschnitt 5.2 gegebenen Definitionen und Invarianten nicht mehr ohne weiteres verwenden. Um dies trotzdem tun zu können, genügt es bereits, den bisher verwendeten Schemabegriff durch den Begriff der Schemaversion zu ersetzen. Damit beschreibt die in Abschnitt 5.2.4.2 gegebene Definition 5.19 der Konsistenz nun nicht mehr Anforderungen an ein Schema, sondern an eine Schemaversion. Daher benötigen wir eine zusätzliche Definition für die Konsistenz eines versionierten Schemas.

Definition 5.32 {konsistentes (versioniertes) Schema}
Ein versioniertes Schema $s = (sid, sname, svids, <^1_{sv})$ *heißt* konsistent*, wenn die folgenden Bedingungen erfüllt sind.*

- *Jede Schemaversion* $sv \in svids$ *ist konsistent nach Definition 5.19.*
- *Eindeutigkeit:*

  - *der Namen und Identitäten der Versionen von s nach Invariante 5.11*
  - *der Klassenversionen der Versionen von s nach Invariante 5.12*

- *Vollständigkeit des Schemas nach Invariante 5.13*

- *Ableitung:*

  - *Der Schemaableitungsgraph* $SDG = (svids, <^1_{sv})$ *ist minimal nach Invariante 5.14*
  - *Die Ableitungsbeziehung zwischen den Versionen einer Klasse c von s ist ein Klassenableitungsbaum nach Definition 5.30.*
    Formal: $\forall c \in classes(s) : (cv(c), \leq^c)$ *ist ein Klassenableitungsbaum über* $cv(c)$.
  - *Für jede Klasse existiert ein Klassenkonvertierungsgraph. Diesen werden wir jedoch erst in den Abschnitten 6.5 (Definition 6.6) und 6.6.2.2 (Definition 6.14) näher betrachten.*

Eine neue Schemaversion darf nur dann akzeptiert werden, wenn diese entsprechend Definition 5.19 als konsistent bezeichnet werden kann. Da eine Schemaänderung Seiteneffekte auf andere Komponenten der Schemaversion bewirken kann, ist es von besonderer Bedeutung, daß diese Seiteneffekte garantiert korrigiert werden, bevor die Schemaversion benutzt wird. Eine Schemaänderung, die die Konsistenz des Schemas so verletzt, daß diese durch korrigierende Maßnahmen nicht wieder hergestellt werden kann, ist abzulehnen.

5.4 Aspekte des praktischen Umgangs mit der Schemaversionierung

Nachdem wir das COAST-Objektmodell für versionierte Schemata in den vorangegangenen Abschnitten 5.2 und 5.3 eingeführt haben und dabei u.a. Klassen- und Schemaversionen, Vererbungs- und Ableitungsbeziehung sowie Invarianten und Schemakonsistenz formal beschrieben wurden, gehen wir nun auf Aspekte beim praktischen Umgang mit der Schemaversionierung ein. Dazu zählt der Prozeß der Ableitung neuer Schemaversionen und dabei mögliche Wiederverwendungen ebenso wie die Applikationsanbindung an ein versioniertes Schema und die Auswahl der jeweils am besten geeigneten Schemaversion.

5.4.1 Der Schemaableitungsprozeß

Wie bereits erläutert, werden aus konzeptioneller Sicht alle Versionen eines Schemas auf genau dieselbe Art und Weise angelegt, genauer gesagt von bereits existierenden Versionen desselben Schemas abgeleitet. Hier soll nun das schrittweise Vorgehen bei dem als *Schemaableitungsprozeß* (engl. *schema derivation process*) bezeichneten Vorgang der Ableitung einer neuen Schemaversion genauer vorgestellt werden.

Zunächst muß dabei eine neue Version sv_v des in Bearbeitung befindlichen Schemas s erzeugt und (innerhalb aller Versionen von s) eindeutig benannt werden. Die Schemaversion sv_v enthält initial keinerlei Klassen mit Ausnahme der für die Konsistenz von sv_v nach Definition 5.19 benötigten Wurzelklasse **Object**.

Im zweiten Schritt werden die Schemaänderungsprimitive der ODL in beliebiger Kombination auf sv_v angewendet, um die neue Schemaversion entsprechend den Anforderungen zu spezifizieren. Um die relative Spezifikation neuer Schemaversionen zu ermöglichen, stehen insbesondere

solche Schemaänderungsprimitive zur Verfügung, die die Integration von Klassen beliebiger[79] existierender Versionen desselben Schemas gestatten. Diejenigen Schemaversionen, aus denen die neue Schemaversion sv_v Komponenten integriert, werden als *Quellschemaversionen des Schemaableitungsprozesses* bezeichnet, sv_v selbst heißt *Zielschemaversion*. Neben der Integration bestehender Komponenten werden Schemaänderungsprimitive zum Hinzufügen neuer und zum Löschen oder Verändern in der Zielschemaversion enthaltener Klassen angeboten.

Daraufhin sind Spezifikationen auf der Objektebene vorzunehmen, auf die wir in Kapitel 6 eingehen werden. Diese spezifizieren, wie das System mit bereits existierenden oder später erzeugten, modifizierten oder gelöschten Objekten umgehen soll.

Zum Abschluß des Schemaableitungsprozesses wird sv_v als direkter Nachfolger aller bei der Ableitung benutzter Quellschemaversionen in den Schemaableitungsgraphen integriert und eingefroren. Diesen Zeitpunkt der endgültigen Fertigstellung einer neuen Schemaversion sv führte Definition 5.22 als *Ableitungszeit* (engl. *derivation time*) der Schemaversion sv. Sie wird formal als $dt(sv)$ notiert. Von nun an kann die Schemaversion sv_v von Applikationen benutzt werden und steht als Quellschemaversion für weitere Schemaänderungen zur Verfügung.

Der hier beschriebene Schemaableitungsprozeß schließt die Implementierung von Methoden nicht ein. Methodenrümpfe werden in externen Dateien (außerhalb der Kontrolle des DBMS) spezifiziert und zum Ableitungszeitpunkt übersetzt. Um die Implementierung einer Methode zu verändern, muß die entsprechende Datei modifiziert werden.

5.4.2 Reduzierung der Zahl notwendiger Schemaversionen bei kapazitätserweiternden Schemaänderungen

Zur Erbringung seiner Leistungsmerkmale muß das Konzept der Schemaversionierung bei der Verwaltung und Konsistenzsicherung von Schema und Datenbank zusätzlichen Aufwand treiben. Um die damit verbundene Beeinträchtigung der Effizienz (Teilziel 3.15) zu vermindern, kommt die Reduzierung der Anzahl der Versionen eines Schemas in Betracht. Dies kann dadurch erreicht werden, daß nicht bei jeder Schemaänderung eine neue Version angelegt wird. Dazu hatten wir bereits die Ausführung mehrerer atomarer Schemaänderungsprimitive zu einer einzigen Schemaänderung zusammengefaßt, so daß die dabei entstehende Schemaversion auch auf der Abstraktionsebene des Schemaentwicklers als sinnvolle und vollständige Modellierung der Diskurswelt erachtet werden kann.

Ein weiterer Schritt zur Reduzierung der Anzahl notwendiger Schemaversionen kann dadurch getan werden, daß selbst Änderungen gefrorener Schemaversionen erlaubt werden. Dies kann jedoch nur unter gewissen Umständen geschehen, nämlich genau dann, wenn sichergestellt ist, daß mögliche Auswirkungen auf Objekte transparent behandelt werden können und daß sich keine Auswirkungen auf existierende Applikationen zeigen. Um dies sicherzustellen, muß eine Analyse der Auswirkungen verschiedener Schemaänderungen durchgeführt werden, so wie wir das in [FL96] am Beispiel von O_2 getan haben.

Grundlage der Überlegungen ist der Begriff der *Informationskapazität eines Schemas*. Dieser beschreibt, wieviel Semantik der Diskurswelt durch ein Schema beschrieben wird und läßt insbesondere eine Kategorisierung in kapazitätsvermindernde (engl. *capacity reducing*), kapazitätserhaltende (engl. *capacity preserving*) und kapazitätserweiternde (engl. *capacity augmenting*) Schemaänderungsprimitive zu.

Kapazitätserweiternde Schemaänderungen, wie das Hinzufügen eines Attributes zu einer Klasse oder das Anlegen neuer Klassen können in der Regel unbesehen auch auf gefrorene Schemaver-

[79] Die verwendeten Schemaversionen müssen allerdings gefroren sein, ansonsten werden sie dem Schemaentwickler aber erst gar nicht zur Verfügung gestellt.

sionen angewendet werden, da dabei benötigte Schemakomponenten erhalten bleiben.[80] Selbst die Erweiterung von Attributtypen zu Obertypen ist möglich, ohne bereits in der Datenbank enthaltene Objekte prüfen zu müssen. Bei der Spezialisierung eines Typs zu einem Untertyp müßte jedoch die gesamte Datenbank auf die Einhaltung der verschärften Bedingung geprüft werden, weswegen diese Schemaänderung z.B. von ORION (siehe Regel 4.6 in Abschnitt 4.5.3.7.2) allein aus Aufwandsgründen nicht erlaubt wird.

Grundsätzlich geht die Möglichkeit zur Änderung gefrorener Schemaversionen selbst bei kapazitätserweiternden Primitiven dann verloren, wenn sehr dynamische Applikationen existieren, die das Metaschema der Datenbank abfragen und auf das Vorhandensein oder Nichtvorhandensein bestimmter Schemakomponenten unterschiedlich reagieren.

Ähnliches gilt auch für kapazitätserhaltende Änderungen wie Umbenennungen, sofern der Zugriff auf Attribute nicht über Identifikatoren durchgeführt wird, die bei Umbenennungen unverändert bleiben.

Kapazitätsvermindernde Schemaänderungen sind bestenfalls dann noch ohne Erzeugung einer neuen Schemaversion durchführbar, wenn das Datenbanksystem Kenntnis all seiner Applikationen hat und auf diesem Wege sicherstellen kann, daß zu entfernende Schemakomponenten gar nicht benutzt werden.

In [FL96] haben wir kapazitätserhaltende und -vermindernde Schemaänderungen zusammengefaßt (engl. *schema modifying primitives*) und dann feiner kategorisiert. Zunächst unterscheiden wir, ob sich Applikationen nach einer Schemaänderung noch unverändert compilieren und ausführen lassen (engl. *compilation safe*) oder nicht (engl. *compilation unsafe*). Bei letzteren kann man weiter unterscheiden, ob eine bereits vorliegende, compilierte Applikation wenigstens noch ablauffähig ist (engl. *execution safe*) oder nicht (engl. *execution unsafe*). In [FL96] haben wir die Schemaänderungsprimitive von O_2 unter Berücksichtigung verschiedener Randbedingungen den genannten Kategorien zugeordnet und die damit erreichbare Reduzierung notwendiger Schemaversionen an einem durchgängigen Beispiel veranschaulicht.

Durch die Entscheidung, ob bei einer Schemaänderung die Durchführung auf einer bereits gefrorenen Schemaversion möglich oder ob die Ableitung einer neuen Schemaversion notwendig ist, kann die Anzahl von Versionen eines Schemas erheblich reduziert werden. Trotzdem ist sichergestellt, daß selbst bei informationsvermindernden Schemaänderungen nichts an Objektsemantik verloren gehen kann, da dann mehrere Versionen eines Objektes gespeichert werden.

5.4.3 Applikationsanbindung an Schemaversionen

Jede Applikation (DB-Klient) wird statisch (d.h. zur Übersetzungszeit) an eine Schemaversion gebunden (siehe Abschnitt 2.1.10). Diese Verbindung spiegelt die spezielle Ansicht der Applikation auf die Datenbasis wider sowie ihre speziellen Anforderungen an die Speicherung und Verwaltung von Informationen. Damit wird festgelegt, Objekte welcher Klassen die Applikation zugreifen kann und in welcher Form (d.h. mit welchen Eigenschaften) sie diese sieht. Wir sagen, solch eine Verbindung definiere einen (Schemaversions-) *Kontext* für die Erzeugung und Veränderung von Daten.

[80] Hierzu ist zu bemerken, daß sich die Ausführungen in [FL96] auf das Datenmodell und auf die Schemaänderungsprimitive von O_2 beziehen. Auch beispielsweise die Änderung eines Attributtyps zu einer Referenz auf eine neu erstellte Klasse, welche zumindest den bisherigen Wert des geänderten Attributes modelliert, ist generell als kapazitätserweiternd einzustufen. Dies gilt zumindest dann, wenn sie entsprechend auf die Datenbank abgebildet wird, d.h. wenn der ursprüngliche Attributwert in ein Objekt der neu erstellten Klasse übernommen und durch eine Referenz auf eben dieses Objekt ersetzt wird. Bei solchen komplexen Schemaänderungen wie dem dargestellten **nest** ist jedoch die fortgesetzte Ausführbarkeit existierender Applikationen ohne Änderung nicht mehr gegeben. Da solche Schemaänderungen in O_2 jedoch nicht unterstützt werden, haben wir von ihrer Berücksichtigung abgesehen.

Eine einzelne Version eines versionierten Schemas entspricht einem kompletten Schema im herkömmlichen Sinne. Daher werden Applikationen nicht mehr an ein Schema insgesamt, sondern an eine Version davon gebunden. Diese für das Übersetzen und Binden einer Applikation benutzte Schemaversion wird als (aktive) Schemaversion dieser Applikation bezeichnet. Demzufolge kann die Applikation auch nur noch auf diejenigen Objekte einer Datenbank zugreifen, die in ihrer aktiven Schemaversion *sichtbar* sind. Diese sog. *Zugriffsbereiche* werden wir in Kapitel 6 genauer vorstellen.

Abbildung 5.6 zeigt beispielhaft, wie 11 Applikationen $app_1, \ldots, app_{11}$ an sieben benutzerdefinierte Versionen eines Schemas s gebunden sein könnten. Eine neu entwickelte Applikation kann dabei durchaus an eine ältere Schemaversion gebunden werden (siehe z.B. Applikation app_4 in Abbildung 5.6). Da in der Praxis einige Schemaänderungen auch korrigierenden Charakter haben werden, wird die Schemaversion einer Applikation zum Zeitpunkt ihrer Implementierung zwar oft ein Blattknoten des Schemaversionsableitungsgraphen sein; dies wird jedoch in keinster Weise durch unser Modell erzwungen.

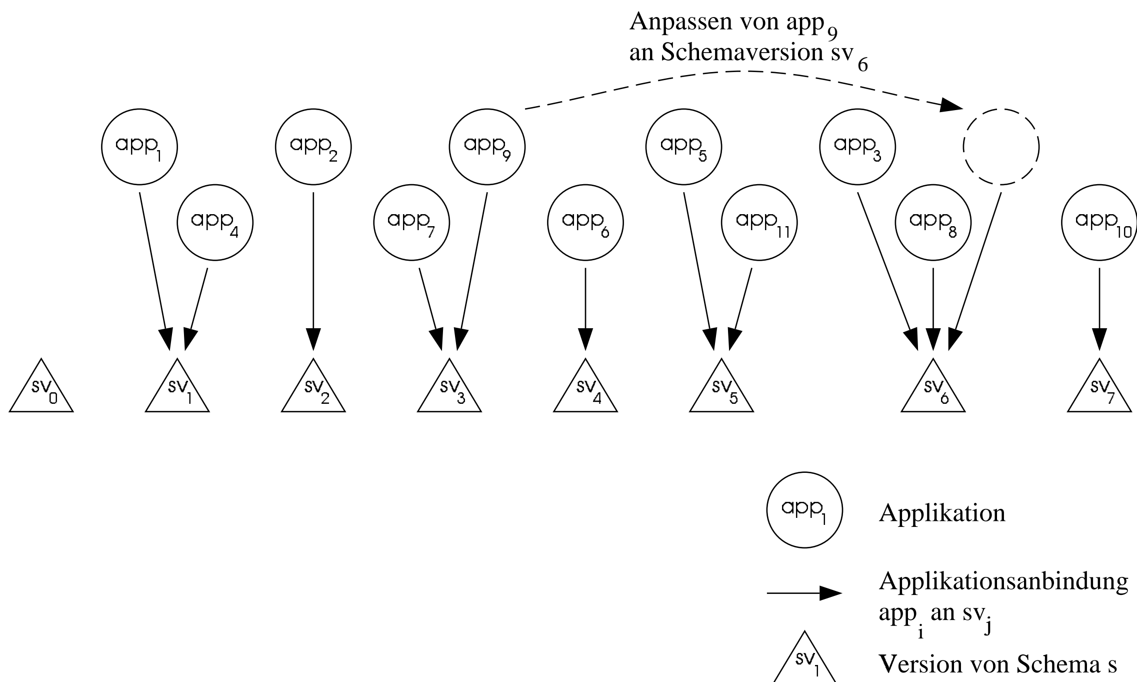


Abbildung 5.6: Beispiel der Anbindung von Applikationen $app_1, \ldots, app_{10}$ an die Versionen eines Schemas s und optionale Anpassung von app_9 an Schemaversion sv_6.

Jede Applikation kann diese Verbindung zu einer Schemaversion ändern, womit eine veränderte Auffassung der Diskurswelt zum Ausdruck kommt. Dafür muß die Applikation i.Allg. angepaßt, zumindest aber recompiliert werden (siehe z.B. Applikation app_9, die in Abbildung 5.6 von sv_3 an sv_6 angepaßt wird). Während die Erzeugung neuer Schemaversionen zunächst keinerlei Einfluß auf existierende Applikationen hat, so kann es doch mitunter erwünscht sein, Applikationen an neuere Schemaversionen anzupassen. Im Gegensatz zur direkten Schemaevolution, wo eine solche Anpassung nach jeder Schemaänderung mit allen Applikationen sofort durchgeführt werden muß, erlaubt die Schemaversionierung beliebige Vorgehensweisen. Beliebig viele (oder auch keine) Applikationen können zu beliebigen Zeitpunkten an neue Schemaversionen angepaßt werden.

5.4.4 Auswahl von Schemaversionen für Applikationen

Die Auswahl einer oder mehrerer Quellschemaversionen für die Ableitung einer neuen Schemaversion hängt vom Ursprung der Schemaänderung ab. Manchmal kommt das Bedürfnis nach einer Schemaänderung von einem bestimmten Klienten, so daß es am natürlichsten erscheint, dessen bisherige Schemaversion als Ursprung für die notwendigen Änderungen heranzuziehen. In anderen Fällen mag es sinnvoller erscheinen, diejenige(n) Schemaversion(en) als Quelle(n) auszuwählen, die dem gewünschten am „ähnlichsten" sind, sowohl syntaktisch (bezüglich des Inhaltes) als auch semantisch (bezüglich der semantischen Verbindungen zu anderen Schemaversionen), wobei existierende Spezifikationen am effektivsten wiederbenutzt werden. Hierzu können besondere Werkzeuge zur Unterstützung des Auswahlprozesses sinnvoll sein (siehe Abschnitt 7.1.2.2). Da mitunter Schemaänderungen auch als Verbesserungen angesehen werden können, scheint es in solchen Fällen angebracht zu sein, Blattknoten des Schemaableitungsgraphen als Quellschemaversionen auszuwählen. Hierbei soll betont werden, daß nahezu jede semantische Beziehung etabliert werden kann und eine geeignete Auswahl der Quellschemaversion(en) bei der Ableitung einer neuen Schemaversion lediglich zur Verringerung des Spezifikationsaufwandes beiträgt.

Eine mit der Auswahl von Quellschemaversionen für die Ableitung neuer Schemaversionen verwandte Fragestellung ist die nach der für eine zu implementierende Applikation am besten geeignete Schemaversion. Die Anforderungen eines Klienten können durch eine bereits existierende Schemaversion in befriedigendem Maße erfüllt sein. Anderenfalls kann die Ableitung einer neuen Schemaversion für die neue Applikation erforderlich sein. Dabei ist eine interessante Frage diejenige nach der Trennung von Datenbank und Applikationsfunktionalität. Ein wichtiges Charakteristikum einer Datenbank ist das der gemeinsamen Nutzung. Von mehreren Applikationen nutzbare Funktionalität sollte in der Datenbank untergebracht werden. Demzufolge muß ein Kompromiß gefunden werden zwischen zahlreichen klientenorientierten Funktionalitäten und der Anzahl entstehender Schemaversionen.

5.5 Die Schemabeschreibungssprache COAST-ODL

Zur textuellen Beschreibung eines Schemas und seiner Versionen wird eine Sprache benötigt, in der die entsprechenden Komponenten spezifiziert werden können. Solche Sprachen werden im ODMG-Standard [Cat96, CB98, CBB+00] als Objektbeschreibungssprachen (engl. *object definition language, ODL*) bezeichnet, wir nennen sie hier genauer Schemabeschreibungssprachen.[81] Die von der ODMG vorgeschlagene ODL bietet jedoch nur Lesezugriffe auf das Schema und diese auch nur für das C++-Binding (über die C++ Schema Access API). Auch SQL3 entsprach nicht den gestellten Anforderungen [Her99]. Weiterhin bieten die in der Literatur vorgestellten Schemabeschreibungssprachen allesamt keine Unterstützung für Schemaversionen, so daß für diese Arbeit eine eigene ODL benötigt wird. Die hier einzuführende COAST-ODL beschränkt sich auf das COAST-Objektmodell und kann daher trotz der Erweiterung auf versionierte Schemata recht knapp gehalten werden. Beim Entwurf der Sprache war die Orthogonalität der Konstrukte auf Schema-, Schemaversions-, Klassen- und Attributebene ein vorrangiges Ziel. Weiterhin sollte sich die COAST-ODL weitgehend an den aus der Literatur bekannten Schemabeschreibungssprachen, insbesondere an der des ODMG-Standards, orientieren.

Odberg [Odb95] unterscheidet zwischen dem Anlegen einer Basisschemaversion und dem Ableiten neuer Schemaversionen von dieser Basisschemaversion. Demzufolge führt er zwei voneinander

[81] Genau genommen handelt es sich bei der ODL der ODMG nicht um eine vollständige Schemabeschreibungssprache, da dort nur Schnittstellen beschrieben werden. Vor diesem Hintergrund wäre die von der Object Management Group (OMG) verwendete Bezeichnung *Schnittstellendefinitionssprache* (engl. *interface definition language, IDL*) auch für die ODL der ODMG treffender.

getrennte und wenngleich ähnliche, so doch nicht identische Sprachen zur Schemabeschreibung ein, nämlich eine Datendefinitionssprache (engl. *data definition language, DDL*) und eine Änderungsspezifikationssprache (engl. *change specification language, CSL*). Im Gegensatz zu Odberg verwendet COAST nur eine Sprache, nämlich die Schemabeschreibungssprache COAST-ODL. Zum einen kann damit auf die Unterscheidung zweier getrennter Sprachen, welche keinerlei Vorteile bietet, verzichtet werden, zum anderen drückt sich damit die gemeinsame Vorgehensweise bei der Ableitung aller Schemaversionen aus. Bereits die erste benutzerdefinierte Schemaversion sv_1 wird in COAST nicht neu angelegt, sondern durch Ableitung aus der systemdefinierten Schemaversion sv_0 erzeugt. Da sv_0 keinerlei benutzerdefinierte Klassen enthält, ist dieser Unterschied zwar rein konzeptioneller Natur, er vereinheitlicht jedoch die Vorgehensweise und manifestiert sich in der Existenz nur einer einzigen Schemabeschreibungssprache in COAST.

Zur sprachlichen Trennung herkömmlicher Operationen von der Operationen der COAST-ODL bezeichnen wir letztere im folgenden als *Schemabeschreibungsprimitive* oder kurz als *Primitive*. Dabei unterscheiden wir zwischen *erzeugenden Primitiven*, die neue Schemata, Schemaversionen, etc. anlegen und *Schemaänderungsprimitiven* (engl. *schema update primitives*), die bestehende Strukturen modifizieren.

Die Primitive der COAST-ODL lassen sich in vier Gruppen untergliedern, die die Spezifikationen und Änderungen auf den Ebenen Schema, Schemaversion, Klasse und Attribut vornehmen. Auf jeder Ebene stehen Primitive zur Erzeugung neuer und zur Veränderung oder Löschung existierender Metaobjekte der jeweiligen Granularität zur Verfügung. Eine vollständige Auflistung sämtlicher Sprachkonstrukte der COAST-ODL befindet sich in Anhang B.

Bevor wir die Primitive der COAST-ODL im einzelnen vorstellen geben wir anhand der folgenden Taxonomie (entsprechend [BKKK87, BM93a, Lau96a]) einen Überblick.

1 Primitive auf Schemaebene

- 1.1 **create schema** sname { ... };
- 1.2 **modify schema** sname { ... };
- 1.3 **rename schema** sname **to** newsname;
- 1.4 **delete schema** sname;

2 Primitive auf Schemaversionsebene

- 2.1 **create schemaversion** svname { ... };
- 2.2 **modify schemaversion** svname { ... };
- 2.3 **rename schemaversion** svname **to** newsvname;
- 2.4 **delete schemaversion** svname;
- 2.5 **freeze schemaversion** svname;
- 2.6 **unfreeze schemaversion** svname;

3 Primitive auf Klassenebene

- 3.1 **integrate class** cname **from** svname ...;
- 3.2 **create class** cname { ... };
- 3.3 **modify class** cname { ... };
- 3.4 **rename class** cname **to** newcname;
- 3.5 **delete class** cname;

4 Primitive auf Attributebene

4.1 Primitive bezüglich Vererbung
  4.1.1 **create inherit** cname;
  4.1.2 **delete inherit** cname;
4.2 Primitive bezüglich Attributen
  4.2.1 **create attribute** aname: typename;
  4.2.2 **overwrite attribute** aname **to** newaname;
  4.2.3 **delete overwrite attribute** aname;
  4.2.4 **retype attribute** aname **to** typename;
  4.2.5 **rename attribute** aname **to** newaname;
  4.2.6 **delete attribute** aname;
4.3 Primitive bezüglich Methoden
  4.3.1 **create method** mname;
  4.3.2 **modify method** mname;
  4.3.3 **rename method** mname **to** newmname;
  4.3.4 **delete method** mname;

5 Primitive für die Propagation

5.1 **create derivation for class** classname **to class** classname **in schemaversion** svname ...;
5.2 **modify derivation for class** classname ...;
5.3 **delete derivation for class** classname;

Die Primitive der Kategorie 5 betreffen die Propagation auf Objektebene. Daher werden wir diese erst in Abschnitt 6.4 erläutern. Auch das Integrationsprimitiv (3.1) läßt die Spezifikation von Propagationsparametern zu, weswegen wir in Abschnitt 6.4 auch darauf zurückkommen werden.

Unser Modell der Schemaevolution ist unabhängig davon, welche Schemaänderungsoperationen (primitiver oder komplexer Natur) zur Verfügung stehen und kann daher relativ leicht erweitert werden. Für die Ergänzung der obigen Taxonomie um weitere Operationen ist allerdings neben deren Effekt auf Schemaebene auch derjenige auf Objektebene zu spezifizieren.

5.5.1 Primitive der COAST-ODL auf Schemaebene

Ein COAST-Schema s namens *sname* kann mittels der ODL erzeugt, verändert, umbenannt oder gelöscht werden. Dazu stehen die folgenden Primitive zur Verfügung.

```
create schema sname {
    ...          // Schemadefinitionsblock
    } /* create schema sname */

modify schema sname {
    ...          // Schemadefinitionsblock
    } /* modify schema sname */

rename schema sname to newsname;

delete schema sname;
```

Die Primitive auf Schemaebene.

Die Primitive **create schema** und **modify schema** beinhalten einen sog. *Schemadefinitionsblock* in dem ein neues Schema erzeugt bzw. ein bestehendes Schema verändert werden kann. Im diesem Schemadefinitionsblock können die im nächsten Abschnitt zu beschreibenden Primitive der Schemaversionsebene verwendet werden, um Komponenten von s zu erzeugen bzw. zu verändern.

Hier wie auch im folgenden muß stets die von Invariante 5.1 geforderte Eindeutigkeit der Namen innerhalb ihres jeweiligen Gültigkeitsraumes gewährleistet sein, was zu den offensichtlichen Einschränkungen beim Anlegen und Umbenennen von Schemata und ihren Komponenten führt. Dies gilt hier für Schemanamen global und im folgenden für Schemaversionsnamen innerhalb ihres Schemas, für Klassen innerhalb ihrer Schemaversion, etc. Wir ersparen uns im folgenden die Wiederholung dieses selbstverständlichen Hinweises.

5.5.2 Primitive der COAST-ODL auf Schemaversionsebene

5.5.2.1 Ableitung von Schemaversionen

In Datenbanksystemen ohne Unterstützung für Schemaversionierung durchläuft das Schema während seines Lebenszyklus verschiedene Zustände in einer seriellen Abfolge. Dabei wird jeder Zustand durch Ausführung von Primitiven in den jeweils nächsten Zustand überführt.

In Anlehnung an die klassische Vorgehensweise werden neue Versionen eines Schemas in COAST nicht komplett definiert sondern durch Ausführung von Primitiven aus existierenden Schemaversionen abgeleitet (Teilziel 3.1). Im Vergleich zu einer vollständigen Beschreibung jeder einzelnen Schemaversion wird durch die Einführung der Schemaänderungsprimitive eine erhebliche Verminderung des notwendigen Schreibaufwandes bei der Spezifikation neuer Schemaversionen erreicht. Desweiteren ist die Verwendung der Primitive notwendige Voraussetzung für die automatische Erkennung von Äquivalenzen zwischen Komponenten verschiedener Versionen eines Schemas, da erst damit beispielsweise die Änderung eines Klassennamens so ermöglicht wird, daß die beiden verschiedennamigen Versionen vom System weiterhin als Ausprägungen derselben Klasse erkannt werden können.

Die Ableitung einer neuen Version eines Schemas s wird durch einen Schemaversionsdefinitionsblock bewerkstelligt, der in den Schemadefinitionsblock von s eingebettet ist. Der Schemaversionsdefinitionsblock legt eine neue Schemaversion sv an, die zunächst lediglich die systemdefinierte Wurzelklasse **Object** ($sv.cv_0$) enthält. Ein Schemadefinitionsblock wird syntaktisch wie folgt dargestellt:

```
create schemaversion svname {
    ...          // Schemaversionsdefinitionsblock
    } /* create schemaversion svname */
```

Anlegen von Schemaversionen.

Regel 5.1 {**Ableiten einer Schemaversion**}
Beliebig viele neue Versionen eines Schemas können zu jeder Zeit von beliebig vielen gefrorenen Versionen desselben Schemas abgeleitet werden.

5.5.2.2 Verändern von Schemaversionen

Die Veränderung einer existierenden Schemaversion kann mit dem folgenden Änderungsprimitiv durchgeführt werden.

```
modify schemaversion svname {
    ...          // Schemaversionsdefinitionsblock
    } /* modify schemaversion svname */
```

Verändern von Schemaversionen.

Die nachträgliche Veränderung einer existierenden Schemaversion ist jedoch nur unter gewissen Voraussetzungen möglich. Insbesondere darf die zu verändernde Schemaversion nicht gefroren sein. In den Abschnitten 5.4.2 und 7.1.2.2 werden weitere Aspekte der Erzeugung und Veränderung von Schemaversionen behandelt. Sind die Voraussetzungen für die nachträgliche Veränderung einer vorhandenen Schemaversion nicht gegeben, so muß eine neue Schemaversion abgeleitet werden.

Regel 5.2 {Verändern einer Schemaversion}
Eine gefrorene Schemaversion, also insbesondere sv_0, kann nicht geändert werden.

5.5.2.3 Umbenennen und Löschen von Schemaversionen

Die Primitive zum Umbenennen und Löschen einer Schemaversion *sv* namens *svname* sind den analogen Primitiven auf Schemaebene sehr ähnlich.

```
rename schemaversion svname to newsvname;

delete schemaversion svname;
```

Umbenennen und Löschen von Schemaversionen.

Eine Schemaversion $sv \neq sv_0$ kann allerdings nur dann gelöscht werden, wenn sie im Schemaableitungsgraphen keine Nachfolgerschemaversion hat.

Regel 5.3 {Umbenennen und Löschen einer Schemaversion}
Eine gefrorene Schemaversion kann nicht umbenannt oder gelöscht werden.

Die auf *sv* operierenden Applikationen können nach dem Löschen von *sv* natürlich nicht mehr ausgeführt werden.

5.5.2.4 Einfrieren und Auftauen von Schemaversionen

Jede Schemaversion ist nach Definition 5.22 entweder *gefroren* (engl. *frozen*) oder *aufgetaut* (engl. *unfrozen*). Eine aufgetaute Schemaversion wird als in der Entwicklung befindlich betrachtet und kann demzufolge nicht benutzt werden. Dahingegen stellt eine gefrorene Schemaversion eine abgeschlossene Modellierung dar und ist folglich unveränderlich. Für sie können Applikationen entwickelt werden, die auf die Datenbank zugreifen, und sie kann als Quellschemaversion weiterer Ableitungen verwendet werden.

Eine neue Schemaversion ist anfänglich aufgetaut und kann durch Verwendung eines Primitives explizit eingefroren werden. Damit wird ihre Ableitungszeit festgelegt und fortan ist sie nicht nur für ihren Erzeuger sondern für alle Schemaentwickler sichtbar. Wenn eine Schemaversion nicht benutzt wird, dann kann sie wieder aufgetaut und modifiziert werden. Der hier verwendete Begriff des Benutzens schließt die Verwendung als Quellschemaversion weiterer Schemaversionen sowie die Verwendung durch Applikationen und die Speicherung von Objekten dieser Schemaversion in der Datenbank ein.[82] Die Primitive zum Auftauen und Einfrieren können daher nur auf Blättern des Schemaableitungsgraphen ausgeführt werden.

Da der Ableitungsprozeß einer Schemaversion mit dem Auftauen gleichsam fortgesetzt wird, muß auch die Ableitungszeit beim nächsten Einfrieren entsprechend aktualisiert werden.

freeze schemaversion `svname;`

unfreeze schemaversion `svname;`

Einfrieren und Auftauen von Schemaversionen.

Regel 5.4 {Einfrieren und Auftauen einer Schemaversion}
Eine Schemaversion darf nur dann aufgetaut werden, wenn sie nicht die systemdefinierte Wurzelschemaversion sv_0 ist und wenn von ihr keine weiteren Schemaversionen abgeleitet sind, keine Applikationen für sie existieren und ihr Zugriffsbereich keine Objekte enthält.

Regel 5.4 verlangt, daß eine aufzutauende Schemaversion weder auf der Ebene der Applikationen noch auf der der Objekte tatsächlich benutzt wird. Wenn diese Voraussetzungen erfüllt sind, so können auch weitere Schemaänderungen durchgeführt werden, was durch das Auftauen erlaubt wird.

5.5.3 Primitive der COAST-ODL auf Klassenebene

Die in den folgenden Abschnitten 5.5.3.1 bis 5.5.3.4 vorzustellenden Primitive der COAST-ODL können bei der Ableitung einer neuen Schemaversion *sv* grundsätzlich in beliebiger Zusammenstellung innerhalb eines Schemaversionsdefinitionsblocks verwendet werden. Es ist für die Konsistenzerhaltung jedoch unbedingt erforderlich, daß die Eindeutigkeit der Namen der Elemente von *sv* zu jedem Zeitpunkt gewährleistet bleibt, da ansonsten spätere Primitive im Ableitungsprozeß von *sv* durch Angabe von Elementnamen nicht mehr eindeutig spezifizieren könnten, mit welchen Elementen von *sv* sie operieren. Zwischenzeitliche Unvollständigkeit der neuen Schemaversion *sv* kann jedoch toleriert werden. Zur Ableitungszeit der neuen Schemaversion *sv*, d.h. beim Verlassen des Schemaversionsdefinitionsblocks, muß ihre Konsistenz jedoch vollständig gegeben sein.

5.5.3.1 Integration von Klassen existierender Schemaversionen

Eine grundlegende Änderungsoperation bei der Ableitung einer neuen Schemaversion sv_v ist die Integrationsoperation **integrate**. Sie dient der Spezifikation derjenigen Schemakomponenten, die aus bereits bestehenden Versionen von Schema *s* übernommen werden sollen. Nach

[82] In unserem Modell geschieht die Anbindung von Applikationen an eine Schemaversion außerhalb der Kontrolle des Datenbanksystems. Daher kann der COAST-Prototyp (siehe Kapitel 7) beim Auftauen nicht überprüfen, ob eine Schemaversion durch eine Applikation benutzt wird. Da jedoch sichergestellt werden kann, daß in der Datenbank keine Objekte der aufzutauenden Schemaversion existieren, kann man wohl davon ausgehen, daß sich dann auch noch keine Applikationen dieser Schemaversion im Einsatz befinden.

der Integration können beliebige Änderungen an den integrierten Komponenten vorgenommen werden.

integrate class `cname` **from** `svname`	[[**no**] **superclasses**] [[**no**] **subclasses**] [[**no**] **componentclasses**] [**keep** | **replace**] [**by classid** | **by classname**] ;

Integration von Klassenversionen.

Voraussetzung: $\exists sv_u \in sv(s), \exists c \in classes(sv_u) : name(sv_u) = svname \wedge name(sv_u.c) = cname$

Eine Klasse hängt in besonderer Weise von ihren Oberklassen und von ihren Komponentenklassen[83] ab. Daher übernimmt die Operation **integrate** per Default (Integrationsoptionen **superclasses** und **componentclasses**) nicht nur die explizit spezifizierte Klasse c der Schemaversion sv_u, sondern implizit auch all ihre Ober- und Komponentenklassen, sofern entsprechende Klassen nicht ohnehin bereits in sv_v enthalten sind. Dieses Verhalten kann allerdings durch die in der obigen Syntax angegebenen Optionen verändert werden. Des weiteren kann insbesondere eine abstrakte Klasse als Grundlage für ihre Unterklassen gesehen werden und macht ohne diese möglicherweise wenig Sinn. Daher kann optional (Default ist die Option **no subclasses**) auch die Mitintegration aller Unterklassen der explizit integrierten Klasse durch Angabe des Schlüsselwortes **subclasses** veranlaßt werden.

Neben der explizit integrierten Klasse c werden also ggf. Ober-, Unter- und Komponentenklassen implizit mitintegriert. Diese implizit mitintegrierten Klassen haben nun aber ihrerseits wieder Ober-, Unter- und Komponentenklassen, über deren implizite Mitintegration zu bestimmen ist. Da die Spezifikation verschiedener Integrationsoptionen auf verschiedenen Ebenen sehr aufwendig und schwer verständlich wäre, gehen wir bei der impliziten Integration von Klassen genau wie bei der explizit integrierten Klasse vor, d.h. deren Ober-, Unter- und Komponentenklassen werden ebenfalls mitintegriert, sofern die entsprechenden Integrationsoptionen gesetzt sind. Wenn nicht alle Klassen mit denselben Integrationsoptionen integriert werden sollen, so ist für die verschiedenen Klassen jeweils eine eigene **integrate**-Operation zu verwenden.

Ist die Oberklasse einer in sv_v zu integrierenden Klasse c nicht bereits in sv_v vorhanden und wird sie auch nicht mitintegriert, so wird sie aus der Oberklassenmenge von c in der Zielschemaversion sv_v gelöscht, da ansonsten deren Vollständigkeit (siehe Invariante 5.3) nicht mehr gegeben wäre. Aus demselben Grund wird in $sv_v.c$ ein lokal definiertes Attribut a vom Referenztyp bei der Integration gelöscht, wenn die referenzierte Komponentenklasse nicht in sv_v enthalten ist.[84] Ist das Attribut a jedoch nicht in der zu integrierenden Klasse $sv_u.c$ lokal angelegt, sondern geerbt,

[83] Unter dem Begriff der *Komponentenklassen* verstehen wir hier alle Klassen, die von einer Klasse referenziert werden, obwohl wir damit keine strenge Beziehung des Enthaltenseins (**is_part_of**) verbinden. Dazu wäre bei der Spezifikation einer Referenz wie in [KBG89] eine Unterscheidung zwischen abhängigen (engl. *dependent, total*) und unabhängigen (engl. *independent, overlapping*) sowie zwischen exklusiv (engl. *exclusive, disjoint*) und nicht-exklusiv (engl. *shared, overlapping*) zugeordneten Komponentenobjekten zu ermöglichen. Die hier bedeutsame Abhängigkeit besteht allerdings in allen genannten Fällen gleichermaßen, so daß wir auf eine Unterscheidung verzichten können.

[84] Die Anforderung der Vollständigkeit des Schemas muß nicht wie hier notwendigerweise immer erfüllt sein. Eine Verminderung der Anforderung dahingehend, daß die Vollständigkeit erst bei der Benutzung der Schemaversion, d.h. bei ihrem Einfrieren gegeben sein muß, ist möglich und wird etwa von O_2 (siehe Abschnitt 4.3.4.2) angeboten. Die damit verbundenen, erhöhten Anforderungen an die Implementierung des Schemamanagers (siehe Abschnitt 7.2.2), der dann auch unvollständig spezifizierte Klassen und Schemata verwalten müßte, erscheinen uns hier jedoch nicht gerechtfertigt, da die Problematik bis auf den Fall sich zyklisch referenzierender Klassen durch Umordnung der Schemaänderungsprimitive innerhalb eines Schemaversionsdefinitionsblocks (**create** oder **modify schemaversion**) umgangen werden kann. Und der genannte Fall zyklischer Referenzen konnte durch eine kleine Modifikation des ODL-Parsers (siehe Abschnitt 7.1.2.3.1) erlaubt werden [Dol99, Her99].

so ist es nicht möglich, das Attribut a ausschließlich in $sv_v.c$ zu löschen (siehe Abschnitt 5.5.4). Eine Möglichkeit wäre nun, das Attribut in der Oberklasse von $sv_v.c$ zu löschen, in der es definiert wird. Damit hätte die Integration einer Klasse, welche ein geerbtes Attribut mit einem nicht existierenden Typ redefiniert, allerdings sogar die Löschung des Attributes in Oberklassen der zu integrierenden Klasse zur Folge. Da dies eine sehr weitreichende und in den meisten Fällen ungewollte Konsequenz einer Integration wäre, löschen wir in solchen Fällen lediglich die Redefinition des Attributes a aus $sv_v.c$. Damit erhält es denselben Typ wie in einer Oberklasse von $sv_v.c$. Auf diese Weise ist das Attribut in der integrierten Klasse (und eventuell in ihren Unterklassen) allgemeiner als in der Quellschemaversion sv_u, es ist jedoch zumindest vorhanden.

Insbesondere bei der impliziten Mitintegration zahlreicher Klassen in eine Schemaversion sv_v kann es passieren, daß bereits entsprechende Klassen in sv_v enthalten sind. Defaultmäßig (Integrationsoption **keep**) werden bereits in sv_v vorhandene Klassen nicht erneut integriert, sondern die vorhandene Klassenversion wird stattdessen unverändert beibehalten und im Falle der impliziten Integration entsprechend als Ober-, Unter- oder Komponentenklassen verwendet. Alternativ kann jedoch durch Angabe der Option **replace** bestimmt werden, daß passende, bereits in sv_v vorhandene Klassen erneut integriert werden sollen, was quasi ein Anpassen der vorhandenen Klasse in sv_v an die Definition der aus sv_u integrierten Klasse zur Folge hat. Dies kann insbesondere dann erwünscht sein, wenn Mengen von Klassen aus verschiedenen Schemaversionen integriert werden, zwischen denen Vererbungs- oder Aggregationsbeziehungen bestehen. Dann sollten nämlich zunächst alle Klassen aus einer Quellschemaversion integriert und dann einige davon durch die Spezifikationen der anderen Quellschemaversion(en) ersetzt werden. Ansonsten würden zwischenzeitlich die oben erwähnten Schritte zur Erhaltung der Vollständigkeit unternommen und Beziehungen zu Ober- und zu Komponentenklassen gelöscht. Diese müßten dann manuell und mit erheblichem Aufwand nach der Integration der Klassen aus den anderen Quellschemaversionen erneut spezifiziert werden.

Bisher war nur davon die Rede gewesen, daß *entsprechende* Klassen in der Zielschemaversion bereits vorhanden seien oder mitintegriert würden. Um diese Formulierung präzisieren zu können, erinnern wir zunächst daran, daß Klassen sich in einer Schemaversion sowohl über eine Identität als auch über einen Namen identifizieren lassen und daß keine bijektive Beziehung zwischen Identitäten und Namen von Klassen bzw. von Klassenversionen besteht. Einerseits kann dieselbe Klasse in verschiedenen Schemaversionen verschiedene Namen haben und andererseits kann derselbe Name in verschiedenen Schemaversionen für Versionen verschiedener Klassen vergeben werden. Damit können bei jeder explizit oder implizit von einer Schemaversion sv_u nach sv_v zu integrierenden Klasse c die folgenden beiden Konfliktfälle auftreten.

- Ein *Identitätskonflikt* tritt auf, wenn bereits eine Klasse d namens `cname` in sv_v enthalten ist, die jedoch nicht mit c identisch ist ($name(sv_u.c) = name(sv_v.d) \wedge c \neq d$).
- Ein *Namenskonflikt* tritt auf, wenn die zu integrierende Klasse c zwar bereits in sv_v enthalten ist,[85] dort allerdings aufgrund einer Umbenennung einen anderen Namen besitzt ($name(sv_u.c) \neq name(sv_v.c)$).

Durch die Auswahl der alternativen Optionen **by classid** (Default) und **by classname** kann der Schemaentwickler bestimmen, wie mit Konflikten bei der Integration zu verfahren ist.

Durch die Option **by classid** wird ausgedrückt, daß Klassen entsprechend ihrer Identität zu vergleichen sind. Tritt dabei ein Identitätskonflikt auf, d.h. es existiert bereits eine andere Klasse d mit dem gewünschten Namen `cname`, so wird die Klasse c in sv_v unter dem neuen Namen

[85] Wir erinnern hier explizit daran, daß jede Klasse zu jeder Zeit nur einmal und damit auch nur von einer Schemaversion integriert sein darf. Ansonsten wäre die Ableitungsstruktur zwischen den Versionen einer Klasse i.Allg. kein Klassenableitungsbaum nach Definition 5.30.

`svuname_cname` integriert und sämtliche, von sv_u übernommenen Referenzen (bezüglich Vererbung oder Aggregation) auf diese Klasse werden entsprechend angepaßt, so daß sie zwar weiterhin auf die Klasse c verweisen, diese nun aber mit ihrem neuen Namen spezifizieren. Wir sagen dann, die Referenzen würden *umbenannt*. Bei einem Namenskonflikt ist die gewünschte Klasse bereits in der Zielschemaversion enthalten und muß daher nicht mehr integriert werden; sie hat jedoch einen anderen Namen. Hier werden lediglich die Referenzen auf c als Ober- oder Komponentenklasse umbenannt, d.h. sie werden so angepaßt, daß sie dieselbe Klasse wie in sv_u referenzieren, jedoch mit deren verändertem Namen. Es kann allerdings ebenfalls passieren, daß Identitäts- und Namenskonflikt gleichzeitig auftreten, d.h. in der Zielschemaversion existiert neben der Klassenversion $sv_v.d$ mit dem gewünschten Namen `cname` auch eine Version der gewünschten Klasse c, allerdings unter einem anderen Namen. In diesem Falle wird bei der Verwendung der Option **by classid** wie beim reinen Namenskonflikt verfahren und die Referenzen umbenannt, so daß sie in Quell- und Zielschemaversion auf dieselbe Klasse verweisen.

Damit sind die integrierten Klassenversionen im Vergleich zu ihren Originalen in der Quellschemaversion inhaltlich unverändert. Bei der Ausgabe der Zielschemaversion anhand eines ODL-Ausdruckes ist dies jedoch aufgrund der ggf. durchgeführten Umbenennungen nicht sofort erkennbar.[86]

Wird hingegen die Option **by classname** gewählt, so werden Vergleiche ausschließlich anhand des Klassennamens durchgeführt. Tritt dabei ein Identitätskonflikt auf, d.h. eine andere Klassenversion $sv_v.d$ hat den gewünschten Namen *cname*, so werden Referenzen der integrierten Klassen von c auf die bereits in sv_v vorhandene Klasse d umgebogen und nichts integriert. Tritt ein Namenskonflikt ein, d.h. die gewünschte Klasse ist bereits integriert, jedoch unter einem anderen Namen, so kommt es darauf an, ob gleichzeitig ein Identitätskonflikt auftritt, d.h. ob in der Zielschemaversion sv_v bereits eine andere Klasse mit dem gewünschten Namen vorhanden ist oder nicht. Ist dies nicht der Fall, so wird die Klasse c verwendet und die Referenzen werden entsprechend umbenannt. Es würde nämlich wenig Sinn machen und weiterhin auch Invariante 5.12 verletzen, die Klasse ein zweites Mal (unter ihrem anderen Namen $name(sv_u.c)$) nach sv_v zu integrieren. Treten Identitäts- und Namenskonflikt gleichzeitig auf, so werden bei Verwendung der Option **by classname** ebenfalls die Referenzen auf c als Ober- oder Komponentenklasse angepaßt, allerdings wird hier die Referenz so geändert, daß sie in sv_v auf eine Version einer anderen Klasse ($d \neq c$) verweist als in sv_u. Diese hat diesmal allerdings denselben Namen, was in der ODL den fälschlichen Eindruck erwecken kann, in sv_v werde dieselbe Klasse referenziert wie in der Quellschemaversion sv_u.

Abbildung 5.7 faßt die obigen Ausführungen tabellarisch zusammen.

Durch die Benutzung des **integrate**-Primitives wird $sv_v.c <^1_{sv,c} sv_u.c$ und dementsprechend wird sv_u in die Menge der Vorgängerschemaversionen von sv_v aufgenommen ($sv_v <^1_{sv} sv_u$). Werden in wenigstens einer Schemaversion $sv \in sv(s)$ Klassen von verschiedenen $sv_{u_1}, \ldots, sv_{u_m}$ $(m > 1)$ integriert, so entsteht also ein allgemeiner Schemaableitungsgraph, ansonsten handelt es sich um den Spezialfall eines Schemaableitungsbaumes. Wird das **integrate**-Primitiv bei der Ableitung einer Schemaversion sv gar nicht benutzt, so wird sv direkte Nachfolgerschemaversion von sv_0.

Durch den Verzicht auf eine optionale Übernahme einer vollständigen Schemaversion sv_u bei der Ableitung von sv_v, wie dies beispielsweise durch ein Primitiv in der Art von "**create schemaversion** svname **from** parentsvname;" möglich gewesen wäre, erreichen wir übrigens eine vollkommen symmetrische Anwendung des **integrate**-Primitives bei der Definition einer neuen Schemaversion, d.h. alle Vorgängerschemaversionen von sv_v werden gleichbehandelt. Trotzdem

[86] Dies liegt daran, daß die Identitäten der Klassen und Attribute in der ODL, wie auch in anderen Schemabeschreibungssprachen, nicht ausgedrückt werden können und die stattdessen verwendeten Namen abweichenden Gesetzmäßigkeiten unterliegen. Wir werden in Abschnitt 5.5.7 (siehe Abbildung 5.11) auf Probleme stoßen, die derselben Ursache zuzurechnen sind.

Fall	Konflikte	$\exists sv_v.c$: $name(sv_v.c) =$ `cname`	$\exists sv_v.c$: $name(sv_v.c) \neq$ `cname`	$\exists sv_v.d$: $name(sv_v.d) =$ `cname`	mit der Option **by classid**	mit der Option **by classname**
1	keine Konflikte				$sv_u.c$ integrieren	
1'	keine Konflikte	x			—[a]	
2	Identitätskonflikt			x	$sv_u.c$ integrieren als `svuname_cname` und Referenz umbenennen	Referenz auf d umbiegen[b]
3	Namenskonflikt		x		Referenz in $name(sv_v.c)$ umbenennen[a]	
4	Namens- und Identitätskonflikt		x	x	Referenz in $name(sv_v.c)$ umbenennen[a]	Referenz auf d umbiegen[b]

[a] Bei Option **replace** ist $sv_u.c$ zuvor durch erneute Integration zu aktualisieren.
[b] Bei Option **replace** ist $sv_u.d$ zuvor durch erneute Integration zu aktualisieren.

Abbildung 5.7: Mögliche Konflikte und deren Behebung bei der Integration einer Klasse c namens `cname` von sv_u nach sv_v.

kann die komplette Übernahme einer existierenden Schemaversion in nur einem zusätzlichen ODL-Ausdruck erreicht werden, indem die Klasse **Object** und ihre Unterklassen und damit alle Klassen von sv_u übernommen werden:

```
create schemaversion svname {
    integrate class Object from parentsvname subclasses;
    ...
    } /* create schemaversion svname */
```

Kopieren einer vollständigen Schemaversion.

Die Operation **integrate** bewirkt, daß die Quell- und Zielklassenversionen als Versionen derselben Klasse aufgefaßt werden. Jede Klasse kann in einer Schemaversion nur höchstens einmal vorliegen, daher werden bereits in der Zielschemaversion vorhandene Klassen nicht wirklich ein zweites Mal integriert sondern höchstens aktualisiert (Option **replace**). Gegebenenfalls sind Namensanpassungen wie oben beschrieben notwendig.

5.5.3.2 Anlegen von Klassen

Das Anlegen neuer Klassen geschieht analog dem Anlegen von Schemaversionen.

```
create class cname {
    ...          // Klassendefinitionsblock
    } /* create class cname */
```

Anlegen von Klassen.

Im Klassendefinitionsblock können die Primitive der Attributebene angewendet werden.

Klassen können mit dem **create class**-Primitiv nur als Blätter der Klassenvererbungsgraphen angelegt werden. Soll eine Klasse als interner Knoten in den Vererbungsgraphen integriert werden, so sind ihre direkten Nachfolger manuell zu modifizieren, indem die neue Klasse zusätzlich in die Mengen ihrer direkten Oberklassen aufgenommen wird.

5.5.3.3 Verändern von Klassen

Wie beim Anlegen wird auch beim Verändern von Klassen ein Klassendefinitionsblock verwendet.

```
modify class cname {
    ...          // Klassendefinitionsblock
    } /* modify class cname */
```

Verändern von Klassen.

Bei der Veränderung einer Klasse treten im Klassenmodifikationsblock so wie beim Anlegen die Primitive der ODL auf Attributebene auf. In Abschnitt 5.5.4 werden wir erläutern, welche Effekte eine Klassenänderung auf ggf. existierende Unterklassen haben kann und wie für die Einhaltung der Invarianten gesorgt werden kann.

5.5.3.4 Umbenennen und Löschen von Klassen

Das Umbenennen und Löschen von Klassen geschieht analog der entsprechenden Primitive auf Schema- und Schemaversionsebene. Im Unterschied zur Schemaversionsebene, wo nur Blätter des Ableitungsgraphen gelöscht werden können, erlauben wir hier auch das Löschen von Klassenversionen mitsamt ihrer Nachfolger im Vererbungsgraphen.

```
rename class cname to newcname;

delete class cname [ [no] subclasses];
```

Umbenennen und Löschen von Klassen.

Beim Umbenennen einer Klasse können die Referenzen in abhängigen Klassen automatisch angepaßt werden.[87] Auch Methoden, die eine umbenannte Klasse verwenden, können automatisch angepaßt werden.

[87] Diese Anpassung erfordert in COAST keinerlei zusätzlichen Aufwand, da der Schemamanager Beziehungen zwischen Klassen im Metaschema durch Referenzen auf Identifikatoren von Klassen (und nicht auf Klassennamen) realisiert. Bei einer späteren Ausgabe des Schemas, z.B. in eine ODL-Datei, wirkt sich die Umbenennung somit automatisch aus.

Eine Klasse kann nur dann gelöscht werden, wenn dadurch die Vollständigkeit der betreffenden Schemaversion nicht beeinträchtigt wird. Daher müssen korrigierende Maßnahmen ergriffen werden, wenn die zu löschende Klasse c anderen Klassen als Ober- oder Komponentenklasse dient oder wenn sie in Methoden anderer Klassen verwendet wird.

Beim Löschen eines inneren Knoten c des Klassenvererbungsgraphen sind nach Brèche [Brè96] zwei voneinander unabhängige Entscheidungen zu treffen, so daß sich vier Varianten des Löschens ergeben. Zum einen besteht die Möglichkeit, die direkten Unterklassen $c''_1, \ldots, c''_n$ der zu löschenden Klasse c zu direkten Unterklassen der direkten Oberklassen $c'_1, \ldots, c'_m$ von c zu machen (Option **reconnect**). Alternativ geht die zuvor indirekt bestandene Vererbungsbeziehung zwischen den direkten Ober- und direkten Unterklassen von c verloren und letztere müssen ggf. zur Erhaltung des Zusammenhanges des Vererbungsgraphen (Definition 5.9) zu direkten Unterklassen von **Object** gemacht werden. Zum anderen ist festzulegen, ob die in c lokal definierten Attribute und Methoden als lokal definierte Eigenschaften in die direkten Unterklassen von c übernommen werden (Option **propagate**) sollen (sofern dies möglich ist), so daß diese Eigenschaften in allen Unterklassen von c erhalten bleiben. Alternativ gehen die Eigenschaften in den Unterklassen von c verloren. In analoger Weise kann durch die Option **propagate** auch bestimmt werden, wie mit Überschreibungen von Attributen und Methoden verfahren werden soll, die die Klasse c vornimmt.

Da wir hier auf die Wiederholung der Ausführungen aus [Brè96] verzichten möchten und diese keine weitere Relevanz für unsere Konzepte haben, legen wir für unser Modell fest, daß stets mit der Semantik von **reconnect** und **no propagate** gelöscht wird. Bei Angabe der Option **subclasses** werden sämtliche Unterklassen mitgelöscht, womit deren Anpassung ohnehin hinfällig wird.

Sollte die gelöschte Klasse c als Komponentenklasse einer (oder mehrerer) anderen Klasse d gedient haben, so sind deren Referenzen anzupassen. Dabei gehen wir hier genauso vor, wie bei durch die Integration verursachten Unvollständigkeiten. Ist das c referenzierende Attribut in d lokal angelegt, so wird es gelöscht und verschwindet damit auch aus allen Unterklassen von d. Wurde das Referenzattribut in d jedoch geerbt und mit dem Typ von c spezialisiert, so wird lediglich die Redefinition gelöscht.

Wurde die gelöschte Klasse c in einer Methode einer Klasse verwendet, die keine Unterklasse von c ist und die daher nicht automatisch mitgelöscht wird, so kann diese Methode in der vorliegenden Form nach der Schemaänderung nicht mehr verwendet werden. Da eine automatische Anpassung der Methoden nicht möglich und eine automatische Löschung sicher i.Allg. nicht erwünscht ist, müssen die Schemaentwickler manuell eingreifen. Spätestens beim nächsten Versuch zu Compilieren werden sie auf die betroffenen Methoden aufmerksam gemacht.

5.5.4 Primitive der COAST-ODL auf Attributebene

Die Attributebene subsumiert alle Primitive, die im Klassendefinitionsblock einer einzelnen Klasse auftreten dürfen. Diese dienen der Spezifikation und Veränderung von direkten Oberklassen, Attributen und Methoden.

5.5.4.1 Spezifikation und Veränderung der Oberklassen einer Klasse

Das Hinzufügen und Entfernen direkter Oberklassen geschieht mit den folgenden Primitiven.

create inherit `cname;`

delete inherit `cname;`

Verändern von Vererbungsbeziehungen.

Beim Hinzufügen einer Oberklasse zu einer Klasse c werden c und ihre Unterklassen um die Eigenschaften (Attribute und Methoden) der neuen Oberklasse erweitert. Dies kann zu Vererbungskonflikten in c und ihren Unterklassen führen. Erbt die Klasse c ein dort bereits vorhandenes Attribut a nun zusätzlich von der neuen Oberklasse, so kann dies natürlich entsprechend unserem Modell der Mehrfachvererbung zu einer Spezialisierung des Typs dieses Attributes in c und deren Unterklassen führen. Ist das betroffene Attribut dort explizit redefiniert (**overwrite attribute**), so tritt eine Verletzung der strukturellen Typkonformität (Invariante 5.7) ein, wenn der sich nun ergebende, speziellere Typ von a spezieller ist als die explizite Redefinition. In solchen Fällen muß die Hinzufügung der Oberklasse abgelehnt werden. Von der Alternative der automatischen Entfernung solcher, die strukturelle Typkonformität verletzender, expliziter Überschreibungen sehen wir hier ab, da dies eine vom Schemaentwickler nicht erwartete und ggf. schwer nachvollziehbare Konsequenz bedeutete, die darüberhinaus nicht auf die modifizierte Klasse beschränkt bliebe, sondern gleichermaßen auch deren Unterklassen betreffen könnte. Auch das Erben zusätzlicher Methoden kann zu Konflikten führen, wenn unsere Regeln bezüglich Eindeutigkeit, Überschreibung oder Überladung von Methoden verletzt werden.

Hatte die Klasse c zuvor eine Oberklasse d, die auch Oberklasse der Klasse namens `cname` ist (siehe Abbildung 5.8), so wird d in der Oberklassenmenge von c durch die Klasse namens `cname` ersetzt, damit die Minimalität der Vererbungsstruktur (Invariante 5.5) gewahrt bleibt. Dies gilt insbesondere für $d =$ **Object**. Die Klasse d ist dann lediglich noch indirekte Oberklasse von c. Ebenfalls zur Erhaltung der Minimalität wird die Operation **create inherit** `cname` abgelehnt, wenn die Klasse namens `cname` bereits eine (direkte oder indirekte) Oberklasse von c ist.

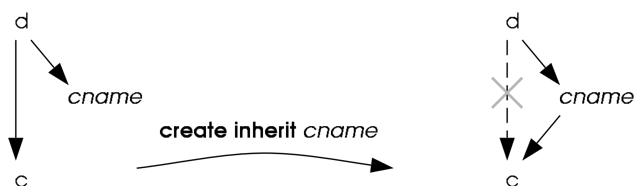


Abbildung 5.8: Um die Minimalität des Klassenvererbungsgraphen zu bewahren, wird die Klasse d aus der Menge der direkten Oberklassen von c entfernt, wenn die Klasse namens `cname` dort eingefügt wird.

Beim Entfernen einer Klasse aus der Oberklassenmenge einer Klasse c gehen die geerbten Eigenschaften in c und ihren Unterklassen verloren, wenn sie nicht mehrfach geerbt wurden. Dabei werden in c und ihren Unterklassen insbesondere auch alle Überschreibungen von Attributen entfernt, die nach Löschung der Vererbungsbeziehung nicht mehr geerbt werden. Dies ist notwendig zur Erhaltung der strukturellen Typkonformität der Klassenvererbung (zweite Teilbedingung in Invariante 5.7).[88] Gehen durch den Verlust einer Oberklasse zuvor geerbte Methoden verloren, so kann dies zu Unvollständigkeiten in sämtlichen Klassen führen, die Methoden von c verwenden.

[88] Im Gegensatz zu vielen objektorientierten Sprachen unterscheidet unsere Schemabeschreibungssprache syntaktisch zwischen dem Anlegen (**create attribute**) und dem Überschreiben (**overwrite attribute**) eines Attributes.

Hat eine Klasse nur **Objekt** als einzige Oberklasse, so kann diese nicht aus der Oberklassenmenge entfernt werden, d.h. **delete inherit Object** ist niemals zulässig. Ist die Klasse namens `cname` die einzige in der Oberklassenmenge von c, so wird sie bei **delete inherit** `cname` durch **Object** ersetzt, damit der Zusammenhang der Klassenvererbungsgraphen (Invariante 5.4) gewahrt bleibt.

Es ist hier zu beachten, daß die Primitive **create** und **delete inherit** keine Inversen voneinander sind, d.h. die Ausführung des einen kann nicht durch das andere aufgehoben werden. Dies liegt an den genannten korrigierenden Maßnahmen, die bei dem jeweils anderen Primitiv nicht rückgängig gemacht werden können.

Es wird in der Praxis relativ häufig der Wunsch auftreten, eine *Spezialisierung einer Oberklasse* c' einer Klasse c vorzunehmen, d.h. c' in der Oberklassenmenge von c durch eine Unterklasse c'' von c' zu ersetzen. Sei beispielsweise die Klasse `Manager` (c) Unterklasse von `Person` (c') und es werde die neue Klasse `Employee` (c'') angelegt, die ebenfalls von `Person` erbe (siehe Abbildung 5.9). Dann wird es angebracht sein, die Oberklasse `Person` der Klasse `Manager` zu `Employee` zu spezialisieren, so daß schließlich gilt: `Manager` $<_c$ `Employee` $<_c$ `Person`. Mit der Spezialisierung der Oberklasse geht also eine Spezialisierung der Klasse `Manager` selbst einher. Analog kann umgekehrt der Fall einer *Generalisierung einer Oberklasse* auftreten.

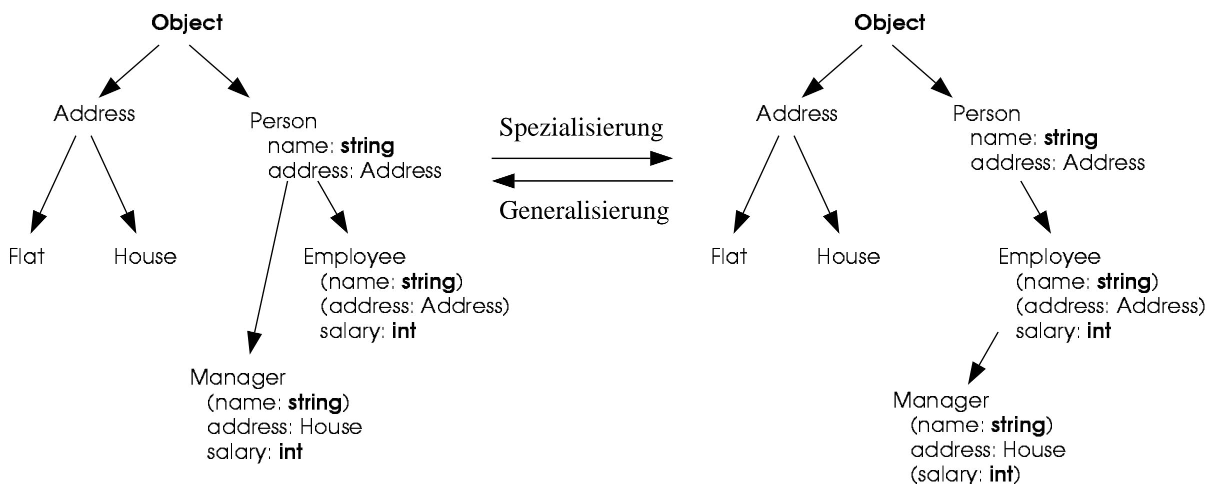


Abbildung 5.9: Beispiel der Spezialisierung und der Generalisierung der Oberklasse von **Manager**.

Sowohl die Spezialisierung als auch die Generalisierung können durch Löschen der bisherigen Oberklasse aus und anschließendes Einfügen der neuen Oberklasse in die Oberklassenmenge der zu verändernden Klasse c erreicht werden, d.h. durch ein **delete inherit** gefolgt von einem **create inherit**.[89] Dabei tritt jedoch das Problem auf, daß die Oberklassenbeziehung zu c' und zu c''

Daher werden wir beim Löschen von Vererbungsbeziehungen übrig gebliebene Überschreibungen von Attributen, die danach nicht mehr geerbt werden, erkennen und entfernen können. In ORION (siehe Abschnitt 4.5.3.7) existiert keine Invariante, die das Überschreiben nicht geerbter Attribute verbietet. Deshalb und aufgrund der syntaktischen Gleichheit zwischen Attributerzeugung und -überschreibung ist anzunehmen, daß bei ORION vorherige Überschreibungen nach dem Löschen der Vererbungsbeziehung automatisch in Erzeugungen von Attributen übergehen, wobei die textuelle Beschreibung der Attribute der vormaligen Unterklasse unverändert bleibt. Das Löschen einer Vererbungskante kann der ORION-Vorgehensweise folgend allerdings zu einem Vererbungskonflikt in einer Unterklasse führen, wenn dort die Eindeutigkeit der Herkunft geerbter Attribute (siehe ORION-Invariante 4.3) verloren geht. Durch die Einführung zweier alternativer Optionen **delete** und **create** könnte man dem Schemaentwickler beim Entfernen von Oberklassen die Wahl zwischen unserer Vorgehensweise (Löschen von Überschreibungen)und der von ORION (Umsetzung von Überschreibungen in Erzeugungen) überlassen.

[89] In dem in Abbildung 5.9 dargestellten Beispiel träte bei sofortigem Einfügen der Vererbungskante zwischen `Manager` und `Employee` nach den Invarianten 5.1 und 5.2 ein Namens- bzw. Herkunftskonflikt auf, da `Manager`

zwischenzeitlich komplett verloren geht, da dann weder c' noch c'' Oberklasse von c sind. Damit müßten aus Gründen der Konsistenzerhaltung, insbesondere wegen der zweiten Teilbedingung der strukturellen Typkonformität (Invariante 5.7) alle Überschreibungen zwischenzeitlich nicht mehr geerbter Attribute entfernt werden.[90] Damit ginge im Beispiel der Abbildung 5.9 die Information verloren, daß Manager nicht in Wohnungen sondern in Häusern wohnen. Da solche Überschreibungen nach dem Hinzufügen der neuen Oberklassen nicht mehr rekonstruiert werden können, wären sie nach der Spezialisierung oder Generalisierung eigentlich verloren, obwohl sie semantisch weiterhin sinnvoll wären.[91] Damit dieser unerwünschte Effekt nicht eintritt, führen wir einen eingeschränkten Transaktionsmechanismus ein, der sich allerdings lediglich auf die Konsistenz bezieht.[92] Dieser verschiebt die Konsistenzprüfung und damit das Entfernen nach Invariante 5.7 nicht gestatteter Überschreibungen bis zu dem Zeitpunkt, an dem die Veränderung einer Klasse abgeschlossen ist, d.h. bis ans Ende des Klassendefinitionsblocks einer **modify class**-Operation. Mit diesem Transaktionsmechanismus sind beliebige Veränderungen der Oberklassenmenge einer Klasse c möglich, wobei anschließend semantisch noch sinnvolle Überschreibungen voll erhalten bleiben. Insbesondere kann eine Klasse c' in der Oberklassenmenge von c damit auch durch eine Klasse c'' ersetzt werden, die weder Unterklasse (dies entspräche einer Spezialisierung) noch Oberklasse (dies entspräche einer Generalisierung) von c' ist. Die in Abbildung 5.9 dargestellte Spezialisierung der Oberklasse von `Manager` kann damit durch die folgende Anweisung erreicht werden.

```
modify class Manager {
    delete attribute salary;
    delete inherit Person;
    create inherit Employee;
    } /* modify class Manager */
```

Beispiel einer Spezialisierung.

Es sei abschließend bemerkt, daß die hier beschriebene Problematik durch einen Verzicht auf die Minimalität des Klassenvererbungsgraphen (Invariante 5.5) hätte komplett vermieden werden können.[93] Dann kann die Hinzufügung einer oder mehrerer neuer Oberklassen nämlich vor dem Löschen der alten Klassen aus der Oberklassenmenge von c durchgeführt werden. Um die Minimalität des Klassenvererbungsgraphen herzustellen, könnte man ergänzend ein explizites Primitiv namens **minimise** oder **reduce** anbieten. Die hier gemachten Ausführung mögen jedoch als Beispiel dafür dienen, wie weitreichend die Konsequenzen einer vermeintlich unscheinbaren Veränderung des Objektmodells sein können.

---

dann über zwei gleichnamige Attribute `salary` unterschiedlicher Identitäten verfügte. Daher muß das in der Klasse `Manager` lokal definierte Attribut `salary` zuerst gelöscht werden (**delete attribute**).

[90] Bei der Spezialisierung einer Oberklasse tritt das hier beschriebene Problem genau genommen nicht auf, da dort auf das vorausgehende **delete inherit** verzichtet werden kann. Stattdessen genügt es, die speziellere Oberklasse durch ein **create inherit** in die Oberklassenmenge von c einzufügen. Um die Erhaltung der Minimalität der Vererbungsstruktur nach Invariante 5.5 zu gewährleisten, würde die generellere Klasse c' beim Einfügen ihrer Unterklasse c'' automatisch aus der Oberklassenmenge von c entfernt (siehe Abbildung 5.8).

[91] Bei der Spezialisierung gilt dies für alle Überschreibungen in c. Bei der Generalisierung, d.h. beim Ersetzen einer Klasse c'' durch eine Oberklasse $c' >_c c''$ davon in der Oberklassenmenge der Klasse c sind die in c'', nicht aber in c' vorhandenen Eigenschaften anschließend nicht mehr in c enthalten (es sei denn, sie wurden mehrfach geerbt) und die sie betreffenden Überschreibungen müssen auf jeden Fall entfernt werden.

[92] Ähnliche Vorschläge wurden in [SGD93] für NO^2 und in [Zic91b] für ESSE gemacht.

[93] Dies wurde in der Implementierung (siehe Kapitel 7) auch tatsächlich gemacht.

5.5.4.2 Spezifikation und Veränderung der Attribute einer Klasse

Für das Hinzufügen, Modifizieren, Umbenennen und Löschen von Attributen stehen die folgenden Primitive zur Verfügung.

create attribute `aname: typename;`

overwrite attribute `aname to typename;`

delete overwrite attribute `aname;`

retype attribute `aname to typename;`

rename attribute `aname to newaname;`

delete attribute `aname;`

Primitive für strukturelle Eigenschaften von Objekten.

Das Primitiv **overwrite** dient zum Überschreiben des Typs eines geerbten Attributes. Wie wir bereits in Invariante 5.7 festgelegt hatten, ist dabei nur eine Spezialisierung des Attributtyps erlaubt. In verschiedenen objektorientierten Programmiersprachen wird ein solches Überschreiben syntaktisch genau wie das Anlegen eines neuen Attributes bewerkstelligt. Da das System das Schema kennt, kann es immer entsprechend reagieren: Existiert ein Attribut mit dem angegebenen Namen bereits, so wird dieses überschrieben, ansonsten wird es neu angelegt. In O_2 (siehe Abschnitt 4.3.4.2) wird sogar die Änderung des Typs eines Attributes im Rahmen der Schemaevolution mit derselben Syntax gemacht. Wir haben für diese drei Aufgaben allerdings mit **create**, **overwrite** und **retype** drei verschiedene Primitive eingeführt. Dies dient der Sicherheit des Schemaentwicklers, die eines der Ziele beim Entwurf der ODL darstellte [Her99]. Damit kann das System nämlich erkennen, ob der Schemaentwickler davon ausgeht, daß noch kein Attribut mit dem von ihm angegebenen Namen (`aname`) existiert (dann verwendet er **create**), oder ob er annimmt, daß das Attribut bereits existiert und lokal angelegt (dann verwendet er **retype**) oder geerbt (dann verwendet er **overwrite**) ist. Damit kann das System den Schemaentwickler mit einer Fehlermeldung aufmerksam machen, falls dieser von einer falschen Annahme ausgeht.

Das Primitiv **create attribute** `aname` dient dem Hinzufügen eines neuen Attributes a. Wenn dadurch die Eindeutigkeit der Namen in der Klasse c verletzt wird, muß stattdessen **overwrite** oder **retype attribute** verwendet werden. Weiterhin kann es in Unterklassen von c zu Konflikten kommen. Auch dann ist die Hinzufügung eines Attributes nicht möglich.

Das Primitiv **overwrite attribute** dient ausschließlich dem Überschreiben geerbter Attribute. Dabei darf der Typ des Attributes im Vergleich zu dem ohne die Redefinition geerbten Typ nur spezialisiert werden; es muß jedoch gleichzeitig allgemeiner bleiben als ggf. vorhandene, explizite Spezialisierungen in Unterklassen der Klasse c (Invariante 5.7). Mit **delete overwrite** kann eine in der betrachteten Klasse bestehende Redefinition eines Attributtyps entfernt werden.

Das Primitiv **retype attribute** ist nur dort anzuwenden, wo ein Attribut lokal definiert wurde. Im Gegensatz zu **create** und **overwrite attribute** ist **retype** bei der ersten Definition eines Schemas nicht notwendig und findet demzufolge auch keine Entsprechung in Systemen ohne Evolutionsprimitive, etwa in Programmiersprachen. Stattdessen wird **retype** ausschließlich zur Veränderung bestehender, zuvor explizit spezifizierter Attributtypen verwendet.

Die Primitive **rename** und **delete attribute** dürfen wie **retype** nur dort verwendet werden, wo ein Attribut lokal definiert wurde und wirken sich auf alle Unterklassen von c aus. Gegebenenfalls dort vorhandene Überschreibungen werden entsprechend angepaßt bzw. mitgelöscht.

Sämtliche hier beschriebenen Primitive können sich auch auf existierende Methoden auswirken. Der Name eines anzulegenden Attributes kann bereits für eine Methode vergeben sein,[94] die verschiedenen Formen der Modifikation eines Attributtyps können zu Typkonflikten bei der Übersetzung von Methoden führen, genau wie Umbenennungen und Löschungen entsprechende Unvollständigkeiten hervorrufen können. Umbenennungen und Löschungen von Attributen können analog behandelt werden, wie die entsprechenden Primitive für Klassen (siehe Abschnitt 5.5.3.4).

5.5.4.3 Spezifikation und Veränderung der Methoden einer Klasse

Das Hinzufügen, Verändern, Umbenennen und Löschen von Methoden geschieht wie folgt.

```
create method mname {
    ...          // Methodendefinitionsblock
    } /* create method mname */;

modify method mname {
    ...          // Methodendefinitionsblock
    } /* modify method mname */;

rename method mname to newmname;

delete method mname;
```

Primitive für verhaltensmäßige Eigenschaften von Objekten.

Wir gehen in dieser Arbeit nicht näher auf die Spezifikation von Methoden ein, setzten der Einfachheit halber allerdings voraus, daß auch beim Verändern einer Methode eine vollständige Spezifikation (wie beim Hinzufügen) angegeben wird.

In COAST werden zur Zeit nur die Namen und die Signaturen, nicht jedoch die Implementierungen der Methoden intern verwaltet. Eine Signatur besteht entsprechend Definition 5.12 aus Name der Klasse, der die Methode zugeordnet ist, Typ des Ergebnisses und einer (möglicherweise leeren) Liste von Eingabeparametern, die jeweils mit Namen und Typ spezifiziert werden.

Da Methoden im Gegensatz zu Attributen keine Zustände beschreiben, die zwischen verschiedenen Schemaversionen zu propagieren sind, können Methoden verschiedener Schemaversionen relativ unabhängig voneinander spezifiziert und in einer separaten Datei abgelegt werden, wobei die Implementierungen die in der ODL spezifizierten Signaturen natürlich einhalten müssen. Um die interne Verwaltung auch der Implementierungen konzeptionell unterstützen zu können, werden hier Primitive zum Umbenennen von Methoden und zum Verändern ihrer Implementierung angegeben, obwohl diese Funktionalitäten auch durch eine Kombination von Löschen und neu Anlegen von Methoden erbracht werden könnte. Neben dem durch die Primitive **modify** und **rename method** erreichten Vorteil einer konzeptionellen Vollständigkeit der ODL im Bezug auf Methoden kann das Umbenennen einer Methode automatisch auch in den Unterklassen der betroffenen Klasse durchgeführt werden.

Die beschriebene externe Verwaltung der Implementierungen von Methoden hat zwar für den in dieser Arbeit vertieft betrachteten Aspekt der Propagation von Datenbankzuständen keine Auswirkung; sie erschwert allerdings die Verwaltung der Methoden. Insbesondere die Ausführung von Primitiven zum Umbenennen oder Löschen von Attributen oder Methoden wirkt sich direkt auf Methoden aus, die diese Attribute und Methoden benutzen. Der Code der dienstnehmenden Methoden muss nämlich entsprechend angepaßt (bei Umbenennungen) oder als nicht

[94] Invariante 5.10 hatte explizit das Überschreiben von Methoden durch Attribute und umgekehrt ausgeschlossen.

mehr übersetzbar (bei Löschungen) gekennzeichnet werden. Derartige Anpassungen sind auch bei externer Verwaltung der Implementierungen von Methoden durchführbar, es besteht jedoch keine Möglichkeit, Änderungen die im Betriebssystem direkt auf Dateiebene oder mit einem Texteditor durchgeführt werden, geeignet zu kontrollieren oder gar einzuschränken.

Man könnte die eigentliche Ursache der geschilderten Problematik als mangelnde Kontrolle des Datenbanksystems über andere, mit ihm eng verbundene Komponenten beschreiben. Aus dieser Perspektive zeigt sich eine gewisse Ähnlichkeit zu einem bereits geschilderten Umstand: Sowohl für die Frage, ob eine Schemaänderung ohne Ableitung einer neuen Schemaversion durchgeführt werden kann (siehe Abschnitt 5.4.2 und [FL96]), als auch für die Frage, ob eine eingefrorene Schemaversion aufgetaut werden darf (siehe Fußnote 82 in Abschnitt 5.5.2.4), müßten die tatsächlich für eine Schemaversion existierenden Applikationen bekannt sein. Derartige Kenntnisse hatten wir jedoch nicht vorausgesetzt. Die erwähnte Ähnlichkeit endet jedoch bei der Ursache der mangelnden Kontrolle. Kenntnis über existierende Applikationen können wir konzeptionell nicht erwarten, da wir ein programmiersprachen- und Compiler-unabhängiges COAST anstreben und weiterhin vom Betriebssystem keine Angaben über Erstellung und Löschung von Applikationen erwarten. Im Gegensatz dazu ist die oben beschriebene externe Verwaltung der Methodenimplementierungen lediglich ein technisches Problem, das sich durch Integration eines Editors in das Datenbanksystem lösen läßt. Diese, beispielsweise von O_2 (siehe Abschnitt 4.3.4.2) realisierte Lösung kam für COAST allein aus Aufwandsgründen nicht in Betracht.

5.5.5 Einige Schemaänderungen in unserem Beispielszenario

Wir kommen nun auf das in Abschnitt 3.1.2 vorgestellte Szenario aus dem Umfeld einer Softwareentwicklungsumgebung zurück und erläutern, wie die dort vorgestellten Schemaversionen durch Ausdrücke unserer Schemabeschreibungssprache erstellt werden könnten.

Die Schemaversion sv_1 wurde von der systemdefinierten Wurzel sv_0 abgeleitet, was bedeutet, daß sie tatsächlich neu erstellt wurde. Dazu wurden die benutzerdefinierten Klassen `Document`, `DocuPart`, `Text` und `Pic` angelegt.

```
create schemaversion sv1 {
    create class Document {
        type tuple (
            author          : string,
            creation_date   : string,
            contents        : DocuPart);
        method ...
        } /* class Document */
    create class DocuPart {
        type tuple (
            visible_width: real,
            visible_height real);
        method ...
        } /* class DocuPart */
    create class Text : DocuPart {
        type tuple (
            contents: string,
            font    : string,
            size    : int);
        method ...
        } /* class Text */
    create class Pic : DocuPart {
        type tuple (
            contents: blob);
        method ...
        } /* class Pic */
    } /* sv1 */
```

Anlegen der Schemaversion sv_1.

Bei der Erstellung der Schemaversion sv_2 wurden die bereits in sv_1 vorhandenen Klassen (`Document`, `DocuPart`, `Text` und `Pic`) mittels des **integrate**-Primitives zunächst unverändert übernommen. Daraufhin wurden der integrierten Klasse `Document` vier zusätzliche Attribute hinzugefügt. Analog zur Erstellung von sv_1 wurden schließlich auch bei sv_2 neue Klassen (`SourceCode`, `Embedded`, `Graph`, `Arc` und `Point`) angelegt.

```
create schemaversion sv2 {
    integrate class Object from sv1 subclasses;
    modify class Document {
        attributes {
            create title           : string,
            create status          : (draft, final);
            create visible_width   : int;
            create visible_height  : int;
            }
        } /* class Document */
    create class SourceCode : Document {
        type tuple (
            contents: Text);
        method ...
        } /* class SourceCode */
    create class Embedded : DocuPart {
        type tuple (
            parts    : list of DocuPart);
        method ...
        } /* class Embedded */
    create class Graph : DocuPart {
        type tuple (
            nodes   : list of Point,
            arcs    : list of Arc,
            color   : string);
        method ...
        } /* class Graph */
    create class Arc {
        type tuple (
            begin   : Point,
            end     : Point);
        method ...
        } /* class Arc */
    create class Point {
        type tuple (
            x : real,
            y : real);
        method ...
        } /* class Point */
    } /* sv2 */
```

Ableiten der Schemaversion sv_2.

Beim Anlegen von sv_3 wurden neben den bereits zuvor benutzten Primitiven auch Umbenennungen von Klassen und Attributen durchgeführt sowie eine Vererbungskante von `KartPoint` zu ihrer Oberklasse `Point` erzeugt.

```
create schemaversion sv3 {
    integrate class Object
        from sv2 subclasses;
    create class Employee {
        type tuple (
            name    : string,
            dept    : int);
        method ...
        } /* class Employee */
    modify class Document {
        attributes {
            retype author to Employee;
            }
        }; /* modify class Document */
    rename class Point to KartPoint;
    create class Point {
        type tuple ();
        method ...
        } /* class Point */
    modify class KartPoint {
        create inherit Point;
        attributes {
            rename x to a;
            rename y to b;
            }
        }; /* modify class KartPoint */
    create class PolarPoint : Point {
        type tuple (
            distance: real,
            angle   : real);
        method ...
        } /* class PolarPoint */
    } /* sv3 */
```

Ableiten der Schemaversion sv_3.

Wie zuvor sv_3 so wurde auch sv_4 von sv_2 abgeleitet, d.h. sv_3 und sv_4 sind typische Alternativen in einem Versionierungskonzept. Die bisherige lineare Entwicklung des Schemas wird dadurch zu einem Ableitungsbaum, in dem die Alternativen nach Abschnitt 2.2 durch Geschwister dargestellt werden. Der Charakter der beiden Alternativen wird jedoch nicht nur durch die Ableitungsstruktur, sondern auch inhaltlich deutlich. Sowohl sv_3 als auch sv_4 erlauben nämlich

die Speicherung von Punkten einer graphischen Darstellung sowohl in kartesischen Koordinaten als auch in Polarkoordinaten. Die Attributmenge der Klasse `Document` wurde zur verbesserten Modellierung des Dokumentenlayouts erheblich verändert.

```
create schemaversion sv4 {
    integrate class Object from sv2 subclasses;
    modify class Document {
        attributes {
            delete visible_width;
            delete visible_height;
            create page_size     : string;
            create border_left   : int;
            create border_right  : int;
            create border_top    : int;
            create border_bottom : int;
            }
        } /* modify class Document */
    modify class Point {
        attributes {
            rename x to a;
            rename y to b;
            create point_type
                    : (kart, polar);
            }
        } /* modify class Point */
} /* sv4 */
```

Ableiten der Schemaversion sv_4.

Wir ergänzen hier das in Abschnitt 3.1.2 vorgestellte Beispielszenario noch um zwei weitere Schemaversionen sv_5 und sv_6.

Die Schemaversion sv_5 wird erst nach Fertigstellung von sv_3 und sv_4 angelegt. Damit ergibt sich die Möglichkeit, aus den beiden Alternativen jeweils diejenigen Klassenversionen zu integrieren, die dem Anforderungsprofil von sv_5 am besten entsprechen. Durch die Integration aus verschiedenen Schemaversionen wird aus dem Ableitungsbaum ein Ableitungsgraph. Wie bereits allgemein beschrieben, läßt sich die Ableitungsstruktur einer isolierten Klasse jedoch stets als Baum repräsentieren, weil das Zusammenmischen zweier Versionen derselben Klasse in unserem Modell nicht vorgesehen ist.

Die Schemaversion sv_5 soll so angelegt werden, daß sie weitgehend sv_3 entspricht. Lediglich die Modellierung von Punkten soll entsprechend sv_4 vorgenommen werden. Eine einfache Möglichkeit zur Erzeugung der gewünschten Schemaversion wäre nun, für jede zu integrierende Klasse einen eigenen **integrate**-Ausdruck zu verwenden. Wir haben uns hier jedoch für einen kürzeren Weg entschieden. Dabei integrieren wir sv_3 vorübergehend komplett und nehmen dann die erforderlichen Anpassungen vor. Hierbei ist zu beachten, daß die Klasse `Point` aus sv_3 in sv_5 nicht explizit gelöscht werden darf. Dies hätte nämlich das automatische Löschen der Attribute `begin` und `end` der Klasse `Arc` zur Folge, da ansonsten ein zwischenzeitlich unvollständiges Schema entstünde. Stattdessen wird die Klasse `Point` in sv_5 durch das abschließende **integrate** unter Verwendung der Option **replace** durch die gewünschte Version aus sv_4 überschrieben.

```
create schemaversion sv5 {
    integrate class Object from sv3 subclasses;
    delete class KartPoint;
    delete class PolarPoint;
    integrate class Point from sv4 replace;
    } /* sv5 */
```

Ableiten der Schemaversion sv_5.

Die weitere Schemaversion sv_6 soll insbesondere dem Zwecke der Archivierung abgeschlossener Softwareentwicklungsprojekte dienen, beispielsweise, um die erreichten Ergebnisse in zukünftigen Projekten wiederverwenden zu können. Auch sv_6 dient damit einer speziellen Gruppe von

Applikationen im Softwareentwicklungsumfeld und verfolgt dementsprechend eigene Ziele. Wie zuvor kann man auch hier nicht von einer generellen Verbesserung vorheriger Schemaversionen sprechen. Stattdessen reflektiert auch sv_6 die speziellen Bedürfnisse einer einzelnen Gruppe von Anwendungen besonders gut, ohne jedoch den Anspruch zu erheben, damit alle vorherigen, älteren Schemaversionen qualitativ und quantitativ zu übertreffen und damit ersetzen zu können.

Im Gegensatz zu den bisherigen Schemaversionen liegt das Augenmerk bei sv_6 auf einer gewissen Form der Minimalität. Nach erfolgreichem Abschluß eines Projektes werden Informationen wie Zeitpläne oder die Zuordnung von Teilaufgaben zu einzelnen Angestellten für die spätere Wiederverwendung der Software nicht mehr benötigt und die sie modellierenden Attribute und Klassen können demzufolge entfernt werden. Anders als bei sv_5 geschieht das Löschen von Klassen hier einfacher implizit, indem diese nämlich erst gar nicht integriert werden.

```
create schemaversion sv6 {
    integrate class Document from sv4;
    modify class Document {
        attributes {
            delete status;
            }
        } /* modify class Document */
    integrate class SourceCode from sv3;
    integrate class Embedded from sv3;
    integrate class Text from sv3;
    integrate class Pic from sv3;
    } /* sv6 */
```

Ableiten der Schemaversion sv_6.

Die Entwicklung der Schemaversion sv_6 möge bereits vor Fertigstellung von sv_5 begonnen worden sein, d.h. die beiden Schemaversionen sind Alternativen im eigentlichen Sinne der Versionierung. Damit resultiert die Ableitungsbeziehung zwischen den Schemaversionen unseres Beispielszenarios in dem in Abbildung 5.10 dargestellten Schemaableitungsgraphen. Dort sind aus Platzgründen nur die Namen der jeweils vorhandenen Klassen aufgeführt. In einer Schemaversion neu angelegte Klassen sind fett gedruckt, z.B. `Document` in sv_1.

5.5.6 Die Erzeugende ODL

Das technische Teilziel 3.19 fordert die Transparenz des Schemaversionierungsmechanismus für die Gruppe der Applikationsentwickler, um diese vor jeglicher zusätzlicher Komplexität zu schützen und ihnen denselben Eindruck zu vermitteln, wie ein unversioniertes Datenbanksystem. Daher soll nun vorgestellt werden, wie eine isolierte Schemaversion, die quasi einem unversionierten Schema entspricht, definiert werden kann, ohne dabei Bezug auf Komponenten zu nehmen, die in anderen Schemaversionen definiert sind. Zu diesem Zwecke ist im wesentlichen nur eine Beschränkung auf die erzeugenden Primitive der ODL notwendig. In Anlehnung an die erzeugenden Operationen abstrakter Datentypen bezeichnen wir den Ausschnitt der ODL, mit dem isolierte Schemaversionen beschrieben werden können, auch als *erzeugende ODL*. Sie umfaßt genau diejenigen ODL-Primitive, die mit dem Schlüsselwort **create** eingeleitet werden. Um die Analogie mit einem unversionierten System noch zu verstärken, kann auch noch auf das Primitiv **create schemaversion** verzichtet und der Schemaversionsdefinitionsblock direkt in das Primitiv **create schema** geschrieben werden.

Bei der Vorstellung der Primitive auf Schemaebene in Abschnitt 5.5.1 hatten wir denselben Schemadefinitionsblock sowohl für die Erzeugung neuer als auch für die Änderung existierender

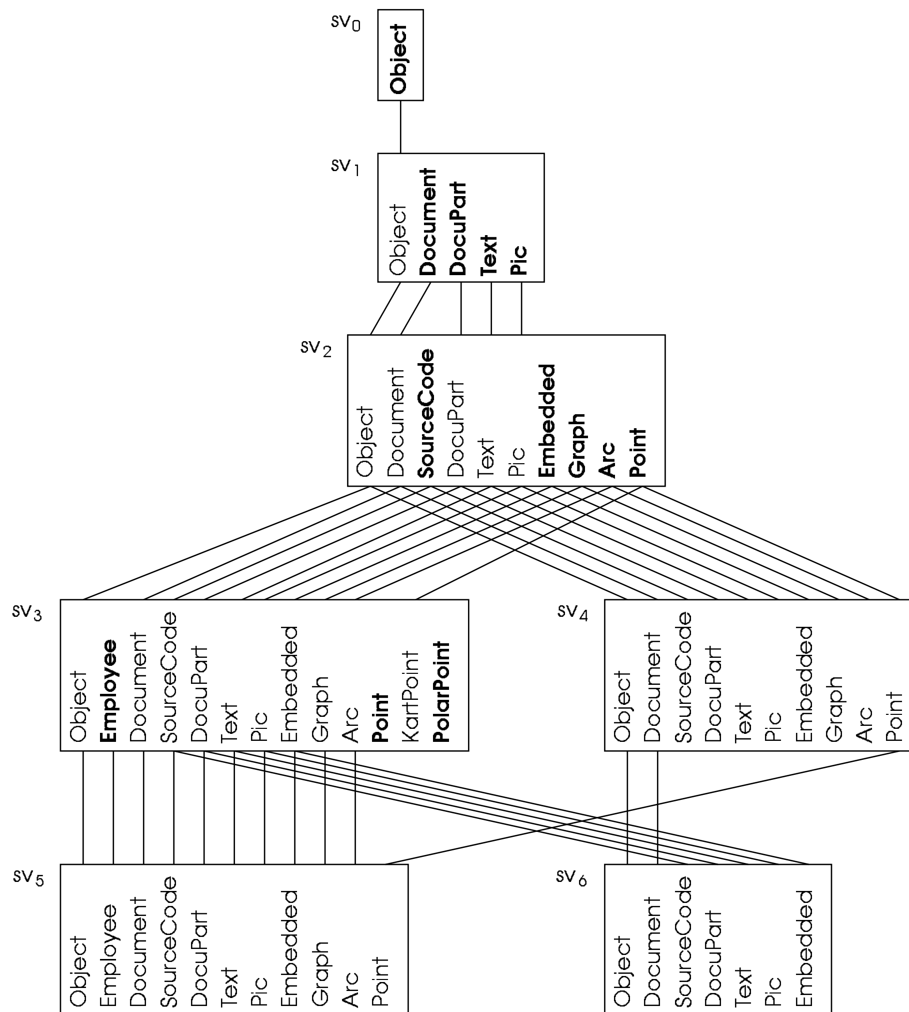


Abbildung 5.10: Darstellung des Beispiels als Schemaableitungsgraph.

Schemata angegeben. Damit hatten wir der Kürze der Beschreibung wegen darauf verzichtet, genau zwischen Schemadefinitionsblock (beim Erzeugen) und Schemamodifikationsblock (beim Verändern) zu unterscheiden. Genau genommen enthielte ein reiner Schemadefinitionsblock allerdings keine Änderungsprimitive auf der Schemaversionsebene oder darunter und die erzeugende ODL kommt mit einem solchen, reinen Schemadefinitionsblock aus, d.h. sie enthält insbesondere keine Primitive zur Veränderung von Klassen, Attributen oder Methoden.

5.5.7 Vergleich der allgemeinen Integrationsproblematik mit der Klassenintegration in COAST

Die allgemeine Aufgabe der Schemaintegration, so wie wir sie in Abschnitt 4.5.4 vorstellten, betrachtet zwei (oder mehr) unversionierte Schemata, die zu einem einzigen Schema zu verschmelzen sind. Dabei muß die Entstehung und Entwicklung der betrachteten Schemata als voneinander unabhängig angesehen werden, was zu den beschriebenen Problemen führt und eine der eigentlichen Integration vorangehende Konsolidierung insbesondere bezüglich der genutzten Datenstrukturen und Namen notwendig macht. Die Kenntnis der in den zu integrierenden Schemata enthaltenen Semantik ist für eine sinnvolle Integration eine notwendige Voraussetzung. Ein Datenbanksystem kann die Semantik eines Datenbankschemas jedoch immer nur zu einem kleinen Teil automatisch erfassen. Die notwendige Verbesserung der Situation kann nun auf zwei

Wegen erreicht werden. Eine manuelle Alternative besteht in der Interaktion des Systems mit dem Datenbankentwickler, welcher die Semantik der zu integrierenden Schemata kennen muß und dem System Fragen bezüglich der Äquivalenz der in den einzelnen Schemata enthaltenen Strukturen beantworten kann. Die auf eine automatische Konsolidierung abzielende Alternative besteht darin, gewisse Annahmen über die Vorgehensweise bei der Entwicklung der Schemata zu machen. Dabei wird man beispielsweise davon ausgehen, daß identische Namen von Schemakomponenten genau dann gewählt wurden, wenn diese Komponenten auch dieselbe Semantik beinhalten, d.h. es treten weder Synonyme noch Homonyme auf. Auf diesem Wege ist jedoch schon eine einfache Umbenennung einer Schemakomponente nicht mehr ohne weiteres automatisch feststellbar.

Das in Abschnitt 5.5.3.1 vorgestellte **integrate**-Primitiv erlaubt auch in COAST die Integration einzelner Klassen oder ganzer Klassenvererbungsgraphen. Die zuvor für den allgemeinen Fall dargelegte Problematik erfährt hier jedoch dadurch eine erhebliche Vereinfachung, daß sämtliche zu integrierende Klassen einem einzigen — allerdings versionierten — Schema entstammen. Damit können bei der Konsolidierung auftretende Fragen nach der Äquivalenz verschiedener Komponenten automatisch beantwortet werden.

Das allgemeine Integrationsproblem ist einer Situation vergleichbar, in der nur die Primitive der erzeugenden COAST-ODL Verwendung finden. Durch das zusätzliche Angebot der Änderungsprimitive der COAST-ODL erlauben wir dem Schemaentwickler, bereits bei der Ableitung einer neuen Schemaversion deren Verhältnis zu existierenden Schemaversionen zu spezifizieren. Das dabei gewonnene Wissen wird im Schemaableitungsgraphen und in den Klassenableitungsbäumen gespeichert und kann dann bei der Integration von Versionen eines Schemas verwendet werden. Durch die Kenntnis der Schemaänderungsoperationen, die bei der Ableitung einer Schemaversion Verwendung fanden, lassen sich nämlich die benötigten Rückschlüsse auf das Verhältnis zwischen verschiedenen Schemakomponenten ziehen. Bei der Spezifikation einer Komponente einer Schemaversion offeriert die COAST-ODL grundsätzlich zwei Möglichkeiten. Zum einen kann eine Komponente stets ausschließlich mit Hilfe der erzeugenden ODL spezifiziert werden, wobei keinerlei Rückgriffe auf bestehende Komponenten anderer Schemaversionen gemacht werden. Zum anderen erlaubt das **integrate**-Primitiv die Übernahme von Komponenten anderer Versionen desselben Schemas, welche dann durch die übrigen Schemaänderungsprimitive nötigenfalls an veränderte Anforderungen angepaßt werden können. Die hier getroffene Annahme über die Vorgehensweise bei der Erzeugung der verschiedenen Versionen eines Schemas besagt nun, daß die Integration und Anpassung einer Schemakomponente genau dann durchgeführt wird, wenn eine semantische Ähnlichkeit zwischen der bestehenden und der neuen Komponente existiert, d.h. genau dann wenn beide dasselbe Konzept der Diskurswelt modellieren. Diese Annahme ist weitaus weniger restriktiv, als die im allgemeinen Fall für eine automatische Integration notwendige Annahme, daß semantisch ähnliche Schemakomponenten denselben Namen tragen. Diese stärkere Einschränkung ist in COAST nicht notwendig, da die Identität von Schemakomponenten automatisch erkannt werden kann, selbst wenn das **rename**-Primitiv angewendet wurde. Desweiteren wird die Einhaltung der genannten Einschränkung zumindest auf Klassenebene schon dadurch gegeben sein, daß eine Propagation von Objekten nur zwischen verschiedenen Versionen derselben Klasse möglich ist. Würde bei der Ableitung einer neuen Schemaversion eine neue Klasse mit Hilfe des **create class**-Primitives angelegt, obwohl eine semantische Ähnlichkeit mit einer existierenden Klasse besteht, so könnte keine Objektpropagation stattfinden. Dadurch wird die Einhaltung der hier gemachten Annahme durch den Schemaentwickler quasi erzwungen. Vergleichbares gilt auch auf Attributebene. Bei der Propagation auf der Objektebene (siehe Kapitel 6) macht es nämlich einen entscheidenden Unterschied, ob ein Attribut a' einer Klassenversion c' aus einem Attribut a der Vorgängerklassenversion c von c' durch Umbenennung hervorging oder ob a bei der Ableitung von c' gelöscht und a' neu hinzugefügt wurde (siehe Abbildung 5.11). Nur im ersteren Fall wird man nämlich defaultmäßig den Attributwert

von a propagieren. Anders als Monk und Sommerville (siehe Abschnitt 4.5.3.3) gehen wir davon aus, daß bei einer reinen Umbenennung eines Attributes dessen Semantik erhalten bleibt. Um dem System die Erzeugung sinnvoller Defaultkonvertierungsfunktionen zu ermöglichen, muß der Schemaentwickler zwischen Umbenennung (mit Erhalt der Semantik) und Löschung des alten und Erzeugung eines neuen Attributes (mit neuer Semantik) unterscheiden.

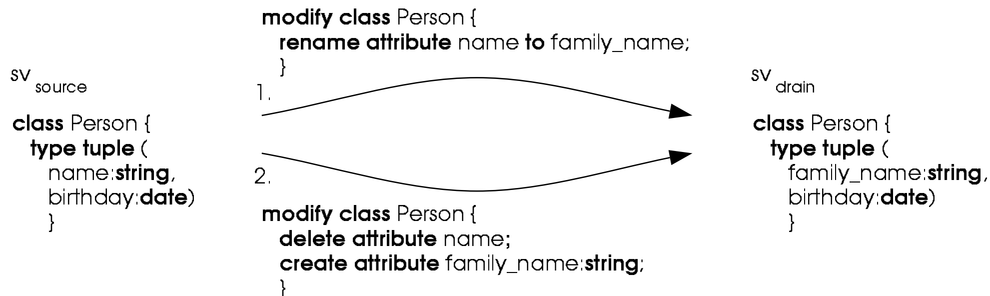


Abbildung 5.11: Zwei Wege, auf denen sv_{drain} von sv_{source} abgeleitet worden sein könnte.

Die in dieser Arbeit vorgestellte Integration von Versionen eines Schemas ist damit relativ leicht zu bewerkstelligen. Die gemachte Annahme über die korrekte Verwendung alternativ möglicher Ableitungswege in der COAST-ODL kann jedoch nicht auf komplett voneinander unabhängige Schemata übertragen werden. Die hier vorgestellte Integration stellt also keineswegs eine Lösung des allgemeinen Problems dar.

5.6 Zusammenfassung und Bewertung

In diesem Kapitel haben wir den Ansatz der Schemaversionierung auf der Ebene des Schemas selbst betrachtet. Als Grundlage entwickelten wir eine formale Definition für das COAST-Objektmodell und eine Menge von Invarianten, die die Konsistenz des Schemas beschreiben und bei Schemaänderungen beachtet werden müssen. Darauf aufbauend stellten wir zunächst die Begriffe der Schemaversion und der Klassenversion vor und erläuterten, in welchem Verhältnis diese untereinander und zueinander stehen. Von erheblicher Bedeutung ist dabei die Erweiterung des Konsistenzbegriffes auf versionierte Schemata und die damit verbundene Formulierung entsprechender Schemainvarianten.

Aus der Sicht des Benutzers eines Datenbanksystems stellt die Schemabeschreibungs- und -änderungssprache einen wesentlichen Aspekt dar. Die Primitive der von uns entworfenen und verwendeten COAST-ODL lassen sich den Ebenen der Schemata, der Schemaversionen, der Klassen und der Attribute und Methoden zuordnen. Neben der darin deutlich werdenden homogenen Struktur wurde insbesondere auf die problemlose Verwendbarkeit der Sprache geachtet. Wir haben die Primitive im einzelnen in ihrer Wirkungsweise vorgestellt und untersucht, welche Schemainvarianten ggf. verletzt werden könnten. Anstelle der trivialen Behandlung solcher Situationen durch Ablehnung der geforderten Schemaänderung haben wir korrigierende Maßnahmen beschrieben, die, soweit möglich, die vom Schemaentwickler in der gegebenen Situation gewünschte Lösung antizipieren und realisieren. Neben der Eingabe von Schemabeschreibungen und -änderungen wird die COAST-ODL auch zur Ausgabe eines im System verwalteten Schemazustandes an den Schemaentwickler eingesetzt. Dazu genügt bereits eine Teilmenge der beschriebenen Primitive, die wir als erzeugende ODL eingeführt haben.

Die Begriffe der Schemaversion und der Klassenversion stehen in engem Zusammenhang. Änderungsoperationen werden zunächst auf Klassen durchgeführt, wenn diese erzeugt, modifiziert oder gelöscht werden. Damit geht jedoch stets auch eine Änderung des die Klassen zusammen-

fassenden Schemas einher. Aus verschiedenen Gründen sind beide Aspekte beim Entwurf einer Schemaevolutionsumgebung zu berücksichtigen.

Das (unversionierte) Schema ist zum einen die Einheit der Beschreibung einer Datenbank aus der Sicht der Applikationen und muß diesen daher an einem Stück zur Verfügung stehen. Zum anderen betreffen Schemaänderungen oft mehrere Klassen. Dies liegt erstens daran, daß in komplexen Szenarien wie etwa bei der Entwurfsunterstützung durch die CAx-Applikationen das Schema in der Praxis oft restrukturiert wird, d.h. Komponenten werden zwischen Klassen verschoben. Zweitens bestreffen selbst lokale Änderungen einer einzelnen Klasse auch andere Klassen, da diese von der ersten über Vererbungs- und Aggregationsbeziehungen abhängen. Damit ist das Schema ebenfalls die Einheit der Änderung von Struktur und Verhalten einer Datenbank. Aus den genannten Gründen ist die Versionierung auf der Ebene des Schemas, so wie sie hier eingeführt wurde, angebracht. Somit ist hier die Schemaversion, als Analogon des Schemas unversionierter Systeme, die Einheit von Beschreibung und Änderung.

Die Klasse ist jedoch gleichzeitig der eigentliche Träger der Strukturinformation und unterliegt während des Evolutionsprozesses Veränderungen. Die Objekte, als eigentliche Träger der Nutzinformation, gehören diesen Klassen an und sollten auch bei deren Veränderung erhalten bleiben. Damit ist auch eine Beziehung zwischen den Klassen verschiedener Schemaversionen zu beschreiben. Dies tun wir, indem wir von Klassenversionen sprechen als Ausprägungen einer Klasse in verschiedenen Modellierungen derselben Diskurswelt, d.h. in verschiedenen Versionen desselben Schemas.

Durch die gleichzeitige Beschreibung von Schemaversionen als auch von Klassenversionen und durch die Etablierung der Zusammenhänge durch Schemaableitungsgraphen und Klassenableitungsbäume haben wir eine angemessene und in dieser Form bisher einmalige Beschreibung des Evolutionsprozesses erreicht. Der vollständige Nutzen der Ableitungsbeziehungen wird sich allerdings erst im folgenden Kapitel voll entfalten.

Aufbauend auf den Vorarbeiten zu unversionierten Schemata war es uns leicht möglich, den zuvor definierten Konsistenzbegriff auf versionierte Schemata auszudehnen, wobei die Analogie zwischen Schemata unversionierter Systeme und den hier besprochenen Schemaversionen so groß ist, daß unsere Mechanismen einem nur eine Schemaversion benutzenden Applikationsentwickler vollkommen transparent bleiben können. Dies wurde unterstützt durch die Definition der erzeugenden ODL als Ausschnitt unserer Schemabeschreibungs- und -änderungssprache, die u.a. der isolierten Beschreibung einer Schemaversion für einen Applikationsentwickler dient.

Beim Entwurf einer Schemabeschreibungs- und -änderungssprache für ein versioniertes Datenbanksystem ist dafür Sorge zu tragen, daß die genannten Ableitungsbeziehungen zwischen Schemaversionen und zwischen Klassenversionen aus den Primitiven extrahiert werden können. Aufgrund dieser Anforderungen wären für die Übernahme einer vorhandenen ODL erhebliche Anpassungen erforderlich. Daher haben wir uns stattdessen entschieden, eine eigene, neue Sprache zu entwickeln, die optimal auf unsere Anforderungen zugeschnitten ist und die aufgrund ihrer einheitlichen Struktur leicht erlernt werden kann.

In diesem Kapitel haben wir die in der Praxis der Schemaevolution vorhandenen Objekte der Datenbank noch nicht betrachtet. Unter Berücksichtigung der Tatsache, daß die hier angestellten Überlegungen ohnehin den notwendigen ersten Schritt auf dem Weg zu einem vollständigen System zur Unterstützung der Schemaevolution darstellen, ist das hier gewählte schrittweise Vorgehen nicht nur akzeptabel sondern sogar empfehlenswert. Im nächsten Kapitel werden wir, quasi als zweiten Schritt unserer Betrachtungen, auf die Objektebene eingehen.

Kapitel 6

Schemaversionierung auf Objektebene

Der Zweck eines Datenbanksystems ist die Verwaltung von Daten, wobei deren gemeinsame Nutzung durch verschiedene Applikationen eine zentrale Rolle spielt. Während in konventionellen Systemen alle kooperierenden Applikationen auf demselben Datenbankschema aufsetzen müssen, hatten wir als technisches Teilziel 3.10 gefordert, daß auch für Applikationen, die auf verschiedenen Versionen eines Schemas aufsetzen, die Möglichkeit zur Kooperation gegeben sein soll. Nachdem wir im vorangegangenen Kapitel die COAST-ODL eingeführt haben, mit der Schemata und Schemaversionen angelegt, verändert und gelöscht werden können, wenden wir uns daher nun der Frage zu, wie die Verwaltung der Instanzen eines versionierten Schemas zu bewerkstelligen ist.

Kooperation entsteht dadurch, daß ein gemeinsam zu benutzendes Objekt zwar von nur einer Applikation angelegt, seine Existenz im Laufe der Zeit jedoch auch anderen Applikationen bekannt wird. Eventuell durch Zugriffsrechte eingeschränkt können dann beliebig viele Applikationen auf gemeinsame Instanzen zugreifen. Dabei werden i.Allg. Synchronisationsmechanismen zur Konsistenzerhaltung benötigt. In einem System mit Schemaversionierung entsteht eine zusätzliche Schwierigkeit dadurch, daß verschiedene Applikationen über verschiedene Schnittstellen, d.h. Versionen des Schemas, auf die gemeinsame Datenbank zugreifen.

Im vorangegangenen Kapitel 5 stellten wir Operationen der COAST-ODL vor, mit denen insbesondere eine neue Version eines Schemas abgeleitet werden kann. Der dort auf Schemaebene beschriebene Prozeß der Ableitung einer neuen Schemaversion (siehe Abschnitt 5.4.1) war jedoch insofern unvollständig geblieben, als noch keinerlei Auswirkungen der Schemaänderungsoperationen auf Objektebene spezifiziert wurden. In diesem Kapitel beschäftigen wir uns daher nun mit der Frage, welche Auswirkungen die Schemaänderungsoperationen auf Objektebene haben und welche Spezifikationen auf Objektebene zur Vervollständigung des Schemaableitungsprozesses noch notwendig sind.

Vorbereitend hatten wir jedoch bereits zwei Aspekte eingeführt, die für die Beschreibung eines neuen, unversionierten Schemas nicht notwendig wären. Dies waren zum einen Beziehungen zwischen Klassen (bzw. Klassenversionen) verschiedener Versionen desselben Schemas gewesen und zum anderen Beziehungen zwischen Klasseneigenschaften, genauer gesagt zwischen Attributen verschiedener Versionen derselben Klasse.

Bei Klassenversionen in verschiedenen Schemaversionen hatten wir streng unterschieden, ob es sich um Versionen derselben Klasse handelte (dies ist durch Integration erreichbar), oder nicht. Letzteres wäre der Fall, wenn eine Klasse neu angelegt wird. Auch bei den Attributen hatten wir bereits darauf hingewiesen, daß es einen Unterschied macht, ob ein existierendes Attribut

gelöscht und ein neues angelegt wird, oder ob ein Attribut umbenannt wird, selbst wenn die in beiden Fällen entstehenden Zielschemaversionen übereinstimmen (siehe Abschnitt 5.5.7). Denn beim Anlegen wird ein neues Attribut erzeugt, das mit dem vorherigen, inzwischen gelöschten Attribut semantisch nichts gemeinsam hat (zumindest nicht für das Datenbanksystem erkennbar), während es sich vor und nach der Durchführung einer Umbenennung um ein und dasselbe Attribut handelt.

Die Etablierung der genannten Beziehungen zwischen Klassenversionen bzw. Attributen verschiedener Schemaversionen können durch die Verwendung unterschiedlicher Schemaänderungsprimitive erreicht oder unterdrückt werden; sie sind jedoch wie erwähnt für eine isolierte Betrachtung der neuen Schemaversion nicht bedeutungsvoll. Sie gewinnen ihre Bedeutung jedoch, sobald es um den Zusammenhang zwischen verschiedenen Schemaversionen geht, da sie dann gewinnbringend eingesetzt werden können, um Auswirkungen auf Objektebene zu ermitteln. Hier macht es für die Vorwärts- und Rückwärtspropagation von Informationen nämlich einen erheblichen Unterschied, ob es sich in zwei Schemaversionen um verschiedene Darstellungen derselben Information handelt oder ob zwei verschiedene und damit unabhängige Informationen modelliert werden. Demzufolge können wir in diesem Kapitel von den zuvor etablierten Beziehungen Gebrauch machen und werden dies insbesondere bei der Erstellung von Defaultkonvertierungsfunktionen auch tun.

Hierdurch wird im Nachhinein ein weiteres Argument klar, das uns dazu veranlaßt hat, eine einzige ODL einzuführen und nicht zwischen einer Schemabeschreibungs- und einer Schemaänderungssprache zu unterscheiden. Mit einer reinen Schemabeschreibungssprache können die oben beschriebenen, semantischen Beziehungen zwischen verschiedenen Schemaversionen nämlich nicht ausgedrückt werden.

Wir führen Konvertierungsfunktionen ein, mit denen ein Objekt in verschiedenen Typen zur Verfügung gestellt werden kann. Die Ausführung dieser Konvertierungsfunktionen kann mittels Propagationsflags gesteuert werden.

6.1 Objektzugriffsbereiche

Entsprechend dem technischen Teilziel 3.12 soll die Sichtbarkeit eines Objektes durch die verschiedenen Schnittstellen, d.h. durch die verschiedenen Versionen des Datenbankschemas gesteuert werden können. Da demzufolge nicht jedes Objekt in jeder Schemaversion sichtbar sein wird, definieren wir zunächst den *Zugriffsbereich* einer Schemaversion.

Definition 6.1 {Zugriffsbereich, IAS}
Der Zugriffsbereich $IAS(sv)$ einer Schemaversion sv (engl. instance access scope) *ist eine Menge, die genau diejenigen Objekte der Datenbank enthält, welche für Applikationen von sv sichtbar sind. Eine Datenbank db besteht damit aus der Vereinigung der Zugriffsbereiche aller Versionen ihres Schemas s(db).*

$$db = \bigcup_{sv \in sv(s(db))} IAS(sv)$$

Der Zugriffsbereich $IAS(sv.c)$ einer Klassenversion *$sv.c$ enthält entsprechend die der Klasse c angehörenden Objekte von $IAS(sv)$.*

Der Zugriffsbereich einer Schemaversion ist in zwei Partitionen aufgeteilt. Die erste Partition wird *direkter Zugriffsbereich* (engl. *direct instance access scope, $DIAS(sv)$*) genannt und enthält alle Objekte, die von Applikationen von sv angelegt wurden. Die zweite Partition wird analog *indirekter Zugriffsbereich* (engl. *indirect instance access scope, $IIAS(sv)$*) genannt und enthält

alle Objekte, die in anderen Schemaversionen als *sv* angelegt und von dort (direkt oder indirekt) nach *sv* propagiert wurden. Offensichtlich gilt $IAS(sv) = DIAS(sv) \cup IIAS(sv)$.

Kooperation zwischen Applikationen verschiedener Schemaversionen kann also genau dann stattfinden, wenn sich deren Zugriffsbereiche überlappen.

6.2 Versionierte Objekte und Konvertierungsfunktionen

Ein Objekt *o*, das in einer Version *sv* eines Schemas *s* von einer Klasse *c* instanziiert wird, entspricht zunächst nur dem Typ von *sv.c*. Jedoch können verschiedene Versionen der Klasse *c* verschiedene Typen spezifizieren und wir machen keinerlei Einschränkungen bezüglich der Typen der verschiedenen Versionen einer Klasse. Insbesondere fordern wir nicht, daß der Typ einer Klasse bei der Ableitung einer neuen Schemaversion nur spezialisiert, d.h. zu einem Untertyp verändert werden darf. Im Allgemeinen kann eine einzelne Repräsentation des Objektes *o* deshalb nicht den Typen aller Versionen seiner Klasse entsprechen. Um die daher notwendige Verwaltung mehrerer Repräsentationen eines Objektes zu ermöglichen, führen wir versionierte Objekte ein.

Definition 6.2 {versioniertes Objekt, o}
Ein (versioniertes) Datenbankobjekt *(kurz* Objekt*) ist eine durch einen eindeutigen Objektidentifikator oid identifizierbare und zu einer bestimmten Klasse c gehörige Einheit, die eine nichtleere Menge von Datenwerten genannt* Objektversionen *enthält. Für jede Schemaversion sv, die die Klasse c enthält, besitzt o höchstens eine Version. Diese Version wird mit sv.o identifiziert und ihr Datenwert gehört dem Typ der Klassenversion sv.c an.*

Ein Objekt *ist ein 6-Tupel* $o = (oid, cid, svid, oct, del_list, ov_list)$ *mit den folgenden Komponenten:*

- $oid \in oId$ *ist der Identifikator des Objektes.*
- $cid \in cId$ *ist der Identifikator der Klasse, der o angehört.*
- *csv ist der Identifikator der Erzeugerschemaversion des Objektes* (engl. creator schema version)*. Darunter verstehen wir diejenige Schemaversion, in der o angelegt wurde.*
- *oct ist die Erzeugungszeit des Objektes* (engl. object creation time).
- $del_list \subseteq svId$ *ist die Menge der Identifikatoren derjenigen Schemaversionen, in denen o gelöscht wurde.*
- *ov_list beschreibt die Menge der Objektversionen von o. Jede Objektversion ist ein 4-Tupel* $ov = (svid, ovct, origin, v)$ *mit den folgenden Komponenten:*
  - *svid ist der Identifikator derjenigen Schemaversion, zu der ov gehört.*
  - *ovct ist die Erzeugungszeit der Objektversion* (engl. object version creation time).
  - *origin ist die Ursprungsschemaversion der Objektversion, also diejenige Schemaversion, in der der logische Wert von ov geschrieben und von wo ov propagiert wurde.*
  - *v ist der Wert der Objektversion.*

Zur Vereinfachung unserer Darstellungen definieren wir $class(o) := cid$, $csv(o) := csv$, $oct(o) := oct$ *und* $ods(o) := del_list$ *für ein Objekt o sowie* $ovct(ov) := ovct$ *und* $origin(ov) := origin$ *für eine Objektversion ov.*

Damit gehört auch jedes versionierte Objekt o genau einer Klasse c an[95] und für jede Version $sv.c$ seiner Klasse enthält o maximal eine Objektversion $sv.o$. Somit lassen sich die Versionen eines Objektes analog den Versionen seiner Klasse in einem, dem Klassenableitungsbaum (siehe Definition 5.30) vergleichbaren Baum anordnen. Da ein Objekt o allerdings nicht zu jeder Version seiner Klasse eine Objektversion enthalten muß, hat der Baum der Objektversionen möglicherweise weniger Knoten als der der Klassenversionen und es besteht damit kein Isomorphismus zwischen den beiden Bäumen.

In Erweiterung aus der Literatur bekannter Modelle der Objektversionierung erlaubt Definition 6.2, daß verschiedene Versionen desselben Objektes verschiedenen Typen entsprechen. Ein Objekt der Klasse c, das in den Schemaversionen $sv_1, \ldots, sv_n$ sichtbar ist, wird dementsprechend aus n Versionen bestehen, die den Typen von $sv_1.c, \ldots, sv_n.c$ angehören.

Das technische Teilziel 3.10 sieht die Möglichkeit vor, daß Applikationen, die auf verschiedenen Versionen eines Schemas aufsetzen, auf gemeinsam genutzten Objekten kooperieren können. Da jede Applikation nur über eine Version des Schemas auf die Datenbank zugreifen kann, muß ein gemeinsam von Applikationen verschiedener Schemaversionen zu nutzendes Objekt zwischen verschiedenen Typen umgewandelt werden können. Diese Typumwandlung der Datenbankobjekte muß automatisch durchgeführt werden können und wird im weiteren Verlauf dieser Arbeit als *Konvertierung* bezeichnet. Die Art und Weise der Durchführung einer solchen Konvertierung wird durch sog. *Konvertierungsfunktionen* spezifiziert.

Definition 6.3 {Konvertierungsfunktion, cf}
Gegeben seien zwei Versionen sv_u, sv_v eines Datenbankschemas s, die beide je eine Version der Klasse c enthalten. Weiterhin sei sv_u eine direkte Vorgängerschemaversion von sv_v.

Eine Vorwärtskonvertierungsfunktion (Rückwärtskonvertierungsfunktion) *von sv_u nach sv_v (von sv_v nach sv_u) für die Klasse c ist ein Programmstück, das eine Objektversion von $sv_u.c$ auf eine Objektversion von $sv_v.c$ (von $sv_v.c$ in $sv_u.c$) abbildet und wird formal ausgedrückt als $fcf_{c,v\leftarrow u}$ ($bcf_{c,u\leftarrow v}$).*

Wir bezeichnen sv_u und sv_v auch als Quell- *und* Zielschemaversion *der entsprechenden Konvertierungsfunktionen.*

Aus Quell und Zielschemaversion ist jederzeit ersichtlich, ob es sich um eine Vorwärts- oder um eine Rückwärtskonvertierungsfunktion handelt. Kommt es darauf jedoch nicht an, so schreiben wir mitunter in beiden Fällen nur $cf_{c,y\leftarrow x}$.

Konvertierungsfunktionen sind klassenspezifisch, d.h. sie können nur auf direkte Instanzen einer Klasse angewendet werden. Eine Vererbung von Konvertierungsfunktionen an Unterklassen und damit eine Wiederverwendung der darin enthaltenen Spezifikationen wäre in eingeschränktem Maße möglich, wird jedoch in dieser Arbeit nicht näher untersucht. Grundsätzlich werden für alle Klassen, die bei Ableitung einer neuen Schemaversion integriert wurden, Konvertierungsfunktionen benötigt. Dies gilt insbesondere auch für solche Klassen, die durch die explizite Integration einer anderen Klasse implizit mit integriert wurden. Dabei ist weiterhin zu beachten, daß die Modifikation einer Klasse durch die Vererbung implizit auch Unterklassen verändern kann. Sofern diese implizit veränderten Unterklassen (explizit oder implizit) integriert wurden, so ist damit ggf. auch die Modifikation der Konvertierungsfunktionen dieser Unterklassen vonnöten.

Ein weiteres Merkmal der in Definition 6.3 eingeführten Konvertierungsfunktionen ist, daß sie lediglich zwischen direkt benachbarten Schemaversionen spezifiziert werden. Konvertierungen zwischen beliebigen Schemaversionen können durch Komposition entsprechender Konvertierungsfunktionen bewerkstelligt werden. Darauf werden wir in Abschnitt 6.5 näher eingehen. Darüber

[95] Da unsere Konzepte Vererbung zwischen Klassenversionen vorsehen, aber nicht zwischen Klassen, müssen wir unsere Aussage an dieser Stelle nicht auf die direkte Klassenzugehörigkeit einschränken.

hinaus werden wir in Abschnitt 6.6 ergänzend sog. *Extrakonvertierungsfunktionen* einführen. Diese werden, anders als die in Definition 6.3 beschriebenen Konvertierungsfunktionen, nicht zwischen Vorgängern und Nachfolgern sondern zwischen Geschwistern in einem Klassenableitungsbaum definiert und erweitern damit die Möglichkeiten zur Propagation von Information zwischen verschiedenen Schemaversionen.

Da Konvertierungsfunktionen die Objektversion der Quellschemaversion nicht modifizieren, geht bei der Konvertierung anders als beim direkten Ansatz zur Unterstützung von Schemaevolution keine Information verloren.

Beispiel 6.1 Abbildung 6.1 zeigt zwei verschiedene Versionen eines Objektes, das ein Dokument repräsentiert.

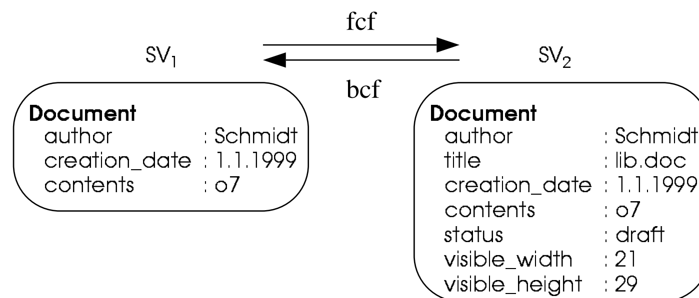


Abbildung 6.1: Zwei verschiedene Versionen eines Objektes der Klasse `Document`.

Die Konvertierungsfunktion $fcf_{Document,2\leftarrow 1}$ könnte beispielsweise wie folgt aussehen.

```
forward conversion {
    new.author         = old.author;
    new.title          = "no title yet";
    new.creation_date  = old.creation_date;
    new.contents       = old.contents;
    new.status         = draft;
    new.visible_width  = 17;
    new.visible_height = 25;
    }
```

Eine einfache Konvertierungsfunktion.

□

Mit den Präfixen **old** und **new** kann sich eine Konvertierungsfunktion auf die Werte der Objektversionen von Quell- und Zielschemaversion (in Beispiel 6.1 sind dies sv_1 und sv_2 für die Vorwärtskonvertierungsfunktion) beziehen. Hierbei ist zu beachten, daß auf die Attribute der Quellschemaversion (Präfix **old**) nur lesend zugegriffen werden kann, auf die der Zielschemaversion (Präfix **new**) nur schreibend.

- Zuweisungen an Attribute der Quellschemaversion (wie etwa **old**.a = x) sind nicht zulässig, da dafür die Konvertierungsfunktion in der anderen Richtung zuständig ist.

- Das Lesen von Attributen der Zielschemaversion (wie etwa **new**.a = **new**.a + 1) ist nicht zulässig, da Probleme bei der verzögerten Ausführung solcher Konvertierungsfunktionen entstehen könnten (siehe Abschnitt 6.3.3.2.2).

Auf die Präfixe kann verzichtet werden, wenn sämtliche benutzten Attributnamen eindeutig sind, z.B. weil die Attribute bei der Ableitung der Zielschemaversion umbenannt wurden.

Sehr einfache Konvertierungsfunktionen können vom Datenbanksystem automatisch aus den zur Ableitung der Zielschemaversion ausgeführten Schemaänderungsoperationen abgeleitet werden. Diese sog. *Defaultkonvertierungsfunktionen* arbeiten wie folgt.

- Attribute, die durch Verwendung von **create attribute** zu einer Klasse hinzugefügt wurden, werden mit einem Nullwert initialisiert. Falls vorhanden, kann auch der Defaultwert eines Attributes verwendet werden.
- Unveränderte oder lediglich umbenannte Attribute übernehmen den Wert der Quellschemaversion in die Zielschemaversion.
- Einfache Typkonvertierungen durch das Primitiv **modify attribute** wie von **int** zu **real** können ebenfalls automatisch berücksichtigt werden.

Prinzipiell bestehen zwei Möglichkeiten für den Umgang mit derartigen Konvertierungen, die automatisch aus den durchgeführten Schemaänderungsoperationen erkannt werden können.

- Zum einen können die entsprechenden Operationen in einer generierten Defaultkonvertierungsfunktion explizit ausgedrückt werden. Will der Schemaentwickler von der durch diese Defaultkonvertierungsfunktion vom System vorgeschlagenen Konvertierung abweichen, so ändert er diese und erzeugt damit eine benutzerdefinierte Konvertierungsfunktion, die dann bei der Propagation anstelle des Defaults ausgeführt wird.
- Zum anderen besteht die Möglichkeit, erkannte Konvertierungen bei der Propagation zunächst auf jeden Fall durchzuführen, womit quasi eine Initialisierung des zu konvertierenden Objektes in der Zielschemaversion erreicht wird. Anschließend wird die eigentliche, vom Schemaentwickler spezifizierte Konvertierungsfunktion ausgeführt. Diese kann allerdings leer sein, wenn die automatisch durchgeführte Konvertierung bereits ausreicht. Diese Möglichkeit wird beispielsweise in O_2 verwendet (siehe Abschnitt 4.3.4.2).

Wir haben uns hier aus den folgenden Gründen für die erste Alternative entschieden. Erstens gibt nur diese Alternative die vom System erkannten Teile der Konvertierungsfunktion explizit an. Die zweite Alternative kann zwar genau dieselben Operationen als Initialisierung durchführen, diese werden aber nicht explizit aufgeführt. Stattdessen muß anhand der Beschreibung der Funktionsweise des Systems bei jeder Konvertierungsfunktion auf die implizit durchgeführte Initialisierung geschlossen werden. Da dies beim praktischen Umgang insbesondere mit umfangreichen Schemata recht schwierig ist, wird man vermutlich einige Operationen sicherheitshalber trotzdem manuell spezifizieren. Damit wird die Konvertierungsfunktion mit Initialisierung entsprechend der zweiten Alternative ähnlich oder genauso lang wie ohne (erste Alternative). Darüber hinaus stellt diese zusätzliche, manuelle Spezifikation bereits automatisch durchgeführter Operationen eine zusätzliche Fehlerquelle dar. Existiert allerdings eine Defaultkonvertierungsfunktion, so sind sämtliche bei der Propagation durchzuführenden Operationen klassenspezifisch notiert. Damit herrscht zum einen stets Klarheit über die Leistungen des Systems und zum anderen sind kleine Veränderungen, wie z.B. das Einfügen eines Faktors beim Reskalieren von numerischen Attributen mit geänderten Maßeinheiten, sehr einfach durchführbar. Über die genannten Vorteile der ersten Alternative bei der Spezifikation von Konvertierungsfunktionen hinaus ergibt sich weiterhin eine Beschleunigung bei der späteren Ausführung der Konvertierungsfunktionen. Vom Schemaentwickler nicht gewünschte Operationen, die bei der Initialisierung entsprechend der

zweiten Alternative auf jeden Fall durchgeführt werden, können nämlich aus der Defaultkonvertierungsfunktion gestrichen werden, womit auch deren Ausführung zur Laufzeit tatsächlich eingespart wird.

Ein automatisch erzeugter Default für $fcf_{Document,2 \leftarrow 1}$ in Beispiel 6.1 hätte den neuen Attributen allerdings nur ihre jeweiligen Defaultwerte zuweisen können (also `""` für `title`, `draft` für `status` und 0 für `visible_width` und `visible_height`). Allgemein wird eine automatisch generierte Konvertierungsfunktion in der Regel nicht genügen, um ein Maximum an Semantik übertragen zu können, wenn eine Klasse im Rahmen der Ableitung einer neuen Schemaversion größere Änderungen durchlaufen hat. Beispiel 6.2 zeigt allerdings, daß die Defaultkonvertierungsfunktion auch in solchen Fällen gut als Grundlage zur Anpassung durch den Schemaentwickler dienen kann. Neben dem Rahmen können zahlreiche Zuweisungen einzelner Attribute unverändert übernommen werden. Zur kompletten Spezifikation der Konvertierungsfunktionen in beide Richtungen genügt die manuelle Anpassung einzelner Zuweisungen an den Stellen, wo der Automatismus nicht ausreicht.

Beispiel 6.2 Für die Beschreibung von $fcf_{Document,4 \leftarrow 2}$ und $bcf_{Document,2 \leftarrow 4}$ verwenden wir hier bereits die in Abschnitt 6.4 vorzustellende Propagationssprache. Sowohl in Vorwärts- als auch in Rückwärtsrichtung konnten die ersten fünf von insgesamt 10 bzw. sieben automatisch erzeugten Attributzuweisungen beibehalten werden.

```
modify schemaversion sv4 {
    modify derivation for class Document
        forward conversion {
            new.author         = old.author;
            new.title          = old.title;
            new.creation_date  = old.creation_date;
            new.contents       = old.contents;
            new.status         = old.status;
            new.page_size      = "DinA4";
            new.border_left    = 0,5 * (21,0 - old.visible_width);
            new.border_right   = 0,5 * (21,0 - old.visible_width);
            new.border_top     = 0,5 * (29,7 - old.visible_height);
            new.border_bottom  = 0,5 * (29,7 - old.visible_height);
            }
        backward conversion[96] {
            new.author         = old.author;
            new.title          = old.title;
            new.creation_date  = old.creation_date;
            new.contents       = old.contents;
            new.status         = old.status;
            new.visible_width  = width (old.page_size)  - old.border_left
                                                        - old.border_right;
            new.visible_height = height (old.page_size) - old.border_top
                                                        - old.border_bottom;
            };
    };
```

Vorwärts- und Rückwärtskonvertierungsfunktionen.

□

[96] Wir gehen hier davon aus, daß die Funktionen `width` und `height` die Breite und die Höhe des übergebenen Seitenformates liefern. Sie ließen sich allerdings auch durch eine Fallunterscheidung mit einer **case**-Anweisung ersetzen.

Wenn eine Konvertierungsfunktion eine Objektreferenz aus der Quell- in die Zielschemaversion übernimmt, wie das z.B. beim Attribut `contents` der Klasse `Document` zwischen sv_1 und sv_2 der Fall ist (siehe $fcf_{Document,2 \leftarrow 1}$ oben), so wird bei der Ausführung zunächst geprüft, ob das referenzierte Objekt (im Beispiel der Klasse `DocuPart`) im Zugriffsbereich der Zielschemaversion (im Beispiel $IAS(sv_2)$) sichtbar ist und nur in diesem Falle wird die Referenz übertragen. Anderenfalls wird dem Attribut in der Zielschemaversion eine **nil**-Referenz zugewiesen. Selbst wenn eine Konvertierungsfunktion die identische Abbildung ist (wie etwa bei der Klasse `Document` zwischen sv_3 und sv_5) können bei der Propagation inhaltlich verschiedene Versionen desselben Objektes entstehen.

Im Gegensatz zu den anderen Konvertierungsfunktionen des Beispiels liest $bcf_{Document,2 \leftarrow 3}$ Werte der referenzierten Objekte der Klasse `Employee`. Dies wird durch die Zuweisung **new**.`author` = **old**.`author.name` spezifiziert. Solche Konvertierungsfunktionen werden *komplex* genannt. Nähere Details zur Propagationssprache und zu komplexen Konvertierungsfunktionen finden sich in [Haa00].

Die folgende Regel bestimmt, welche Konvertierungsfunktionen bei der Ableitung einer neuen Schemaversion benötigt werden.

Regel 6.1 {**Definieren von Konvertierungsfunktionen bei Schemaableitung**}
Bei der Ableitung einer neuen Schemaversion sv_v *von bereits existierenden Schemaversionen* $sv_{u_1}, \ldots, sv_{u_m} \in sv(s)$ $(m \in \mathbb{N})$ *werden genau eine Vorwärts- und eine Rückwärtskonvertierungsfunktion für jede abgeleitete Klasse c definiert, nämlich* $fcf_{c,v \leftarrow u_i}$ *und* $bcf_{c,u_i \leftarrow v}$ $(i \in \{1, \ldots, m\})$. *Dabei muß* $sv_{u_i}.c$ *natürlich existieren. Für neu erzeugte Klassen* $(csv(c) = sv_v)$ *werden keine Konvertierungsfunktionen spezifiziert.*

Regel 6.1 besagt insbesondere, daß auch bei der Integration von $m > 1$ Schemaversionen $sv_{u_1}, \ldots, sv_{u_m}$ in eine neue Schemaversion sv_v nur eine Vorwärts- und nur eine Rückwärtskonvertierungsfunktion pro abgeleiteter Klasse c definiert werden darf.

Wie sich durch Beispiel 6.5 bestätigen wird, ist die in Regel 6.1 gemachte Einschränkung, daß die Vorwärts- und die Rückwärtskonvertierungsfunktion einer Klasse c für dieselbe Vorgängerschemaversion sv_{u_i} $(1 \leq i \leq m)$ von sv_v definiert werden, notwendig.

Wir möchten hier festhalten, daß sich eine Klasse bei der Integration in eine neue Schemaversion nicht notwendigerweise ändern muß. Die Werte der Objekte dieser Klasse können sich in den Zugriffsbereichen verschiedener Schemaversionen jedoch trotzdem unterscheiden, wenn die verwendete Konvertierungsfunktion von der Identität verschieden ist. So können beispielsweise Reskalierungen erreicht werden.

Abschließend soll noch ein grundsätzlicher Unterschied betont werden zwischen unserem Ansatz und dem in Abschnitt 4.5.3.2 beschriebenen Konzept der Handler, die von Skarra und Zdonik vorgeschlagen wurden. Diesem Konzept folgend verwendet ENCORE für jedes Objekt nur eine einzelne Repräsentation und diese entspricht dem Zustand des Typs zum Zeitpunkt der Erzeugung des Objektes. Mit Ausnahme der Verwendung des einen Typs, für den eine physikalische Repräsentation vorliegt, müssen alle folgenden Zugriffe umgesetzt werden. Im Gegensatz dazu legen wir ein Objekt in verschiedenen Objektversionen an, die, sofern sie gerade den aktuellen logischen Zustand des Objektes repräsentieren, direkt und damit schnell zugegriffen werden können. Weiterhin ist jede Objektversion mit einem eigenen Speicherbereich ausgestattet, so daß auch für später hinzugefügte Attribute Werte zugewiesen und persistent abgelegt werden können. Damit folgen wir der Kritik, die auch Monk und Sommerville (siehe Abschnitt 4.5.3.3) am Handlerkonzept von ENCORE üben.

6.3 Die Propagationssteuerung

In diesem Abschnitt gehen wir auf die Steuerung der Propagation auf Objektebene ein. Dazu analysieren wir zunächst die Aufgabenstellung und gehen erste, allgemeine Schritte der Modellbildung. Auf diesen Grundlagen erarbeiten wir die Details der Propagation zum Zeitpunkt der Ableitung einer neuen Schemaversion und danach. Abschließend gehen wir auf Wechselwirkungen zwischen Propagationsflags verschiedener Klassen ein und stellen Einsatzmöglichkeiten der Mechanismen in unserem durchgängigen Beispiel dar.

6.3.1 Analyse und erste Modellbildung

Das technische Teilziel 3.12 fordert, daß die Sichtbarkeit eines Objektes durch die verschiedenen Versionen eines Schemas kontrollierbar sein muß. Die Versionen eines Schemas stehen untereinander durch die Ableitungsbeziehung in Verbindung. Es ist demzufolge angebracht, eine analoge Ableitungsbeziehung auch auf Objektebene zwischen den Zugriffsbereichen der Schemaversionen zu etablieren. Damit wird die Sichtbarkeit eines Objektes in einer Schemaversion *sv* von seiner Sichtbarkeit in den Vorgänger- und Nachfolgerversionen von *sv* abgeleitet und nicht für jede Schemaversion unabhängig von anderen spezifiziert.

Wird ein Objekt von einer Schemaversion in eine andere propagiert, so sind die beiden Objektversionen i.Allg. nicht identisch, da sie verschiedenen Typen entsprechen und ihre Attributwerte möglicherweise trotz gemeinsamen Typs verschiedene Semantik tragen (etwa bei Reskalierungen von Inch zu Zentimeter). Sie stellen jedoch zwei verschiedene Modellierungen desselben Zustands des modellierten Objektes dar. Um dies ausdrücken zu können führen wir die folgende Sprechweise ein.

Definition 6.4 {logische Objektwerte}
Wir sagen zwei Versionen eines Objektes repräsentieren denselben logischen Objektwert *(oder denselben* logischen Objektzustand*), wenn die eine in die andere propagiert wurde oder wenn beide durch (direkte oder indirekte) Propagation aus einer gemeinsamen dritten Objektversion hervorgingen.*

Das technische Teilziel 3.13 fordert in Erweiterung von Teilziel 3.12, daß nicht nur über die Sichtbarkeit eines Objektes generell entschieden werden kann, sondern daß ggf. Applikationen verschiedener Schemaversionen verschiedene logische Objektzustände sehen sollen. Daher genügt es nicht, lediglich über die Zugehörigkeit eines Objektes zum Zugriffsbereich einer Schemaversion zu entscheiden. Ist ein Objekt sichtbar, so muß weiterhin festgelegt werden, welcher seiner logischen Zustände sichtbar sein soll.

Diesen beiden Überlegungen folgend ergeben sich zwei Feststellungen. Zum einen ist die Sichtbarkeit eines Objektes im Zugriffsbereich einer Schemaversion *sv* bei der Ableitung von *sv* in Abhängigkeit von den Zugriffsbereichen der Vorgängerschemaversionen von *sv* zu spezifizieren. Da die Schemaversion *sv* zu ihrer Ableitungszeit noch keine Nachfolgerschemaversionen besitzt, können diesbezügliche Abhängigkeiten auch noch nicht festgelegt werden. Zum anderen muß im Falle der Sichtbarkeit bestimmt werden, welcher der logischen Objektzustände sichtbar sein soll. Das COAST-Objektmodell unterstützt bisher keine für die Applikationen nutzbare Objektversionierung wie sie in Abschnitt 2.2 vorgestellt wurden. Die in Definition 6.2 beschriebenen Objektversionen dienen ausschließlich dem Schemaversionierungsmechanismus und sind dem technischen Teilziel 3.19 folgend gerade nicht für Applikationen sichtbar. Demzufolge ist in jeder Schemaversion zu jeder Zeit höchstens ein Zustand jedes Objektes sichtbar. Neue Objektzustände können sich jedoch durch nachträgliche Modifikationen der Datenbank durch die

Applikationen ergeben. Wird eine solche Objektänderung in einer Vorgänger- oder Nachfolgerschemaversion von *sv* durchgeführt, dann entsteht auch die Frage, ob im Zugriffsbereich von *sv* nun der neue oder weiterhin der alte Zustand des Objektes sichtbar sein soll.

Die Weitergabe von Datenbankzuständen und -änderungen zwischen Zugriffsbereichen verschiedener Versionen eines Objektes bezeichnen wir als *Objektpropagation* (oder kurz *Propagation*). Der Begriff der Propagation schließt damit die ggf. notwendige Konvertierung von Objektwerten mit ein. Analog zu den Konvertierungsfunktionen unterscheiden wir zwischen der Propagation von älteren Schemaversionen zu neueren und der umgekehrten Richtung und sprechen dabei von *Vorwärts-* und *Rückwärtspropagation.*

Die beiden oben dargestellten Aspekte der Objektpropagation, nämlich die Entscheidung über die Sichtbarkeit generell und die Auswahl des ggf. sichtbaren Zustandes, können bei der Betrachtung der Situation aus einem anderen Blickwinkel wiedergefunden werden. Bei der Untersuchung der Frage, welche Spezifikationen bezüglich der Sichtbarkeit verschiedener Zustände eines Objektes im Zugriffsbereich einer neuen Schemaversion *sv* überhaupt sinnvollerweise gemacht werden können, stößt man nämlich auf eine analoge zweiteilige Antwort (siehe Abbildung 6.2). Zum einen kann über die Übernahme von zur Ableitungszeit von *sv* in den Zugriffsbereichen ihrer Vorgängerschemaversionen existierender Objekte entschieden werden (siehe Abbildung 6.2.a). Eine Rückwärtspropagation ist hier nicht sinnvoll, da der direkte Zugriffsbereich von *sv* zum Ableitungszeitpunkt noch keine Objekte enthält. Zum anderen können die gewünschten Auswirkung zukünftiger Veränderungen in den Zugriffsbereichen der Vorgängerschemaversionen von *sv* auf den Zugriffsbereich von *sv* festgelegt werden (siehe Abbildung 6.2.b1). Analog können in der Rückwärtsrichtung die Auswirkungen zukünftiger Veränderungen im Zugriffsbereich von *sv* auf die Zugriffsbereiche der Vorgängerschemaversionen von *sv* festgelegt werden (siehe Abbildung 6.2.b2).

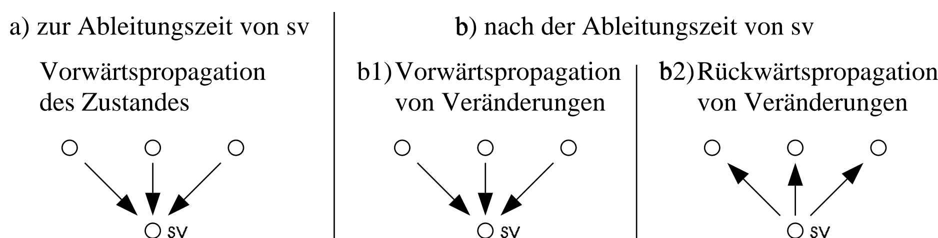


Abbildung 6.2: Verschiedene Typen der Propagation.

Die Propagation von zur Ableitungszeit existierenden Objekten und von später durchgeführten Modifikationen lassen sich beide mit Hilfe derselben Konvertierungsfunktion bewerkstelligen. Die Propagation modifizierter Objektzustände kann nämlich einfach durch wiederholte Ausführung der Konvertierungsfunktion erreicht werden. Darüberhinaus können auch die zu den beiden Zeitpunkten gewünschten Propagationen mit einem einzigen technischen Konzept beschrieben werden. Dazu unterscheiden wir vier verschiedene Propagationstypen, die jeweils durch ein *Propagationsflag* ein- und ausgeschaltet werden können.

Die Spezifikation der gewünschten Objektpropagation ist der letzte Schritt bei der Ableitung einer neuen Schemaversion und ihres Zugriffsbereiches. Der komplette *Schemaableitungsprozeß* (siehe Abschnitt 5.4.1) beinhaltet damit die folgenden Schritte.

1 auf Schemaebene

- Spezifikation der neuen Schemaversion durch die COAST-ODL
- ggf. Optimierung der Ableitungsbeziehungen

2 auf Objektebene

- Spezifikation der Konvertierungsfunktionen
- Spezifikation der Propagationsflags

3 Einfrieren der Schemaversion

Die Entscheidung, ob zur Ableitungszeit in den Zugriffsbereichen der Vorgängerschemaversionen existierende Objekte in den Zugriffsbereich von sv zu übernehmen sind oder nicht, wird durch das sog. *Schnappschußflag* (engl. *snapshot-flag*) (s-Flag) ausgedrückt. Die möglicherweise nach der Ableitung von sv in Zugriffsbereichen von Vorgänger- oder Nachfolgerschemaversionen durch die Applikationen dieser Schemaversionen ausgeführten Veränderungen haben wir in Objekterzeugung (engl. *creation*), Objektmodifikation (engl. *modification*) und Objektlöschung (engl. *deletion*) klassifiziert. Dementsprechend bestimmen *Erzeugungsflag* (c-Flag), *Modifikationsflag* (m-Flag) und *Löschungsflag* (d-Flag) die Auswirkungen zukünftiger Veränderungen in den Zugriffsbereichen der Vorgänger- und Nachfolgerschemaversionen von sv auf den Zugriffsbereich von sv. Die Objektpropagation umfaßt damit die Weitergabe von existierenden Objekten, von Objekterzeugungen, von Objektmodifikationen und von Objektlöschungen zwischen Zugriffsbereichen verschiedener Versionen eines Schemas.

Da die Propagationsflags den Konvertierungsfunktionen zugeordnet sind, ist es angebracht, sie gemeinsam mit diesen zu spezifizieren. Analog zu den Konvertierungsfunktionen werden daher bei der Ableitung einer neuen Schemaversion sv die Propagationsflags für die Vorwärtspropagation von Vorgängerschemaversionen von sv nach sv (siehe Abbildung 6.2.a und b1) und für die Rückwärtspropagation von sv zu ihren Vorgängerschemaversionen (siehe Abbildung 6.2.b2) bestimmt. Die Propagation von Nachfolgerschemaversionen von sv zu sv und umgekehrt wird entsprechend bei der Spezifikation dieser Nachfolgerschemaversionen festgelegt.

Wie bereits angedeutet, schließt die Propagation von Objektzuständen die Ausführung entsprechender Konvertierungsfunktionen ein. Da die Konvertierungsfunktionen klassenspezifisch festgelegt werden, liegt es nahe auch die Propagation für die Extension jeder Klasse separat zu spezifizieren. Weiterhin ist zwischen Vorwärts- und Rückwärtspropagation zu unterscheiden. Damit besteht sowohl bezüglich der Granularität als auch bezüglich der Orientierung eine hohe Ähnlichkeit zwischen der Spezifikation der Konvertierungsfunktionen und der Spezifikation der gewünschten Propagation. Dieser Zusammenhang wird sich auch im Entwurf unserer Propagationssprache (siehe Abschnitt 6.4) niederschlagen.

Mit Hilfe der Propagationsflags kann also gesteuert werden, wo sich die Zugriffsbereiche zweier Schemaversionen überlappen. Regel 6.2 faßt einige wichtige Kriterien, die für alle Propagationsflags gelten, zusammen.

Regel 6.2 $\left\{\begin{array}{l}\textbf{Propagationsflags von Vorwärts- und} \\ \textbf{Rückwärtskonvertierungsfunktionen}\end{array}\right\}$

Die Propagationsflags *einer Schemaversion sv*

- *kontrollieren die Propagation von Objekten aus den Zugriffsbereichen der direkten Vorgängerschemaversionen von sv in den Zugriffsbereich von sv (entlang der Vorwärtskonvertierungsfunktionen) und umgekehrt (entlang der Rückwärtskonvertierungsfunktionen),*
- *werden für jede abgeleitete Klasse separat spezifiziert und triggern die entsprechenden Konvertierungsfunktionen,*
- *dürfen analog zu Konvertierungsfunktionen (siehe Regel 6.1) auch bei der Integration mehrerer Schemaversionen nur für eine Vorgängerschemaversion spezifiziert werden,*

- *können bei mehrschrittiger Propagation entlang eines Konvertierungspfades mit Hilfe der Komposition ermittelt werden. Darauf werden wir in Abschnitt 6.5 näher eingehen.*

Nach Definition 5.30 kann jede Klasse einer Schemaversion sv nur von einer Vorgängerschemaversion von sv abgeleitet werden und nach Regel 6.1 darf dafür jeweils nur eine Konvertierungsfunktion in Vorwärts- und eine in Rückwärtsrichtung definiert werden. Da die Propagationsflags klassenspezifisch sind, müssen wir demzufolge jeweils nur eine Vorgängerschemaversion berücksichtigen.

Wir werden in den folgenden beiden Abschnitten genauer auf die verschiedenen Propagationsflags eingehen. Dabei betrachten wir jeweils eine Klasse c einer Schemaversion sv_v, die von $sv_u.c$ ($sv_v.c <^1_{sv,c} sv_u.c$) abgeleitet sei.

6.3.2 Propagation des Zustandes zum Ableitungszeitpunkt

Wir hatten bereits festgestellt, daß zwischen der Propagation des zum Ableitungszeitpunkt einer neuen Schemaversion sv aktuellen Datenbankzustandes in den Zugriffsbereich von sv und der Propagation von späteren Zustandsänderungen unterschieden werden muß. Wir gehen hier zunächst auf die Propagation des Zustandes (siehe Abbildung 6.2.a) ein.

6.3.2.1 Das Schnappschußflag (s-Flag)

Bei der Ableitung von sv_v kann der Schemaentwickler entscheiden, ob die zur Ableitungszeit von sv_v im Zugriffsbereich von $sv_u.c$ enthaltenen Objekte in den Zugriffsbereich von $sv_v.c$ übernommen werden sollen. Entsprechend schaltet er das Schnappschußflag, das der Vorwärtspropagation für die Klasse c zugeordnet ist, ein (s) oder aus ($\overline{s}$).

Ist das Schnappschußflag eingeschaltet, so wird zur Ableitungszeit von sv_v für jedes Objekt o, das im Zugriffsbereich von $sv_u.c$ enthalten ist, eine neue Objektversion für $sv_v.c$ angelegt. Ihr Wert bestimmt sich durch Anwendung der entsprechenden Vorwärtskonvertierungsfunktion: $sv_v.o = fcf_{c,v\leftarrow u}(sv_u.o)$.

Ein Objekt der Klasse `Document` beispielsweise, das bereits im Zugriffsbereich von sv_1.`Document` existierte, als sv_2 abgeleitet wurde, wird auch in sv_2 zugreifbar sein, wenn das Schnappschußflag (s-Flag) für die Klasse `Document` bei der Ableitung von sv_2 eingeschaltet wurde.

Im weiteren Verlauf dieses Kapitels werden wir uns zur Verbesserung der Übersichtlichkeit mitunter auf eine vereinfachte Form der Darstellung eines Objektes beschränken. Dabei betrachten wir zunächst nur ein einziges Objekt o der Klasse c und stellen seine Sichtbarkeit in den Zugriffsbereichen verschiedener Versionen eines Schemas dar. Der Propagationsmechanismus arbeitet selbstverständlich für alle Klassen (entsprechend der für sie gesetzten Propagationsflags) analog.

In den folgenden Abbildungen zur Veranschaulichung der Funktionsweise der verschiedenen Flags werden Schemaversionen als Kreise dargestellt. Dabei wird nur eine Klasse c berücksichtigt. Ist das betrachtete Objekt $o \in c$ in einer Schemaversion nicht sichtbar, so ist der entsprechende Kreis leer, ansonsten trägt er ein Symbol, das einen logischen Objektwert repräsentiert. Dabei wird in verschiedenen Schemaversionen einer Abbildung genau dann dasselbe Symbol verwendet, wenn dort auch derselbe logische Objektwert sichtbar ist.

Abbildung 6.3 zeigt, wie die Sichtbarkeit des zur Ableitungszeit von sv_v bereits in sv_u enthaltenen Objektes o vom Schnappschußflag für die Vorwärtspropagation von sv_u nach sv_v abhängt. Ist das Schnappschußflag wie im oberen Teil von Abbildung 6.3 dargestellt eingeschaltet (s), dann wird das Objekt o (ebenso wie alle Objekte aus dem Zugriffsbereich von sv_u, für deren Klasse das

Schnappschußflag für die Vorwärtspropagation gesetzt ist) zur Ableitungszeit $t_2 = dt(sv_v)$ von sv_v im Zugriffsbereich von sv_v sichtbar. Da der Wert der Objektversion $sv_v.o$ durch Anwendung der Konvertierungsfunktion $cf_{c,2\leftarrow 1}$ aus dem Wert $sv_u.o$ abgeleitet wurde, repräsentieren beide Objektversionen zum Zeitpunkt t_2 denselben logischen Objektwert und sind demzufolge mit demselben Symbol dargestellt.

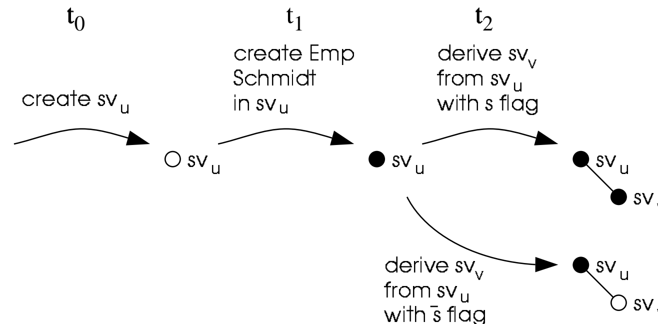


Abbildung 6.3: Die Semantik des Schnappschußflags.

Ist das Schnappschußflag jedoch wie im unteren Teil von Abbildung 6.3 dargestellt ausgeschaltet ($\overline{s}$), dann wird das Objekt o nicht im Zugriffsbereich von sv_v sichtbar. Entsprechend ist der sv_v repräsentierende Kreis leer dargestellt.

Da der Zugriffsbereich einer Schemaversion zu ihrer Ableitungszeit noch keine Objekte enthält, macht es keinen Sinn, ein Schnappschußflag für die Rückwärtspropagation vorzusehen.

Definition 6.5 {Propagationsflags einer Klasse, $cpo(sv, sv', c)$}
Die Propagationsflags (engl. class propagation options) $cpo(sv, sv', c)$ *einer Klasse c von einer Schemaversion sv in eine Schemaversion sv' ist die Menge der für diese Propagation eingeschalteten Flags. Dabei müssen die Klassenversionen $sv.c$ und $sv'.c$ existieren und $sv' \in dpred(sv) \cup dsucc(sv)$ muß gelten. Die Flags werden als s, c, m und d oder auch als s-flag, c-flag, m-flag und d-flag notiert.*

Invariante 6.1 {Schnappschußflag}
Wenn eine neue Schemaversion sv von $sv_1, \ldots, sv_n$ abgeleitet wird, dann gilt

$$IAS(sv) = \bigcup_{i=1}^{n} \bigcup_{\substack{c \in sc(sv) \cap sc(sv_i) \\ s\text{-flag} \in cpo(sv_i, sv, c)}} IAS(sv_i.c).$$

6.3.3 Propagation des Zustandes nach dem Ableitungszeitpunkt

Nach der Ableitung von sv_v können Applikationen von sv_u weiterhin Änderungen in dessen Zugriffsbereich durchführen. Der Effekt solcher nachträglicher Änderungen auf den Zugriffsbereich von sv_v kann mit Erzeugungsflag (c-Flag), Modifikationsflag (m-Flag) und Löschungsflag (d-Flag) eingestellt werden. Die Propagationen dieser Änderungen kann sowohl in Vorwärts- wie auch in Rückwärtsrichtung gleichermaßen durchgeführt werden. Wenn wir von der Propagationsrichtung abstrahieren möchten, benutzen wir eine beliebige Konvertierungsfunktion $cf_{c,j\leftarrow i}$ und sprechen von sv_i als *Quell-* und von sv_j als *Zielschemaversion.*

6.3.3.1 Das Erzeugungsflag (c-Flag)

Ist das Erzeugungsflag für die Vorwärtspropagation von Objekten der Klasse c von sv_u nach sv_v eingeschaltet (c[97]), so werden Objekte, die nach der Ableitung von sv_v im Zugriffsbereich von sv_u erzeugt werden, in den Zugriffsbereich von sv_v propagiert (siehe oberen Teil von Abbildung 6.4). Jedes nach der Ableitung von sv_v in sv_u erzeugte Objekt o wird damit auch im Zugriffsbereich von sv_v enthalten sein. Ebenso wie beim Schnappschußflag wird hierfür eine Version $sv_v.o$ von o angelegt und ihr Wert durch Anwendung der Konvertierungsfunktion bestimmt: $sv_v.o = fcf_{c,v \leftarrow u}(sv_u.o)$. Demzufolge handelt es sich bei beiden Objektversionen um denselben logischen Objektwert.

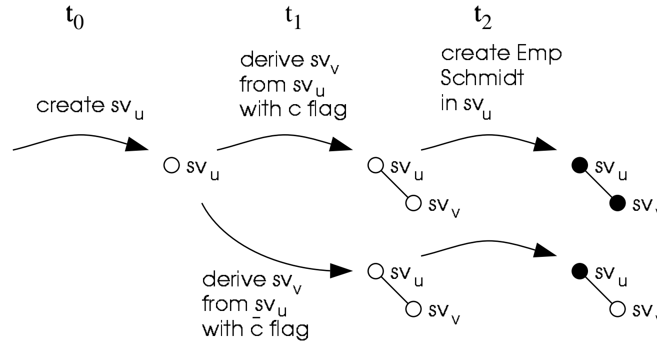


Abbildung 6.4: Die Semantik des Erzeugungsflags.

Ist das Erzeugungsflag jedoch wie im unteren Teil von Abbildung 6.4 dargestellt ausgeschaltet ($\overline{c}$), dann wird das Objekt o nicht im Zugriffsbereich von sv_v sichtbar.

Der Unterschied zwischen Schnappschußflag und Erzeugungsflag liegt damit in ihrem Wirkungsbereich. Das Schnappschußflag regelt die Sichtbarkeit von zur Ableitungszeit bereits im Zugriffsbereich von sv_u existierenden Objekten, während sich das Erzeugungsflag nur auf solche Objekte bezieht, die erst nach der Ableitung von sv_v im Zugriffsbereich der Vorgängerschemaversion sv_u erzeugt werden.

Bei der Rückwärtspropagation funktioniert der Mechanismus analog. Dabei muß dann allerdings das Erzeugungsflag für die Rückwärtsrichtung gesetzt sein und die Rückwärtskonvertierungsfunktion $bcf_{c,u \leftarrow v}$ verwendet werden.

Invariante 6.2 {**Erzeugungsflag**}
Wenn ein Objekt o einer Klasse c im Zugriffsbereich einer Schemaversion sv erzeugt wird, so wird o genau dann auch im Zugriffsbereich einer Schemaversion sv' sichtbar werden, wenn das folgende gilt.

$$sv' \in dsucc(sv.c) \cup dpred(sv.c) \wedge c\text{-flag} \in cpo(sv, sv', c).$$

Da wir uns in diesem Abschnitt zunächst auf die Propagation zwischen benachbarten Schemaversionen beschränken, treten in Invariante 6.2 nur die direkten Vorgänger- ($dpred$) und Nachfolgerschemaversionen ($dsucc$) auf. In Abschnitt 6.5 werden wir die transitive Propagation untersuchen.

[97] Wir verwenden das Symbol c in dieser Arbeit gleichermaßen für eine Klasse wie auch für das Erzeugungsflag. Da es sich dabei jedoch um zwei klar getrennte Bereiche handelt, sollte hier keine Verwechslungsgefahr bestehen.

6.3.3.2 Das Modifikationsflag (m-Flag)

Ist das Modifikationsflag wie im oberen Teil von Abbildung 6.5 dargestellt für die Vorwärtspropagation der Klasse c von sv_u nach sv_v eingeschaltet (m), so werden Modifikationen, die an Objekten im Zugriffsbereich von $sv_u.c$ durchgeführt werden, nach sv_v propagiert. Voraussetzung dazu ist allerdings, daß das Objekt o zum Zeitpunkt der Modifikation im Zugriffsbereich von sv_u auch im Zugriffsbereich von sv_v enthalten ist, was durch ein gesetztes Schnappschuß- oder Erzeugungsflag verursacht sein kann. Im Gegensatz zum Schnappschuß- und Erzeugungsflag legt das Modifikationsflag nämlich keine Objektversion $sv_v.o$ an, sondern setzt deren Existenz voraus.

Die Propagation der Modifikation geschieht dadurch, daß die Konvertierungsfunktion $fcf_{c,v \leftarrow u}$ zum Zeitpunkt der Modifikation auf $sv_u.o$ angewendet wird. Wurde das Objekt o ursprünglich in sv_u angelegt, so ist dies zumindest die zweite Anwendung von $fcf_{c,v \leftarrow u}$ auf o. Das Objekt kann jedoch auch in sv_v erzeugt und durch $bcf_{c,u \leftarrow v}$ nach sv_u propagiert worden sein. Der resultierende Wert der Konvertierung wird der bereits existierenden Objektversion $sv_v.o$ zugewiesen und der bisherige Wert von $sv_v.o$ dabei überschrieben.

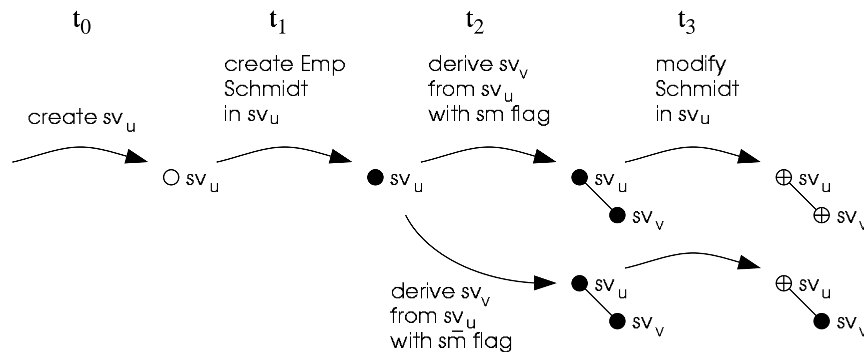


Abbildung 6.5: Die Semantik des Modifikationsflags.

Ist das Modifikationsflag jedoch ausgeschaltet ($\overline{m}$), dann wirken sich nach der Ableitung von sv_v im Zugriffsbereich von sv_u durchgeführte Änderungen nicht mehr auf Objekte im Zugriffsbereich von sv_v aus. Wie im unteren Teil von Abbildung 6.5 dargestellt repräsentieren die Objektversionen $sv_u.o$ und $sv_v.o$ dann verschiedene logische Objektwerte: Während $sv_v.o$ weiterhin den alten Wert enthält, repräsentiert $sv_u.o$ nun einen neuen Wert, der in der Abbildung durch ein anderes Symbol dargestellt ist.

Die Rückwärtspropagation von Modifikationen geschieht analog der Vorwärtspropagation. Insbesondere kommt es dabei nicht darauf an, ob das Objekt ursprünglich in sv_u oder in sv_v erzeugt wurde. Es muß lediglich in den Zugriffsbereich der jeweils anderen Schemaversion propagiert worden sein.

Invariante 6.3 {**Modifikationsflag**}
Wenn ein Objekt o einer Klasse c im Zugriffsbereich einer Schemaversion sv modifiziert wird, so wird o mit dem modifizierten logischen Objektwert genau dann auch im Zugriffsbereich einer Schemaversion sv′ sichtbar werden, wenn das folgende gilt.

$$sv' \in dsucc(sv.c) \cup dpred(sv.c) \wedge m\text{-flag} \in cpo(sv, sv', c) \wedge o \in IAS(sv'.c)$$

6.3.3.2.1 Überschreiben von Objektmodifikationen

Bei ausgeschaltetem Propagationsflag können in den Zugriffsbereichen verschiedener Schemaversionen auch verschiedene logische Werte eines Objektes sichtbar sein. Dabei kann es in der

entgegengesetzten Richtung auch passieren, daß eine Modifikation eines älteren logischen Objektwertes einen neueren logischen Wert überschreibt. Dazu betrachten wir die in Abbildung 6.6 dargestellte Entwicklung.

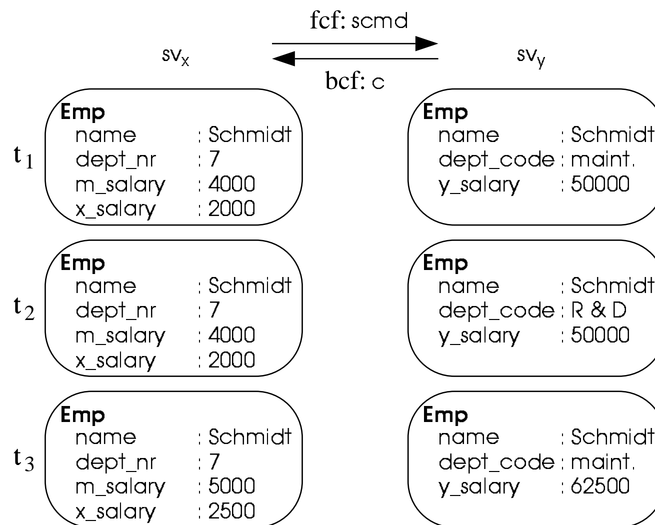


Abbildung 6.6: Die zu t_2 im Zugriffsbereich von sv_y durchgeführte Änderung von Objekt Schmidt wird durch die Propagation zu t_3 überschrieben.

Zum Zeitpunkt $t_1 > dt(sv_y)$ wird der Angestellte Schmidt in Schemaversion sv_y erzeugt und aufgrund des für die Rückwärtspropagation eingeschalteten Erzeugungsflags mit Hilfe der Rückwärtskonvertierungsfunktion $bcf_{Emp,x \leftarrow y}$ automatisch in den Zugriffsbereich von sv_x propagiert.

Zum Zeitpunkt $t_2 > t_1$ werde Herr Schmidt zu einer anderen Abteilung versetzt, weshalb eine Applikation von sv_y den Attributwert von `dept_code` entsprechend modifiziert. Da das Modifikationsflag für die Rückwärtspropagation von sv_y nach sv_x ausgeschaltet ist, wird diese Modifikation nicht in den Zugriffsbereich von sv_x propagiert, d.h. das Herrn Schmidt repräsentierende Objekt hat damit verschiedene (und möglicherweise inkonsistente) logische Werte in den Zugriffsbereichen der beiden Schemaversionen. Dabei ist insbesondere der logische Objektwert in sv_y nun neuer als derjenige in sv_x. Aber damit verhält sich das System genau so, wie es der Schemaentwickler durch Ausschalten der Rückwärtspropagation von Änderungen spezifiziert hat.

Nun erhalte Herr Schmidt zum Zeitpunkt $t_3 > t_2$ eine Gehaltserhöhung, die durch eine Applikation von sv_x in deren Zugriffsbereich eingebracht wird. Dies löst nun jedoch aufgrund des für die Vorwärtspropagation von sv_x nach sv_y eingeschalteten Modifikationsflags die Ausführung der Vorwärtskonvertierungsfunktion $fcf_{Emp,y \leftarrow x}$ aus. Damit wird jedoch der zum Zeitpunkt t_2 im Zugriffsbereich von sv_y modifizierte Wert des Attributes `dept_code` überschrieben, d.h. die auf diesem Attribut zum Zeitpunkt t_2 ausgeführte Modifikation geht zum Zeitpunkt t_3 wieder verloren.

In CLOSQL (siehe Abschnitt 4.5.3.3) werden in einer ähnlichen Situation Zwischenwerte in einem speziellen Systembereich der Datenbank gespeichert. Wenn ein Objekt aus einer Klassenversion $sv_x.c$ in eine Version $sv_y.c$ mit weniger Attributen propagiert wird, so werden die Werte dieser „verlorengegangenen“ Attribute zwischengespeichert. Wird dasselbe Objekt später von $sv_y.c$ zurück nach $sv_x.c$ propagiert, so werden die zwischengespeicherten Werte wiederhergestellt und nicht die durch die Konvertierungsfunktion berechneten Werte verwendet. Wir folgen diesem Ansatz aus zwei Gründen nicht: Zum einen kann sich die Semantik des Objektes in der Zwischenzeit erheblich geändert haben, so daß der zwischengespeicherte Wert gar nicht mehr zu dem propagierten „paßt“. Zum anderen können wir automatisch gar nicht entscheiden, wie sinnvoll die von einer Konvertierungsfunktion (insbesondere von einer durch den Schemaentwickler

definierten) spezifizierten Zuweisungen überhaupt sind. Damit haben wir aber auch nicht das Recht, diese Werte als unsinnig zu verwerfen und durch zwischengespeicherte alte Werte zu ersetzen. Zu welchen Problemen solch eine Strategie führen kann, hatten wir bereits in Abbildung 4.7 dargestellt.

6.3.3.2.2 Betrachtung von Konvertierungsfunktionen mit Werterhaltung

Wir wollen in diesem Abschnitt auf eine besondere Aufgabenstellung hinweisen, die eng mit dem Einsatz des Modifikationsflags im Zusammenhang mit neu hinzugefügten Attributen verknüpft ist. Wir gehen dabei von einem Ausschnitt eines Schemas aus, der zwei Versionen einer Klasse c enthält: $sv_1.c$ und $sv_2.c$. Dabei sei letztere Klassenversion durch Integration von sv_1 und anschließendes Hinzufügen eines Attributes z entstanden.

```
create schema s {
    create schema version sv1 {
        create class c {
            create x type_of_x;
            create y type_of_y;
            }; /* create class c */
        } /* create schema version sv1 */
    create schema version sv2 {
        integrate class c from sv1;
        modify class c {
            attributes {
                create z type_of_z;
            }; /* modify class c */
        } /* create schema version sv2 */
} /* create schema s */
```

Beispiel eines Schemas.

Die Vorwärtskonvertierungsfunktion initialisiere den Wert des Attributes z in $sv_2.c$ mit dem Nullwert des entsprechenden Typs.

```
forward conversion {
    new.x = old.x;
    new.y = old.y;
    new.z = nil;
    }
```

Die Vorwärtskonvertierungsfunktion zum obigen Beispiel.

Weiterhin sei das Modifikationsflag für die Vorwärtspropagation eingeschaltet. Schließlich sei das Schnappschuß- oder das Erzeugungsflag in Vorwärts- oder das Erzeugungsflag in Rückwärtsrichtung eingeschaltet. Damit kann der folgende Fall eintreten: Ein Objekt o ist in beiden Schemaversionen sv_1 und sv_2 sichtbar und hat in sv_2 einen Wert, der ihm von einer Applikation von sv_2 zugewiesen wurde. Wird nun genau dieses Objekt o in sv_1 verändert, so wird aufgrund des Modifikationsflags die Konvertierungsfunktion $fcf_{c,2\leftarrow 1}$ angestoßen und damit der vormalige Wert von z durch einen Nullwert überschrieben.

Das beschriebene Verhalten wird mitunter nicht erwünscht sein. Schließlich mußte der z-Wert bei in sv_1 angelegten Objekten manuell durch eine Applikation von sv_2 gesetzt werden und der dafür investierte Aufwand soll nicht verloren gehen. Stattdessen wird man in einigen Situationen sicher vorziehen, daß z den bisherigen Wert behält anstatt von der Konvertierungsfunktion auf den Nullwert zurückgesetzt zu werden.

Um dies zu erlauben, führen wir hier eine Erweiterung der Propagationssprache ein, die es gestattet, die folgenden drei Fälle zu unterscheiden:

1 Das Attribut der Zielobjektversion soll auf jeden Fall mit einem in der Konvertierungsfunktion notierten Wert überschrieben werden. Dies entspricht dem bisherigen und wird durch eine Zuweisung an **new**.z ausgedrückt.

2 Das Attribut der Zielobjektversion soll nur dann geschrieben werden, wenn das Objekt erst durch diese Propagation in der Zielschemaversion sichtbar wird, d.h. wenn es sich um eine Propagation aufgrund des Schnappschuß- oder des Erzeugungsflags handelt.

3 Das Attribut der Zielobjektversion soll nur dann überschrieben werden, wenn das Objekt bereits zuvor in der Zielschemaversion enthalten war, d.h. wenn es sich um die Propagation einer Veränderung (Modifikationsflag) handelt.

Damit der Schemaentwickler bei der Spezifikation einer Konvertierungsfunktion $cf_{c,d \leftarrow s}$ ausdrücken kann, welcher der drei oben genannten Fälle vorliegt, muß ihm eine entsprechende Unterscheidungsmöglichkeit gegeben werden. Dabei bestehen drei Alternativen (Es sei a ein beliebiges Attribut der Zielklassenversion $sv_d.c.$):

1 Es könnte ein Prädikat **m-propagated** angeboten werden, das in einer **if-then-else**-Anweisung ausgewertet werden kann.

```
new.a = expr_1;                      // Fall 1
if NOT (m-propagated)
    then    { new.a = expr_2; }      // Fall 2
    else    { new.a = expr_3; }      // Fall 3
```

Alternative 1 zur Werterhaltung.

2 Die **if-then-else**-Anweisung aus Alternative 1 könnte zu einem speziellen Zuweisungsoperator **assign** zusammengefaßt werden. Ein den obigen Anweisungen äquivalenter Ausdruck würde sich damit wie folgt darstellen:

```
new.a = expr_1;             // Fall 1
assign (a, expr_2, expr_3); // Fall 2 bzw. 3
```

Alternative 2 zur Werterhaltung.

Um auszudrücken, daß in Fall 2 oder 3 keine Zuweisung erfolgen soll, müßten die Ausdrücke expr_2 und expr_3 der **assign**-Anweisung optional sein. Die dabei entstehende Syntax mit zwei direkt aufeinanderfolgenden Kommata (falls in Fall 2 keine Zuweisung erfolgen soll) oder einem Komma gefolgt von einer schließenden Klammer (falls in Fall 3 keine Zuweisung erfolgen soll) wäre allerdings nur schwer lesbar.

3 Schließlich könnten drei verschiedene Variablen **new**.a, **create**.a und **modify**.a angeboten werden, wobei **create**.a nur in Fall 2 und **modify**.a nur in Fall 3 zugewiesen würden.

```
new.a = expr_1;    // Fall 1
create.a = expr_2; // Fall 2
modify.a = expr_3; // Fall 3
```

Alternative 3 zur Werterhaltung.

In den Arbeiten zur direkten Schemaevolution in O_2 (siehe Abschnitt 4.3.4.2) wird vorgeschlagen, daß zu jeder benutzerdefinierten Konvertierungsfunktion eine systemdefinierte Defaultkonvertierungsfunktion erzeugt werden solle, die zum Zwecke der Initialisierung stets vor einer benutzerdefinierten Konvertierungsfunktion auszuführen sei. In COAST sehen wir die beiden Typen von Konvertierungsfunktionen hingegen als Alternativen, von denen stets nur eine zu verwenden ist. Bei einer benutzerdefinierten Konvertierungsfunktion auftretende Mängel können schließlich bereits während der Spezifikation beseitigt werden, womit man sich zur Laufzeit den Aufwand für die zusätzliche Ausführung der Defaultkonvertierungsfunktion erspart. Davon abgesehen könnten mit der beschriebenen Erweiterung auch für COAST die bei der von O_2 gewählten Vorgehensweise benötigten Defaultkonvertierungsfunktionen erzeugt werden.

Die hier vorgestellte Erweiterung ist in COAST zum gegenwärtigen Zeitpunkt nicht implementiert und wird daher in Kapitel 7 auch nicht behandelt. Wir möchten stattdessen hier einige Bemerkungen bezüglich anderer Ansätze der Literatur und zu einer möglichen Implementierung der beschriebenen Erweiterung machen.

Bei Ansätzen der direkten Schemaevolution (siehe Abschnitt 4.3) wird nur in Vorwärtsrichtung propagiert und selbst bei einer verzögerten Implementierung, wie sie beispielsweise für O_2 vorgeschlagen wurde (siehe Abschnitt 4.3.4.2), ist die hier besprochene Erweiterung sinnlos, da jeder Konvertierungsschritt (pro Objekt) nur einmal durchgeführt wird, d.h. es findet keine Propagation von Änderungen statt. Bei der Beschreibung der Versionierungsansätze (siehe Abschnitt 4.5) finden sich keinerlei Hinweise auf die hier besprochene Problematik, so daß davon auszugehen ist, daß diese dort nicht berücksichtigt wurde. Sichten schließlich werden ebenfalls von einer Datenquelle zu einer Senke hin spezifiziert, so daß auch auf Sichten aufbauende Simulationsmechanismen (siehe Abschnitt 4.4) keine Unterstützung für die besprochene Erweiterung bieten.

Im Rahmen unserer Arbeit ist hier zunächst an das technische Teilziel 3.15 zu erinnern, das eine effiziente Realisierung der Propagationsmechanismen fordert. Dies ist besonders dann möglich, wenn auf die Durchführung von Propagationen in dieselbe Zielschemaversion sv_d verzichtet werden kann, solange das betroffene Objekt dort (d.h. in sv_d) zwischenzeitlich nicht zugegriffen wird. Damit ist die Durchführung einer Reihe von Konvertierungsfunktionen $cf_{c,d \leftarrow s_n} \circ \cdots \circ cf_{c,d \leftarrow s_1}$ mit derselben Zielschemaversion sv_d äquivalent zur Ausführung lediglich der letzten Konvertierungsfunktion $cf_{c,d \leftarrow s_n}$ dieser Reihe. Dann muß in den Fällen, in denen das Objekt in der Zielschemaversion sv_d zwischenzeitlich nicht benutzt wird, nur einmal, nämlich unmittelbar vor dem Zugriff durch sv_d propagiert werden und dazu genügt die Ausführung einer einzigen Konvertierungfunktion. Um diese sehr hilfreiche Eigenschaft zu erreichen, muß sichergestellt sein, daß keine Konvertierungsfunktion Zuweisungen in Abhängigkeit von vorherigen Werten von Attributen der Zielobjektversion macht.

Bei der Implementierung ist besonders darauf zu achten, daß der Unterschied zwischen den Fällen 2 und 3 physikalisch erkennbar bleibt, damit wie oben beschrieben, auf der logischen Ebene verschieden verfahren werden kann.

- Eine Objektversion existiert nicht und hat noch nie existiert. Dabei handelt es sich dann um Fall 2.
- Eine Objektversion hat bereits existiert, sie wurde jedoch aufgrund einer Optimierung auf der physikalischen Ebene vom Propagationsmechanismus gelöscht. Dann handelt es sich um Fall 3.

Es sei abschließend bemerkt, daß eine beliebige Zuweisung von Werten an Attribute in Abhängigkeit des vorherigen Zustandes von Attributen in der Zielschemaversion unter den gegebenen Prämissen nicht erlaubt werden kann. Dies liegt darin begründet, daß dabei die beschriebene

Einsparung der Ausführung von Propagationsschritten nicht möglich wäre. Schon im einfachsten Fall, wenn $s := s_1 = \cdots = s_n$ gilt, es sich also stets nur um ein und dieselbe Konvertierungsfunktion handelt, müssen i.Allg. alle Konvertierungsschritte ausgeführt werden, d.h. es sind n Ausführungen von $cf_{c,d \leftarrow s}$ erforderlich. Dies erkennt man leicht am Beispiel einer Konvertierungsfunktion $cf_{c,d \leftarrow s}$ mit $\mathbf{new}.z = \mathbf{new}.z + 1$. Hierbei wird offensichtlich die Zahl der Ausführungen mitgezählt und ein korrektes Ergebnis wird nur dann erreicht, wenn alle Konvertierungsschritte auch durchgeführt werden. In der obigen Erweiterung waren wir daher sorgsam darauf bedacht, nur solche Operationen zuzulassen, die den Wert in der Zielobjektversion höchstens erhalten, ihn aber nicht in Abhängigkeit seines früheren Wertes verändern.

6.3.3.3 Das Löschungsflag (d-Flag)

Vorwärtspropagation von Löschungen bedeutet, daß ein Objekt o, das aus dem Zugriffsbereich von sv_u gelöscht wird, automatisch auch aus dem Zugriffsbereich von sv_v entfernt wird. Voraussetzung dazu ist neben dem eingeschalteten Löschungsflag (d) für die Vorwärtspropagation natürlich wie bei der Propagation von Modifikationen, daß das Objekt in der Zielschemaversion sv_v überhaupt sichtbar ist. Der obere Teil von Abbildung 6.7 zeigt die Propagation der Löschung von Objekt o.

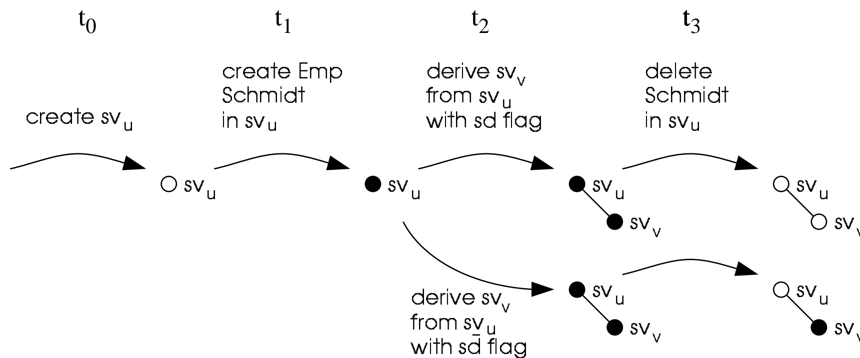


Abbildung 6.7: Die Semantik des Löschungsflags.

Ist das Löschungsflag für Propagationen von Objekten der Klasse c von sv_u nach sv_v wie im unteren Teil von Abbildung 6.7 dargestellt ausgeschaltet ($\overline{d}$), so wird das Objekt zwar aus dem Zugriffsbereich von sv_u gelöscht; im Zugriffsbereich von sv_v bleibt es jedoch weiterhin unverändert enthalten.

Die Rückwärtspropagation von Löschungen geschieht wiederum analog. Auch hier kommt es nicht darauf an, ob das Objekt zunächst in sv_u erzeugt und dann nach sv_v propagiert worden war oder umgekehrt.

Allein für die Propagation von Objektlöschungen (d.h. wenn lediglich das Löschungsflag eingeschaltet ist) werden natürlich keine Konvertierungsfunktionen benötigt. Wir gehen jedoch im folgenden zumindest formal von der Existenz von Konvertierungsfunktionen aus, selbst wenn diese nie benötigt werden. Damit ist die Existenz einer Konvertierungsfunktion gleichbedeutend damit, daß entlang dieser Kante propagiert werden kann.

Invariante 6.4 {**Löschungsflag**}
Wenn ein Objekt o einer Klasse c im Zugriffsbereich einer Schemaversion sv gelöscht wird, so wird o genau dann auch im Zugriffsbereich einer Schemaversion sv' gelöscht werden, wenn das folgende gilt.

$$sv' \in dsucc(sv.c) \cup dpred(sv.c) \wedge d\text{-flag} \in cpo(sv, sv', c).$$

Während des Ableitungsprozesses einer Schemaversion sv_v von sv_u muß der Schemaentwickler die gewünschten Propagationstypen spezifizieren. Dazu muß er für jede benötigte Vorwärtspropagation von sv_u nach sv_v und für jede Rückwärtspropagation von sv_v nach sv_u die entsprechenden Flags und Konvertierungsfunktionen angeben. In Abschnitt 6.4 werden wir die Sprache vorstellen, mit der diese Angaben spezifiziert werden.

6.3.4 Einige Propagationsflags in unserem Beispielszenario

Wir möchten nun anhand einiger ausgewählter Ausschnitte erläutern, wie die Propagationsflags die Modellierung unseres Beispielszenarios aus Abschnitt 3.1.2 unterstützen können.

Die Schemaversion sv_6 dient als dauerhafte Ablage für abgeschlossene Softwareentwicklungsprojekte und kommt demzufolge ohne Zustandsattribute, ohne Projektpläne und ohne Zwischenberichte und dergleichen aus. Keines der durch sv_6 zugreifbaren Dokumente wird durch eine lokale Applikation, d.h. durch eine Applikationen dieser Schemaversion erstellt. Stattdessen kommen sie ausnahmslos durch Propagation von anderen Schemaversionen in den Zugriffsbereich von sv_6. Dementsprechend sind Schnappschuß- und Erzeugungsflags in Richtung sv_6 eingeschaltet. Auch Modifikationen sollen in dieser Richtung weitergegeben werden. Löschungen in Vorgängerschemaversionen dürfen hingegen nicht propagiert werden. In Rückwärtsrichtung soll keinerlei Propagation stattfinden und demzufolge wird für sv_6 auch keine Rückwärtskonvertierungsfunktion benötigt.

Konvertierungsfunktionen, die die komplette Semantik der Quellschemaversion in die Zielschemaversion übertragen, nennen wir auch *informationserhaltend*. Bei der Umrechnung zwischen kartesischen Koordinaten und Polarkoordinaten gilt dies sogar in beide Richtungen. In solchen Fällen können alle Propagationsflags bedenkenlos eingeschaltet werden, falls eine maximale Ähnlichkeit zwischen den Zugriffsbereichen zweier Schemaversionen gewünscht wird.

Wenn eine Konvertierungsfunktion allerdings Attribute mit Nullwerten beschreibt, so ist es ggf. ratsam, auf die Propagation von Modifikationen zu verzichten. Die Schemaversion sv_2 hat beispielsweise die Klasse `Document` um die Attribute `Titel` und `Status` ergänzt, welche von der Konvertierungsfunktion $fcf_{Document,2 \leftarrow 1}$ mit Defaultwerten initialisiert werden. Diese Defaultwerte könnten mit Hilfe einer Applikation von sv_2 manuell durch sinnvollere Einträge ersetzt werden. Erfolgt nun jedoch noch eine Änderung eines Dokumentes in sv_1, so wird $fcf_{Document,2 \leftarrow 1}$ bei eingeschaltetem Modifikationsflag erneut ausgeführt und setzt dabei die manuell eingetragenen Titel- und Status-Informationen wieder auf die Defaultwerte zurück. Wie mit derlei Problemen umzugehen ist, hängt im Einzelfall von den Applikationen der beteiligten Schemaversionen ab und kann nicht grundsätzlich entschieden werden. Abschnitt 6.3.3.2.2 diskutierte dieses Problem und bot eine konzeptionelle Lösung an.

Für wichtige Aufgaben des Managements wie das Erzeugen und Bearbeiten von Projektplänen, die Zuordnung von Angestellten zu Aufgaben, die Festlegung von Zeitvorgaben oder das Löschen von Dokumenten nach Projektabschluß mögen ausgezeichnete Applikationen zur Verfügung stehen, die nicht von jedem Angestellten benutzt werden dürfen. Zum Schutz vor unerwünschten Änderungen kritischer Dokumente kann die Propagation zu den Schemaversionen hin, auf denen die entsprechenden Management-Applikationen aufsetzen, abgeschaltet werden. Damit erfüllt unser Modell in eingeschränktem Maße auch Autorisierungsaufgaben.

6.3.5 Kombinationsmöglichkeiten der Propagationsflags

Die Propagationsflags s, c, m und d können bei der Ableitung einer Klasse nahezu beliebig verwendet und kombiniert werden. Selbst die alleinige Verwendung des Modifikations- oder des

Löschungsflags für die Propagation von einer Schemaversion sv_x zu einer benachbarten Schemaversion sv_y ist möglich, wenn in der umgekehrten Richtung (also von sv_y zu sv_x) zumindest das Schnappschuß-[98] oder das Erzeugungsflag gesetzt ist. Werden Objekte jedoch weder in Vorwärts- noch in Rückwärtsrichtung propagiert (d.h. weder s- noch c-Flag sind zwischen sv_x und sv_y eingeschaltet), so kann kein Objekt in den Zugriffsbereichen beider Schemaversionen sichtbar werden und demzufolge macht auch die Verwendung von m- oder d-Flag in einer solchen Situation keinen Sinn.

Weitere Einschränkungen bei der Kombination von Propagationsflags ergeben sich bei der Betrachtung mehrerer Klassen einer Schemaversion, da diese über Vererbungs- und Aggregationsbeziehungen zueinander in Beziehung stehen. Betrachtet man die Propagation verschiedener Klassen zwischen zwei Schemaversionen, so sind demzufolge Konsistenzbedingungen zu erwarten, die die Menge der verwendbaren Propagationsflags einschränken. Dazu wäre ggf. eine Erweiterung des Objektmodells angebracht.

Bertino [BM93a] erlaubt beispielsweise eine Klassifikation von Aggregationsbeziehungen (siehe dazu Fußnote 83 auf Seite 161). Eine vorstellbare Einschränkung könnte hiermit ausdrücken, daß Modifikationen einer Klasse zumindest dann propagiert werden müssen, wenn sie abhängige und exklusiv zugeordnete Komponenten von Objekten einer Klasse darstellt, deren Modifikationen ebenfalls propagiert werden. Analog sollte sich die Propagation von Löschungen komplexer Objekte auch auf deren abhängige Komponenten erstrecken. Im Bereich der Versionierung komplexer Objekte treten ähnliche Fragestellungen auf [TOC93]. Dort ist etwa bei der Veränderung einer Komponente zu entscheiden, ob auch von übergeordneten, zusammengesetzten Objekten neue Versionen anzulegen sind.

Grundsätzlich kann hier noch bemerkt werden, daß das Einschalten aller Flags in beide Richtungen im Zusammenhang mit informationserhaltenden Konvertierungsfunktionen dem Sichtenkonzept sehr nahe kommt. Werden jedoch alle Flags ausgeschaltet, so resultiert ein Mechanismus, der konzeptionell isolierten Datenbanken gleicht.

6.4 Die Propagationssprache

In Abschnitt 5.5 hatten wir die Schemaänderungsprimitive der COAST-ODL vorgestellt. Die dort bereits erwähnten Primitive der Kategorie 5 beziehen sich auf die Propagation von Objekten und damit auf die Objektebene. Daher hatten wir deren Erläuterung zurückgestellt, so daß wir hier zunächst die konzeptionellen Grundlagen der Schemaversionierung auf Objektebene legen konnten. Wir holen nun die in Abschnitt 5.5 zurückgestellte Vorstellung der Primitive für die Propagation nach. Auch das Integrationsprimitiv (3.1) der COAST-ODL läßt die Angabe von Propagationsflags zu, weswegen wir auch dessen vollständige Syntax erst hier angeben. Da die hier vorgestellten Primitive zusammen mit der nicht näher angegebenen Syntax der Konvertierungsfunktionen die Propagation auf Objektebene steuern, bezeichnen wir diesen Teil der ODL auch als *Propagationssprache*.

Das Integrationsprimitiv erlaubt die Spezifikation der Propagationsflags für die integrierten Klassen. Durch Angabe von + oder − werden die Flags ein- bzw. ausgeschaltet. Bei bereits integrierten Klassen werden die Propagationsflags nicht verändert. Die anderen angegebenen Optionen hatten wir bereits in Abschnitt 5.5.3.1 vorgestellt. Da ein Aufruf von **integrate** in der Regel mehrere Klassen integriert und die integrierten Klassen noch verändert werden können, haben

[98] Das Schnappschußflag kann wie bereits dargelegt nur für die Vorwärtspropagation gesetzt werden, d.h. wenn sv_x die direkte Vorgängerschemaversion von sv_y ist.

wir von der Angabe der Konvertierungsfunktionen direkt beim Integrieren abgesehen. Für diesen Zweck ist das unten beschriebene **modify derivation**-Primitiv zu verwenden.

```
integrate class cname from svname    [[no] superclasses]
                                     [[no] subclasses]
                                     [[no] componentclasses]
                                     [keep | replace]
                                     [by classid | by classname]
                                     [forward propagation s+|− c+|− m+|− d+|−]
                                     [backward propagation c+|− m+|− d+|−];
```

Integrieren von Klassenversionen.

Mit **create derivation** können nachträglich Ableitungsbeziehungen zu mit **create class** erzeugten Klassen hinzugefügt werden. Damit wird ein Zustand erreicht der demjenigen entspricht, der bei Integration der Klasse *sv.c* (anstelle des Anlegens einer neuen Klasse) erreicht worden wäre. Wir werden in Abschnitt 7.1.2.2 zeigen, wie dieses Primitiv durch ein Schema-Werkzeug realisiert werden kann.

```
create derivation for class cname to class sourcecname in schemaversion svname
                                     [forward propagation s+|− c+|− m+|− d+|−]
                                     [backward propagation c+|− m+|− d+|−];
                                     [forward conversion {
                                         ... // Konvertierungsanweisungen
                                         }]
                                     [backward conversion {
                                         ... // Konvertierungsanweisungen
                                         }];
```

Anlegen von Ableitungsbeziehungen.

Für die Änderung bereits durch **integrate class** etablierter Ableitungsbeziehungen ist das **modify derivation**-Primitiv zu verwenden. Dieses Primitiv erlaubt die nachträgliche Veränderung bestehender Ableitungsbeziehungen. Zum einen können die Propagationsflags verändert werden, z.B. wenn mit einem Aufruf des Integrationsprimitives mehrere Klassen integriert wurden, die aber nicht alle dieselben Flags haben sollen. Zum anderen wird **modify derivation** insbesondere für die Spezifikation der Konvertierungsfunktionen von integrierten Klassen benötigt, da das Integrationsprimitiv selbst keine Spezifikation der Konvertierungsfunktionen zuläßt.

```
modify derivation for class cname    [forward propagation s+|− c+|− m+|− d+|−]
                                     [backward propagation c+|− m+|− d+|−];
                                     [forward conversion {
                                         ... // Konvertierungsanweisungen
                                         }]
                                     [backward conversion {
                                         ... // Konvertierungsanweisungen
                                         }];
```

Verändern von Ableitungsbeziehungen.

Die Zielsetzung von **integrate** und **create derivation** entspricht damit ungefähr der des Gleichheitsoperators der CSL von Odberg [Odb95].

Analog zum Anlegen ist auch das nachträgliche Löschen von Ableitungsbeziehungen möglich. Die Klasse namens `cname` wird nach Verwendung des **delete derivation**-Primitives so verwendet, als wäre sie mit **create class** lokal angelegt worden, d.h. es findet keinerlei Propagation zu anderen, bereits bestehenden Schemaversionen statt.

delete derivation for class `cname`;

Löschen von Ableitungsbeziehungen.

Weitere Erläuterungen zur Propagationssprache von COAST finden sich in [Her99].

6.5 Transitive Objektkonvertierung und -propagation

Bisher haben wir uns auf die Propagation zwischen direkten Nachbarn im Schemaableitungsgraphen beschränkt. Damit folgten wir den technischen Teilzielen 3.16 (geringer Spezifikationsaufwand) und 3.17 (Lokalität). Diese stehen jedoch in einem teilweisen Widerspruch zum technischen Teilziel 3.11 (vollständige Propagation). Dieses Problem lösen wir, indem wir automatisch eine Komposition mehrerer Konvertierungsfunktionen immer dann durchführen, wenn keine direkte Propagation möglich ist.

Abbildung 6.8 zeigt die transitive Propagation einer Objekterzeugung. Schemaversion sv_w ist von sv_v abgeleitet, welche wiederum von sv_u abgeleitet ist. Die Klasse c ist in allen drei genannten Schemaversionen enthalten und wurde in sv_v und in sv_w jeweils von der direkten Vorgängerschemaversion integriert. Weiterhin wurde das Erzeugungsflag bei beiden Ableitungen eingeschaltet. Wir betrachten wieder nur ein Objekt der Klasse c, das zum Zeitpunkt t_2 noch nicht in der Datenbasis enthalten sei. Zum Zeitpunkt t_3 werde $o \in c$ nun im Zugriffsbereich von sv_u erzeugt. Aufgrund des für die Vorwärtspropagation der Klasse c von sv_u nach sv_v eingeschalteten Erzeugungsflags wird Objektversion $sv_v.o$ angelegt und mit dem Wert $sv_v.o = fcf_{c,v\leftarrow u}(sv_u.o)$ initialisiert. Doch dies wird als Objekterzeugung im Zugriffsbereich von sv_v klassifiziert und löst demzufolge nun die Propagation von sv_v nach sv_w aus, d.h. auch Objektversion $sv_w.o$ wird angelegt und mit $sv_w.o = fcf_{c,w\leftarrow v}(sv_v.o) = fcf_{c,w\leftarrow v} \circ fcf_{c,v\leftarrow u}(sv_u.o)$ initialisiert. Demzufolge kann man sagen, daß die Propagation des Objektwertes von sv_u nach sv_w mittels der Komposition zweier Konvertierungsfunktionen gelungen ist: $cf_{c,w\leftarrow u} := fcf_{c,w\leftarrow v} \circ fcf_{c,v\leftarrow u}$.

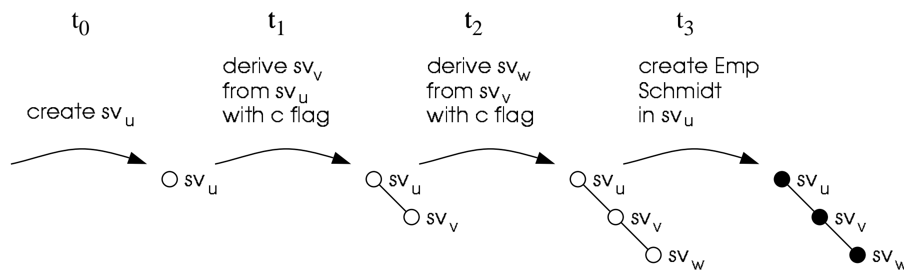


Abbildung 6.8: Transitive Propagation von Objekterzeugungen.

Auf die beschriebene Art und Weise können grundsätzlich Erzeugungen, Modifikationen und Löschungen von Objekten von einer Version sv eines Schemas s in eine beliebige Teilmenge aller Versionen von s propagiert werden.

Wir waren bisher davon ausgegangen, daß die direkte Ableitung einer Klassenversion $sv_v.c$ von $sv_u.c$ ($sv_v <_{sv} sv_u$) stets mit der Objektpropagation zwischen $sv_u.c$ und $sv_v.c$ mit Hilfe von Konvertierungsfunktionen $fcf_{c,v\leftarrow u}$ und $bcf_{c,u\leftarrow v}$ einhergeht. Um die Ableitung einer Klassenversion

von der Propagation ihrer Objekte zu entkoppeln, unterscheiden wir ab jetzt allerdings zwischen beiden Konzepten. Während sich Klassenableitungsbäume (definiert durch $<^1_{sv,c}$) und (direkte) Vorgänger- und Nachfolgerklassenversionen ($pred(sv.c)$, $dpred(sv.c)$, $succ(sv.c)$, $dsucc(sv.c)$) auf die Ableitung beziehen, führen wir nun Klassenkonvertierungsbäume ein, die sich auf die Existenz von Konvertierungsfunktionen und damit auf die Möglichkeit zur Objektpropagation beziehen. Im Allgemeinen müssen die beiden Baumtypen nicht deckungsgleich sein, d.h. eine Klassenversion $sv.c$ kann von einer Schemaversion $sv_1 \in dpred(sv)$ abgeleitet sein, während Konvertierungsfunktionen zwischen $sv.c$ und $sv_2.c$ ($sv_1 \neq sv_2$) existieren. Diese Unterscheidung nehmen wir insbesondere auch mit Hinblick auf eine später durchzuführende Verallgemeinerung von Konvertierungsbäumen zu Konvertierungsgraphen vor.

Definition 6.6 {**Klassenkonvertierungsbaum,** CCT_c}
Gegeben sei ein Schemaableitungsgraph $SDG = (sv(s), <^1_{sv})$.

Ein Klassenkonvertierungsbaum[99] für eine Klasse c (engl. class conversion tree) *stellt sämtliche zwischen Versionen der Klasse c definierten Vorwärts- und Rückwärtskonvertierungsfunktionen dar. Formal wird ein Klassenkonvertierungsbaum CCT_c beschrieben durch $CCT_c = (sv(c), <^1_{cc})$. Für eine Klasse $c \neq$* **Object** *gilt $sv_v.c <^1_{cc} sv_u.c$ genau dann, wenn von $sv_v.c$ nach $sv_u.c$ eine Konvertierungsfunktion existiert. Da die systemdefinierte Klasse* **Object** *keine direkten Instanzen hat, benötigen wir für sie keinen Klassenkonvertierungsbaum.*

Genau genommen handelt es sich bei einem Klassenkonvertierungsbaum um einen zusammenhängenden Graphen, denn entsprechend Regel 6.1 wird bei der Ableitung einer neuen Schemaversion für jede abgeleitete Klasse c sowohl eine Vorwärts- als auch eine Rückwärtskonvertierungsfunktion definiert. Da die Quellschemaversion der Vorwärtskonvertierungsfunktion dabei entsprechend Regel 6.1 mit der Zielschemaversion der Rückwärtskonvertierungsfunktion identisch sein muß, ist die Beziehung $<^1_{cc}$ hier symmetrisch ($(sv_x.c <^1_{cc} sv_y.c) \Leftrightarrow (sv_y.c <^1_{cc} sv_x.c)$). Durch die Definition der Beziehung $(sv_v.c <^1_{xx} sv_u.c) :\Leftrightarrow ((sv_v.c <^1_{cc} sv_u.c) \wedge (sv_u.c <^1_{cc} sv_v.c) \wedge (sv_v \in dsucc(sv_u)))$ (oder gleichbedeutend $(sv_v.c <^1_{xx} sv_u.c) :\Leftrightarrow (\exists\ fcf_{c,v\leftarrow u} \wedge \exists\ bcf_{c,u\leftarrow v})$) fassen wir ein Paar aus Vorwärts- und Rückwärtskonvertierungsfunktion zu einer Kante zusammen und damit ergibt sich mit $(sv(c), <^1_{xx})$ tatsächlich ein Baum. Daher haben wir den Begriff des Baumes gewählt. Außerdem werden wir den Klassenkonvertierungsbaum in Definition 6.14 zu einer asymmetrischen Graphstruktur verallgemeinern, die sich nicht mehr wie oben als Baum darstellen läßt.

Der Klassenkonvertierungsbaum beschreibt ähnlich dem Klassenableitungsbaum eine Beziehung zwischen den Versionen einer Klasse. Im Gegensatz zu der im Ableitungsbaum dargestellten Ableitungsbeziehung handelt es sich hier um die Existenz von Konvertierungsfunktionen. Zwischen im Klassenkonvertierungsbaum direkt benachbarten Versionen einer Klasse existiert sowohl eine Vorwärts- als auch eine Rückwärtskonvertierungsfunktion. Abbildung 6.11 wird einige Beispiele zeigen.

Regel 6.1 erlaubt die Definition von Konvertierungsfunktionen nur zwischen direkten Vorgänger- und Nachfolgerschemaversionen. Die sog. *transitive Objektpropagation* zwischen beliebigen Versionen eines Schemas erfolgt entlang von Konvertierungspfaden.

Definition 6.7 {**Konvertierungspfad,** cp}
Ein Konvertierungspfad *cp für eine Klasse c ist ein linearer Pfad $cp = \langle sv_{x_1}, \ldots, sv_{x_m} \rangle$ durch eine Menge von Schemaversionen,*[100] *so daß die Versionen $sv_{x_1}.c, \ldots, sv_{x_m}.c$ der Klasse c existie-*

[99] Der Klassenkonvertierungsbaum wurde in [Lau97a] als Konvertierungsbaum (engl. *conversion tree*) bezeichnet und mit ct symbolisiert.

[100] An dieser Stelle führt der Konvertierungspfad präzise ausgedrückt durch den Klassenkonvertierungsbaum der Klasse c. Wie bereits angedeutet werden wir den Klassenkonvertierungsbaum in Definition 6.14 zu einem

ren und Konvertierungsfunktionen $cf_{c,x_{(i+1)}\leftarrow x_i}$ für $i \in \{1,\ldots,m-1\}$ zwischen ihnen vorhanden sind, d.h. es gilt jeweils $sv_{x_{(i+1)}}.c <^1_{cc} sv_{x_i}.c$ oder $sv_{x_i}.c <^1_{cc} sv_{x_{(i+1)}}.c$.

Die Propagation eines Objektes entlang des Konvertierungspfades cp wird durch die Komposition der entsprechenden Konvertierungsfunktionen bewerkstelligt und wir verwenden diese auch zur Repräsentation des Pfades: $cf_{c,x_m\leftarrow x_1} := cf_{c,x_m\leftarrow x_{(m-1)}} \circ \cdots \circ cf_{c,x_2\leftarrow x_1}$.

Für die Propagation von Löschungen sind natürlich keine Konvertierungsfunktionen erforderlich. Trotzdem werden Pfade benötigt, entlang derer sich Löschungen entsprechend des Löschungsflags transitiv fortpflanzen können. Hier verfolgen wir die einfachste und naheliegendste Lösung, indem wir Löschungen wie Erzeugungen und Modifikationen entlang der definierten Konvertierungspfade propagieren. Dabei stellt es kein Problem dar, daß unsere Definition von Konvertierungspfaden auf Konvertierungsfunktionen beruht und diese für die Propagation von Löschungen eigentlich gar nicht benötigt werden.

Die Existenz und Eindeutigkeit von Konvertierungspfaden wird in den nächsten beiden Abschnitten näher untersucht.

6.5.1 Bestimmung des Konvertierungspfades

Nach der kurzen Motivation für die Funktionsweise der transitiven Propagation von Objekterzeugungen, -modifikationen, und -löschungen soll nun zunächst geklärt werden, wie die transitive Propagation i.Allg. zu bewerkstelligen ist. Dabei ist die Frage nach der Auswahl der für die transitive Propagation zu verwendenden Konvertierungsfunktionen von besonderem Interesse. Für die Propagation von Objektlöschungen werden zwar keinerlei Konvertierungsfunktionen benötigt, die Erweiterung von der Propagation zwischen direkt benachbarten Schemaversionen auf die transitive Variante zwischen beliebigen Schemaversionen geschieht jedoch analog zu der für die transitive Propagation von Objekterzeugungen und -modifikationen durchgeführten Komposition von Konvertierungsfunktionen entlang eines Konvertierungspfades.

Die folgenden Betrachtungen gehen von dem allgemeinen Fall aus. Gegeben seien eine Quellschemaversion sv_{source} und eine Zielschemaversion sv_{drain}, die beide Versionen desselben Schemas s seien und die beide die Klasse c enthalten ($sv_{source}.c, sv_{drain}.c \in sv(c) \subseteq sv(s)$). Gesucht ist eine Möglichkeit, ein Objekt der Klasse c von sv_{source} nach sv_{drain} propagieren zu können.

Wir gehen ohne Beschränkung der Allgemeinheit davon aus, daß die beiden folgenden Pfade[101] von der Erzeugerversion ($csv(c)$) der Klasse c durch ihren Klassenkonvertierungsbaum zu ihren Versionen in sv_{source} und sv_{drain} existieren:[102]

$$\begin{array}{lclcl} \langle csv(c) & = & sv_{x_0} >^1_{cc} sv_{x_1} >^1_{cc} \cdots >^1_{cc} sv_{x_p} & = & sv_{source}\rangle \\ \langle csv(c) & = & sv_{y_0} >^1_{cc} sv_{y_1} >^1_{cc} \cdots >^1_{cc} sv_{y_q} & = & sv_{drain}\rangle \end{array}$$

In Abbildung 5.10 ist der Pfad der Klasse `Document` von $csv(Document) = sv_1$ nach sv_6 ($x_0 = 1$, $x_1 = 2$, $x_2 = 4$, $x_3 = 6$):

$$\langle sv_1 >^1_{cc} sv_2 >^1_{cc} sv_4 >^1_{cc} sv_6\rangle$$

allgemeinen Graphen erweitern. Um die hier gegebene Definition von Konvertierungspfad nicht später auf einen Pfad durch den Klassenkonvertierungsgraphen verallgemeinern zu müssen, haben wir an dieser Stelle absichtlich die etwas unpräzisere Formulierung als „Menge von Schemaversionen“ gewählt.

[101] Der Kürze der Darstellung wegen weichen wir hier und im folgenden von der bisherigen Darstellung von Pfaden nach Definition 2.8 ab und stellen Ordnungsbeziehungen zwischen den Knoten eines Pfades mit dar. Wir schreiben $\langle a < b < c\rangle$ für den Pfad $\langle a, b, c\rangle$, wenn die entsprechende Relation gilt.

[102] Den Nachweis der Existenz dieser Konvertierungspfade wird Lemma 6.1 erbringen.

Die Konvertierung von Objekten der Klasse c von sv_{source} nach sv_{drain} könnte auf dem folgenden Konvertierungspfad vorgenommen werden:

$$\begin{aligned} cf_{c,drain\leftarrow source} &= fcf \circ bcf \\ fcf &= fcf_{c,drain\leftarrow y_{(q-1)}} \circ \cdots \circ fcf_{c,y_1\leftarrow y_0} \\ bcf &= bcf_{c,x_0\leftarrow x_1} \circ \cdots \circ bcf_{c,x_{(p-1)}\leftarrow source} \end{aligned}$$

Existiert dabei jedoch mindestens ein $j > 0$, so daß $\forall i \in \{0,\ldots,j\} : x_i = y_i$ gilt, dann sieht der mittlere Ausschnitt des Konvertierungspfades wie folgt aus (dabei sei *scp* das maximale j mit der beschriebenen Eigenschaft):

$$\begin{aligned} cf_{c,drain\leftarrow source} = \; & \cdots \circ fcf_{c,x_{scp}\leftarrow x_{(scp-1)}} \circ fcf_{c,x_{(scp-1)}\leftarrow x_{(scp-2)}} \circ \cdots \circ fcf_{c,x_1\leftarrow x_0} \circ \\ & bcf_{c,x_0\leftarrow x_1} \circ \cdots \circ bcf_{c,x_{(scp-2)}\leftarrow x_{(scp-1)}} \circ bcf_{c,x_{(scp-1)}\leftarrow x_{scp}} \circ \cdots \end{aligned}$$

Das heißt in dem oben angegebenen Pfad werden *scp* Konvertierungsschritte erst rückwärts und dann vorwärts ausgeführt (siehe Abbildung 6.9). Mit dieser Lösung würde jedoch nicht nur überflüssiger Aufwand getrieben, es könnte sogar Objektsemantik verloren gehen, weil die durch die Konvertierungsfunktionen implementierten Abbildungen nicht notwendigerweise bijektiv sind, weswegen eine Rückwärtskonvertierungsfunktion $bcf_{c,x\leftarrow y}$ dann auch nicht die Umkehrfunktion der Vorwärtskonvertierungsfunktion $fcf_{c,y\leftarrow x}$ ist, d.h. es gilt nicht notwendigerweise $fcf_{c,y\leftarrow x} \circ bcf_{c,x\leftarrow y} = \boldsymbol{id}$. Die Umkehrfunktion $fcf^{-1}_{c,y\leftarrow x}$ existiert i.Allg. gar nicht.

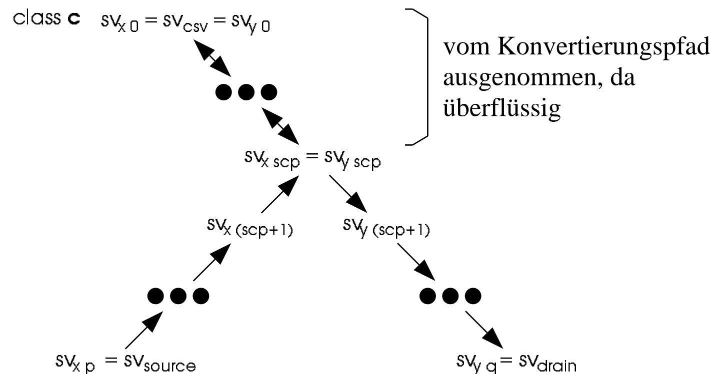


Abbildung 6.9: Der reguläre Konvertierungspfad via scp.

Der oben dargestellte mittlere Ausschnitt des Konvertierungspfades muß daher von der Durchführung der Propagation ausgenommen werden. Zur Ermittlung dieses mittleren Ausschnittes definieren wir zunächst den *kleinsten gemeinsamen Vorgänger zweier Schemaversionen.*

Definition 6.8 {kleinste gemeinsame Vorgängerschemaversion, $scp_c(sv_x, sv_y)$**}**
Gegeben seien zwei Versionen sv_x and sv_y eines Schemas s, die beide die Klasse c enthalten, welche von $csv(c)$ entlang der beiden folgenden Pfade abgeleitet sei ($csv(c) = sv_{x_0} = sv_{y_0}$):

$$\begin{aligned} \langle csv(c) &= sv_{x_0} >^1_{cc} sv_{x_1} >^1_{cc} \cdots >^1_{cc} sv_{x_p} &= sv_x \rangle \\ \langle csv(c) &= sv_{y_0} >^1_{cc} sv_{y_1} >^1_{cc} \cdots >^1_{cc} sv_{y_q} &= sv_y \rangle \end{aligned}$$

Die kleinste gemeinsame Vorgängerschemaversion scp_c (engl. smallest common predecessor) *der beiden Schemaversionen sv_x und sv_y bezüglich der Klasse c ist $scp_c(sv_x, sv_y) := sv_{x_{scp}}$ mit $sv_{x_{scp}} <^1_{cc} sv_x$, $sv_{x_{scp}} <^1_{cc} sv_y$ und $scp = \max\{j | x_i = y_i \; \forall i \in \{0,\ldots,j\}\}$.*

Beispiel 6.3 Abbildung 6.10 zeigt ein Beispiel eines Schemaableitungsgraphen, wobei eine Klasse c in $csv(c) = sv_1$ erzeugt und in $sv_2, \ldots, sv_9$ abgeleitet wurde. Die Klasse c wurde in sv_8 und in sv_9 entlang der folgenden Pfade abgeleitet:

$$\langle sv_1 >^1_{cc} sv_3 >^1_{cc} sv_5 >^1_{cc} sv_8 \rangle$$
$$\langle sv_1 >^1_{cc} sv_3 >^1_{cc} sv_4 >^1_{cc} sv_7 >^1_{cc} sv_9 \rangle$$

Die kleinste gemeinsame Vorgängerschemaversion von sv_9 und sv_8 bezüglich der Klasse c ist demnach $scp_c(sv_9, sv_8) = sv_3$. □

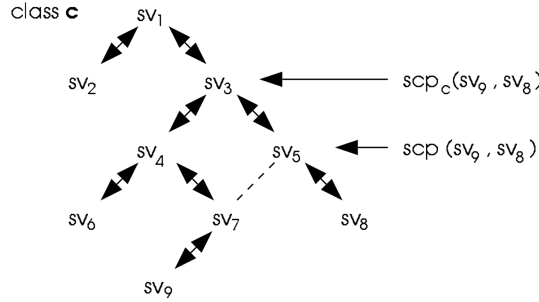


Abbildung 6.10: Ein Beispiel für unterschiedliche scp_c und scp.

Die kleinste gemeinsame Vorgängerschemaversion $scp_c(sv_x, sv_y)$ bezüglich einer Klasse c ist i.Allg. übrigens nicht identisch mit dem kleinsten gemeinsamen Vorgänger $scp(sv_x, sv_y)$ von sv_x und sv_y im Schemaableitungsgraphen. In Abbildung 6.10 soll die gestrichelte Linie zwischen sv_5 und sv_7 darstellen, daß die entsprechende Kante zwar zum Schemaableitungsgraphen gehört, nicht aber zum Klassenkonvertierungsbaum der Klasse c. Damit unterscheidet sich $sv_3 = scp_c(sv_9, sv_8)$ von $sv_5 = scp(sv_9, sv_8)$.

Jedoch gilt stets $scp(sv_x, sv_y) \leq_{sv} scp_c(sv_x, sv_y)$. Desweiteren sind alle Schemaversionen, die auf den beiden Ableitungspfaden unterhalb der kleinsten gemeinsamen Vorgängerschemaversion liegen, voneinander verschieden: $\forall a \in \{1, \ldots, \min(p, q)\} : (sv_{x_a} <_{sv} scp_c(sv_x, sv_y) \Rightarrow (x_a \neq y_a))$. Es kann jedoch trotzdem passieren, daß zwei verschiedene auf dem Ableitungspfad unterhalb der kleinsten gemeinsamen Vorgängerschemaversion liegende Schemaversionen sv_{x_a} und sv_{y_b} dieselbe[103] Version von Klasse c definieren. Es existieren also möglicherweise $a \in \{0, \ldots, p\}$ und $b \in \{0, \ldots, q\}$, so daß $sv_{x_a}, sv_{y_b} <_{sv} scp_c(sv_x, sv_y)$ und $sv_{x_a}.c = sv_{y_b}.c$. Zur Vermeidung solcher Situationen soll das in Abschnitt 7.1.2.2 zu beschreibende Werkzeug zur Verbesserung des Schemaableitungsgraphen beitragen.

Um Objekte der Klasse c von sv_{source} nach sv_{drain} zu konvertieren, benutzen wir den folgenden Pfad, der den im obigen Vorschlag noch enthaltenen überflüssigen und ggf. sogar störenden mittleren Ausschnitt nicht enthält ($sv_{x_{scp}} := scp_c(sv_{source}, sv_{drain})$).

$$\begin{aligned} cf_{c,drain \leftarrow source} &= fcf \circ bcf \\ fcf &= fcf_{c,drain \leftarrow y_{(q-1)}} \circ \cdots \circ fcf_{c,y_{(scp+1)} \leftarrow y_{scp}} \\ bcf &= bcf_{c,x_{scp} \leftarrow x_{(scp+1)}} \circ \cdots \circ bcf_{c,x_{(p-1)} \leftarrow source} \end{aligned}$$

Damit arbeitet die transitive Propagation entlang des sog. *regulären Konvertierungspfades*.

[103] Hier sollen zwei Versionen einer Klasse als *gleich* angesehen werden, wenn sie dieselben Attribute (bezüglich Name und Typ) und dieselben Methoden (bezüglich Name und Signatur) anbieten.

Definition 6.9 {regulärer Konvertierungspfad}
Der regulärer Konvertierungspfad *von sv_{source} nach sv_{drain} bezüglich einer in diesen beiden Schemaversionen enthaltenen Klasse c ist der eindeutige Konvertierungspfad, der von sv_{source} via $scp_c(sv_{source}, sv_{drain})$ nach sv_{drain} führt.*

6.5.2 Eindeutigkeit des Konvertierungspfades

Teilziel 3.12 drückt die Forderung nach einer möglichst flexiblen Steuerung der Propagation aus. Dabei blieb bisher unberücksichtigt, daß die spezifizierte Propagation auch entsprechend der Vorstellung des Schemaentwicklers ablaufen sollte. Dieser Anspruch schließt insbesondere die Eindeutigkeit und Unveränderlichkeit von Propagationspfaden ein. Daher muß der für beliebige, transitive Propagationen von einer Quellschemaversion sv_{source} in eine Zielschemaversion sv_{drain} zu benutzende Pfad für den Schemaentwickler klar erkennbar sein und darf sich nicht verändern. In Graphen existieren i.Allg. mehrere Pfade zwischen zwei Knoten. Eine solche Mehrdeutigkeit kann für die Propagation jedoch nicht in Kauf genommen werden, da sich auf verschiedenen Pfaden verschiedene Werte für das Objekt in der Zielschemaversion ergeben können und das Ergebnis der Propagation dann für den Schemaentwickler schwer oder gar nicht vorhersagbar ist. Auch eine nachträgliche Änderung eines Konvertierungspfades ist auszuschließen, da Objekte sonst zu verschiedenen Zeiten mit verschiedenen Werten propagiert würden, was insbesondere bei der Verwendung einer Strategie mit verzögerter Propagation zu unvorhersehbaren oder gar zufällig erscheinenden Auswirkungen führen könnte.

Wie wir im folgenden nachweisen werden, geschieht die Auswahl eines Konvertierungspfades zwischen zwei Versionen eines Schemas auf der Grundlage eines fest vorgegebenen Konvertierungsgraphen, d.h. bei einer gegebenen Menge von Konvertierungsfunktionen in eindeutiger Art und Weise. Die vorausgesetzte Unveränderlichkeit eines Konvertierungsgraphen wird durch die Regeln 5.2 und 5.4 gewährleistet, sobald eine erste Propagation in den Zugriffsbereich einer betrachteten Schemaversion sv stattgefunden hat. Ab diesem Zeitpunkt existiert nämlich zumindest ein Objekt im Zugriffsbereich von sv. Daraus ergibt sich nach Regel 5.4, daß die Schemaversion sv nicht mehr aufgetaut werden kann und damit nach Regel 5.2, daß auch die Menge ihrer Konvertierungsfunktionen zu anderen existierenden Schemaversionen unveränderlich ist. Bei der Ableitung neuer Schemaversionen werden natürlich zusätzliche Konvertierungsfunktionen, insbesondere auch Rückwärtskonvertierungsfunktionen zu bereits existierenden Schemaversionen (also z.B. auch zu sv) definiert, so daß sich gegebene Konvertierungsgraphen ändern können. Diese Änderungen sind jedoch ausschließlich Ergänzungen, die keinen Einfluß auf die Propagation zwischen bereits zuvor existierenden Schemaversionen haben.

Wir sprechen hier und im folgenden allgemein von der Eindeutigkeit von Propagationspfaden, ohne zwischen der Propagation von Erzeugungen, Änderungen und Löschungen zu unterscheiden. Bei der Propagation von Löschungen kommt es allerdings lediglich darauf an, ob zwischen der Schemaversion, wo die Löschung stattfindet und einer betrachteten Zielschemaversion ein Pfad mit dem Löschungsflag an allen Kanten existiert oder nicht; seine Eindeutigkeit ist jedoch hier genaugenommen nicht erforderlich. Da wir für die Propagation von Objekterzeugungen und -modifikationen aber ohnehin für eindeutige Propagationspfade sorgen müssen, verwenden wir diese auch für die Propagation von Löschungen.

Wir wollen nun zeigen, daß die im vorigen Abschnitt getroffene Wahl des regulären Konvertierungspfades eindeutig ist.

Wir betrachten nochmals Abbildung 5.10 und gehen dabei davon aus, daß sämtliche Konvertierungsfunktionen jeweils entlang der Ableitungskanten definiert wurden. Für die Klasse `Graph` sind dies beispielsweise $fcf_{Graph,3\leftarrow 2}$, $bcf_{Graph,2\leftarrow 3}$, $fcf_{Graph,4\leftarrow 2}$, $bcf_{Graph,2\leftarrow 4}$, $fcf_{Graph,5\leftarrow 3}$

und $bcf_{Graph,3\leftarrow 5}$. Abbildung 6.11 zeigt die Klassenkonvertierungsbäume einiger Klassen unseres durchgängigen Beispiels.

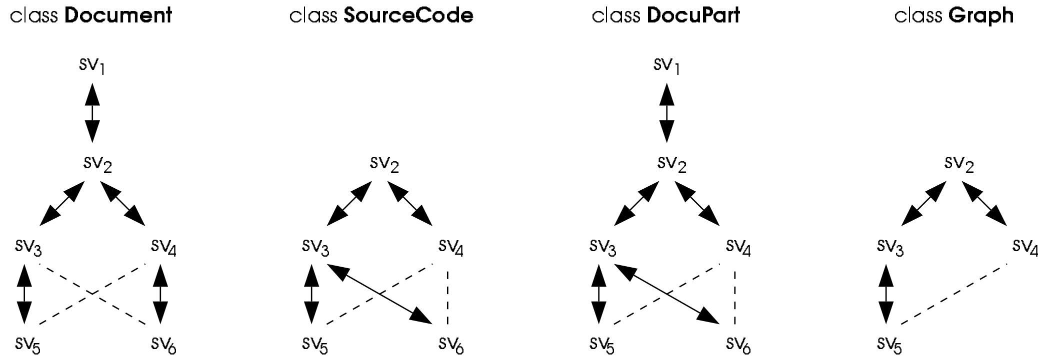


Abbildung 6.11: Die Klassenkonvertierungsbäume einiger Klassen des Beispiels aus Abschnitt 3.1.2.

Für die in Abbildung 6.11 gestrichelt dargestellten Linien wurden keine Konvertierungsfunktionen definiert, d.h. beispielsweise existiert keine Konvertierungsfunktion $fcf_{Document,6\leftarrow 3}$. Regel 6.1 hatte ja gerade die Definition dieser fehlenden Vorwärtskonvertierungsfunktion verboten, da bereits $fcf_{Document,6\leftarrow 4}$ für die Klasse `Document` bei der Integration von sv_3 und sv_4 zu sv_6 definiert wurde, und pro Klasse nur eine Vorwärtskonvertierungsfunktion (und entsprechend nur eine Rückwärtskonvertierungsfunktion) zwischen sv_6 und ihren Vorgängerschemaversionen sv_3 und sv_4 erlaubt ist.

Eine Applikation von Schemaversion sv_3 könnte jedoch Objekte der Klasse `Document` erzeugen, die auch im Zugriffsbereich von sv_6 sichtbar werden sollen. Die dazu notwendige Propagation kann jedoch aufgrund des Fehlens der Vorwärtskonvertierungsfunktion $fcf_{Document,6\leftarrow 3}$ nicht direkt durchgeführt werden. In solchen Fällen wird die Propagation entlang des regulären Konvertierungspfades durchgeführt. In dem beschriebenen Beispiel ist dies der indirekte Pfad $\langle sv_3, sv_2, sv_4, sv_6\rangle$ und die benötigte Konvertierungsfunktion kann durch Komposition existierender Konvertierungsfunktionen wie folgt automatisch berechnet werden: $cf_{Document,6\leftarrow 3} := fcf_{Document,6\leftarrow 4} \circ fcf_{Document,4\leftarrow 2} \circ bcf_{Document,2\leftarrow 3}$.

Die Verletzung von Regel 6.1 durch die zusätzliche Definition einer Vorwärtskonvertierungsfunktion $fcf_{Document,6\leftarrow 4}$ würde in einer Mehrdeutigkeit des für die Propagation von Objekten der Klasse `Document` von sv_4 nach sv_6 zu benutzenden Konvertierungspfades resultieren. In diesem Fall wäre zwar eine automatische Auflösung der Mehrdeutigkeit anhand der Pfadlänge möglich. Diesem Kriterium folgend würde die direkte Konvertierung in einem Schritt der indirekten in drei Schritten vorgezogen. Zumindest das Kriterium der Pfadlänge genügt i.Allg. jedoch nicht. Die zusätzliche Definition der Vorwärtskonvertierungsfunktion $fcf_{Document,6\leftarrow 3}$ hätte nämlich ebenfalls eine Mehrdeutigkeit bei der Propagation von sv_2 nach sv_6 verursacht: Sowohl $fcf_{Document,6\leftarrow 3} \circ fcf_{Document,3\leftarrow 2}$ als auch $fcf_{Document,6\leftarrow 4} \circ fcf_{Document,4\leftarrow 2}$ kommen hier als Alternativen in Betracht, jedoch ist anhand des Längenkriteriums keine Entscheidung zu Gunsten eines Pfades möglich, weil beide Pfade gleichlang sind. Auch der Einsatz anderer denkbarer Kriterien zur Gewichtung von Konvertierungspfaden mit dem Ziel immer eine eindeutige Auswahl treffen zu können hat sich als nicht erfolgversprechend herausgestellt [Eig97]. Dies liegt u.a. auch daran, daß die Auswahl natürlich zugunsten desjenigen Pfades getroffen werden sollte, auf dem mehr Objektsemantik übertragen werden kann. Dies zu messen ist jedoch äußerst problematisch.

Es soll an dieser Stelle nochmal explizit darauf hingewiesen werden, daß der Auswahl des Konvertierungspfades eine erhebliche Bedeutung zukommt. Schließlich können verschiedene Konver-

tierungsfunktionen der Objektversion der Zielschemaversion komplett andere Werte zuweisen. Wir betrachten dazu das folgende Beispiel.

Beispiel 6.4 Wir gehen davon aus, daß die Klasse `Document` in sv_2 keine Attribute hat und daß sv_1, sv_3 und sv_4 dieselbe Version der Klasse c mit zumindest einem Attribut spezifizieren (siehe Abbildung 6.12). In diesem Fall kann $fcf_{Document,4\leftarrow 2}$ die Attribute von Objekten der Klasse sv_4.`Document` lediglich mit **nil** initialisieren, d.h. auf dem ersten möglichen Konvertierungspfad von sv_1 nach sv_4 via sv_2 könnte keinerlei Objektsemantik übertragen werden. Im Gegensatz dazu wäre $fcf_{Document,4\leftarrow 3} = fcf_{Document,3\leftarrow 1} = \boldsymbol{id}$ und damit könnte auf dem Konvertierungspfad via sv_3 die komplette Semantik von Objekten der Klasse `Document` von sv_1 nach sv_4 übertragen werden.

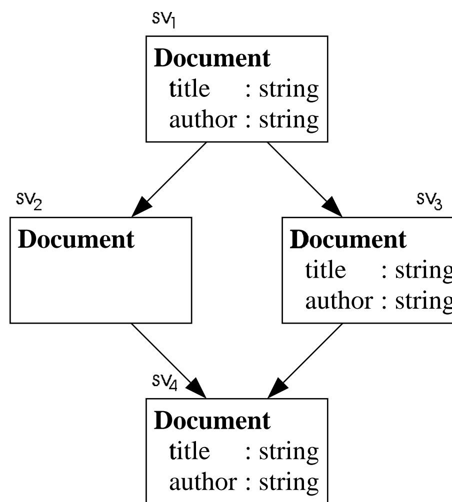


Abbildung 6.12: Semantik kann von sv_1 nach sv_4 nur auf dem Konvertierungspfad via sv_3 propagiert werden.

Auch wenn dieses Beispiel zur Hervorhebung möglicher Unterschiede recht künstlich ist, so sollte doch trotzdem ersichtlich sein, daß derartige Unterschiede auch schon beim Fehlen nur eines Attributes auftreten können. □

Das folgende Beispiel zeigt, warum die Forderung in Regel 6.1 notwendig gewesen war, daß Vorwärts- und Rückwärtskonvertierungsfunktion einer Klasse Objekte von bzw. nach derselben Klassenversion konvertieren.

Beispiel 6.5 Wenn Vorwärts- und Rückwärtskonvertierungsfunktion einer Klasse c bei der Integration von sv_2 und sv_3 zu sv_4 nicht wie von Regel 6.1 gefordert von bzw. zu derselben Klassenversion konvertieren, so kann das zu mehrdeutigen Konvertierungspfaden führen. In Abbildung 6.13 wurden $fcf_{c,4\leftarrow 2}$ und $bcf_{c,3\leftarrow 4}$ spezifiziert und damit wäre die Propagation von sv_2 nach sv_3 sowohl via sv_1 als auch via sv_4 möglich. □

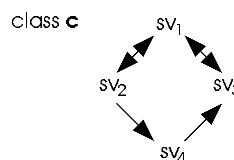


Abbildung 6.13: Mehrdeutige Konvertierungspfade.

Neben der unterschiedlichen Objektsemantik, die auf mehrdeutigen Konvertierungspfaden übertragen würde, könnten außerdem Konflikte bei der Komposition der entlang der verschiedenen Pfade ein- bzw. ausgeschalteten Propagationsflags auftreten.

Lemma 6.1 wird zeigen, daß Regel 6.1 das Auftreten mehrdeutiger Konvertierungspfade ausschließt. Die durch Regel 6.1 eingeführte Begrenzung auf genau eine Konvertierungsfunktion pro Klasse in jeder Richtung stellt jedoch keine echte Einschränkung der Flexibilität unseres Ansatzes dar. In unserem Beispielszenario wurde die Klasse `Document` bei der Ableitung von sv_6 nur geringfügig modifiziert; viele Klassen wurden unverändert in die Schemaversionen sv_5 und sv_6 integriert. In solchen Fällen entspricht die Konvertierungsfunktion der Identität und durch den zusätzlichen Propagationsschritt geht keinerlei Semantik verloren.

Definition 6.10 {Wohldefiniertheit eines Schemaableitungsgraphen}
Ein Schemaableitungsgraph $SDG = (sv(s), <^1_{sv})$ *heißt* wohldefiniert *(bezüglich seiner Konvertierungsfunktionen) genau dann, wenn für jedes Paar von Schemaversionen* $sv_x, sv_y \in sv(s)$, $(x \neq y)$ *und für jede Klasse* c *in* sv_x *und in* sv_y *das folgende gilt:*

Existenz: *ein Konvertierungspfad für die Klasse* c *von* sv_x *nach* sv_y *existiert und*

Eindeutigkeit: *ist eindeutig bestimmt.*

Lemma 6.1 (Wohldefiniertheit eines Schemaableitungsgraphen)
Wenn jede Schemaversion $sv \in sv(s)$ *entsprechend der Regeln 6.1 und 6.2 abgeleitet wurde, dann ist der resultierende Schemaableitungsgraph* $SDG = (sv(s), <^1_{sv})$ *wohldefiniert (bezüglich seiner Konvertierungsfunktionen).*

Beweis: Wir beweisen Lemma 6.1 durch vollständige Induktion auf der Menge $sv(c)$ aller Schemaversionen des Schemaableitungsgraphen, die die Klasse c enthalten, in der Reihenfolge der zeitlichen Ableitung der Schemaversionen.

Induktionsanfang ($\{sv_0\}$)
Lemma 6.1 gilt trivialerweise für $\{sv_0\}$.

Induktionsschritt ($\{sv_0, \ldots, sv_n\} \rightarrow \{sv_0, \ldots, sv_{(n+1)}\}$)
Nach Induktionsannahme gelte Lemma 6.1 nun für $\{sv_0, \ldots, sv_n\}$. Wir betrachten die abgeleitete Klasse c aus $sv_{(n+1)}$.

Fall 1: Die Schemaversion $sv_{(n+1)}$ wurde von nur einer einzigen Schemaversion sv_u abgeleitet.

Existenz: Nach Induktionsannahme können Objekte von jeder Klassenversion $sv_j.c$ ($j \in \{1, \ldots, n\}$) nach $sv_u.c$ propagiert werden, d.h. es existiert[104] eine Konvertierungsfunktion $cf_{c,u \leftarrow j}$. Aufgrund unserer Auswahl der Induktionsmenge existiert $sv_{(n+1)}.c$, nach Regel 6.1 existiert auch $fcf_{c,(n+1) \leftarrow u}$ und damit kann die Konvertierung nach $sv_{(n+1)}.c$ durch die folgende Komposition erledigt werden: $cf_{c,(n+1) \leftarrow j} := fcf_{c,(n+1) \leftarrow u} \circ cf_{c,u \leftarrow j}$.
Eine analoge Konstruktion zeigt, daß Objekte umgekehrt auch von der Klassenversion $sv_{(n+1)}.c$ in jede Klassenversion $sv_j.c$ ($j \in \{1, \ldots, n\}$) propagiert werden können: $cf_{c,j \leftarrow (n+1)} := cf_{c,j \leftarrow u} \circ bcf_{c,u \leftarrow (n+1)}$.

[104] Wie man beispielsweise an der Konvertierungsfunktion $cf_{Document,4 \leftarrow 3}$ in Abbildung 6.11 sieht, muß diese Konvertierungsfunktion nicht notwendigerweise ausschließlich aus Vorwärtskonvertierungsfunktionen bestehen.

Eindeutigkeit: Nach Induktionsannahme sind alle Konvertierungspfade innerhalb der Menge $\{sv_0, \ldots, sv_n\}$ eindeutig. Nach Regel 6.1 werden lediglich $fcf_{c,(n+1)\leftarrow u}$ und $bcf_{c,u\leftarrow(n+1)}$ zu der Menge der Konvertierungsfunktionen hinzugefügt. Wegen $sv_{(n+1)} \notin \{sv_0, \ldots, sv_n\}$ können dadurch keine Zyklen bei der Ableitung von $sv_{(n+1)}$ entstehen.

Fall 2: Die Schemaversion $sv_{(n+1)}$ wurde durch Integration mehrerer Schemaversionen $sv_{u_1}, \ldots, sv_{u_m} \in \{sv_0, \ldots, sv_n\}$ abgeleitet.

Existenz: Nach Regel 6.1 wurde genau eine Vorwärts- (und eine Rückwärts-) Konvertierungsfunktion für die Klasse c bei der Ableitung von $sv_{(n+1)}$ definiert. Wir nehmen ohne Beschränkung der Allgemeinheit an, diese sei $fcf_{c,(n+1)\leftarrow u_1}$ ($bcf_{c,u_1\leftarrow(n+1)}$). Damit können wir dieselbe Konstruktion wie in Fall 1 anwenden.

Eindeutigkeit: Wie in Fall 1 können durch das Hinzufügen einer Konvertierungsfunktion, deren Quell- und Zielschemaversion nicht beide in $\{sv_0, \ldots, sv_n\}$ liegen, keine Zyklen in $\{sv_0, \ldots, sv_n\} \cup \{sv_{(n+1)}\}$ entstehen.

$\Rightarrow$ Lemma 6.1 gilt auch für $\{sv_0, \ldots, sv_{(n+1)}\} = \{sv_0, \ldots, sv_n\} \cup \{sv_{(n+1)}\}$. □

Lemma 6.1 besagt, daß die Konvertierungsfunktionen jeder Klasse eines wohldefinierten Schemaableitungsgraphen einen Klassenkonvertierungsbaum definieren. Abbildung 6.11 zeigt einige der Klassenkonvertierungsbäume unseres durchgängigen Beispiels aus Abschnitt 3.1.2.

Für die Durchführung der Propagation (siehe Abschnitt 7.1.1.3) wird entscheidend sein, welche Zugriffsbereiche der Versionen eines Schemas bei einem schreibenden Zugriff auf die Datenbank angepaßt werden müssen. Greift eine Applikation erzeugend, modifizierend oder löschend auf ein Objekt der Klasse c im Zugriffsbereich einer Schemaversion sv zu, so sind zunächst die direkten Vorgänger und Nachfolger von sv im Klassenkonvertierungsbaum von c zu betrachten. Bei Schemaversionen, wo sich die Änderung auswirkt, sind dann deren noch nicht betrachtete Vorgänger und Nachfolger zu untersuchen, um die transitive Fortpflanzung der Propagation zu realisieren.

Der Wirkungsbereich einer Änderung kann sich potentiell über den gesamten Klassenkonvertierungsbaum von c erstrecken. Abhängig vom Typ der Änderung (Erzeugung, Modifikation oder Löschung eines Objektes) kann der Wirkungsbereich aufgrund der zu betrachtenden Propagationsflags (c-, m- oder d-Flag) weiter eingegrenzt werden. Dazu definieren wir im folgenden drei Typen von Propagationsbäumen, die im Vergleich zum Klassenkonvertierungsbaum nur noch jeweils diejenigen Versionen einer Klasse enthalten, in deren Zugriffsbereich sich eine Änderung in $sv.c$ aufgrund der spezifizierten Propagationsflags auswirken kann.

Zwei Hinweise zu Propagationsbäumen sind an dieser Stelle angebracht. Zum einen existiert für jede Klasse nur ein Klassenkonvertierungsbaum, die drei Propagationsbäume müssen jedoch für jede Klassenversion $sv.c$ definiert werden, da bei einer Änderung in $sv.c$ genau die Flags an den von $sv.c$ weg laufenden Kanten des Klassenkonvertierungsbaumes von c untersucht werden müssen. Zum zweiten schränkt der entsprechende Propagationsbaum den potentiellen Wirkungsbereich einer Änderung im Vergleich zum Klassenkonvertierungsbaum zwar weiter ein, bei Modifikationen und Löschungen hängt der tatsächliche Wirkungsbereich jedoch zusätzlich von der Sichtbarkeit des modifizierten oder gelöschten Objektes in den verschiedenen Zugriffsbereichen ab (siehe Abschnitt 6.5.3). Die transitive Propagation einer Modifikation oder Löschung erfaßt daher nicht notwendigerweise sämtliche Knoten des entsprechenden Propagationsbaumes.

Definition 6.11 {**c-, m- und d-Propagationsbaum**}
Der c- (m-, d-) Propagationsbaum einer Klassenversion *$sv.c$ ist der maximale, zusammenhängende Teilgraph*[105] *des Klassenkonvertierungsbaumes von c, der neben $sv.c$ genau alle Versionen*

[105] Ein Teilgraph enthält nur solche Knoten und Kanten, die auch bereits im Originalgraphen enthalten sind.

der Klasse c enthält, die von sv.c aus direkt oder indirekt erreicht werden können, wobei sämtliche begangene Kanten das c- (m-, d-) Flag in der entsprechenden Richtung eingeschaltet haben müssen.

Formal werden c-, m- und d-Propagationsbaum einer Klassenversion sv.c als $cst(sv.c)$, $mst(sv.c)$ und $dst(sv.c)$ notiert.

Beispiel 6.6 (siehe Abbildung 7.7 auf Seite 255):
$dst(sv_4.c) = \{sv_4.c, \ldots, sv_7.c, sv_{10}.c, sv_{12}.c, sv_{16}.c, sv_{17}.c, sv_{18}.c, sv_{21}.c, sv_{24}.c, sv_{25}.c\}$ □

Die Propagationsbäume werden also durch die Propagationsflags spezifiziert. Wurde bei der Ableitung einer Klassenversion das s-, c- oder m-Flag eingeschaltet, so wird in der entsprechenden Richtung eine Konvertierungsfunktion benötigt. Da die Propagationsbäume jedoch Teilgraphen von Klassenkonvertierungsbäumen sind, ist die Existenz der entsprechenden Konvertierungsfunktionen sichergestellt. In der Praxis gehen wir jedoch den umgekehrten Weg, d.h. der Schemaentwickler spezifiziert bei der Ableitung einer neuen Klassenversion zuerst die Flags für die Vorwärts- und Rückwärtspropagation. Nur wenn in einer Richtung das s-, c- oder m-Flag eingeschaltet ist, wird eine Konvertierungsfunktion in dieser Richtung benötigt. In diesem Falle wird dem Schemaentwickler eine automatisch generierte Defaultkonvertierungsfunktion vorgeschlagen, welche dieser bei Bedarf beliebig verändern kann. Wir haben uns zu dieser Vorgehensweise entschlossen, da der Schemaentwickler dem System so mitteilen kann, wo überhaupt eine Konvertierungsfunktion benötigt wird. Der Aufwand für die Implementierung einer Konvertierungsfunktion wird durch die angebotenen Defaultkonvertierungsfunktionen verringert oder entfällt sogar komplett. Beim Einschalten von Flags muß natürlich zunächst geprüft werden, ob Regel 6.1 noch eingehalten wird, wenn die entsprechend der spezifizierten Flags benötigten Konvertierungsfunktionen implementiert werden. Es kann natürlich auch Regel 6.2 direkt geprüft werden.

Das Integrationsprimitiv ist nützlich, wenn verschiedene Schemaversionen zusammengefaßt werden sollen. Weiterhin erlaubt es eine flexible Ableitung neuer Schemaversionen, da der Schemaentwickler die für seine Zwecke am besten geeigneten Implementierungen der verschiedenen Klassen aus den verschiedenen Schemaversionen zusammenstellen kann (siehe Abbildung 5.10). Jedoch kann nicht nur der Entwurf eines Schemas in verschiedenen Varianten enden, die zusammengesetzt werden sollen. Auch der Entwurf einer einzelnen Klasse kann in verschiedenen Alternativen resultieren, deren Objekte in einer gemeinsamen Klassenextension zugreifbar sein sollen. In diesen Fällen stellt das Integrationsprimitiv keine Hilfe dar. Wir werden in Abschnitt 6.6 Mechanismen zur Lösung dieses Problems vorstellen.

Da die bisherigen Beispiele fast nur Vorwärtskonvertierungsfunktionen beinhalteten, möchten wir explizit darauf hinweisen, daß die Transitivität natürlich auch für solche Konvertierungsfunktionen gegeben ist, die (ggf. ausschließlich) Rückwärtskonvertierungsfunktionen beinhalten. Im unserem Beispiel (siehe Abbildung 6.11), wird eine Änderung eines Dokument-Objektes in sv_6 durch $cf_{Document,2 \leftarrow 6} = bcf_{Document,2 \leftarrow 4} \circ bcf_{Document,4 \leftarrow 6}$ nach sv_2 propagiert. Aufgrund der automatischen Komposition der Konvertierungsfunktionen entlang der Konvertierungspfade muß der Schemaentwickler bei n Schemaversionen höchstens $2n - 2$ Konvertierungsfunktionen pro Klasse spezifizieren. Dies ist ein erheblicher Vorteil gegenüber dem Ansatz von Skarra und Zdonik (siehe Abschnitt 4.5.3.2).

6.5.3 Löcher in Konvertierungspfaden

Wir möchten nun noch auf eine Konsequenz der bisher vorgestellten Prinzipien hinweisen: „Löcher in Konvertierungspfaden werden bei der transitiven Propagation nicht traversiert“. Zur Erklärung dieser Aussage betrachten wir die im oberen Teil der Abbildung 6.14 dargestellte

Entwicklung bis zum Zeitpunkt t_3. Die in der Abbildung dargestellten Propagationsflags gelten jeweils in Vorwärtsrichtung; die Rückwärtspropagation sei abgeschaltet.[106] In der dargestellten Konfiguration können Modifikationen des Herrn Schmidt repräsentierenden Objektes zum Zeitpunkt $t_5 > t_3$ von sv_u nach sv_v und transitiv auch nach sv_w propagiert werden.

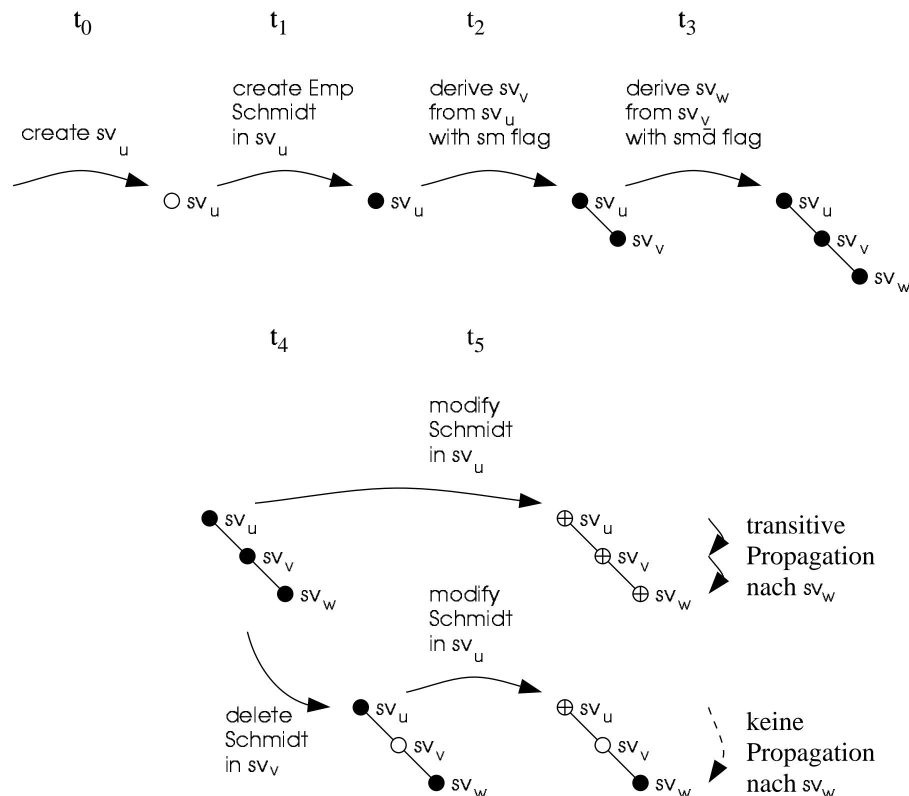


Abbildung 6.14: Transitive Propagation von Modifikationen.

Nehmen wir nun jedoch an, das Herrn Schmidt repräsentierende Objekt würde zum Zeitpunkt t_4 ($t_5 > t_4 > t_3$) durch eine Applikation von sv_v aus dem Zugriffsbereich von sv_v gelöscht, wie im unteren Teil von Abbildung 6.14 dargestellt. Diese Löschung würde nicht nach sv_w propagiert, weil das Löschungsflag der Klasse `Employee` in sv_w ausgeschaltet ist. Damit wäre das Objekt in den Zugriffsbereichen von sv_u und sv_w enthalten, nicht jedoch in dem von sv_v. Eine solche Konstellation bezeichnen wir als *Loch* im Konvertierungspfad von sv_u nach sv_w bezüglich der Existenz des Herrn Schmidt repräsentierenden Objektes.

Wenn nun in dieser Situation die beschriebene Modifikation des Objektes in sv_u zum Zeitpunkt t_5 durchgeführt wird, so kann sie trotz des für die Klasse `Employee` in sv_v gesetzten Modifikationsflags nicht nach sv_v propagiert werden, weil das Objekt nicht mehr im Zugriffsbereich von sv_v enthalten ist. Damit findet also keine Modifikation im Zugriffsbereich von sv_v statt, die transitiv nach sv_w propagiert werden könnte. Dies wäre jedoch der Fall gewesen, wenn das Objekt nicht aus dem Zugriffsbereich von sv_v gelöscht worden wäre. Demzufolge kann der Zustand eines Objektes im Zugriffsbereich von sv_w von der Sichtbarkeit dieses Objektes in sv_v abhängen.

Um dieses ggf. unerwünschte Verhalten zu vermeiden, bestünden beispielsweise die folgenden beiden Möglichkeiten, die Semantik des Modifikationsflags zu ändern (oder ein weiteres Flag m' einzuführen):

[106] Für das Beispiel ist nur wesentlich, daß Löschungen in sv_v nicht nach sv_u zurückpropagiert werden.

- Zum einen könnte das Objekt in dem Loch zum Zeitpunkt einer späteren Modifikation erneut angelegt werden. Dieses Wiederanlegen könnte dann wie das erstmalige Erzeugen eines Objektes behandelt und die Propagation der Modifikation somit durchgeführt werden. Im Beispiel würde das Herrn Schmidt repräsentierende Objekt zum Zeitpunkt t_5 im Zugriffsbereich von sv_v automatisch neu angelegt und in den Zugriffsbereich von sv_w propagiert.

- Zum anderen könnten Modifikationen im Zugriffsbereich einer Schemaversion sv grundsätzlich zu allen anderen Schemaversionen des m-Propagationsbaumes von $sv.c$, deren Zugriffsbereich das modifizierte Objekt enthält, propagiert werden, unabhängig von der Existenz des Objektes in den Zugriffsbereichen anderer Schemaversionen auf dem jeweiligen Konvertierungspfad. Im Beispiel würde die Modifikation somit nach sv_w propagiert, obwohl das modifizierte Objekt nicht im Zugriffsbereich von sv_v enthalten ist.

Wir haben uns jedoch für die zuvor beschriebene Alternative, bei der Löcher in Konvertierungspfaden nicht traversiert werden, entschieden, da die schrittweise Propagation von einer Schemaversion zur nächsten die geradlinigste Anwendung der Propagationsmechanismen darstellt. Indem wir bei jedem Konvertierungsschritt nur die beiden direkt beteiligten Schemaversionen betrachten, folgen wir auch dem technischen Teilziel 3.17 (Lokalität). Desweiteren würden die beiden dargestellten Alternativen sowohl die Formalisierung als auch die Implementierung unserer Mechanismen erheblich verkomplizieren.

Bei der Propagation von Löschungen kann kein durch Löcher verursachtes Problem entstehen. Um dies einzusehen, ist zunächst festzustellen, daß Löcher ausschließlich durch Löschung entstehen können. Entsprechend der schrittweisen Propagation eines Objektes kann es nicht vorkommen, daß bereits bei der Erzeugung eines Objektes ein Loch entsteht, wobei auf dem Propagationspfad zwischen zwei Schemaversionen, in denen ein Objekt sichtbar wird, eine Schemaversion liegt, deren Zugriffsbereich das Objekt nicht enthält. Nachdem wir festgestellt haben, daß ein Loch nur durch eine Löschung entstehen kann, bleibt lediglich zu bedenken, daß bereits diese erste Löschung, die zur Entstehung des Loches führte, überall hin propagiert worden ist, wohin eine Löschung von dieser Schemaversion in der jeweiligen Klasse überhaupt hinpropagiert werden kann. Demzufolge hätte die Propagation einer späteren Löschung hinter dem Loch sowieso keine weitere Auswirkung. Eine Situation, wie sie in Abbildung 6.15 dargestellt ist, kann also gar nicht auftreten. Denn dort hätte bereits bei der Entstehung des Loches die Löschung in den Bereich 2 propagiert werden müssen. Die dargestellte Situation bezüglich der Sichtbarkeit des Objektes kann lediglich dann auftreten, wenn die Propagation von Löschungen von dem Loch aus in den Bereich 2 hinein abgeschaltet wäre. Dann aber hätte eine spätere Löschung in Bereich 1 ohnehin keine Auswirkung auf Bereich 2.

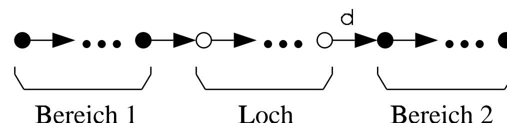


Abbildung 6.15: Eine unmögliche Situation.

Das folgende Beispiel soll die Funktionsweise der transitiven Propagation deutlich machen.

Beispiel 6.7 Abbildung 6.16 zeigt drei Schemaversionen sv_x, sv_y und sv_z. In sv_y wurde die Abteilungsnummer vom Typ **Integer** durch ein textuelles Abteilungskürzel ersetzt sowie Monatsgehalt (`m_salary`) und Zulagen (`x_salary`) aus $sv_x.Emp$ zu einem Jahresgehalt (`y_salary`) zusammengefaßt. Dessen Währung wurde in sv_z schließlich von US$ auf ECU geändert.

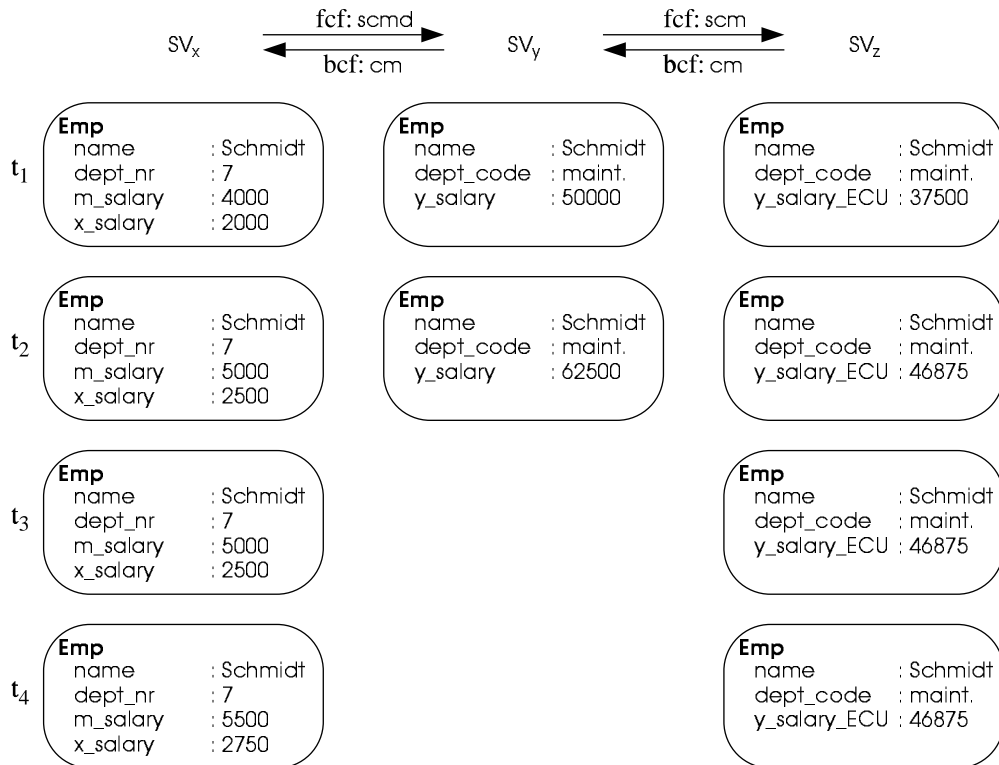


Abbildung 6.16: Die Transitivität des Propagationsmechanismus.

Die Konvertierungsfunktion $fcf_{Emp,y \leftarrow x}$ könnte beispielsweise wie folgt aussehen.

```
modify derivation for class Emp in schemaversion sv_y
    forward derivation s+c+m+d+;
    forward conversion {
        case old.dept_nr of
            1: new.dept_code = "contr.";
            ...
            7: new.dept_code = "maint.";
        end case;
        new.y_salary = old.x_salary + 12 * old.m_salary;
        }
```

Beispiel einer Konvertierungsfunktion mit Propagationsflags.

Die eingeschalteten Propagationsflags sind in der Abbildung ebenso dargestellt, wie die vier vorhandenen Konvertierungsfunktionen. Das Objekt Schmidt werde zu einem Zeitpunkt $t_1 > dt(sv_z)$ in sv_x erzeugt. Damit wird es aufgrund des eingeschalteten c-Flags nach sv_y propagiert, was in einer Objekterzeugung in $IAS(sv_y)$ resultiert. Eine solche Objekterzeugung muß nun wiederum nach sv_z propagiert werden, da das c-Flag auch für $fcf_{Emp,z \leftarrow y}$ eingeschaltet ist. Auf diesem Wege werden Erzeugungen bzw. Modifikationen von Objekten transitiv von der Erzeugerschemaversion des Objektes bzw. von der Schemaversion wo die Modifikation des Objektes stattfand zu jeder anderen Schemaversion im c- bzw. m-Propagationsbaum dieser Schemaversion bezüglich der Klasse c propagiert.

Zur Erläuterung der Voraussetzung, daß ein Objekt bereits im Zugriffsbereich der Zielschemaversion existieren muß, damit Modifikationen propagiert werden können, betrachten wir nochmals Abbildung 6.16.

Zum Zeitpunkt $t_2 > t_1$ werde das Objekt Schmidt durch eine Applikation von sv_x modifiziert. Da $o_{Schmidt}$ ebenfalls in $IAS(sv_y)$ enthalten ist, wird die Modifikation nach $IAS(sv_y)$ propagiert und folglich dann auch nach $IAS(sv_z)$. Bisher verhält sich alles so wie bei der Propagation von Erzeugungen und Löschungen.

Nun werde jedoch $o_{Schmidt}$ zum Zeitpunkt $t_3 > t_2$ aus $IAS(sv_y)$ gelöscht. Da das d-Flag weder für $bcf_{Emp,x \leftarrow y}$ noch für $fcf_{Emp,z \leftarrow y}$ gesetzt ist, bleibt das Objekt sowohl in sv_x als auch in sv_z sichtbar.

Wenn nun eine der obigen Modifikation zu t_2 ähnliche Modifikation zum Zeitpunkt $t_4 > t_3$ passiert, dann wird diese nicht von $IAS(sv_x)$ nach $IAS(sv_z)$ propagiert werden. Dies liegt darin begründet, daß die Modifikation nicht in einem ersten Schritt nach sv_y propagiert werden kann, da die Voraussetzung dafür nicht gegeben ist ($o_{Schmidt} \notin IAS(sv_y)$). Damit hat sich das Verhalten des System zwischen t_2 und t_4 verändert, was die Löschung von $o_{Schmidt}$ aus $IAS(sv_y)$ reflektiert. □

6.6 Extrakonvertierungsfunktionen

6.6.1 Motivation

Die in Abschnitt 6.5.1 vorgestellte automatische Komposition von Konvertierungsfunktionen entlang regulärer Konvertierungspfade ist aus zahlreichen Gründen sinnvoll: Erstens muß der Schemaentwickler beim Ableiten einer neuen Schemaversion nicht den gesamten Schemaableitungsgraphen kennen. Stattdessen genügt entsprechend dem technischen Teilziel 3.17 (Lokalität) die Definition der Beziehungen zwischen der neuen Schemaversion und ihren direkten Vorgängerschemaversionen. Zweitens wird der für die Implementierung der Konvertierungsfunktionen notwendige Aufwand erheblich dadurch reduziert, daß pro abgeleiteter Klasse lediglich eine Vorwärts- und eine Rückwärtskonvertierungsfunktion benötigt werden. In vielen Fällen genügt gar die automatisch generierte Defaultkonvertierungsfunktion. Das folgende Beispiel stellt jedoch eine Situation dar, in der der Schemaentwickler durch die Definition zusätzlicher Konvertierungsfunktionen mehr Semantik zwischen verschiedenen Versionen einer Klasse transportieren könnte.

Beispiel 6.8 Abbildung 6.17 zeigt die Schemaversionen sv_1, sv_2 und sv_3, die alle eine Version der Klasse `Person` beinhalten. Das Attribut `address` wurde in der Klassenversion sv_2.`Person` hinzugefügt, während das Geburtsdatum bei der Ableitung der Klasse `Person` in Schemaversion sv_3 ergänzt wurde. Wie bei den Schemaversionen sv_5 und sv_6 in Abbildung 5.10 wird auch hier vermutlich der nächste durchzuführende Schritt die Integration der bei der Ableitung von sv_2.`Person` und sv_3.`Person` erreichten Verbesserungen in einer neuen Schemaversion sv_4 sein. □

Im Gegensatz zu Abbildung 5.10, wo Verbesserungen an mehreren Klassen durchgeführt worden waren, wurden in Beispiel 6.8 zwei verschiedene Versionen derselben Klasse abgeleitet. Die Integration von sv_2 und sv_3 in sv_4 wäre mit den bisher beschriebenen Konzepten zwar möglich, hätte jedoch einen bedeutenden Nachteil: Da eine einzelne Klasse nur von einer einzigen Vorgängerschemaversion abgeleitet werden kann, dürfen Konvertierungsfunktionen für die Klasse `Person` Regel 6.1 folgend nur entweder (a) zwischen sv_2 und sv_4 oder (b) zwischen sv_3 und sv_4 implementiert werden. Nehmen wir zunächst an, daß Alternative (a) folgend die Konvertierungsfunktionen $fcf_{Person,4 \leftarrow 2}$ und $bcf_{Person,2 \leftarrow 4}$ bei der Ableitung von sv_4 spezifiziert werden. Dabei ist es jedoch nicht möglich, das Geburtsdatum einer Person zwischen sv_3 und sv_4 auf dem regulären Konvertierungspfad via sv_1 zu propagieren, da sv_1 und sv_2 kein Attribut für die

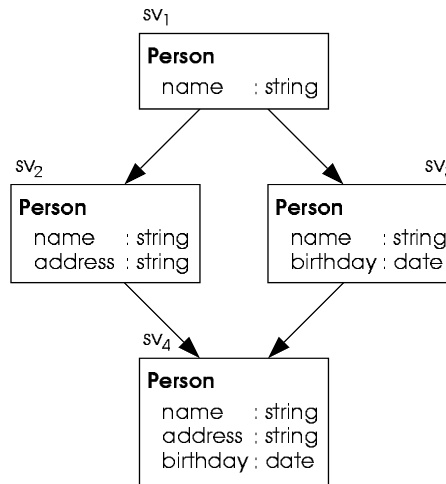


Abbildung 6.17: Beispiel 6.8 ohne zusätzliche Konvertierungsfunktionen.

Speicherung eines Geburtstages haben. Sowohl sv_3.`Person` als auch sv_4.`Person` besitzen jedoch das Attribut `birthday`, d.h. man würde sicher erwarten, daß die in diesem Attribut gespeicherte Semantik zwischen sv_3 und sv_4 propagiert werden kann. Wären wir entgegen der obigen Annahme Alternative (b) gefolgt, so wäre die Propagation der Geburtstages ohne Umstände möglich; nun ergäbe sich allerdings bei der Propagation von Adressinformation zwischen sv_2 und sv_4 ein ähnliches Problem wie oben.

Um das Problem zu lösen erlauben wir in Ergänzung von Vorwärts- und Rückwärtskonvertierungsfunktionen zusätzlich die Implementierung sog. *Extrakonvertierungsfunktionen*, die einen Ersatz mehrerer Vorwärts- und Rückwärtskonvertierungsfunktionen auf dem regulären Konvertierungspfad darstellen können. Diese „Abkürzung“ des Konvertierungspfades kann dazu genutzt werden bei der Konvertierung solche Schemaversionen auszulassen, die weniger Semantik als Quell- und Zielschemaversion beschreiben können. Die Propagation von Objektzuständen und -änderungen in die ausgelassenen Schemaversionen bleibt auf dem regulären Konvertierungspfad weiterhin möglich. Im Gegensatz zu Vorwärts- und Rückwärtskonvertierungsfunktionen werden Extrakonvertierungsfunktionen nicht zwischen direkt benachbarten Schemaversionen spezifiziert. Da Lemma 6.1 insbesondere die Existenz aller benötigter Konvertierungspfade gezeigt hat, ist die zusätzliche Spezifikation von Extrakonvertierungsfunktionen optional.

6.6.2 Formale Beschreibung von Extrakonvertierungsfunktionen

Definition 6.12 {Tiefe einer Klasse, $depth_c(sv)$}
Gegeben sei ein Schemaableitungsgraph $SDG = (sv(s), <^1_{sv})$ und eine Klasse c, die in einer Schemaversion $sv \in sv(s)$ enthalten ist.

Die Tiefe einer Klasse *c in einer Schemaversion sv ist definiert als die Länge des (eindeutigen und existierenden) Pfades von der Erzeugerschemaversion $csv(c)$ der Klasse c nach sv durch den Klassenkonvertierungsbaum CCT_c. Wir sprechen synonym auch von der* Tiefe der Klassenversion *$sv.c$. Die Tiefe einer Klasse c in einer Schemaversion sv wird formal notiert als $depth_c(sv)$.*

Beispiel 6.9 Abbildung 6.18 stellt eine Situation dar, wo $depth_{Person}(sv_3) = 1$ gilt, während gleichzeitig $depth_{Document}(sv_3) = 2$ ist. Dies zeigt, daß die Tiefe zweier Klassen derselben Schemaversion verschieden sein kann, sogar dann wenn beide dieselbe Erzeugerschemaversion ($csv(Document) = csv(Person) = sv_1$) haben. □

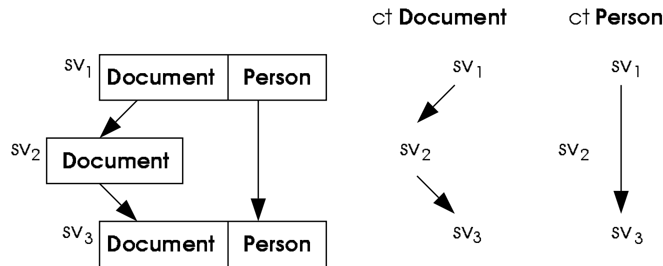


Abbildung 6.18: Tiefe einer Klasse.

Definition 6.13 {Extrakonvertierungsfunktion, xcf}
Gegeben seien die Versionen sv_x, sv_y von Schema s, die beide die Klasse c enthalten, sv_y sei zeitlich nach sv_x abgeleitet worden, und es gelte $depth_c(sv_x) = depth_c(sv_y)$.

Eine Extrakonvertierungsfunktion[107] *von sv_x nach sv_y für eine Klasse c ist ein Programmstück, das eine Objektversion von $sv_x.c$ auf eine Objektversion von $sv_y.c$ abbildet und wird formal ausgedrückt als $xcf_{c,y\leftarrow x}$. Die* Tiefe *einer Extrakonvertierungsfunktion ist definiert als $depth_c(sv_x)$.*

Auch bei der Auswahl und Implementierung hilfreicher Extrakonvertierungsfunktionen kann ein Entwurfswerkzeug wertvolle Unterstützung bieten. Hat der Schemaentwickler die Notwendigkeit einer bestimmten Extrakonvertierungsfunktion zum Ausdruck gebracht, so kann das Schema-Werkzeug automatisch eine Defaultkonvertierungsfunktion anbieten. Desweiteren kann er bei der Ableitung einer neuen Schemaversion darauf aufmerksam machen, wo durch die Definition einer Extrakonvertierungsfunktion möglicherweise zusätzliche Semantik übertragen werden könnte (siehe Abschnitt 7.1.2.2).

Die Lösung für das in Beispiel 6.8 aufgetretene Problem ist, die neue Klasse sv_4.`Person` von sv_1.`Person` abzuleiten, so daß Extrakonvertierungsfunktionen wie in Abbildung 6.19 dargestellt spezifiziert werden können. Die Extrakonvertierungsfunktionen $xcf_{Person,4\leftarrow 2}$ und $xcf_{Person,2\leftarrow 4}$ propagieren Werte der Attribute `name` und `address`, während $xcf_{Person,4\leftarrow 3}$ und $xcf_{Person,3\leftarrow 4}$ Werte von `name` und `birthday` propagieren.[108]

Da Extrakonvertierungsfunktionen in Ergänzung zu Vorwärts- und Rückwärtskonvertierungsfunktionen eingeführt werden, entstehen Mehrdeutigkeiten bezüglich der zu benutzenden Konvertierungspfade. Natürlich wollen wir auch weiterhin wohldefinierte Schemaableitungsgraphen erhalten. Daher müssen wir Einschränkungen bei der Spezifikation von Extrakonvertierungsfunktionen machen und eine Strategie festlegen, unter welchen Umständen welche Extrakonvertierungsfunktionen zu verwenden sind.

6.6.2.1 Eingeschränkte Spezifikation von Extrakonvertierungsfunktionen

Extrakonvertierungsfunktionen für eine Klasse c dürfen nur zwischen einer neu abgeleiteten Schemaversion sv und existierenden Schemaversionen $sv_i \in sv(s)$ spezifiziert werden, wenn die Tiefe der Klasse c in den Schemaversionen identisch ist, d.h. wenn $depth_c(sv) = depth_c(sv_i)$ gilt.

[107] In [Lau97a] hatten wir Vorwärtsextrakonvertierungsfunktionen (engl. *forward extra conversion functions*) und Rückwärtsextrakonvertierungsfunktionen (engl. *backward extra conversion functions*) eingeführt. Diese Unterscheidung entsprechend der Propagationsrichtung basiert allerdings nicht auf einer Vorgänger- oder Nachfolgerbeziehung sondern lediglich auf der durch die verschiedenen Ableitungszeiten festgelegten Reihenfolge zwischen Quell- und Zielschemaversion. Daher können wir hier auf diese Unterscheidung verzichten.

[108] In Abhängigkeit von den sich ergebenden Propagationsflags (siehe Regel 6.4) können auch bei Extrakonvertierungsfunktionen möglicherweise unbeabsichtigte Überschreibungen von Objektmodifikationen analog unserer Beschreibung in Abschnitt 6.3.3.2.1 auftreten. Abschnitt 6.3.3.2.2 beschreibt, wie derartige, unerwünschte Überschreibungen vermieden werden können.

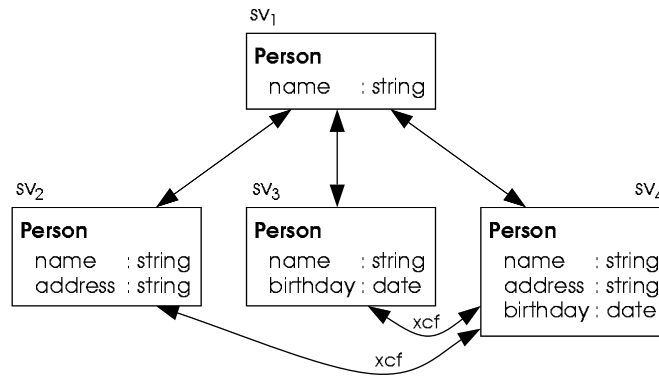


Abbildung 6.19: Beispiel 6.8 unter Zuhilfenahme von Extrakonvertierungsfunktionen.

Um die Notwendigkeit dieser Einschränkung zu motivieren, betrachten wir Abbildung 6.20. Die dort dargestellte Situation wäre ohne die gemachte Einschränkung zulässig und würde beispielsweise die Propagation von sv_3 nach sv_5 gleichermaßen via sv_2 oder via sv_4 ermöglichen. Eine solche Mehrdeutigkeit kann allerdings durch die im anschließenden Abschnitt 6.6.2.2 vorzustellenden Einschränkungen bei der Benutzung vorhandener Extrakonvertierungsfunktionen nicht aufgelöst werden. Die hier getroffene Einschränkung verhindert derartige Mehrdeutigkeiten.

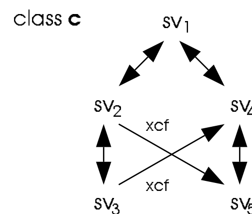


Abbildung 6.20: Extrakonvertierungsfunktionen, deren Quell- und Zielklassenversionen verschiedene Tiefen haben, sind nicht erlaubt.

Die geschilderte Einschränkung ist in der Praxis nicht besonders groß, da die Tiefe jeweils klassenspezifisch definiert ist (siehe Beispiel 6.9). In Abschnitt 6.7 werden wir andeuten, wie diese Beschränkung im Notfall auch umgangen werden kann.

6.6.2.2 Eingeschränkte Benutzung von Extrakonvertierungsfunktionen

Extrakonvertierungsfunktionen werden nur dann benutzt, wenn die beiden folgenden Bedingungen gelten.

- Der reguläre Konvertierungspfad von sv_{source} nach sv_{drain} enthält einen Vorwärts- und einen Rückwärtsast, d.h. es gilt $sv_{source} \not\leq_{sv} sv_{drain}$ (dies wäre ein reiner Rückwärtskonvertierungspfad) und $sv_{drain} \not\leq_{sv} sv_{source}$ (dies wäre ein reiner Vorwärtskonvertierungspfad).

- Höchstens eine Extrakonvertierungsfunktion wird auf einem Konvertierungspfad genutzt und diese muß direkt vom Rückwärtsast zum Vorwärtsast des regulären Konvertierungspfades führen. In dem in Abbildung 6.21 dargestellten Schemaableitungsgraphen würde zur Propagation von sv_2 nach sv_4 der reguläre Konvertierungspfad via sv_1 genutzt, weil keine Extrakonvertierungsfunktion existiert, die direkt von sv_2 nach sv_4 führt. Ansonsten könnten Mehrdeutigkeiten bei der Bestimmung des Konvertierungspfades auftreten. Abbildung 6.22 zeigt ein Beispiel einer solchen Situation. Der Konvertierungspfad von sv_2 nach

sv_5 wäre ohne die hier eingeführte Einschränkung, nur eine Extrakonvertierungsfunktion auf einem Konvertierungspfad zu nutzen, mehrdeutig (via sv_3 oder via sv_4).

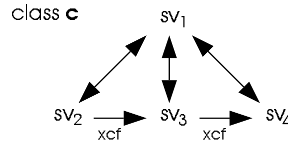


Abbildung 6.21: Die Komposition mehrerer Extrakonvertierungsfunktionen ist nicht erlaubt.

Wenn in verschiedenen Tiefen ($0 < i \leq \min(depth_c(sv_{source}), depth_c(sv_{drain}))$) Extrakonvertierungsfunktionen $xcf_{c,y_i \leftarrow x_i}$ existieren, benutzen wir diejenige maximaler Tiefe, d.h. diejenige mit maximalem i, da sie die meisten Rückwärts- und Vorwärtskonvertierungsfunktionen ersetzt und damit den kürzestmöglichen Konvertierungspfad ergibt. An dieser Stelle wird deutlich, warum die im vorangegangenen Abschnitt 6.6.2.1 gemachte Einschränkung bezüglich der Tiefe von Quell- und Zielklassenversion einer Extrakonvertierungsfunktion notwendig war. Ohne sie könnten wir hier nämlich keine eindeutige Auswahl treffen.[109]

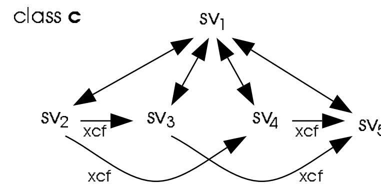


Abbildung 6.22: Ein Beispiel eines Klassenkonvertierungsgraphen.

Sind in einem Schema s Extrakonvertierungsfunktionen für eine Klasse c spezifiziert, so ist die Konvertierungsstruktur dieser Klasse kein Baum mehr. Wir definieren daher nun den *Klassenkonvertierungsgraphen* in Verallgemeinerung des Klassenkonvertierungsbaumes aus Definition 6.6.

Definition 6.14 {Klassenkonvertierungsgraph, CCG}
Gegeben sei ein Schemaableitungsgraph $SDG = (svids, <^1_{sv})$.

Ein Klassenkonvertierungsgraph CCG über einer Menge von Klassenversionen $cvids$ (engl. class conversion graph) *ist ein gerichteter (möglicherweise zyklischer) Graph, dessen Knoten die Klassenversionen aus cvids repräsentieren und dessen Kanten die Konvertierungsbeziehung zwischen*

[109] Theoretisch wäre es allerdings auch ohne die genannte Einschränkung möglich, eine eindeutige Auswahl zu treffen. Die beiden in Abbildung 6.20 dargestellten Extrakonvertierungsfunktionen könnten damit erlaubt werden. Für die eindeutige Bestimmung des Konvertierungspfades könnte man aus der Menge der vom Rückwärtsast zum Vorwärtsast führenden Konvertierungsfunktionen zunächst diejenigen in eine Teilmenge selektieren, die eine maximale Verkürzung des regulären Konvertierungspfades erreichen. Diese Teilmenge enthält dann ggf. noch mehrere, alternative Extrakonvertierungsfunktionen (in Abbildung 6.20 wären dies $xcf_{c,4\leftarrow3}$ und $xcf_{c,5\leftarrow2}$). Um nun eine eindeutige Auswahl aus der vorselektierten Teilmenge zu erreichen, könnte man die Schemaversionen entlang des regulären Konvertierungspfades betrachten und beispielsweise die erste abgehende Extrakonvertierungsfunktion aus der betrachteten Teilmenge benutzen. In Abbildung 6.20 würde demzufolge von sv_3 nach sv_5 entlang des Konvertierungspfades $cf_{c,5\leftarrow3} = fcf_{c,5\leftarrow4} \circ xcf_{c,4\leftarrow3}$ verwendet werden.

Das beschriebene oder ähnliche Verfahren wären für den Schemaentwickler in der Praxis schwer nachvollziehbar. Daher haben wir uns hier für etwas stärkere Einschränkungen zugunsten eines einfacheren Verfahrens entschieden.

Das allgemeine Problem der Suche eines eindeutigen Pfades durch einen beliebigen Graphen erleichtert sich in der hier betrachteten Aufgabenstellung durch zwei spezielle Umstände: Zum einen können wir von dem bereits eindeutig vorliegenden regulären Konvertierungspfad ausgehen und zum zweiten haben wir uns auf die Nutzung maximal einer Extrakonvertierungsfunktion je Konvertierungspfad beschränkt.

diesen Klassenversionen widerspiegeln. Eine Kante von Klassenversion cv_v nach Klassenversion c_u ($c_u, c_v \in cvids$) bedeute dabei, daß eine Konvertierungsfunktion von cv_v nach cv_u existiert.

Formal wird ein Klassenkonvertierungsgraph dargestellt als ein Paar $CCG = (cvids, <^1_{cc})$ mit den folgenden Komponenten:

- *$cvids \subseteq cId$ ist die Menge der Identifikatoren derjenigen Klassenversionen, die zu dem Klassenkonvertierungsgraphen gehören.*

  Es handelt sich bei den cvids ausschließlich um Versionen derselben Klasse c, d.h. $cvids = \{sv_1.c, \ldots, sv_n.c\}$.

- *$\leq_{cc} \subseteq cvids^2$ ist eine Ordnungsrelation auf cvids entsprechend Definition 2.3.*

Da die systemdefinierte Klasse **Object** *keine direkten Instanzen hat, benötigen wir für sie keinen Klassenkonvertierungsgraphen.*

Entsprechend Definition 2.4 verwenden wir auch die Symbole $<_{cc}$, $\leq_{cc}$, $\leq^1_{cc}$, $>_{cc}$, $\geq_{cc}$, $>^1_{cc}$ und $\geq^1_{cc}$ zur Darstellung der Konvertierungsbeziehung.

Hierbei ist zu beachten, daß die Kanten eines Klassenkonvertierungsgraphen nicht notwendigerweise mit Kanten im Schemaableitungsgraphen korrespondieren müssen, d.h. die Quellklassenversion der Vorwärtskonvertierung zu einer Klassenversion cv und die Zielklassenversion der Rückwärtskonvertierung von cv müssen nicht notwendigerweise mit derjenigen Klassenversion übereinstimmen, von der cv integriert wurde.

Der Klassenkonvertierungsgraph unterscheidet sich damit deutlicher vom Klassenableitungsbaum als der Klassenkonvertierungsbaum.

6.6.3 Bestimmung des Konvertierungspfades

Regel 6.3 {Definieren von Extrakonvertierungsfunktionen bei Schemaableitung}
Für jede abgeleitete Klasse c einer neuen Schemaversion $sv_{(n+1)}$ und für jede existierende Schemaversion $sv_{x_i} \in sv(s)$ mit $depth_c(sv_{x_i}) = depth_c(sv_{(n+1)})$ dürfen Extrakonvertierungsfunktionen $xcf_{c,(n+1)\leftarrow x_i}$ und $xcf_{c,x_i\leftarrow(n+1)}$ spezifiziert werden.

Wie sich in Lemma 6.2 zeigen wird, sind die hier gemachten Einschränkungen bezüglich der Definition (siehe Abschnitt 6.6.2.1) und Benutzung (siehe Abschnitt 6.6.2.2) von Extrakonvertierungsfunktionen hinreichend, um die Wohldefiniertheit des sich ergebenden Schemaableitungsgraphen zu garantieren. Für viele Einschränkungen haben wir in den vorangegangenen Beispielen auch die Notwendigkeit gezeigt.

Definition 6.15 {Tiefe eines Konvertierungspfades, $level_c$}
Die Tiefe *eines Konvertierungspfades $level_c(sv_{source}, sv_{drain})$ für eine Klasse c von sv_{source} nach sv_{drain} ist definiert als die maximale Tiefe einer existierenden Extrakonvertierungsfunktion für die Klasse c von einem Vorgänger von sv_{source} zu einem Vorgänger von sv_{drain} im Klassenkonvertierungsgraphen der Klasse c. Falls keine solche Extrakonvertierungsfunktion existiert, definieren wir $level_c(sv_{source}, sv_{drain}) := depth_c(scp_c(sv_{source}, sv_{drain}))$.*

Beispiel 6.10 In Abbildung 6.19 ist die Tiefe eines Konvertierungspfades für die Klasse `Person` von sv_2 nach sv_4 gegeben durch $level_{Person} = depth(xcf_{Person,4\leftarrow 2}) = 1$. □

Um Objekte einer Klasse c von sv_{source} nach sv_{drain}[110] zu propagieren, benutzen wir den folgenden Konvertierungspfad ($lvl := level_c(sv_{source}, sv_{drain})$):

[110] Wir benutzen hier weiterhin die in Abschnitt 6.5.1 eingeführte Notation.

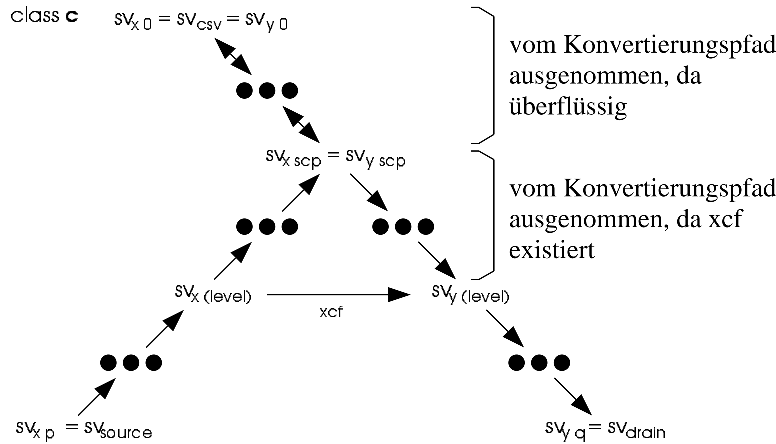


Abbildung 6.23: Ein Konvertierungspfad mit einer Extrakonvertierungsfunktion.

$$
\begin{aligned}
cf_{c,drain \leftarrow source} &= fcf \circ xcf \circ bcf \\
fcf &= fcf_{c,drain \leftarrow y_{(q-1)}} \circ \cdots \circ fcf_{c,y_{(lvl+1)} \leftarrow y_{lvl}} \\
xcf &= xcf_{c,y_{lvl} \leftarrow x_{lvl}} \\
bcf &= bcf_{c,x_{lvl} \leftarrow x_{(lvl+1)}} \circ \cdots \circ bcf_{c,x_{(p-1)} \leftarrow source}
\end{aligned}
$$

Wie aus Abbildung 6.23 ersichtlich ist, verkürzt die Benutzung einer Extrakonvertierungsfunktion die Länge des jeweiligen Konvertierungspfades. Dabei werden zwar einige Schemaversionen auf dem regulären Pfad ausgelassen, es kommen jedoch keine neuen Schemaversionen hinzu. Damit werden Probleme mit fehlenden Konvertierungsfunktionen oder mehrdeutigen Propagationsflags vermieden.

Regel 6.4 {Propagationsflags von Extrakonvertierungsfunktionen}
Die einer Extrakonvertierungsfunktion $xcf_{c,y \leftarrow x}$ *zugeordneten Propagationsflags werden durch Komposition der Propagationsflags der Klasse c auf dem regulären Konvertierungspfad von* $sv_x.c$ *nach* $sv_y.c$ *berechnet. Die Komposition der als boolsche Werte aufgefaßten Flags geschieht durch den Konjunktionsoperator. Für ein Flag* $f \in \{c, m, d\}$ *seien* $f_1, f_2 \in \{f, \overline{f}\}$*. Dann gilt* $f_1 \circ f_2 = f_1 \wedge f_2$*.*

Beispiel 6.11 Gegeben sei der in Abbildung 6.24 dargestellte Schemaableitungsgraph. Hier könnte zunächst der Eindruck entstehen, es bestünden zwei Alternativen für die Propagation von sv_4 nach sv_3 unter Benutzung von Extrakonvertierungsfunktionen: Zum einen könnte mit $xcf_{c,3 \leftarrow 2}$ via sv_2 propagiert werden, zum anderen mit $xcf_{c,5 \leftarrow 4}$ via sv_5. Wir hatten uns jedoch für die Benutzung des ersteren Konvertierungspfades entschieden, da die zweite Alternative zahlreiche Schwierigkeiten verursachen könnte. Dies liegt darin begründet, daß der Konvertierungspfad der zweiten Alternative mit sv_5 eine Schemaversion enthält, die nicht auf dem regulären Konvertierungspfad von sv_4 nach sv_3 liegt. Damit könnten sich die auf dem Konvertierungspfad via sv_5 spezifizierten Propagationsflags von denjenigen unterscheiden, die als Komposition der Flags auf dem Pfad via sv_2 für die Propagation von sv_4 nach sv_3 entsprechend Regel 6.4 berechnet werden. Desweiteren wurde Schemaversion sv_5 erst nach den Schemaversionen auf dem regulären Konvertierungspfad abgeleitet. Damit war bis zur Ableitungszeit von sv_5 nur ein eindeutiger Propagationspfad — nämlich der via sv_2 — gegeben, auf dem eventuell bereits Objekte aus dem Zugriffsbereich von sv_4 nach sv_3 propagiert wurden. Die Benutzung der zweiten vorgestellten Alternative würde damit bedeuten, daß Objekte von $sv_4.c$ vor und nach der Ableitungszeit von

sv_5 auf verschiedenen Konvertierungspfaden in den Zugriffsbereich von $sv_3.c$ propagiert würden, was sicher überraschende und nicht nachvollziehbare Resultate im Zugriffsbereich von sv_3 ergeben würde. Sogar die durch eine Objektmodifikation im Zugriffsbereich von sv_4 ausgelöste wiederholte Propagation eines Objektes nach sv_3 würde nach der Ableitung von sv_5 entlang eines anderen Propagationspfades erfolgen als bei den vorangegangenen Propagationen desselben Objektes. Auf diesem Wege würde eventuell ein und dasselbe Objekt vor und nach der Ableitung von sv_5 mit komplett verschiedenen Semantiken von sv_4 nach sv_3 propagiert.

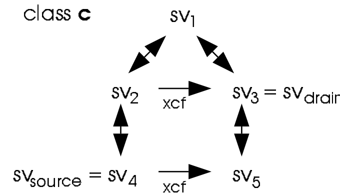


Abbildung 6.24: Auswahl eines Konvertierungspfades mit einer Extrakonvertierungsfunktion.

Unserer Strategie folgend nehmen wir die Propagation von sv_4 nach sv_3 entlang des Konvertierungspfades via sv_2 vor und vermeiden damit die genannten Probleme. Insbesondere werden alle Propagationen von $sv_4.c$ nach $sv_3.c$ zu allen Zeiten entlang desselben Konvertierungspfades vorgenommen. Man beachte hier und i.Allg., daß alle Schemaversionen auf dem regulären Konvertierungspfad ab dem Beginn der Propagation von der Quell- zur Zielschemaversion eingefroren sind und sich das Angebot potentiell benutzbarer Konvertierungsfunktionen und Extrakonvertierungsfunktionen damit nicht mehr nachträglich ändern kann.

Für die Propagation von sv_4 nach sv_5 wird jedoch sehr wohl die Extrakonvertierungsfunktion $xcf_{c,5\leftarrow 4}$ verwendet. Hier sind Schwierigkeiten der oben genannten Art jedoch auch ausgeschlossen, da sv_5 in diesem Fall (anders als oben) auf dem regulären Propagationspfad von $sv_{source} = sv_4$ nach $sv_{drain} = sv_5$ liegt. $\square$

6.6.4 Eindeutigkeit des Konvertierungspfades

Lemma 6.2 (Wohldefiniertheit der Konvertierung)
Wenn jede Schemaversion $sv \in sv(s)$ entsprechend der Regeln 6.1, 6.2, 6.3 und 6.4 abgeleitet wurde, dann ist der resultierende Schemaableitungsgraph $SDG = (sv(s), <^1_{sv})$ wohldefiniert (bezüglich seiner Konvertierungsfunktionen).

Beweis: Wir beweisen Lemma 6.2 durch vollständige Induktion auf der Menge $sv(c)$ aller Schemaversionen des Schemaableitungsgraphen, die die Klasse c enthalten, in der Reihenfolge der zeitlichen Ableitung der Schemaversionen.

Induktionsanfang ($\{sv_0\}$)
Lemma 6.2 gilt trivialerweise für $\{sv_0\}$.

Induktionsschritt ($\{sv_0, \ldots, sv_n\} \rightarrow \{sv_0, \ldots, sv_{(n+1)}\}$)
Nach Induktionsannahme gelte Lemma 6.2 nun für $\{sv_0, \ldots, sv_n\}$. Wir wählen eine beliebige abgeleitete Klasse c aus $sv_{(n+1)}$.

Existenz: Durch Hinzufügen von Extrakonvertierungsfunktionen wird die Existenz eines Konvertierungspfades nicht beeinträchtigt.

Eindeutigkeit: Wir betrachten Konvertierungen von $sv_x.c$ nach $sv_y.c$.

Fall 1: $sv_x, sv_y \in \{sv_0, \ldots, sv_n\}$
Da nach Lemma 6.3 keine Extrakonvertierungsfunktionen zwischen bereits existierenden Schemaversionen aus $\{sv_0, \ldots, sv_n\}$ hinzugefügt werden dürfen, ist die Eindeutigkeit aufgrund der Induktionsannahme garantiert.

Fall 2: $sv_x = sv_{(n+1)}$, $sv_y \in \{sv_0, \ldots, sv_n\}$
Wenn ein $i \in \{1, \ldots, n\}$ existiert und eine Extrakonvertierungsfunktion $xcf_{c,y_i \leftarrow x}$ so daß $sv_y <_{cc} sv_{y_i}$ gilt, dann benutzen wir $xcf_{c,y_i \leftarrow x}$.
Regel 6.3 garantiert, daß die ausgewählte Extrakonvertierungsfunktion eindeutig ist, weil kein $j \neq i$ existieren kann, so daß $\exists\ xcf_{c,y_i \leftarrow x}$, $sv <_{cc} sv_{y_i}$ und $\exists\ xcf_{c,y_j \leftarrow x}$, $sv <_{cc} sv_{y_j}$.
Ansonsten benutzen wir $bcf_{c,pred \leftarrow x}$ (mit $sv_x <^1_{cc} sv_{pred}$).
Damit ist der erste Schritt auf dem Konvertierungspfad von $sv_x.c$ nach $sv_y.c$ eindeutig. Der Rest des Konvertierungspfades von sv_x nach $sv_y \in sv(s)$ beinhaltet lediglich Vorwärts- und Rückwärtskonvertierungsfunktionen innerhalb von $sv(s)$ und ist aufgrund der Induktionsannahme eindeutig.

Fall 3: $sv_x \in sv(s)$, $sv_y = sv_{(n+1)}$
Analog zu Fall 2.

$\Rightarrow$ Lemma 6.2 gilt auch für $\{sv_0, \ldots, sv_{(n+1)}\} = \{sv_0, \ldots, sv_n\} \cup \{sv_{(n+1)}\}$. $\Box$

6.7 Entwurfsunterstützung bei der Ableitung neuer Schemaversionen

In einem Schemaversionierungsmodell wie es beispielsweise in [KC88] vorgestellt wurde, kann eine neue Schemaversion nur von einer direkten Vorgängerschemaversion abgeleitet werden, wodurch die Möglichkeiten zur Propagation von Objekten erheblich eingeschränkt werden. Wir haben in den Abschnitten 5.5.3.1 und 6.6 zwei weitere Konzepte vorgestellt: zum einen die Integration von Schemaversionen und zum anderen die Benutzung von Extrakonvertierungsfunktionen.

Das Integrationskonzept gibt uns die Möglichkeit, die am besten passende Implementierung jeder Klasse aus verschiedenen Schemaversionen auszuwählen, d.h. anstelle eines Ableitungsbaumes für Schemaversionen erhalten wir einen Schemaableitungsgraphen, der als Vereinigung der Ableitungsbäume aller Klassen eines Schemas angesehen werden kann. Damit kann die Propagation von Objekten klassenspezifisch vorgenommen werden.

Durch die Benutzung von Extrakonvertierungsfunktionen werden die Möglichkeiten der Objektpropagation erweitert, ohne daß die Wohldefiniertheit des Schemaableitungsgraphen verloren geht.

Bei der Ableitung neuer Schemaversionen kann jedoch die Frage entstehen, welches der angebotenen Konzepte idealerweise zu nutzen sei. Anders ausgedrückt muß entschieden werden, welche Version einer Klasse bei der Ableitung einer neuen Schemaversion zu integrieren sei. Einerseits sollte die integrierte Klassenversion den Anforderungen in der neuen Schemaversion so ähnlich wie möglich sein, um die Anzahl der notwendigen Anwendungen der Schemaänderungsprimitive zu minimieren. Andererseits kann es in besonderen Fällen (wie etwa in Beispiel 6.8) vorteilhaft sein, Extrakonvertierungsfunktionen zu spezifizieren.

Als Daumenregel kann man grob sagen, daß Extrakonvertierungsfunktionen soweit als möglich vermieden werden sollten, da sie die Komplexität des Konvertierungsgraphen erhöhen und weiterhin voraussetzen, daß sich Quell- und Zielschemaversion in derselben Tiefe befinden. Daher wird eine neue Klassenversion nicht als Nachfolger von derjenigen Klassenversion, die ihr

am ähnlichsten ist, abgeleitet, sondern als Geschwisterversion. Dies verkompliziert jedoch den Schemaableitungsprozeß und zerstört zumindest teilweise die klare Ableitungssemantik zwischen den Versionen einer Klasse, weil die Tatsache, daß eine Klassenversion $sv_v.c$ von $sv_u.c$ abgeleitet wurde, nicht länger impliziert, daß $sv_v.c$ und $sv_u.c$ sehr ähnliche Versionen einer Klasse sind. Stattdessen könnte es bedeuten, daß $sv_v.c$ einer von $sv_u.c$ abgeleiteten Klassenversion ähnlich ist.

In Beispiel 6.8 könnte sv_4.Person sowohl von sv_2.Person als auch von sv_3.Person leichter abgeleitet werden als von sv_1.Person. Da Definition 6.6 erfordert, daß $<^1_{cc} \subseteq <^1_{sv}$ können wir die Klassenversion sv_4.Person nicht von sv_2.Person oder von sv_3.Person ableiten und anschließend sv_4 zu einer Geschwisterversion von sv_2 und sv_3 im Klassenkonvertierungsgraphen $CCG_{SD,Person} = (SV, <^1_{SV,\texttt{Person}})$ machen. Aber wir können sv_4.Person dennoch mit einem Trick von sv_2.Person oder von sv_3.Person ableiten: Wir integrieren bei der Ableitung der neuen Schemaversion sv_4 zunächst irgend eine beliebige Klasse von sv_1 in sv_4. Damit teilen wir dem System mit, daß sv_4 eine direkte Nachfolgerschemaversion von sv_1 ist ($sv_4 <^1_{sv} sv_1$) (zusätzlich zu $sv_4 <^1_{sv} sv_2$ oder $sv_4 <^1_{sv} sv_3$). Damit können wir Vorwärts- und Rückwärtskonvertierungsfunktionen zwischen sv_1 und sv_4 für die Klasse Person spezifizieren. Damit wird sv_4 eine Geschwisterversion von sv_2 und sv_3 im Klassenkonvertierungsgraphen $CCG_{SD,\texttt{Person}}$ was uns schließlich die Definition der gewünschten Extrakonvertierungsfunktionen gestattet. Aber dieser Trick ist natürlich eine sehr unschöne Lösung, die unbedingt vermieden werden sollte.

In der Situation von Beispiel 6.8 (siehe Abbildung 6.19) ist die Ableitung von sv_4.Person von sv_1.Person durch zweimalige Anwendung des **add attribute** Schemaänderungsprimitives dringend angeraten.

Wenn die Menge der Attribute einer neuen Klassenversion $sv_v.c$ eine Teilmenge der Attribute einer existierenden Klassenversion $sv_u.c$ ist, dann kann $sv_v.c$ natürlich von sv_u abgeleitet werden. Wenn die Attributmengen verschiedener Versionen einer Klasse jedoch überlappen (wie es in Beispiel 6.8 der Fall ist) kann die Verwendung von Extrakonvertierungsfunktionen unumgänglich sein.

Weiterhin kann, bei der Entscheidung von welcher Version einer Klasse abzuleiten sei, ggf. vorhandenes Wissen über die Zugriffsbereiche verschiedener Schemaversionen berücksichtigt werden. Existiert etwa nur eine Version einer Klasse, in der tatsächlich Objekte erzeugt werden (beispielsweise wenn nur eine Applikation existiert, die Personen registriert), so kann die Propagation von dieser Klassenversion in die neu abzuleitende Schemaversion geprüft werden, um die gewünschte Funktionsweise des Mechanismus zu gewährleisten.

6.8 Löschen von Schemaversionen und deren Zugriffsbereichen

Das Primitiv **delete schemaversion** zum Löschen von Schemaversionen haben wir bereits in Abschnitt 5.5.2.3 eingeführt. Dabei hatten wir, wie im gesamten Kapitel 5, zunächst nur die Schemaebene betrachtet. Auf der Basis der in Kapitel 6 erarbeiteten Konzepte untersuchen wir nun die Auswirkungen des Primitives auf Objektebene.

Das Löschen einer Version sv eines Schemas s dient zunächst der Bereinigung von s um nicht mehr benötigte Schemaversionen. Damit wird die Aktualität und die Übersichtlichkeit des gesamten Schemas bewahrt. Auf Objektebene sollte darüber hinaus der belegte Speicherplatz freigegeben werden, indem der Zugriffsbereich von sv und die darin enthaltenen Objektversionen gelöscht werden.

In Abschnitt 5.5.2.3 hatten wir gefordert, daß eine Schemaversion nur dann gelöscht werden darf, wenn sie keine Nachfolgerschemaversionen im Schemaableitungsgraphen hat. Durch diese

Forderung werden einerseits zwar Abhängigkeiten anderer Schemaversionen[111] von *sv* vermieden, was die Realisierung des Primitives erheblich vereinfacht. Andererseits wird sein praktischer Nutzen dadurch deutlich eingeschränkt.

Soll das Primitiv **delete schemaversion** derart verallgemeinert werden, daß beliebige Schemaversionen damit gelöscht werden können, sind einige Aspekte näher zu untersuchen, auf die wir im folgenden kurz eingehen. Ungeachtet dieser Aspekte läßt sich eine Realisierung des Primitives natürlich dadurch erreichen, daß zu löschende Schemaversionen für Schema- und Applikationsentwickler unsichtbar gemacht werden. Obwohl eine Schemaversion dann von außerhalb des Datenbanksystems nicht mehr zugreifbar ist, würde sie intern wie jede andere Schemaversion weitergeführt und auch ihr Zugriffsbereich bliebe physikalisch bestehen und würde bei Propagationen ggf. sogar noch aktualisiert werden. Diese Lösung wäre zwar sehr einfach, hätte aber mehrere erhebliche Nachteile. Konzeptionell wäre der Schemaableitungsgraph nicht mehr zusammenhängend[112] und die zur Erzeugung einer Schemaversion durchgeführten Änderungen wären in einer generierten ODL-Datei nicht mehr vollständig enthalten, da die zu *sv* führenden Schritte fehlten. Physikalisch schließlich könnte die von einer Löschung erwartete Ersparnis an Speicherplatz und Propagationszeit nicht erbracht werden.

Wir gehen nun auf die Aspekte ein, die für eine echte Löschung einer inneren Schemaversion *sv* des Schemaableitungsgraphen zu berücksichtigen sind. Dabei sei sv_u eine direkte Vorgänger- und sv_v eine direkte Nachfolgerschemaversion von *sv*.

- Wie bereits erwähnt, geht der Zusammenhang des Schemaableitungsgraphen beim Löschen innerer Knoten zunächst verloren und muß wieder hergestellt werden, beispielsweise indem die direkten Vorgängerschemaversionen von *sv* zu direkten Vorgängerschemaversionen aller Nachfolgerschemaversionen von *sv* gemacht werden. Dabei ist die mit den Ableitungskanten verknüpfte Semantik zu berücksichtigen, d.h. die zur Erzeugung einer Nachfolgerschemaversion sv_v durchgeführten Schemaänderungen müssen vom Datenbanksystem angegeben werden können. Die Konkatenation der zur Erzeugung von *sv* und von sv_v ursprünglich verwendeten Primitive würde sich als ein erster, ggf. noch zu überarbeitender Ansatz anbieten.

- Entsprechend der sich neu ergebenden Integrationsbeziehungen bzw. der Klassenableitungsbäume müßten die Konvertierungsfunktionen und die Propagationsflags zwischen ehemaligen direkten Vorgängern und Nachfolgern von *sv* bestimmt werden. Dabei können wir uns auf die Betrachtung solcher Klassen *c* beschränken, die sv_v von *sv* integriert hat und *sv* von sv_u. Auch hierbei wäre eine Form der Konkatenation zu verwenden. Eine automatisch konkatenierte Konvertierungsfunktion könnte Hilfsvariablen verwenden, die genau den Attributen der jeweils betrachteten Klasse in Version *sv.c* entsprechen und die als Ersatz für eine nicht mehr vorhandene Objektversion von *sv.c* das temporäre Zwischenergebnis zwischen den beiden zu konkatenierenden Konvertierungsfunktionen aufnehmen könnten. Die sich ergebenden Konvertierungsfunktionen sollten anschließend allerdings noch vereinfacht werden.

- Durch die erforderlichen Anpassungen der Klassenableitungsbäume und der Klassenkonvertierungsgraphen würde sich in den Nachfolgerschemaversionen von *sv* die Tiefe der von

[111] Applikationen von *sv* hängen natürlich ebenfalls von der Existenz dieser Schemaversion ab. Sie müssen, um nach der Löschung von *sv* wieder ausgeführt werden zu können, zunächst an eine andere Schemaversion angepaßt werden (siehe Abschnitt 5.4.3).

[112] Dieser Nachteil ließe sich dadurch beheben, daß man die Vorgängerschemaversionen der gelöschten Schemaversion *sv* nun als Vorgänger der ehemaligen Nachfolgerschemaversionen von *sv* darstellt, ähnlich der Option **reconnect** beim Löschen von inneren Klassen eines Vererbungsgraphen (siehe dazu den Hinweis auf [Brè96] in Abschnitt 5.5.3.4).

sv integrierten Klassen ändern. Aufgrund der in Definition 6.13 geforderten und in Abschnitt 6.6.2.1 erläuterten Einschränkung könnten Extrakonvertierungsfunktionen von und zu allen Nachfolgerschemaversionen von sv unzulässig werden. Da ein derartiger, nachträglicher Wegfall von Konvertierungsfunktionen nicht hingenommen werden kann, wären die Einschränkungen bezüglich der Spezifikation von Extrakonvertierungsfunktionen entsprechend aufzuweichen, was das Verständnis des Mechanismus erschweren würde.

- Aufgrund der physikalisch verzögert wirkenden Propagation sind Änderungen von Instanzen im Zugriffsbereich von sv nicht jederzeit vollständig an betroffene Vorgänger- und Nachfolgerschemaversionen weitergegeben. Bevor die Instanzen aus dem Zugriffsbereich von sv gelöscht werden können, müssen daher noch nicht durchgeführte Propagationsschritte nachgeholt werden. Dazu kann die in Abschnitt 7.2.3.3.3 vorzustellende Methode **PMpropagate** verwendet werden.
- In Abschnitt 6.5.3 hatten wir auf Löcher in Konvertierungspfaden und auf ihre Auswirkungen auf die Propagation hingewiesen. Stellte sv für ein Objekt o ein solches Loch dar und ist o in den Zugriffsbereichen von sv_u und sv_v enthalten, so muß dafür gesorgt werden, daß o nach dem Löschen von sv nicht wieder zwischen sv_u und sv_v propagiert werden kann. Dies könnte prinzipiell geschehen, da das Loch durch die Löschung verschwunden ist. Die Maßnahmen, die notwendig sind, um die neuerliche Propagation zu verhindern, würden zumindest einen erheblichen Suchaufwand in der Datenbank verursachen.

Aufgrund der Komplexität einiger der geschilderten Aspekte haben wir auf die Löschung von inneren Knoten des Schemaableitungsgraphen verzichtet.

6.9 Zusammenfassung und Bewertung

In diesem Kapitel haben wir den Ansatz der Schemaversionierung auf der Ebene der Datenbank betrachtet und damit den Schemaänderungsprozeß vervollständigt. Dabei haben wir zunächst die Objektzugriffsbereiche der verschiedenen Schemaversionen und damit den für Applikationen einer Schemaversion sichtbaren Ausschnitt der Datenbank definiert. Die Kooperation zwischen Applikationen verschiedener Schemaversionen ist dabei genau auf denjenigen Objekten möglich, die in den Zugriffsbereichen mehrerer Schemaversionen enthalten sind. Da die Klasse eines Objektes in verschiedenen Schemaversionen verschiedene Typen haben kann, mußten wir das Konzept der Objektversionierung so erweitern, daß auch verschiedene Versionen desselben Objektes verschiedene Typen aufweisen können. Ein Objekt enthält damit für jede Schemaversion, in der es sichtbar ist, genau eine Objektversion und diese hat den erforderlichen, von der entsprechenden Klassenversion spezifizierten Typ. Die Abbildung zwischen den verschiedenen Objektversionen werden durch klassenspezifische Konvertierungsfunktionen beschrieben. Für die Steuerung der Propagation stehen vier Propagationsflags zur Verfügung, die die Weitergabe existierender Objekte zum Ableitungszeitpunkt neuer Schemaversionen sowie die Weitergabe späterer Objekterzeugungen, -modifikationen und -löschungen regeln. Für die Angabe der Konvertierungsfunktionen und der Propagationsflags wurde eine Propagationssprache in die COAST-ODL eingebettet.

Die Durchführung eines Propagationsschrittes führt zu einer Veränderung im Zugriffsbereich der Zielschemaversion und diese wird, in Abhängigkeit von den gesetzten Flags, selbst wieder propagiert, was uns zum Konzept der transitiven Propagation führte. Durch die Verwendung von Klassenableitungsbäumen haben wir wohldefinierte Propagationspfade erreicht, ohne die Möglichkeit der Integration verschiedener Schemaversionen aufzugeben. Wie wir festgestellt haben, kann insbesondere bei der Propagation über lange Konvertierungspfade Semantik der

propagierten Objekte verloren gehen. Um dieses Problem einzuschränken, haben wir die zuvor eingeführten Vorwärts- und Rückwärtskonvertierungsfunktionen, welche stets entlang der Ableitungsbeziehungen verlaufen, um Extrakonvertierungsfunktionen ergänzt. Diese sind optional, verlaufen horizontal in den Klassenableitungsbäumen und können unter gewissen Voraussetzungen in Ergänzung der Vorwärts- und Rückwärtskonvertierungsfunktionen spezifiziert und benutzt werden. Dabei ist jedoch auf die Eindeutigkeit und Unveränderlichkeit von Konvertierungspfaden zu achten. Mit der Einführung der Extrakonvertierungsfunktionen haben wir einen Mittelweg beschritten zwischen dem Vorteil der einfachen Spezifikation bei Verwendung einer minimalen Menge von Konvertierungsfunktionen und dem Vorteil der maximalen Erhaltung von Objektsemantik bei der Spezifikation einer kombinatorischen Vielfalt von Konvertierungsfunktionen von jeder Schemaversion zu jeder anderen. Abschließend hatten wir einige Hinweise zur Verwendung von Extrakonvertierungsfunktionen beim Entwurf neuer Schemaversionen gegeben.

Um die Allgemeinheit und Flexibilität unserer Propagationskonzepte zu belegen, deuten wir abschließend einige Erweiterungsmöglichkeiten an, die sich gut in das Grundgerüst integrieren lassen.

- Die automatische Erzeugung von Defaultkonvertierungsfunktionen erreicht eine erhebliche Verringerung des Spezifikationsaufwandes. Alternativ dazu oder ergänzend könnte ein Vererbungsmechanismus für Konvertierungsfunktionen entworfen werden, so daß bei gemeinsamem Integrieren einer Klasse mit ihren Oberklassen die Umsetzung geerbter Attribute nicht erneut spezifiziert werden muß.

- Die Steuerung der Propagation könnte, wenn auch nur unter Anstieg der Komplexität, feiner granuliert werden. Das COAST-Objektmodell bietet Containertypen (Menge und Liste) zur Beschreibung entsprechend strukturierter Instanzen der Diskurswelt. Wird nun beispielsweise ein mengenwertiges Attribut durch Einfügen eines neuen Elementes verändert, so resultiert dies im gegenwärtigen Modell in einer erneuten Propagation des gesamten Objektes. Alternativ dazu mag es manchmal angebracht sein, nur die einzelne Operation zu propagieren und etwa ein in der Quellobjektversion eingefügtes Element auch in der Zielobjektversion einzufügen, anstatt die gesamte Menge zu propagieren. Dazu wäre eine feinere Unterscheidung möglicher Änderungsoperationen und ein Angebot entsprechender Propagationsflags notwendig.

- Ein Anwendungsbereich unserer Mechanismen, der außerhalb der von uns verfolgten Ziele liegt, findet sich bei verteilten Datenbanken mit replizierten Objekten. Die Knoten einer verteilten Datenbank entsprechen dabei den Schemaversionen unseres Modells, die Replikate eines Objektes sind den Objektversionen analog und der Datentransport zum Zwecke der Aktualisierung von Replikaten ist der Ausführung von Konvertierungsfunktionen vergleichbar. Ein den Propagationsflags sehr ähnlicher Mechanismus könnte verwendet werden, um zu definieren, wie schnell Replikate bei Änderungen ihres Originals nachgeführt werden sollen. Während dabei zunächst alle Schemaversionen identisch wären, ließe sich ein darüber hinausgehendes Modell entwickeln, das verschiedene Schemaversionen an unterschiedlichen oder sogar an ein und demselben Knoten der verteilten Datenbank erlaubt.

Nachdem wir im vorangegangenen Kapitel 5 die Schemaversionierung in einem ersten Schritt auf der Abstraktionsebene des Schemas betrachtet hatten, konnten wir hier die Auswirkungen auf der Objektebene untersuchen und passende Konzepte entwickeln. Ohne Entsprechung in vorhandenen Systemen und damit in der beschriebenen Form komplett neu sind insbesondere die Steuerung der Ausführung von Konvertierungsfunktionen durch Propagationsflags, die Untersuchung der transitiven Propagation und die Möglichkeit der Spezifikation von Extrakonvertierungsfunktionen. Die Untersuchungen auf der Objektebene stellen den zweiten Schritt dar, der unseren Ansatz der Unterstützung von Schemaevolutionsprozessen durch den Einsatz

von Versionierungskonzepten vervollständigt und somit die konzeptionelle Entwicklung in der vorliegenden Arbeit abschließt.

Kapitel 7

Prototypische Realisierung

Das COAST-Projekt (**C**omplex **O**bject **A**nd **S**chema **T**ransformation) wurde an der Universität Frankfurt am Main initiiert, um die Realisierbarkeit der in dieser Arbeit vorgestellten Mechanismen zu belegen. Folglich war das Ziel die Entwicklung und Implementierung eines Objektdatenbanksystems mit Unterstützung von Schemaevolution entsprechend der technischen Teilziele aus Abschnitt 3.2. Dabei haben wir uns auf diejenigen Konzepte konzentriert, die insbesondere den Schemaversionierungsmechanismus betreffen und die in anderen Systemen nicht enthalten sind. Zu diesen spezifischen Konzepten gehören u.a. die Verwaltung versionierter Schemata und die Propagation von Datenbankzuständen und -änderungen zwischen den Zugriffsbereichen verschiedener Schemaversionen.

In Rahmen von COAST arbeiteten einige Diplomanden[113] an der prototypischen Realisierung eines objektorientierten Datenbanksystems mit einem Ausschnitt der beschriebenen Schemaversionierungsmechanismen in der Programmiersprache C++ [Str97].[114] Die dabei erreichten Ergebnisse wurden u.a. auf der CeBIT'98 [BL98] und '99 [Lau99b] am Stand der Hessischen Hochschulen sowie auf der ECOOP'99 [LADH99] vorgestellt und sind der Öffentlichkeit weiterhin über das World Wide Web [Lau00] zugänglich.

Dieses Kapitel gibt zunächst einen Überblick über die Architektur des COAST-OODBMS Prototypen und geht dabei auf die eingesetzten Komponenten und ihre Aufgabenstellungen ein. Daraufhin wird die Implementierung einiger dieser Komponenten genauer vorgestellt.

7.1 Die Architektur des COAST-OODBMS Prototypen

Der Prototyp des COAST-OODBMS lehnt sich in seiner Architektur (siehe Abbildung 7.1) an dokumentierte experimentelle und kommerzielle Produkte an. Den wesentlichen Bestandteil bildet die Datenbankmaschine, die aus Objekt-, Schema- und Propagationsmanager besteht.

Der *Objektmanager* und der *Schemamanager* erbringen die wesentliche Funktionalität vergleichbarer Komponenten anderer Systeme und erlauben in Erweiterung dieser auch die Verwaltung versionierter Objekte und Schemata. Die Konzepte des Schemaversionierungsmechanismus wer-

[113] Zu nennen sind hier Sabbas Apostolidis [Apo00, LADH99], Alexander Doll [Dol99, LADH99], Patricia Eigner [Eig97, LEW97], Michael Großmann [Gro00], Jan Haase [Haa00, LADH99], Detlef Herchen [Her99], Manfred Prien [Pri98], Christian Wöhrle [Wöh96, LEW97] und Kay Wölfle [Wöl98].

[114] Trotz einiger Tücken [Sak88b] ist C++ im industriellen Umfeld wohl die am häufigsten eingesetzte Programmiersprache für objektorientierte Anwendungen größeren Umfangs. Neben den damit verbundenen Vorteilen bezüglich Stabilität, Verfügbarkeit und Werkzeugunterstützung wurde C++ für die Realisierung des COAST-Servers insbesondere deshalb ausgewählt, weil der als Grundlage dienende EOS-Server (siehe Abschnitt 7.1.1.1) ausschließlich in dieser Programmiersprache eine Schnittstelle anbietet.

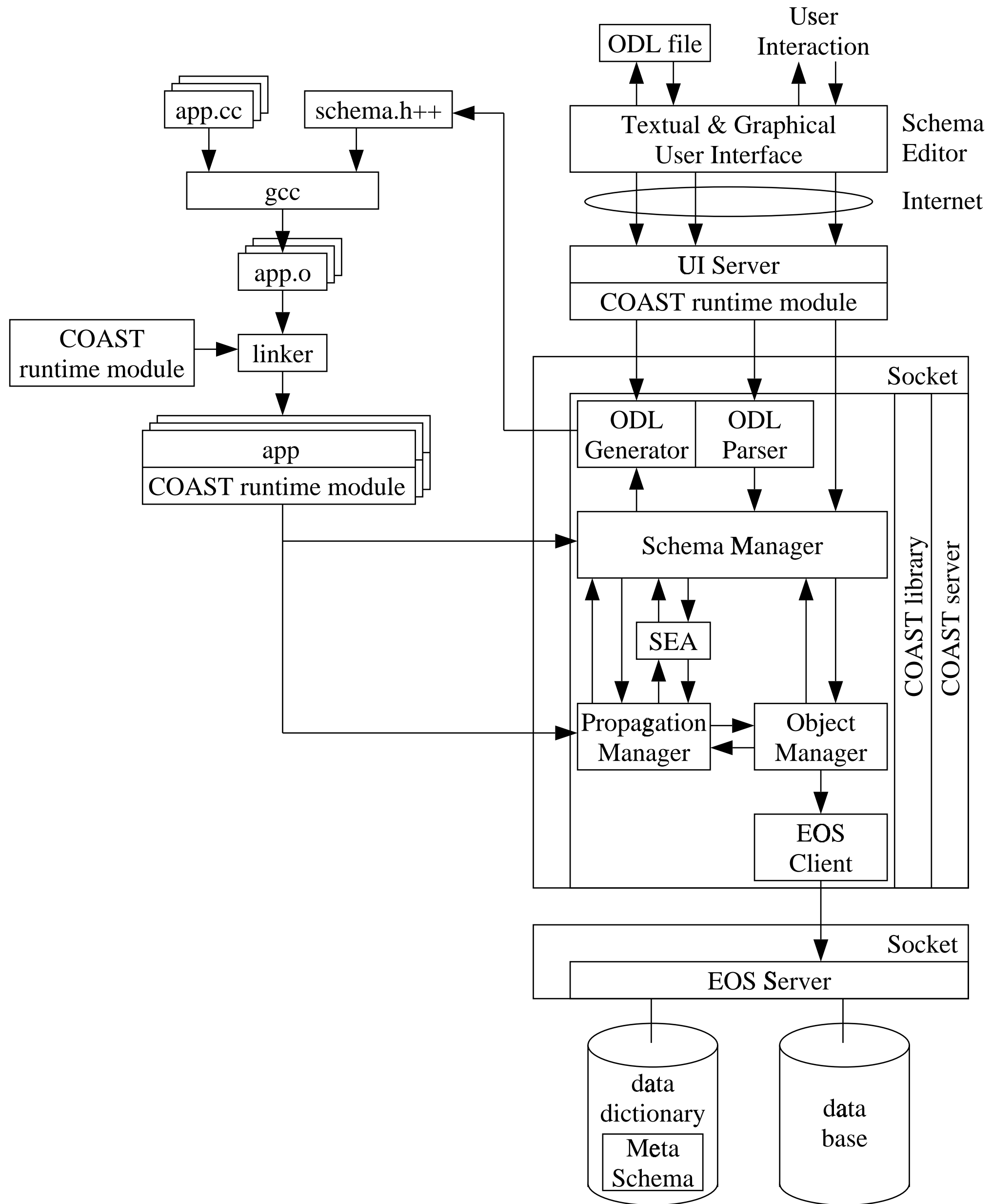


Abbildung 7.1: Die Architektur des COAST OODBMS.

den von der dritten Komponente der Datenbankmaschine realisiert, welche wir als *Propagationsmanager* bezeichnen.

Bei der Anbindung von Applikationen an das OODBMS folgen wir der Vorgehensweise, die von der ODMG vorgeschlagen wird [Cat96, CB98, CBB+00]. Die bereits in Abschnitt 5.5 vorgestellte COAST-ODL kann verwendet werden, um sowohl einzelne Schemaversionen als auch Schemaänderungen zu beschreiben. Eine solche Beschreibung wird vom *ODL-Parser* analysiert und in Aufrufe von Methoden der Anwendungsschnittstelle (engl. *Application Programmer Interface, API*) des Schemamanagers umgesetzt. Alternativ können Schemaversionen an der textuellen oder an der graphischen Benutzeroberfläche interaktiv beschrieben und manipuliert werden. Aus den im Schemamanager gespeicherten Schemaversionen erzeugt der *ODL-Generator* eine Headerdatei (`schema.h++`) für die Programmiersprache C++, welche in die Quellen der Applikationen (`app.cc`) einzubinden ist, um auf eine COAST-Datenbank zugreifen zu können. Diese C++-Headerdatei beinhaltet alle Deklarationen und Anweisungen, die für die persistente Speicherung von Objekten in der Datenbank benötigt werden.

Die drei Komponenten der Datenbankmaschine werden durch den Schemaassistenten und den ODL-Parser und -Generator zum COAST-Server ergänzt, welcher als Betriebssystem-Prozeß von den Applikationen, also von den COAST-Klienten kontaktiert werden kann. Die dazu notwendige Socket-Kommunikation wird von den Applikationsentwicklern verborgen in den generierten Headerdateien bewerkstelligt.

Der sog. *Schemaeditor* stellt eine textuelle und eine graphische Benutzeroberfläche dar und erlaubt neben der Beschreibung und Modifikation von Schemaversionen auch die Spezifikation der Propagationsflags und der Konvertierungsfunktionen.

7.1.1 Die Datenbankmaschine

Der Kern des COAST-OODBMS, die sog. *COAST-Datenbankmaschine* stellt die Methoden zur Verfügung, die von Applikationen über deren Programmierschnittstelle (API) genutzt werden können.

Die COAST-Datenbankmaschine besteht aus drei interagierenden Komponenten. Diese sind der *Objektmanager*, der *Schemamanager* und der *Propagationsmanager*. Die beiden erstgenannten Komponenten sind mit ähnlicher Funktionalität und Schnittstelle in zahlreichen prototypischen und kommerziellen OODBMS[115] zu finden. In Erweiterung der dort vorhandenen Funktionalität ermöglichen die COAST-Komponenten die Verwaltung eines versionierten Datenbankschemas. Der Propagationsmanager als dritte Komponente der COAST-Datenbankmaschine kontrolliert die Propagation von Objekten zwischen den Zugriffsbereichen verschiedener Versionen eines Schemas. Demzufolge findet er keine Entsprechung in anderen Datenbanksystemen.

7.1.1.1 Der Objektmanager

Der Objektmanager[116] erledigt alle Aufgaben, die mit dem Zugriff auf in der Datenbank enthaltene Objekte in Zusammenhang stehen. Die dazu notwendigen Zugriffsoperationen zum Erzeugen, Lesen und Modifizieren von Objekten können dabei in Transaktionen gekapselt werden, die der ACID-Semantik [HR83] entsprechen. Damit sind Transaktionen atomar, konsistenzerhaltend, isoliert und dauerhaft.

[115] In O_2 [BDK92] heißen diese Komponenten auch Objekt- und Schemamanager, bei GemStone [BOS91] wird von Stone- und Gem-Prozeß gesprochen, bei IRIS [FBC+87, FAB+89] von Storage- und Objekt-Manager und bei ORION [KGBW90] von Storage- und Objekt-Subsystem.

[116] Der Objektmanager wurde von Kay Wölfle [Wöl98] implementiert.

Zur Verringerung des notwendigen Implementierungsaufwandes setzt der COAST-Objektmanager auf einem frei erhältlichen Objektmanager namens *Extended Object Store* (EOS, [BP94]) auf, der u.a. auch für die Implementierung des Datenbanksystems ODE [AG89] verwendet wurde. EOS bietet an seiner Schnittstelle bereits die für den physikalischen Datenzugriff notwendigen Methoden an. Die Verwaltung der untypisierten Objekte geschieht in EOS bereits unter der Kontrolle eines Transaktionsmechanismus. Eine nähere Beschreibung von EOS findet sich in [Wöl98]. Die Erweiterung zum COAST-Objektmanager ergänzt die Verwaltung versionierter Objekte, wobei die Versionen eines Objektes im Gegensatz zu den Konzepten der Objektversionierung wie sie in Abschnitt 2.2 beschrieben wurden, auch unterschiedlichen Typen angehören können. Wir werden in Abschnitt 7.2.1.1 auf die vom Objektmanager benutzten Datenstrukturen eingehen.

Der Objektmanager benutzt den Schemamanager zur Interpretation der Objekte der Datenbank, so wie es auch im ODMG-Standard beschrieben ist [Cat96, CB98, CBB+00].

Die Entwicklung eines Nachfolgers von EOS unter dem Namen *Bell Laboratories Storage System* (BeSS) [BP96] kam nicht über den Zustand eines vorläufigen Prototyps hinaus, so daß dessen Verwendung hier nicht in Betracht gezogen werden konnte.

7.1.1.2 Der Schemamanager

Der Schemamanager[117] ist derjenige Teil der Datenbankmaschine, der die Verwaltung der Schemata bewerkstelligt. Dazu verwaltet er Informationen über die Versionen eines gerade geöffneten Schemas, über die darin enthaltenen Klassen und deren Versionen, über die zwischen den Klassenversionen bestehenden Ableitungs- und Vererbungsbeziehungen, sowie über deren Attribute und Methoden.

An seiner Programmierschnittstelle (API) bietet der Schemamanager Methoden zum Anlegen und Verändern eines versionierten Schemas und seiner Komponenten an. Diese Methoden decken den in der COAST-ODL spezifizierbaren Funktionsumfang unserer Schemaänderungsprimitive ab.

Der Schemamanager benutzt den Objektmanager zur Speicherung des Schemas entsprechend einem fest vorgegebenen Metaschema (siehe Abschnitt 7.2.2.2).

7.1.1.3 Der Propagationsmanager

Der Propagationsmanager[118] ist ein Modul der COAST-Datenbankmaschine, das in dieser Form einzigartig ist. Zu den grundlegenden Aufgaben des Propagationsmanagers zählt die Verwaltung der Konvertierungsfunktionen und der Propagationsflags. Darauf aufbauend muß er zur Laufzeit für die korrekte Ausführung der Konvertierungsfunktionen sorgen. Zur Verbesserung der Effizienz (technisches Teilziel 3.15) werden hier Algorithmen eingesetzt, die die notwendige Propagationen nicht sofort, sondern verzögert durchführen. Dabei wird ein Objekt idealerweise immer erst dann in den Zugriffsbereich einer bestimmten Schemaversion propagiert, wenn eine Applikation dieser Schemaversion auf das Objekt zugreift. Auch wenn dieser Idealfall in der Praxis nicht immer zu erreichen ist, da stets ein Kompromiß zwischen Teilzielen wie Zugriffszeit beim Schreiben und Lesen einerseits und Speicherplatzverbrauch andererseits eingegangen werden muß, können durch die verzögerte Propagation erhebliche Verbesserungen erreicht werden. Ein insbesondere bei großen Datenbanken lang anhaltender Konvertierungsprozeß, während dem die Datenbank gesperrt werden muß, ist nicht notwendig und der Verbrauch von Speicherplatz

[117] In [CB98, CJR98] wird der Schemamanager *Schema Repository Module* genannt. Der Schemamanager wurde von Alexander Doll [Dol99, LADH99] und Manfred Prien [Pri98] implementiert.

[118] Der Propagationsmanager wurde von Patricia Eigner [Eig97, LEW97], Jan Haase [Haa00, LADH99] und Christian Wöhrle [Wöh96, LEW97] implementiert.

und CPU-Zeit wird möglichst lange verzögert. Schließlich müssen zahlreiche Konvertierungen gar nicht durchgeführt werden, wenn ein Objekt von Applikationen einer einzigen Schemaversion mehrmals geändert wird und in der Zwischenzeit keine Applikationen anderer Schemaversionen auf dieses Objekt zugreifen. Es genügt dann nämlich den letzten Stand zu propagieren, die Konvertierungen der zwischenzeitlichen Objektzustände können mit dem verzögerten Mechanismus komplett eingespart werden. Wird auf ein Objekt durch eine bestimmte Schemaversion gar nicht zugegriffen, so kann auf die Berechnung und Speicherung der entsprechenden Objektversion gänzlich verzichtet werden.

7.1.2 Weitere Komponenten

7.1.2.1 Der Schemaeditor

Der Schemaeditor[119] stellt dem Schemaentwickler eine Benutzeroberfläche zur Verfügung, die ihm den Zustand des Schemas und seiner Komponenten visualisiert und mit der er auf zwei Wegen arbeiten kann. Zum einen kann der Schemaeditor eine Liste von Schemaänderungsprimitiven mitsamt den darin enthaltenen Anweisungen der Propagationssprache aus einer ODL-Datei lesen und unüberwacht im Stapelverarbeitungsbetrieb ausführen. Zum zweiten bietet der Schemaeditor eine textuelle (engl. *Textual User Interface, TUI*) und eine graphische Benutzerschnittstelle (engl. *Graphical User Interface, GUI*) an, mit der Schemata interaktiv spezifiziert und verändert werden können (siehe Abbildungen 7.2 und 7.3). Während im nicht-interaktiven Betrieb beim Abarbeiten der in einer Datei fest vorgegebenen ODL-Anweisungen ein Fehler erst erkannt wird, nachdem alle Anweisungen schon spezifiziert sind, können im interaktiven Betrieb beispielsweise Konsistenzprüfungen jeweils sofort nach jeder Einzelanweisung durchgeführt werden.

Wie mit der ODL und der darin enthaltenen Propagationssprache, so können auch an der textuellen oder an der graphischen Benutzeroberfläche Schemaänderungen nebst Konvertierungsfunktionen und Propagationsflags spezifiziert werden.

Der Schemaeditor ist als gewöhnliche COAST-Applikation implementiert, die die COAST-Laufzeitbibliothek benutzt, um Kontakt zum COAST-Server aufzunehmen. Die graphische Komponente des Schemaeditors ist als Applet und als Applikation in der Programmiersprache Java [GJS96] implementiert, um Interessenten die Möglichkeit zu geben, die Funktionalität des Schemaversionierungsmechanismus mit Hilfe eines üblichen Web-Browsers ausprobieren zu können. Die jeweils neueste Version des Schemaeditors steht auf der COAST-Homepage [Lau00] zur Verfügung.

7.1.2.2 Der Schemaassistent und weitere Werkzeuge

Der Schemaassistent[120] (engl. *Schema Evolution Assistant, SEA*) soll dem Schemaentwickler Hilfestellung bei der Analyse existierender und bei der Ableitung neuer Schemaversionen geben. Dabei sind zahlreiche Einsatzfelder denkbar.

- Der Schemaassistent kann ähnliche Hilfestellungen leisten, wie sie schon bei der Entwicklung unversionierter Schemata wünschenswert sind, also z.B. Erkennung unbenutzter Klassen, Attribute oder Methoden, Unterstützung bei der Verbesserung einer gegebenen Vererbungsstruktur, etc. Da sich diese Aspekte auf die Verbesserung einer isoliert betrachteten Schemaversion beziehen und damit keinen speziellen Zusammenhang mit dieser Arbeit haben, werden sie hier nicht weiter untersucht.

[119] Der Schemaeditor wurde von Michael Großmann [Gro00] implementiert.
[120] Der Schemaassistent wurde von Sabbas Apostolidis [Apo00, LADH99] implementiert.

```
~coast/SNiFF+/private/Lautemann/coast for lauteman@woerth (SunOS 5.7)
.../Lautemann/coast> coast n

*****************************************************************************
* Complex Object And Schema Transformation (COAST) v0.02, Demo              *
* copyright (c) 1995-99, The COAST Team, University of Frankfurt/M., Germany *
*****************************************************************************

COAST  Main Menu   - 1 -  Schemamanager
                   - 2 -  Propagationmanager
                   - 3 -  Objectmanager
                   - 4 -  ODL Parser/Generator
                   - 5 -  Persistent List
                   - 6 -  Transient List
                   - 7 -  SEA-Tool

                   - i -  Info
                   - q -  Quit

Your choice (1-7, i, q): 1


**********************************
* COAST v0.02, Demo, (c) 1995-99 *
**********************************

No Current Schema


Schema Manager Menu  - 1 -  Create Schema
                     - 2 -  Delete Schema
                     - 3 -  Select Schema
                     - 4 -  Load Schema (prepared example)
                     - 5 -  Load Schema (file)
                     - 6 -  Save Schema (file)
                     - 7 -  Load/Modify Schema (parse ODL file)
                     - 8 -  Close Schema
                     - 9 -  Check Consistency
                     - s -  EOSFORM + Load Handel
                     - t -  EOStest

                     - q -  Quit

Your choice (1-9, s, t, q): 1

Create a Schema
Enter name for new Schema. --> mySchema
Database 'mySchema' successfully created!
Schema mySchema created.
```

Abbildung 7.2: Die textuelle Schnittstelle des COAST-Schemaeditors.

- Zur Erhaltung der Konsistenz müssen die Schemaänderungsprimitive nach der Durchführung der eigentlichen Änderung ggf. noch korrigierende Maßnahmen ergreifen. Diese stellen also indirekte Konsequenzen einer Schemaänderung dar und sind dem Schemaentwickler daher nicht immer sofort offensichtlich. Beinhaltet das Löschen einer Klasse beispielsweise auch das Entfernen sämtlicher Unterklassen, so kann der Schemaassistent als interaktives Werkzeug auf diese Konsequenz aufmerksam machen. Noch anspruchsvoller ist die auch in [CPLZ92d] geforderte Fähigkeit, eine Folge illegaler Schemaänderungsprimitive in eine legale umzusetzen. Dabei muß die hinter einer nicht durchführbaren Schemaänderung stehende Absicht ggf. durch Rückfrage beim Schemaentwickler ermittelt werden und, falls möglich, ein alternativer Weg zur Erreichung des Gewünschten gefunden werden. Im obigen Beispiel könnten Unterklassen der zu löschenden Klasse erhalten werden, wenn vor dem Löschen die Vererbungsbeziehungen zu den Unterklassen geeignet entfernt oder ersetzt werden. Ähnliche Aufgabenstellungen hatten wir bei der Veränderung der Vererbungsstruktur (siehe Abschnitt 5.5.4.1) gefunden. Unerwünschte Konsequenzen können außerdem eintreten, wenn eine bereits durchgeführte Schemaänderung durch eine vermeintliche inverse Änderung zurückgenommen werden soll. Ein gelöschtes Attribut beispielsweise mag durch Neuanlegen auf Schemaebene wiederherzustellen sein, auf Objektebene gehen jedoch ggf. Daten verloren.

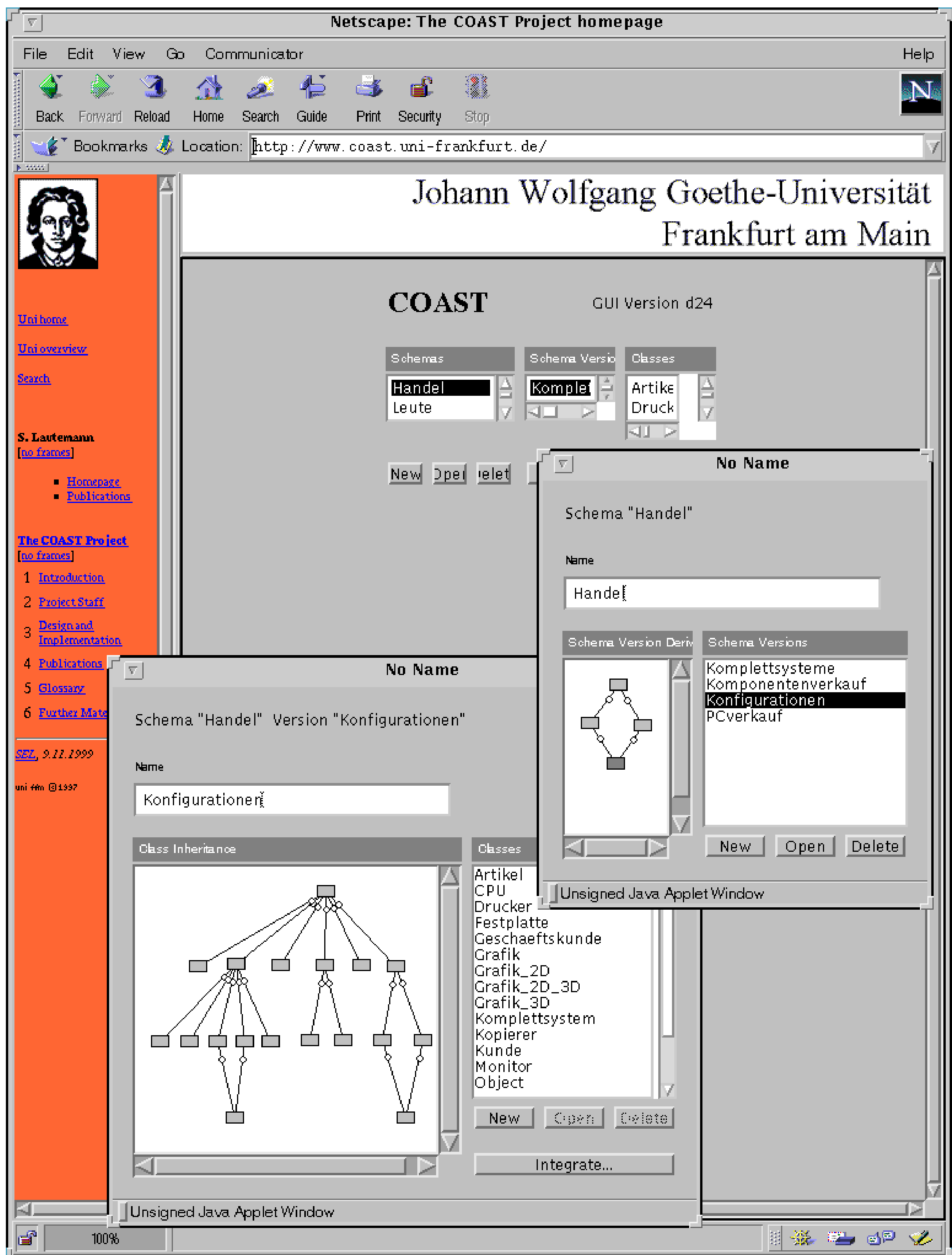


Abbildung 7.3: Die graphische Schnittstelle des COAST-Schemaeditors.

- Die Vielfalt verschiedener Möglichkeiten bei der Ableitung einer neuen Schemaversion durch Erzeugung neuer oder Integration und Veränderung bestehender Klassen kann insbesondere bei großen Schemata leicht unübersichtlich werden. Durch Analyse einer in Entwicklung befindlichen Schemaversion und Vergleich mit bereits existierenden können Ähnlichkeiten festgestellt werden, die sich durch Klassenintegration nutzen lassen. Damit können Schemabeschreibungen lesbarer werden, Ähnlichkeiten deutlicher hervor treten und weitergehende Propagationen ermöglicht werden.

  Insbesondere bei extern erstellten Schemaversionen (siehe Teilziel 3.4) kann der Schemaassistent bei deren Einbringung in das Datenbanksystem zur Erkennung von Ähnlichkeiten und damit zur Erstellung der in Teilziel 3.8 geforderten, semantisch bedeutungsvollen Ableitungsbeziehungen eingesetzt werden. Auf diesem Wege kann auch die Integration von Versionen anderer Schemata erreicht werden, obwohl unsere ODL keine Primitive zur Integration von Klassen anderer Schemata anbietet.

- Bisher hatten wir zumeist einen „korrekten“ Umgang mit den Schemaänderungsprimitiven vorausgesetzt, d.h. der Schemaentwickler muß sich nicht nur der Konsequenzen eingesetzter Primitive für das Schema bewußt sein, sondern er muß auch spätere Auswirkungen auf Objektebene bedenken. Insbesondere in Abschnitt 5.5.7 hatten wir verschiedene Listen von Schemaänderungsprimitiven gezeigt, die bei isolierter Betrachtung der resultierenden Schemaversionen, d.h. bei deren Ausgabe in erzeugender ODL, identische Resultate liefern, die jedoch unterschiedliche Zusammenhänge mit anderen Schemaversionen etablieren und damit auch unterschiedliche Auswirkungen auf die Propagation von Objekten haben.

  Eine erste Aufgabe für ein Werkzeug besteht hierbei also darin, potentielle Problemstellen zu erkennen und den Schemaentwickler zunächst einmal darauf aufmerksam zu machen. Idealerweise werden mögliche Änderungen vorgeschlagen, mit denen die intendierte Wirkung tatsächlich erreicht werden kann.

  In Abschnitt 6.4 wurden Schemaänderungsprimitive vorgestellt, die ein nachträgliches Etablieren von Beziehungen zwischen Klassen verschiedener Versionen eines Schemas und zwischen Attributen verschiedener Versionen einer Klasse erlauben (**create derivation**). Damit können bei der Erstellung einer Schemaversion gemachte Fehler noch zu einem späteren Zeitpunkt korrigiert werden. Alternativ dazu kann auf diese zusätzlichen Schemaänderungsprimitive verzichtet werden, wenn der Schemaassistent in der Lage ist, eine *verbesserte Kopie* einer existierenden Schemaversion anzulegen. Darunter soll eine Kopie verstanden werden, die bei einer isolierten Betrachtung, d.h. in der erzeugenden ODL, dem Original gleicht, die diesem jedoch nicht identisch ist. Der Unterschied besteht gerade darin, daß die oben genannten, eigentlich erwünschten Beziehungen bereits beim Anlegen der verbesserten Kopie etabliert werden und somit keine spätere Korrektur notwendig ist. Damit entfällt auch der Bedarf für die zusätzlichen Schemaänderungsprimitive **modify** und **delete derivation**. Da die Kopie dem Original bei isolierter Betrachtung vollkommen gleicht, ist die Übertragung ggf. existierender Applikationen oder selbst von Objekten der Datenbank kein Problem. Wenn das Werkzeug abschließend das Original löscht und der Kopie durch Umbenennen den Namen der Originalschemaversion gibt, so entsteht für den Schemaentwickler genau dasselbe Ergebnis, das er auch von der Ausführung des **modify derivation**-Primitives erwarten würde.

- Die Erstellung von Defaultkonvertierungsfunktionen sowie die Prüfung von durch den Schemaentwickler manuell erstellten Konvertierungsfunktionen kann durch ein Werkzeug geleistet werden. Dies stellt insbesondere mit Blick auf implizit integrierte Klassen eine wichtige Aufgabe dar, da auch für diese Konvertierungsfunktionen benötigt werden. Weiterhin haben wir kein Konzept für die Vererbung von Konvertierungsfunktionen eingeführt, so daß bei der Veränderung einer Klasse auch die Konvertierungsfunktionen deren integrierter Un-

terklassen ggf. überarbeitet werden müssen, sofern diese Unterklassen durch die Vererbung implizit mitverändert wurden.

Die Erzeugung von Defaultkonvertierungsfunktionen muß sich natürlich auf alle integrierten Klassen erstrecken und dabei auch durch die Vererbung verursachte, implizite Veränderungen in Unterklassen berücksichtigen. Damit stellt der Schemaassistent einen Ersatz für einen Mechanismus zur Vererbung von Konvertierungsfunktionen dar.

- Einem Schemaentwickler sind Einschränkungen bei den Definitionsmöglichkeiten von Extrakonvertierungsfunktionen (siehe Abschnitte 6.6.2.1 und 6.6.2.2) und die damit verbundenen Konsequenzen für die Übertragbarkeit von Objektsemantik mitunter nicht sofort bewußt. Da Veränderungen an bereits in Benutzung befindlichen Schemaversionen nicht mehr ohne Ableitung neuer Schemaversionen möglich sind, soll der Schemaassistent den Schemaentwickler bereits bei der Erstellung einer neuen Schemaversion auf ggf. vorhandene Verbesserungsmöglichkeiten aufmerksam machen.

- Die Forderung nach einer Abschaltbarkeit der Objektpropagation war in Teilziel 3.12 u.a. durch die Möglichkeit eines Testbetriebes motiviert worden, in dem ggf. vorhandene Schwächen von Datenmodellierung und Applikationen sich nicht auf tatsächlich genutzte Objekte auswirken sollten. Das in dieser Arbeit vorgestellte Modell erlaubt jedoch keine nachträgliche Veränderung von Konvertierungsfunktionen oder Propagationsflags mehr, sobald sich die betroffene Schemaversion im Einsatz befindet, d.h. sobald sie nicht mehr aufgetaut werden kann. Damit ergibt sich zunächst das Problem, daß im Testbetrieb befindliche Schemaversionen und Applikationen nach erfolgreich abgeschlossenen Tests nicht direkt verwendet werden können.

  Ein Werkzeug kann jedoch trotzdem eine nachträgliche Veränderung von Konvertierungsfunktionen und Propagationsflags ermöglichen, indem es eine neue Schemaversion erstellt, die der ursprünglichen in struktureller und verhaltensorientierter Hinsicht vollkommen gleicht, die jedoch andere Propagationsparameter spezifiziert. Weiterhin muß das Werkzeug die vorhandenen Applikationen übertragen, wobei aufgrund der Identität der beiden Schemaversionen jedoch keinerlei Anpassungen notwendig sind. Lediglich die beim initialen Betrieb der ursprünglichen Schemaversion durchgeführten Veränderungen ihres Zugriffsbereiches werden dabei nicht übernommen. Da der initiale Betrieb allerdings dem Test gewidmet war, wäre das ohnehin nicht wünschenswert.

- Ein breites Spektrum für weitere Werkzeuge ist auf der Objektebene zu finden. Hier könnte gezielt Einfluß auf die physikalische Struktur einer Datenbank genommen werden, etwa um Zugriffszeiten durch Indexierung oder Clusterung zu verringern. Neben diesen allgemeinen Aspekten ist im Umfeld unserer Arbeit insbesondere die Betrachtung der versionierten Objekte und des Propagationsmanagers von vorrangigem Interesse. Zur Verringerung des Platzbedarfes könnten wiederherstellbare Objektversionen, ggf. sogar in Abhängigkeit von im Betrieb ermittelten Zugriffshäufigkeiten, gelöscht werden. Zeiten geringer Systemlast könnten für die Nachführung aufgrund der verzögerten Vorgehensweise noch nicht vollzogener Propagationsschritte genutzt werden, was vornehmlich den Zeitbedarf zukünftiger Zugriffe verringern kann.

- Schließlich sind Funktionen zum Anzeigen und Vergleichen von Schemaversionen hilfreich, insbesondere wenn eine passende Schemaversion für neue Applikationen ausgewählt werden muß. Weiterhin kann der Umgang mit dem System durch Reportgeneratoren erleichtert werden, die etwa intern gesammelte statistische Informationen aufbereiten und ausgeben.

7.1.2.3 Der ODL-Parser und der ODL-Generator

Für die Interaktion von Schema- und Applikationsentwicklern mit COAST wird eine textuelle Schnittstelle benötigt, über die das vom Schemamanager gerade verwaltete Schema oder Teile davon ausgegeben und ggf. Änderungsanforderungen eingegeben werden können. Die Erzeugung und Veränderung eines Datenbankschemas geschieht hierbei durch Angabe von Schemaänderungsprimitiven der bereits in Abschnitt 5.5 vorgestellten Schemabeschreibungssprache COAST-ODL. Die nicht-interaktive Schnittstelle von COAST wird durch den ODL-Parser und den ODL-Generator realisiert. Die Eingabe des Schemas wird dabei vom ODL-Parser entgegengenommen und verarbeitet, während der ODL-Generator Ausgaben verschiedener Arten leistet. Die textuelle Schnittstelle erfüllt in COAST insbesondere die folgenden Aufgaben:

- Die Eingabe eines oder mehrerer ODL-Ausdrücke kann sowohl zur Spezifikation neuer Schemata und Komponenten davon als auch zur Veränderung vorhandener Schemata benutzt werden.

- Die Ausgabe eines ODL-Ausdruckes, der das gegenwärtig vom Schemamanager verwaltete Schema oder Teile daraus beschreibt, dient sowohl der Dokumentation einer Datenbank als auch der Information des Schemaentwicklers. Dies ist deshalb notwendig, weil sich der gegenwärtige Schemazustand in der Regel nicht nur aufgrund der Eingabe eines einzelnen ODL-Ausdruckes ergeben hat. Stattdessen wird sich der aktuelle Zustand durch eine schrittweise Spezifikation einzelner Schemaversionen ergeben, die darüber hinaus auch noch nachträglich verändert werden können. Schließlich können Schemaänderungen auch über die interaktive Schnittstelle des Schemaeditors (siehe Abschnitt 7.1.2.1) gemacht worden sein, d.h. der Schemaentwickler hat für diese Änderungen gar keine ODL-Spezifikation erstellt.

  Für die Zwecke der Dokumentation und zum Arbeiten mit dem System ist eine Beschreibung des aus mehreren Änderungsschritten resultierenden Ergebnisses deutlich besser geeignet als die Beschreibung eines Ausgangszustandes und einer (bei Benutzung des Schemaeditors ggf. unvollständigen) Liste darauf nachträglich angewandter Änderungen. Der gegenwärtige Zustand repräsentiert ja gerade das Ergebnis der Anwendung diverser Deltas auf den Ausgangszustand.

  Für die Ausgabe des gegenwärtigen Schemazustandes sind zwei Modi zu unterscheiden und diese werden auch beide von der Implementierung des ODL-Generators (siehe Abschnitt 7.2.4) unterstützt.

  - Zum einen kann die Ausgabe des Schemas als Menge isolierter Schemaversionen erfolgen. Dabei werden keinerlei Ableitungsbeziehungen zwischen Komponenten verschiedener Schemaversionen dargestellt, was bedeutet, daß insbesondere das **integrate**-Primitiv hierbei nicht verwendet wird. In diesem Modus kann sich der ODL-Generator sogar komplett auf die erzeugende ODL (siehe Abschnitt 5.5.6) beschränken. Andere Primitive, wie etwa das Umbenennen von Klassen werden dabei nicht benötigt, da die Klasse in der erzeugenden ODL direkt mit dem aktuellen Namen angegeben werden kann. Dieser Modus kann für die Analyse einzelner Schemaversionen beispielsweise zur Applikationsentwicklung hilfreich sein, da dann keine Querverweise zu anderen Schemaversionen betrachtet werden müssen.
  - Für die Ausgabe des Schemas inklusive aller darin enthaltenen Ableitungsbeziehungen sind allerdings i.Allg. auch Umbenennungen erforderlich, und zwar gerade dann, wenn eine Klasse in verschiedenen Schemaversionen verschiedene Namen hat. Auch das Löschen und das Ändern von Name oder Typ eines Attributes wird in diesem Modus benötigt, um integrierte Klassen anpassen zu können.

- Die Ausgabe des Schemas in der Syntax einer objektorientierten Programmiersprache kann als Grundlage für die Anbindung von in dieser Sprache geschriebenen Applikationen an COAST dienen. Da COAST zum größten Teil in C++ implementiert ist, liegt die Ausgabe des Schemas in Form von C++-Klassendeklarationen nahe, welche dann als Headerdatei in Applikationen eingebunden werden können.[121] Für eine vollständige und entsprechend unserem technischen Teilziel 3.19 transparente Applikationsanbindung sind allerdings über die Klassendeklarationen hinausgehende Maßnahmen notwendig, wie etwa die Bereitstellung von Methoden für den lesenden und schreibenden Zugriff auf die Attribute persistenter Objekte.

- Die Aus- und Eingabe von Schemaspezifikationen in Form von ODL-Ausdrücken ist weiterhin notwendige Voraussetzung für die Realisierung des externen Schemaänderungsansatzes entsprechend Teilziel 3.4. Weiterhin ist dafür allerdings ein Werkzeug (siehe Abschnitt 7.1.2.2) zur Erkennung ggf. bestehender Ableitungsbeziehungen zu extern erstellten Schemaversionen notwendig.

- Die Aus- und Eingabe von ODL-Ausdrücken dient in COAST des weiteren der Spezifikation von Konvertierungsfunktionen. Diese sind integraler Bestandteil der ODL und nach Teilziel 3.16 ist insbesondere die Einstellung der Propagationssteuerung möglichst einfach zu halten. Zu diesem Zweck können in COAST Defaultkonvertierungsfunktionen erzeugt werden, die der Schemaentwickler bei Bedarf manuell ändern und dann verwenden kann. Auch hierfür kann die textuelle Schnittstelle von COAST verwendet werden. Um dem Schemaentwickler die Defaultkonvertierungsfunktionen zu übergeben, wird vom ODL-Generator eine Datei erzeugt, die auf dem **modify derivation**-Ausdruck basiert und genau alle Konvertierungsfunktionen für eine Schemaversion enthält. Diese Datei kann dann vom Schemaentwickler in einem herkömmlichen Texteditor modifiziert werden. Der Propagationsmanager bekommt die benutzerdefinierten Konvertierungsfunktionen schließlich über den ODL-Parser mitgeteilt.

7.1.2.3.1 Der ODL-Parser

Der ODL-Parser[122] erhält ODL-Spezifikationen in einer Textdatei und prüft darin zunächst die syntaktische Korrektheit. Ist diese gegeben, so bildet er die ODL-Anweisungen auf die Programmierschnittstelle (API) des Schemamanagers ab. Die Funktionalität der ODL wird also letztlich durch die API des Schemamanagers realisiert. Der Parser geht bei der Abbildung der ODL-Anweisungen auf die API des Schemamanagers in zwei Schritten vor. In einem ersten Durchlauf werden zunächst nur Klassen und ihre Vererbungsbeziehungen innerhalb der Schemaversionen etabliert. Erst im zweiten Durchlauf werden diese „leeren“ Klassen mit Attributen und Methoden versehen. Auf diese Weise wird die in Fußnote 84 auf Seite 161 erwähnte Problematik beim Parsen zyklischer Referenzen sehr einfach umgangen. Auf Fehler reagiert der ODL-Parser mit entsprechenden Meldungen.

7.1.2.3.2 Der ODL-Generator

Die Aufgabe des ODL-Generators[122] umfaßt die Erstellung von ODL-Beschreibungen des gesamten Schemas oder einzelner Schemaversionen, letzteres ggf. unter Beschränkung auf die erzeugende ODL (siehe Abschnitt 5.5.6). Weiterhin produziert der ODL-Generator Headerdateien für einzelne Schemaversionen, die dann von den Applikationen dieser Schemaversionen einzubinden sind, um den Versionierungsansatz gemäß dem technischen Teilziel 3.19 der Transparenz vor den

[121] Wie man hier sieht, ist die Bezeichnung *ODL-Generator* eigentlich zu restriktiv, da der Generator auch C++-Code erzeugen kann.

[122] Der ODL-Parser und -Generator wurde von Detlef Herchen [Her99] implementiert.

Applikationsentwicklern verbergen zu können. In allen Fällen ermittelt der ODL-Generator die Beschreibung der benötigten Schemaversionen durch die API des Schemamanagers.

Für die Beschreibung eines Schemas bzw. einer Schemaversion existieren viele verschiedene, im Ergebnis jedoch äquivalente Möglichkeiten. Beispielsweise kann eine Schemakomponente erst unter einem vorläufigen Namen angelegt und dann umbenannt oder gleich mit dem endgültigen Namen angelegt werden. Desweiteren können Klassen bei **Object** beginnend entsprechend der Vererbungshierarchie von oben nach unten jeweils gleich komplett spezifiziert werden oder Vererbungskanten werden nachträglich eingefügt. Aufgrund der Existenz dieser verschiedenen Varianten wird in [Her99] eine kanonische Darstellung definiert und verwendet, die ein möglichst kurze und leicht lesbare ODL-Beschreibung eines Schemas bzw. einer Schemaversion liefert.

7.2 Die Implementierung des COAST-OODBMS Prototypen

Die Konzepte der Schemaversionierung bedeuten zahlreiche Erweiterungen elementarer Eigenschaften herkömmlicher OODBMS. Eine für die Benutzer transparente Realisierung dieser Erweiterungen erfordert daher Modifikationen am Kern existierender OODBMS [Wöh96], insbesondere am Objekt- und am Schemamanager, die im Rahmen des COAST-Projektes aufgrund fehlender Verfügbarkeit entsprechender Programmquellen nicht für kommerzielle Systeme durchgeführt werden können. Daher wurden Objekt- und Schemamanager des COAST-Prototypen neu entwickelt.

7.2.1 Implementierung des Objektmanagers

7.2.1.1 Die Datenstrukturen für Objekte

Wie bereits erwähnt kann jedes Objekt o jeder Klasse c in Abhängigkeit von den für c gesetzten Propagationsflags in eine beliebige Anzahl von Schemaversionen propagiert werden. Für jede Schemaversion sv, für die $o \in IAS(sv)$ gilt, wird logisch eine Objektversion $sv.o$ von o in dem Typ der Klasse c in Schemaversion sv gespeichert. Wie zuvor bezeichnen wir diese Klassenversion mit $sv.c$. Aufgrund der verzögerten Strategie ist die Zahl der physikalisch gespeicherten Objektversionen stets kleiner oder gleich der Zahl logisch sichtbarer Objektversionen. Die notwendigen Objektversionen werden erst bei Bedarf angelegt bzw. aktualisiert, d.h. wenn ein Objekt durch eine Applikation einer bestimmten Schemaversion zugegriffen wird.

Abbildung 7.4 zeigt die Entwicklung der physikalischen Datenstruktur eines Objektes.[123] Das Objekt enthält den Objektidentifikator (*oid*), den Klassenidentifikator ($class(o)$), die Identität der Erzeugerschemaversion ($csv(o)$), den Erzeugungszeitstempel sowie die Löschungs- und die Objektversionenliste. Die Löschungsliste (*del_list*) beinhaltet die Identitäten derjenigen Schemaversionen, in denen das Objekt gelöscht wurde. Schemaversion sv_4 ist im Beispiel allerdings zu keinem Zeitpunkt in der Löschungsliste, da das Objekt dort nie sichtbar gewesen war. Die Objektversionenliste (*ov_list*) enthält die Objektversionen, welche jeweils aus Schemaversionsidentifikator (*svid*), Zeitstempel (*ovct*), Ursprungsschemaversion ($origin(ov)$) und Objektwert bestehen. Die *Ursprungsschemaversion* einer Objektversion bezeichnet diejenige Schemaversion, in der der dargestellte, logische Objektwert durch eine Applikation gesetzt wurde. Die später durchzuführende Propagation wird einen Objektwert im Propagationsgraphen immer nur von seiner Ursprungsschemaversion weg, aber nie in Richtung zu dieser hin propagieren.

[123] Wir werden dasselbe Objekt in Beispiel 7.2 nochmals betrachten und dort auch die Operationen beschreiben, die zu seiner Entwicklung führten. Abbildung 7.12 zeigt die Entwicklung des Objektes in einer anderen Darstellungsart.

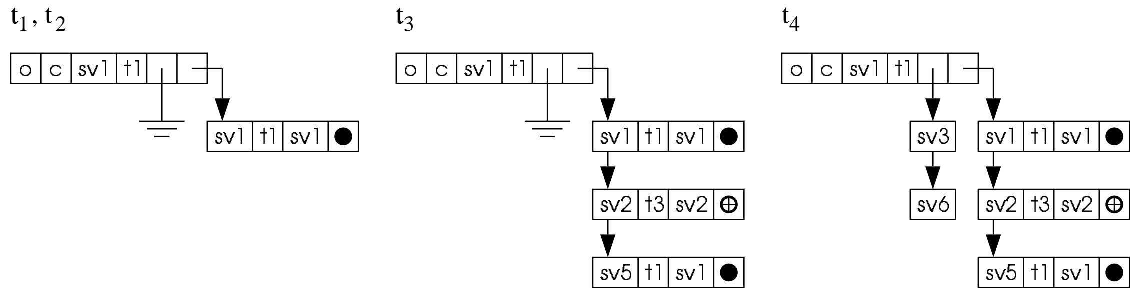


Abbildung 7.4: Eine beispielhafte Entwicklung der physikalischen Speicherstruktur eines Objektes.

Im Gegensatz zur Erzeugerschemaversion eines Objektes ($csv(o)$) bezieht sich die Ursprungsschemaversion ($origin(ov)$) nur auf eine Objektversion. Die Erzeugerschemaversion eines Objektes wird später benötigt werden, um festzustellen, welche Flags zu prüfen sind für die Entscheidung, ob das Objekt überhaupt in einer betrachteten Zielschemaversion sichtbar ist. Dahingegen wird aufgrund der Ursprungsschemaversion einer Objektversion entschieden, ob genau diese eine Objektversion in die Zielschemaversion propagiert werden kann, oder ob damit entgegen der Ausbreitungsrichtung einer Propagation vorgegangen würde, welche stets von der durch eine Applikation veränderten Objektversion weg zu den übrigen Versionen des Objektes hin verläuft.

Zum direkten Zugriff auf die physikalische Repräsentation von Objekten und Objektversionen bietet der Objektmanager u.a. die Methoden **OMcreate**, **OMread**, **OMwrite** und **OMdelete** an. Diese nehmen keinerlei Rücksicht auf ggf. aufgrund der verzögerten Propagation noch vorzunehmende Maßnahmen und dienen daher nur der internen Verwendung innerhalb des COAST-OODBMS. Applikationen müssen stattdessen auf analog benannte Methoden des Propagationsmanagers (siehe Abschnitt 7.2.3) zurückgreifen. Diese verwenden die Methoden des Objektmanagers zur Realisierung der verzögerten Propagation.

Die physikalische Repräsentation eines Objektes hat eine Reihe von Invarianten[124] einzuhalten, die in zweierlei Hinsicht für die in Abschnitt 7.2.3.3 vorzustellenden Zugriffsmethoden **PMcreate**, **PMread**, **PMwrite** und **PMdelete** von Bedeutung sind.

- Einerseits beschreiben die Invarianten Bedingungen, deren Erfüllung durch die physikalischen Objektrepräsentationen garantiert wird. Damit dienen sie als Grundlage für effiziente Implementierungen der erwähnten Zugriffsmethoden und zum Beweis deren Korrektheit.
- Andererseits müssen die ein Objekt verändernden Zugriffsmethoden die physikalische Speicherstruktur in einem Zustand hinterlassen, der die gegebenen Invarianten einhält, so daß auch die folgenden Zugriffe korrekt durchgeführt werden können.

Die in diesem Abschnitt vorzustellenden physikalischen Invarianten betreffen weitestgehend den Objektmanager und können durch diesen sichergestellt werden. In Abschnitt 7.2.3 über die Implementierung des Propagationsmanagers werden wir einige weitere physikalische Invarianten ergänzen.

Invariante 7.1 {Sichtbarkeit}
Wenn in einer Schemaversion sv eine Referenz auf ein Objekt o existiert, dann ist (oder zumindest war) dieses Objekt auch in dieser Schemaversion sichtbar, d.h. es gilt $o \in IAS(sv)$ *(zumindest zu einem früheren Zeitpunkt).*

[124] Diese Invarianten gehen über die beispielsweise in [KM94] als *schema / object consistency constraints* bezeichneten Anforderungen zur Sicherstellung der Verträglichkeit zwischen Objekten und den Klassen, denen diese entstammen, hinaus.

Da wir einen expliziten Löschoperator **PMdelete** zur Verfügung stellen, kann die referenzielle Integrität wie in unversionierten Datenbanksystemen durch Löschung noch referenzierter Objekte zerstört werden. Die Invariante 7.1 besagt allerdings, daß eine Verletzung der referenziellen Integrität nicht durch unseren Propagationsmechanismus verursacht worden sein kann. Dieser verhindert nämlich, daß eine Referenz auf ein Objekt in eine Schemaversion propagiert wird, in die das referenzierte Objekt gar nicht propagiert werden darf. Eine Referenz eines Objektes o' auf ein Objekt o, welches in einer Schemaversion sv nicht sichtbar ist, wird bei der Propagation von o' nach sv auf **nil** gesetzt. Damit erhalten Applikationen einer Schemaversion sv nach Invariante 7.1 nur Referenzen auf solche Objekte o, die auch nach sv propagiert werden können.

Invariante 7.2 {Objektstruktur}
Ein Objekt o einer Klasse c erfüllt alle folgenden Bedingungen zu jeder Zeit:

- *Für jedes Objekt wird zumindest eine Objektversion gespeichert, d.h. $ov_list(o) \neq \emptyset$.*
- *Höchstens eine Version eines Objektes o wird für jede Schemaversion sv gespeichert und jedes $sv.o$ respektiert den Typ der Klassenversion $sv.c$.*
- *Wenn o in einer Schemaversion gelöscht wird, dann muß die entsprechende Version auch aus der Objektversionenliste entfernt werden, d.h. $sv(ov_list(o)) \cap del_list(o) = \emptyset$.[125]*
- *Objektversionen, die denselben logischen Objektwert repräsentieren, enthalten denselben Zeitstempel.*
- *Schemaversionsidentifikator und Ursprungsschemaversion einer Objektversion sind genau dann identisch, wenn der zugehörige Wert durch eine Applikation geschrieben (und nicht durch Propagation ermittelt) wurde.*

7.2.1.2 Klassenextensionen

Der Objektmanager verwaltet für jede Klasse eine *globale Extension*. Eine solche globale Klassenextension enthält alle Objekte ihrer Klasse, die im Zugriffsbereich zumindest einer Schemaversion enthalten sind. Konzeptionell wäre ebenso die Verwendung *lokaler Extensionen* möglich gewesen. Dabei wird für jede Schemaversion sv, die eine Klasse c enthält, eine eigene Extension verwaltet, die lediglich die im Zugriffsbereich $IAS(sv)$ von Schemaversion sv sichtbaren Objekte der Klasse c enthält. Durch die Propagation von Objekten überlappen sich solche lokalen Klassenextensionen genau in den Objekten, die von Applikationen mehrerer Schemaversionen gemeinsam benutzt werden können.

Für die Implementierung wurde das Konzept der globalen Extensionen gewählt, da diese weniger Platz in Anspruch nehmen und sehr viel stabiler sind, d.h. seltener angepaßt werden müssen. Zunächst existiert für jede Klasse nur eine globale Extension, während lokale Extensionen für alle Schemaversionen, in denen die Klasse sichtbar ist, oder zumindest für alle Schemaversionen, in die Objekte der Klasse propagiert werden, erforderlich sind. Da sich lokale Extensionen einer Klasse in den propagierten Objekten überschneiden, benötigen sie zusammen mehr Speicherplatz als eine globale Extension. Des weiteren sind die bei Änderungen (Objekterzeugungen und -löschungen) erforderlichen Anpassungen mit globalen Extensionen leichter durchzuführen.

Bei der Erzeugung eines Objektes muß nur die eine globale Extension der betroffenen Klasse ergänzt werden, während bei der Verwendung lokaler Extensionen mehrere Anpassungen, nämlich genau der Extensionen derjenigen Klassenversionen, in die das neue Objekt propagiert werden

[125] Bei einer gegebenen Liste ov_list von Objektversionen liefert $sv(ov_list)$ die Menge der Schemaversionen, deren Zugriffsbereiche das Objekt enthalten.

kann, durchgeführt werden müßten. Bei der Löschung von Objekten sind stets, wiederum in Abhängigkeit von der Propagation, ein oder mehrere lokale Extensionen anzupassen. Globale Extensionen müssen hingegen nur verändert werden, wenn die Löschung in alle Schemaversionen propagiert wird, wo das Objekt existiert und das Objekt damit vollständig gelöscht wird. Dieser Fall ist jedoch sehr einfach daran zu erkennen, daß die Objektversionenliste des Objektes durch die Löschung einer Objektversion leer wird. Schließlich müssen bei der Ableitung einer neuen Schemaversion keine neuen globalen, wohl aber neue lokale Extensionen angelegt und mit Objektidentitäten gefüllt werden.

Für alle genannten Änderungen (Objekterzeugungen und -löschungen sowie Ableiten neuer Schemaversionen) läßt sich zusammenfassend sagen, daß bei der Verwendung lokaler Extensionen die Vorteile der verzögerten Propagation erheblich eingeschränkt würden. Extensionen stellen redundante, physikalische Datenstrukturen dar, die keinen Sinn hätten, wenn sie nicht stets aktuell wären. Daher müssen die Extensionen bei jeder Änderung sofort angepaßt werden. Dazu ist zwar im Gegensatz zur sofortigen Propagation noch keine Anwendung von Konvertierungsfunktionen auf Objektwerte erforderlich, jedoch muß zumindest die Sichtbarkeit eines Objektes in den Zugriffsbereichen der verschiedenen Schemaversionen jeweils sofort ermittelt werden, was eine deutliche Einschränkung der Vorteile der verzögerten Propagation bedeuten würde. Insbesondere müßten damit wieder Stillstandszeiten bei der Ableitung neuer Schemaversionen in Kauf genommen werden.

Auch hier sind also wieder zwei grundsätzliche Alternativen bei der Vorgehensweise erkennbar. Mit lokalen Extensionen können die im Zugriffsbereich einer Schemaversion sichtbaren Objekte direkt und damit sehr schnell bestimmt werden. Dies entspricht einem lesenden Zugriff. Dafür muß mehr Speicherplatz investiert werden und Änderungsoperationen sind aufwendiger. Dahingegen ist die Sichtbarkeit eines Objektes einer Klasse bei der Verwendung globaler Extensionen algorithmisch und damit zeitaufwendiger zu bestimmen, dafür sind Änderungen schneller zu realisieren und es wird weniger Speicherplatz beansprucht.

Wie in unversionierten Systemen können hier grundsätzlich sowohl *flache Extensionen*, die nur die direkten Instanzen einer Klasse enthalten, als auch *tiefe Extensionen*, die ebenfalls die Objekte der Unterklassen einschließen, verwendet werden. Da sich die vom Schemaentwickler vergebenen Klassennamen genaugenommen nur auf Versionen von Klassen beziehen und wir keine, jeweils für alle Versionen einer Klasse gültigen und im versionierten Gesamtschema eindeutigen Namen haben, setzen wir einen eindeutigen Namen einer globalen Klassenextension aus zwei Komponenten zusammen, nämlich aus dem Namen der Erzeugerschemaversion $csv(c)$ der Klasse c und aus dem Namen der Klassenversion $csv(c).c$. Da Klassen(versions)namen in einer Schemaversion und Schemaversionsnamen in einem Schema eindeutig sind, erhalten wir so global eindeutige Namen für die globalen Klassenextensionen. Diese Namen sind weiterhin unveränderlich, da die Existenz einer Klassenextension eine gefrorene Schemaversion voraussetzt, in der damit keinerlei Umbenennungen mehr möglich sind.

Bei einem Zugriff über eine Schemaversion sv auf eine globale Extension wird vom Propagationsmanager für jedes Objekt der Extension algorithmisch geprüft, ob dieses in der betroffenen Schemaversion sichtbar ist. Nur die sichtbaren Objekte der Extension ($o \in IAS(sv)$) werden an eine Applikation von sv geliefert. Details zur Implementierung der Klassenextensionen finden sich in [Wöl98].

7.2.2 Implementierung des Schemamanagers

Der COAST-Schemamanager bedient sich wie vergleichbare Komponenten anderer Datenbanksysteme eines sog. *Metaschemas*, um einen Zustand eines Datenbankschemas zu speichern und zu verändern. Die an der Schnittstelle des Schemamanagers angeforderten Operationen der COAST-

ODL werden durch die API des Schemamanagers auf die Datenstruktur des Metaschemas abgebildet.

7.2.2.1 Das Metaschema

Das COAST-Metaschema ist ein Teil des Schemamanagers, der die vom Schemaentwickler spezifizierten, versionierten Schemata speichert und verwaltet. Dazu enthält es Klassen, die eine ähnliche Aufgabe wahrnehmen wie die Systemtabellen in relationalen Datenbanksystemen: ihre Instanzen repräsentieren Komponenten benutzerdefinierter Schemata. Eine weitere Ähnlichkeit ist dadurch gegeben, daß das Metaschema in beiden Fällen mit Hilfe der vom jeweiligen Datenmodell angebotenen Konzepte selbst beschrieben wird. Damit kann es mit denselben Methoden zugegriffen werden, die auch für die Zugriffe auf die vom Benutzer eingegebenen Nutzdaten verwendet werden. Änderungen an den Instanzen des Metaschemas sind allerdings nicht direkt möglich, sondern nur über die Methoden des Metaschemas. Diese stellen sicher, daß die durch die Invarianten beschriebene Konsistenz von Schema und Datenbank gewahrt bleibt.

Das Metaschema besteht aus Metaklassen, die die Komponenten benutzerdefinierter Schemata speichern. Dazu dienen die Metaklassen **Schema**, **SV**, **Class**, **CV** und **Attrib** sowie **Method**. Abbildung 7.5 stellt ein erweitertes Entity-Relationship (EER) Diagramm [Che76, TYF86] dar, das neben den genannten Metaklassen auch Klassen des Objekt- und des Propagationsmanagers enthält und damit Querbeziehungen zwischen den Modulen des COAST-Prototypen verdeutlicht. Wir gehen an dieser Stelle jedoch nicht näher auf das Metaschema ein, sondern verweisen auf die detaillierten Beschreibungen der Attribute und Methoden der Metaklassen in [Dol99, Pri98].

7.2.2.2 Alternativen bei der Speicherung des Schemas

Da das Schema die Struktur der Objekte der Datenbank beschreibt und für deren korrekte Interpretation benötigt wird, muß es genau wie die Datenbank selbst persistent gespeichert werden. Dazu bieten sich zunächst drei Alternativen an.

1 Die COAST-ODL enthält neben den Schemaänderungsprimitiven insbesondere auch Anweisungen zur Beschreibung eines Schemazustandes. Wir hatten diese in Abschnitt 5.5.6 unter der Bezeichnung *erzeugende ODL* bereits vorgestellt. Daher liegt es zunächst nahe, diese zu verwenden und das Schema in einer ODL-Textdatei des Betriebssystems abzulegen.

2 Das Schema kann, einem eigenen Format entsprechend, in einer Datei des Betriebssystems abgelegt werden.

3 Da das Schema wie die Nutzdaten in Klassen des Datenbanksystems, eben in den Metaklassen, beschrieben ist und damit dieselben Zugriffsoperationen wie für die Nutzdaten verwendet werden können, besteht eine dritte Alternative darin, das Schema wie eine gewöhnliche Datenbank unter Verwendung der Methoden des Objektmanagers zu speichern. Damit stehen auch bei der Speicherung und Verwaltung von Schemata die Vorteile von ACID-Transaktionen zur Verfügung.

Eine bei der Auswahl einer geeigneten Alternative zu berücksichtigende Randbedingung ist dadurch gegeben, daß die Zuordnung von Objekten und deren Werten zu den Schemaversionen, den Klassen und den Attributen des Schemas nicht über Namen, sondern über Schemaversions-, Klassen- und Eigenschaftsidentifikatoren (*svid*s, *cid*s und *pid*s) erfolgt. Diese sind also mit den

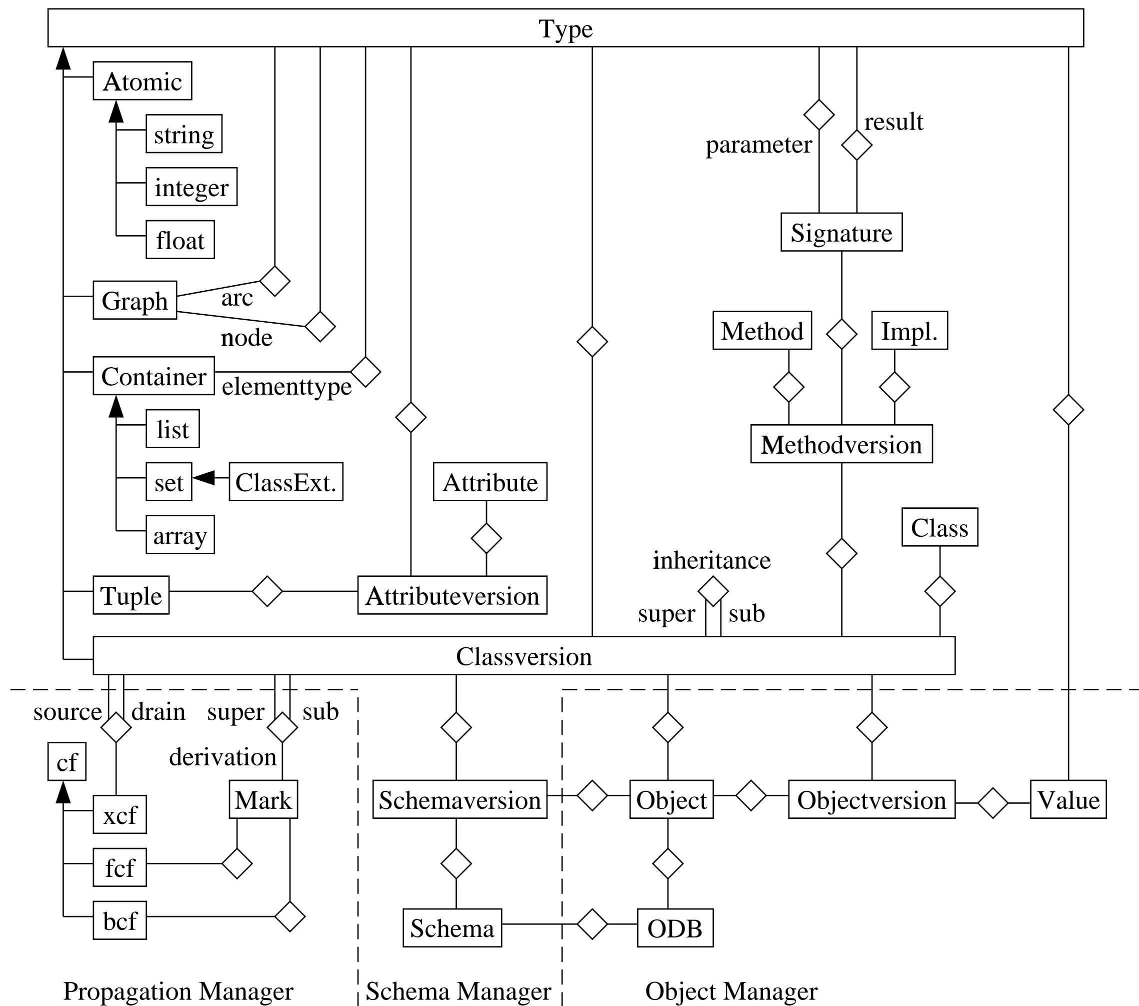


Abbildung 7.5: EER-Diagramm der zentralen Klassen des COAST-Prototypen.

Objekten in der Datenbank gespeichert und müssen, damit sie im Schema nach einem Neustart des Datenbanksystems wieder wie zuvor zur Verfügung stehen, abgespeichert werden. Die Identifikatoren treten jedoch für den Benutzer des Datenbanksystems nicht explizit zutage und sind demzufolge auch nicht in der ODL enthalten. Stattdessen werden sie beim Anlegen von Klassen und ihren Eigenschaften implizit vergeben, wobei es u.a. auf die Reihenfolge der Erzeugungen ankommt. Da sich diese durch Umstellungen der Anweisungen der ODL, insbesondere bei deren Ausführung durch den Parser verändern kann, kommt die erste der oben genannten Alternativen hier nicht in Betracht. Die zweite Alternative käme zwar prinzipiell in Frage, jedoch erscheint die Speicherung des Schemas mit den Methoden des Objektmanagers (Alternative 3) am einfachsten und konzeptionell am geradlinigsten, da sie die Ähnlichkeit zwischen dem Schema als Instanz des Metaschemas und der Datenbank als Instanz des Schemas auch auf Implementierungsebene erhält und somit die Gleichbehandlung unterstreicht. Weitere Details zu den Alternativen bei der Speicherung des Schemas finden sich in [Dol99].

7.2.3 Implementierung des Propagationsmanagers

Der Propagationsmanager realisiert einen wesentlichen Teil der speziellen Funktionalität des hier vorgestellten Schemaversionierungsansatzes und stellt daher das für diese Arbeit bedeutendste Modul des COAST-OODBMS Prototypen dar. Aus diesem Grunde soll er in diesem Abschnitt näher vorgestellt werden. Zur Steigerung der Effizienz wurde eine verzögerte Vorgehensweise

bei der Durchführung der Propagation auf Objektebene gewählt, die zunächst motiviert werden soll (Abschnitt 7.2.3.1). Danach gehen wir auf die transitive Propagation ein (Abschnitt 7.2.3.2) und ergänzen die bereits in Abschnitt 7.2.1.1 vorgestellten Invarianten für physikalische Datenstrukturen, um uns schließlich den Propagationsalgorithmen selbst zuwenden zu können (Abschnitt 7.2.3.3).

7.2.3.1 Verzögerte physikalische Propagation

Die in Kapitel 5 vorgestellten Konzepte bieten Möglichkeiten zur flexiblen und fein granulierten Steuerung der Propagation, wobei in jeder Klasse unterschiedliche Arten von Objektänderungen unabhängig voneinander in andere Schemaversionen propagiert werden können. Logisch, d.h. aus der Sicht der Applikationen, werden alle notwendigen Propagationen immer sofort nach der Änderung eines Objektes oder nach der Ableitung einer neuen Schemaversion durchgeführt. Die sofortige Propagation aller Objektänderungen und die Speicherung aller Objektversionen in den Zugriffsbereichen neu angelegter Schemaversionen würde bei der tatsächlichen physikalischen Durchführung jedoch viel Zeit und Platz in Anspruch nehmen, und zwar umso mehr, je größer die Datenbank ist. Damit verbunden ist das Problem, daß die Datenbank den Applikationen während der Propagation ihrer Objekte in neue Schemaversionen nicht zur Verfügung steht [FMZ94b].

Daher verwenden wir dem technischen Teilziel 3.15 der Effizienz folgend auf der physikalischen Ebene einen verzögert arbeitenden Ansatz. Dieser garantiert insbesondere die Zeitäquivalenz (siehe Abschnitt 4.3.4.2), d.h. aus der Sicht der Applikationen kann die verzögerte physikalische Durchführung nicht von einer sofortigen unterschieden werden. Ein physikalisch gespeicherter Wert eines Objektes kann aufgrund der Verzögerung nun jedoch von dessen logischem Wert abweichen, welchen das Objekt in einer Schemaversion tatsächlich hat und welcher demzufolge bei Zugriffen an Applikationen zu liefern ist.

Die verzögerten Mechanismen propagieren erst dann, wenn eine Applikation auf ein Objekt zugreift und vermeiden damit Stillstandszeiten der Datenbank während der Ableitung neuer Schemaversionen. Weitere Vorteile der verzögerten Propagation sind, daß Konvertierungsfunktionen nur dann ausgeführt werden, wenn dies notwendig ist und daß auch erst dann Speicherplatz belegt wird.

Objekte einer Schemaversion können auch dann noch verändert werden, nachdem Nachfolgerschemaversionen abgeleitet wurden. Eine Änderung eines Objektes in einer anderen Schemaversion kann die Ergebnisse zuvor durchgeführter Konvertierungen ungültig machen. Sei sv_j von sv_i abgeleitet und das Objekt o der Klasse c möge in beiden Schemaversionen sichtbar sein. Ist nun das Modifikationsflag für die Propagation von $sv_i.c$ nach $sv_j.c$ eingeschaltet, dann würden n aufeinanderfolgende Veränderungen von o in sv_i bei einer sofortigen Propagation n Ausführungen der Konvertierungsfunktion $fcf_{c,j \leftarrow i}$ auslösen. Wird o jedoch zwischenzeitlich nicht von einer Applikation von sv_j zugegriffen, dann waren $n-1$ der n Ausführungen der Konvertierungsfunktion überflüssig und können bei Verwendung verzögerter Mechanismen eingespart werden. Daher ist eine möglichst späte Durchführung der Propagation effizient.

Wie bereits dargestellt verwenden wir Objektversionen, um die Werte, die ein Objekt in verschiedenen Schemaversionen hat, zu repräsentieren. Zur Abbildung von Modifikationen einer Objektversion in die das Objekt repräsentierende Datenstruktur sind zahlreiche Varianten denkbar. Die einfachste darunter ist die sofortige Propagation, d.h. wann immer eine Objektversion durch eine Applikation verändert wird, werden für alle Schemaversionen, in denen das Objekt sichtbar ist, neue Werte berechnet und gespeichert. Damit ist die physikalische Repräsentation einer Datenbank zu jedem Zeitpunkt ihrem logischen Zustand, so wie er sich dem Benutzer darstellt, vollkommen identisch. Diese Variante ist einerseits leicht zu realisieren, andererseits aber

bei der Anwendung sehr zeit- und speicherplatzintensiv. Das andere Extrem wäre, Objektveränderungen nur zu protokollieren, sie aber nicht tatsächlich in der Datenbank durchzuführen. Bei einem lesenden Zugriff müßten dann, ähnlich der Wiederherstellung eines konsistenten Datenbankzustandes nach Hard– oder Softwarefehlern (engl. *recovery*), durchgeführte Änderungen anhand von Protokollen zunächst nachgeführt werden, bevor der eigentliche Zugriff stattfindet. Damit würde jeweils nur das sofort getan, was für die spätere Rekonstruktion aller Versionen eines Objektes notwendig ist. Diese Strategie wäre meist weniger speicheraufwendig als die zuerst dargestellte, es könnte jedoch auch passieren, daß veraltete oder gelöschte Werte in Objektversionen nicht entfernt werden, da dies nicht unbedingt erforderlich ist. Während die erstere der obigen Strategien schnelle Lesezugriffe ermöglicht, weil der gesuchte Wert bereits vorhanden ist und nur gelesen werden muß, ist die letztere sicher bei Schreibzugriffen schneller. Eine dritte Variante wäre es, hauptsächlich eine Verringerung des Speicherplatzbedarfes anzustreben, indem keine veralteten Objektversionen aufbewahrt werden. Mit dem genannten Ziel könnten sogar aktuelle Objektversionen gelöscht werden, sofern sie sich durch Propagation rekonstruieren lassen.

Die hier vorzustellende Variante ist durch einige Nebenbedingungen vereinfacht, so daß wir die verwendeten Algorithmen zumindest grob vorstellen können. Die Konsistenzbedingungen für die Datenstrukturen werden in Form einer Menge physikalischer Invarianten vorgestellt.

Wie bereits dargelegt, müssen wir eine komplette, sofortige Propagation der Datenbank zum Zeitpunkt der Ableitung einer neuen Schemaversion vermeiden. Dies würde zuviel Zeit und Platz benötigen. Daher verwenden wir eine verzögerte Strategie, die ein Objekt idealerweise erst dann in eine Schemaversion sv propagiert, wenn eine Applikation von sv darauf zugreift. Wir sagen dann, *der (logische) Wert des Objektes o für Schemaversion sv (formal als sv.o notiert) werde zugegriffen.* Auf der physikalischen Ebene verwenden wir Datenstrukturen, die mehrere Versionen eines Objektes o enthalten können. Die ein Objekt repräsentierende Datenstruktur wächst damit, sobald das Objekt aus einer Schemaversion heraus zugegriffen wird, für die noch keine physikalische Repräsentation in Form einer Objektversion vorliegt. In solchen Fällen muß die korrekte Quellobjektversion (z.B. $sv'.o$) gefunden und nach sv propagiert werden.

Bei Schreibzugriffen auf ein Objekt durch eine Schemaversion sv muß eventuell der bisherige, logische Wert von $sv.o$, unabhängig davon, ob eine aktuelle oder veraltete Objektversion $sv.o$ bereits physikalisch gespeichert war oder nicht, in Vorgänger- und Nachfolgerschemaversionen propagiert werden, die die Veränderung nicht sehen sollen (m-Flag ausgeschaltet).

Die konkrete Zahl von Operationen, die für einen Lese- oder Schreibzugriff durchgeführt werden müssen, hängt von den Invarianten für die physikalische Repräsentation der Objekte und ihrer Versionen ab. Wir hatten bereits verschiedene Strategien hierzu umrissen und angedeutet, daß sich daraus verschiedene physikalische Invarianten ergeben. Bei den Strategien, die einen Großteil der Arbeit bereits beim Schreibzugriff erbringen, ist der physikalische Objektzustand dem logischen ähnlicher. Das schlägt sich in strengeren physikalischen Konsistenzbedingungen und damit auch in leichter zu implementierenden Algorithmen für die Propagation vor den eigentlichen Zugriffen nieder. Man kann grob davon ausgehen, daß die Geschwindigkeit beim Schreiben umsomehr fällt und beim Lesen umsomehr steigt, je mehr Propagationsschritte sofort erledigt werden. Dabei ist der Gewinn auf der einen Seite allerdings nicht proportional dem Verlust auf der anderen, so daß die Wahl der Variante nicht einfach nur aufgrund des zu erwartenden Verhältnisses zwischen Lese- und Schreibzugriffen getroffen werden kann. Die oben erwähnte Einsparung überflüssiger Propagationen im Falle einer gewissen Lokalität der Zugriffe, d.h. wenn mehrere aufeinanderfolgende Zugriffe auf ein Objekt von Applikationen derselben Schemaversion kommen, ist hierbei nämlich nicht berücksichtigt.

Für die Erläuterung der verzögerten Propagation stellen wir zunächst gewisse Überlegungen an und führen einige Begriffe ein.

Wir nennen die Menge der (physikalischen bzw. logischen) Versionen eines Objektes die *(physikalische bzw. logische) Versionenmenge des Objektes* (engl. *version set*).

Um den für die Speicherung der Versionen eines Objektes benötigten Platzbedarf zu reduzieren, führen wir eine Kategorisierung von Objektversionen nach der Notwendigkeit ihrer physikalischen Speicherung ein. Die dabei zugrunde liegende Idee ist ähnlich der von Clamen (siehe Abschnitt 4.5.3.4) auf Attribute angewandten.

Definition 7.1 {überflüssige, ableitbare und benötigte Objektversionen}
Entsprechend der Notwendigkeit zur physikalischen Speicherung des Wertes einer Objektversion sv.o eines Objektes o für eine Schemaversion sv unterscheiden wir die folgenden Fälle.

- *Eine physikalische Objektversion sv.o heißt* überflüssig, *wenn der von ihr repräsentierte logische Objektwert in keiner Schemaversion aktuell ist und sie nicht für die Propagation in eine andere Schemaversion verwendet wird.*
- *Eine physikalische Objektversion sv.o heißt* ableitbar (engl. derivable), *wenn der von ihr repräsentierte logische Objektwert durch Anwendung von Konvertierungsfunktionen auf den Wert einer anderen physikalischen Version desselben Objektes berechnet, also rekonstruiert werden kann.*
- *Eine physikalische Objektversion sv.o heißt* benötigt (engl. required), *wenn sie weder überflüssig noch ableitbar ist. Der von ihr repräsentierte Objektwert ist in sv selbst aktuell oder wird für die Propagation in eine andere Schemaversion benötigt und kann nicht aus einer anderen Objektversion rekonstruiert werden.*

Eine logische Objektversion heißt abhängig, *wenn die zugehörige physikalische Objektversion nicht lokal gespeichert ist,* unabhängig *sonst.*

Die zugehörige, physikalische Objektversion einer abhängigen logischen Objektversion muß ableitbar sein, sonst kann der Objektwert nicht rekonstruiert werden.

Überflüssige Objektversionen sollten eigentlich nicht gespeichert werden. Dies würde allerdings voraussetzen, daß bei jeder Änderung eines Objektwertes alle gespeicherten Objektversionen geprüft werden, was in der Regel zu viel Aufwand bedeutet. Zu Zeiten geringer Last kann eine solche Prüfung gewinnbringend durchgeführt werden. Natürlich kann dann sogar eine sofortige Durchführung aller aufgrund der verzögernden Vorgehensweise bisher noch nicht durchgeführter Propagationen angestoßen werden, so wie dies für den direkten Ansatz zur Schemaevolution von Zicari (siehe Abschnitt 4.3.4.2) vorgeschlagen wird. Im Gegensatz zu der dort behandelten Situation muß beim Schemaversionierungsansatz allerdings beim vollständigen Nachholen aller noch nicht durchgeführter Propagationen zusätzlicher Speicherplatz investiert werden, weil dann i.Allg. auch in solche Schemaversionen propagiert wird, wo das Objekt bisher gar nicht physikalisch gespeichert ist, weil es dort noch von keiner Applikation zugegriffen wurde. Daher wäre hier eine Einschränkung dahingehend angebracht, daß nur bereits physikalisch gespeicherte Objektversionen aktualisiert oder eventuell sogar gelöscht werden, jedoch keine neuen Objektversionen angelegt werden.

Für die Ableitbarkeit einer Objektversion von einer anderen Version desselben Objektes ist notwendige Voraussetzung, daß die beiden Objektversionen denselben logischen Wert repräsentieren, d.h. sie werden in unseren vereinfachenden Diagrammen (siehe z.B. Abbildungen 6.3 und 6.4) mit demselben Symbol dargestellt. Derselbe logische Objektwert ist allerdings keine hinreichende Bedingung für die Ableitbarkeit einer Objektversion. Er kann nämlich dadurch entstanden sein, daß $sv_x.o$ aus $sv_y.o$ abgeleitet wurde oder umgekehrt. Im ersteren Fall ist zwar $sv_x.o$ aus $sv_y.o$ ableitbar, die Umkehrung muß allerdings nicht gelten, weil die Konvertierungsfunktionen in der

Rückrichtung nicht notwendigerweise die Umkehrfunktion der Konvertierungsfunktionen in der Hinrichtung darstellen ($cf_{c,x \leftarrow y} \circ cf_{c,y \leftarrow x}(sv_x.o) = sv_x.o$). oder weil die Flags in der Rückrichtung nicht entsprechend gesetzt sind. Analoges gilt für den zweiten Fall, daß nämlich $sv_y.o$ aus $sv_x.o$ abgeleitet worden war.

Ableitbare Objektversionen stellen somit eine Form der Redundanz dar, wobei Speicherplatz zur Verbesserung der Zugriffsgeschwindigkeit investiert wird. Die Ableitbarkeit ist den obigen Ausführungen zufolge allerdings keine symmetrische Beziehung. Stattdessen ist zwischen der Quelle und dem Ziel der Ableitung zu unterscheiden, wobei die Quellobjektversion benötigt wird und demzufolge nicht gelöscht werden darf.

Lemma 7.1 (Ableitbare physikalische Objektversionen)
Eine physikalische Objektversion $sv.o$ ist ableitbar, wenn eine andere Version $sv'.o$ desselben Objektes physikalisch gespeichert ist, die denselben logischen Objektwert repräsentiert und die nach $sv.o$ propagiert werden kann. Dabei ist weitere Voraussetzung, daß die Propagation zuvor bereits von $sv'.o$ nach $sv.o$ erfolgte oder, wenn die Propagation von $sv.o$ nach $sv'.o$ erfolgte, daß die Konvertierungsfunktionen der Klasse von o zwischen sv und sv' informationserhaltend sind. Wurden $sv.o$ und $sv'.o$ aus einer dritten Schemaversion sv'' propagiert, so muß auch die Konvertierung zwischen sv' und sv'' informationserhaltend sein, damit $sv'.o$ informationserhaltend über sv'' nach $sv.o$ propagiert werden kann.

Neben der in Definition 7.1 gegebenen Unterteilung können wir weiterhin *gemeinsame* Objektversionen ausmachen. Diese liegen dann vor, wenn Konvertierungsfunktionen zwischen zwei Schemaversionen die Umkehrfunktion voneinander darstellen und die Flags so gesetzt sind, daß dieselben Änderungen in beide Objektversionen propagiert werden. Eine zwei oder mehr Schemaversionen gemeinsame Objektversion muß nur einmal gespeichert werden, da ihr Wert in den jeweils anderen Schemaversionen rekonstruiert werden kann.

Ein Objekt o einer Klasse c sei in den Schemaversionen sv_u und sv_v sichtbar, wobei sv_v direkt von sv_u abgeleitet und das Modifikationsflag sowohl für $fcf_{c,v \leftarrow u}$ als auch für $bcf_{c,u \leftarrow v}$ eingeschaltet sei. Wenn von den beiden Objektversionen von o diejenige in $IAS(sv_u)$ (also $sv_u.o$) zuletzt durch eine Applikation geändert wurde, dann ist $sv_v.o$ von $sv_u.o$ ableitbar. Daher genügt es, $sv_u.o$ zu speichern.

Wenn $sv_u.c$ und $sv_v.c$ desweiteren *informationsäquivalent* (engl. *information equivalent*) sind, wenn also die Konvertierungsfunktionen zwischen $sv_u.c$ und $sv_v.c$ informationserhaltend sind (d.h. $bcf_{c,u \leftarrow v} \circ fcf_{c,v \leftarrow u}(sv_u.o) = sv_u.o$ und $fcf_{c,v \leftarrow u} \circ bcf_{c,u \leftarrow v}(sv_v.o) = sv_v.o$), dann können wir sogar statisch entscheiden, beispielsweise alle Objekte der Klasse c stets nur in der sv_u entsprechenden Version zu speichern. Diese Entscheidung kann etwa auf Grundlage der Speicherplatzanforderungen von $sv_u.o$ und $sv_v.o$ oder aufgrund der erwarteten Zugriffshäufigkeiten durch die verschiedenen Schemaversionen getroffen werden. In dem oben beschriebenen, allgemeineren Fall müssen wir jedoch zumindest die zuletzt veränderte Objektversion speichern, d.h. manchmal $sv_u.o$ und manchmal $sv_v.o$. Die Entscheidung kann dann also nicht statisch getroffen werden.

Wenn das Modifikationsflag nur für eine Richtung eingeschaltet ist, etwa für $fcf_{c,v \leftarrow u}$, müssen wir $sv_u.o$ immer speichern. Die Objektversion $sv_v.o$ muß zusätzlich immer dann gespeichert werden, wenn sie nach der letzten Änderung von $sv_u.o$ modifiziert wurde.

Die Strategie mit möglichst später Durchführung von Propagationen und minimalem Speicherplatzbedarf ist vorzuziehen, wenn Zugriffe meist schreibender Natur sind und in viele Schemaversionen propagiert werden müssen. Wenn lesende Zugriffe dominieren und ausreichend Speicherplatz zur Verfügung steht, können natürlich alle logisch sichtbaren Versionen auch physikalisch gespeichert werden. Hier werden wir allerdings auf eine Strategie eingehen, die bevorzugt die Speicherplatzanforderungen reduziert.

Bei einem Schreib- oder Lese- bzw. Löschzugriff genügt es nicht, einen ggf. vorhandenen, physikalisch gespeicherten Wert zu überschreiben bzw. zu löschen, weil davon weitere Objektversionen abhängig sein könnten.

Beispiel 7.1 Bei der Einführung der Propagationsflags ab Abschnitt 6.3.2.1 hatten wir eine vereinfachte Form der Darstellung für die in verschiedenen Schemaversionen sichtbaren logischen Werte eines Objektes eingeführt. Diese soll nun auch hier wieder verwendet werden, allerdings in einer leicht erweiterten Variante (siehe Abbildung 7.6), die uns eine symbolische Darstellung der Speicherstrukturen eines Objektes erlaubt. Ist ein Wert des betrachteten Objektes für eine bestimmte Schemaversion physikalisch gespeichert, so drücken wir dies durch zwei konzentrische Kreise aus. Für einige Schemaversionen in Abbildung 7.6 sind also aufgrund der verzögerten Propagation noch keine physikalischen Werte gespeichert. Während das Füllmuster des inneren Kreises weiterhin den logisch sichtbaren Objektwert darstellt, zeigt der zusätzliche, äußere Ring den tatsächlich physikalisch für die jeweilige Schemaversion gespeicherten Wert an. Wenn der physikalische Wert einer Objektversion deren logischen Wert repräsentiert, so wird der äußere Ring leer dargestellt, ansonsten zeigt sein Füllmuster, welcher Wert physikalisch gespeichert ist.[126] Die Darstellung der in den jeweiligen Abbildungen bedeutsamen Propagationsflags geschieht wiederum an den Kanten der Konvertierungsgraphen.

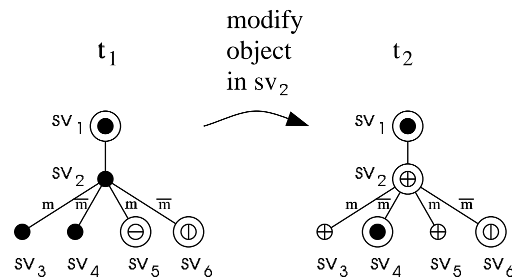


Abbildung 7.6: Veränderungen von Objekten.

Unter der Annahme, daß ausschließlich die Vorwärtspropagation eingeschaltet sei, sind auf der linken Seite von Abbildung 7.6 die Objektversionen $sv_2.o$, $sv_3.o$ und $sv_4.o$ abhängig von $sv_1.o$. Wenn die in Abbildung 7.6 dargestellte Änderung des Objektes in $IAS(sv_2)$ stattfindet, unterscheiden wir die Ausgangssituation entsprechend der folgenden vier Fälle.

(1) abhängige Objektversion mit m-Flag (sv_3)

(2) abhängige Objektversion mit $\overline{m}$-Flag (sv_4)

(3) unabhängige Objektversion mit m-Flag (sv_5)

(4) unabhängige Objektversion mit $\overline{m}$-Flag (sv_6)

Während in Fall 1 nichts unternommen werden muß, ist es in Fall 2 erforderlich, den alten Objektwert nach sv_4 zu propagieren (und dort zu speichern), bevor die Objektversion von sv_2 tatsächlich überschrieben wird. In Fall 3 muß zumindest festgehalten werden, daß der alte, lokal gespeicherte Objektwert von sv_5 nicht mehr aktuell ist, so daß zukünftige Lesezugriffe in sv_5 den korrekten Wert in der Vorgängerversion sv_2 finden. Dies kann durch entsprechend gesetzte Zeitstempel oder durch Löschung des alten Wertes erreicht werden. Da der alte Wert von $sv_5.o$ ($\ominus$) noch in von $sv_5.o$ abhängigen Objektversionen benötigt werden könnte, muß die Propagation rekursiv fortgesetzt werden. In Fall 4 muß nichts unternommen werden.

[126] Erst in Abbildung 7.12 wird sich zum Zeitpunkt t_3 eine Situation ergeben, wo sich logischer und physikalischer Objektwert einer Schemaversion unterscheiden, und zwar in $IAS(sv_1)$.

Schließlich ist der physikalische Objektwert von $sv_{svid}.o$ zu überschreiben oder neu anzulegen, sofern er (wie im Beispiel) bisher nicht existierte.

Im Beispiel wurden $sv_2.o$ und $sv_4.o$ unabhängig, während $sv_3.o$ und $sv_5.o$ nun von $sv_2.o$ abhängen. □

Bei der Betrachtung komplexer Konvertierungsfunktionen hängt die Frage, ob eine bestimmte Objektversion noch benötigt werde, auch von Instanzen anderer Klassen ab und die zweite von Invariante 7.2 gemachte Forderung kann eventuell nicht aufrecht erhalten werden. Damit wird die Implementierung sowohl des Objektmanagers als auch des Propagationsmanagers deutlich aufwendiger. Um dies zu vermeiden bietet es sich ggf. an, komplexe Konvertierungsfunktionen in bestimmten Situationen sofort auszuführen.

7.2.3.2 Transitive Propagation von Löschungen und Veränderungen

In diesem Abschnitt zeigen wir an generellen Beispielen, wie die transitive Propagation von Löschungen und Veränderungen physikalisch abläuft. Dazu nehmen wir in beiden Fällen vereinfachend an, daß nur Propagationsflags in Vorwärtsrichtung eingeschaltet seien.

Wie bereits zuvor stellen wir die Existenz einer physikalischen Objektversion durch einen zweiten Kreis um die logische Objektversion dar. In Vorwärtsrichtung eingeschaltete Propagationsflags sind neben den entsprechenden Ableitungskanten angeschrieben.

7.2.3.2.1 Löschen einer Objektversion

Abbildung 7.7 zeigt einen Klassenpropagationsbaum, der aus 27 Klassenversionen ($sv_1, \ldots, sv_{27}$) besteht. In den folgenden Beispielen betrachten wir wieder ein beliebiges Objekt o.

Wir zeigen nun, wie die Löschung von Objekt o aus Schemaversion sv_4 in der von Abbildung 7.7 dargestellten Situation durchgeführt wird.

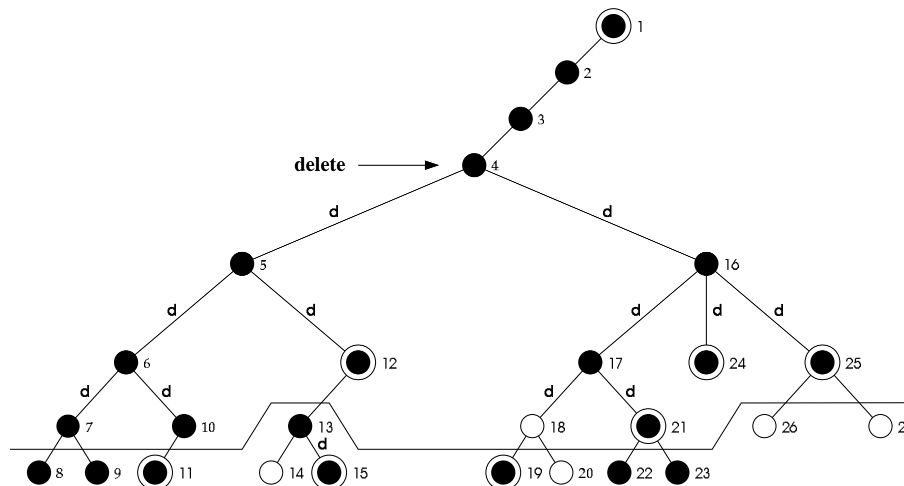


Abbildung 7.7: Der d-Propagationsbaum von $sv_4.c$ vor der Löschung von $sv_4.o$.

Da wir hier nur die Vorwärtspropagation betrachten, bleibt das Objekt o in sv_1, sv_2 und sv_3 sichtbar, es muß jedoch nicht nur aus sv_4, sondern auch aus allen Schemaversionen dessen d-Propagationsbaumes $dst(sv_4.c)$ entfernt werden. Das untere Ende von $dst(sv_4.c)$ ist in den Abbildungen 7.7 und 7.8 durch eine Linie markiert. Nicht alle Schemaversionen des dst müssen o in ihrem Zugriffsbereich haben, aber nur ein „Blatt-Teilbaum“ des dst kann leer sein. In sv_{18} wurde das Objekt bereits zuvor gelöscht.

Bevor wir die Objektversionen des *dst* jedoch löschen, müssen wir sicherstellen, daß jede direkte Nachfolgerversion der Blätter des *dst* ($sv_8.o$, $sv_9.o$, $sv_{11}.o$, $sv_{13}.o$, $sv_{19}.o$, $sv_{20}.o$, $sv_{22}.o$, $sv_{23}.o$, $sv_{26}.o$ und $sv_{27}.o$), die abhängig ist ($sv_8.o$, $sv_9.o$, $sv_{13}.o$, $sv_{22}.o$ und $sv_{23}.o$) die benötigte Objektversion physikalisch speichert. Ansonsten könnten Löcher entstehen und Information verloren gehen.

- Die Objektversionen $sv_8.o$ und $sv_9.o$ sind von $sv_1.o$ abhängig. Daher muß der Wert von $sv_1.o$ via $sv_2, \ldots, sv_7$ nach $sv_8.o$ und $sv_9.o$ propagiert werden.
- Die Schemaversion sv_{11} besitzt bereits eine lokale und aktuelle Kopie des Wertes von o (d.h. $sv_{11}.o$ ist unabhängig) und benötigt daher keine Propagation.
- Die Objektversion $sv_{13}.o$ ist von $sv_{12}.o$ abhängig, d.h. $sv_{12}.o$ muß nach $sv_{13}.o$ propagiert werden, bevor $sv_{12}.o$ gelöscht wird.
- Der für sv_{21} gespeicherte Objektwert muß nach sv_{22} und sv_{23} propagiert werden.
- Die Objektwerte von sv_{24} und sv_{25} müssen nicht propagiert werden, weil keine Nachfolgerschemaversionen von ihnen abhängen. Die Schemaversion sv_{24} hat keine Nachfolgerschemaversionen und die Nachfolgerschemaversionen von sv_{25} haben o nicht in ihren Zugriffsbereichen.

Für indirekte Nachfolgerschemaversionen von Blättern von $dst(sv_4.c)$ (im Beispiel sv_{14} und sv_{15}) muß nichts getan werden, weil die direkten Nachfolgerschemaversionen (im Beispiel sv_{13}) den Objektwert lokal speichern, sofern er benötigt wird (Löcher können nicht entstehen).

Schließlich können alle in Knoten von $dst(sv_4.c)$ gespeicherten Objektwerte gelöscht werden. Das Resultat ist in Abbildung 7.8 dargestellt.

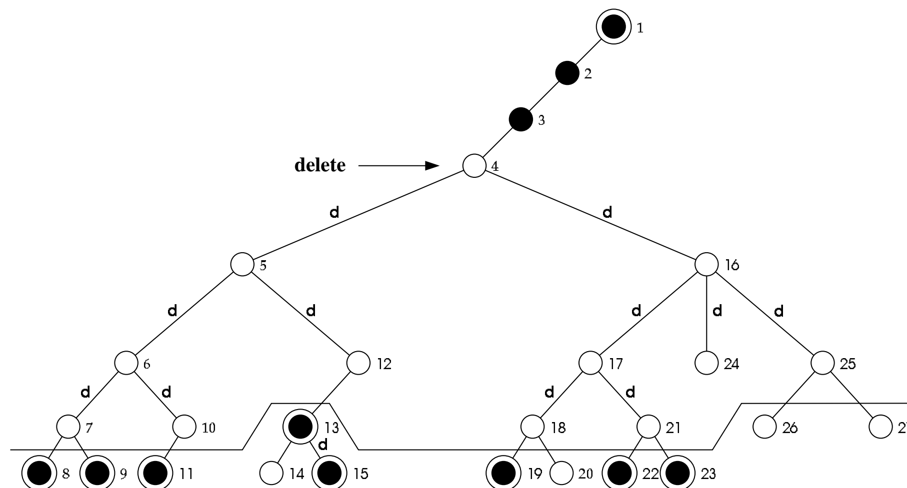


Abbildung 7.8: Der d-Propagationsbaum von $sv_4.c$ nach der Löschung von $sv_4.o$.

Der Algorithmus zum Löschen von o in sv_4 traversiert den d-Propagationsbaum *dst* entsprechend der Tiefensuche (engl. *depth first order*); d.h. im Beispiel werden die Schemaversionen entsprechend ihrer Nummer (4 bis 27, außer 14 und 15) besucht.

In den Blättern des *dst* müssen jeweils die direkten Nachfolgerschemaversionen geprüft werden. Für jede dieser Schemaversionen sv, deren Objektversion von einer anderen Objektversion abhängt, welche in *dst* enthalten sein könnte (im Falle von sv_{13} im Beispiel) oder oberhalb (sv_8 im Beispiel) müssen wir das Objekt von der Ursprungsschemaversion $origin(ov)$ konvertieren und lokal in sv speichern.

Während der Tiefensuche müssen lokal existierende Objektversionen (falls vorhanden) nach Durchführung ggf. notwendiger Propagationsschritte gelöscht werden.

Die Konvertierung kann prinzipiell auf drei verschiedene Weisen durchgeführt werden:

- Der Objektwert wird beim Abstieg immer mitkonvertiert und steht dann sofort zur Verfügung, falls er gebraucht werden sollte. Die Konvertierung wird allerdings nicht immer benötigt (z.B. die Konvertierung von sv_5 nach sv_{12} oder von sv_{17} nach sv_{18} oder nach sv_{21}), weil keine Nachfolgerschemaversion des *dst* den berechneten Objektwert benötigt.
- Der Objektwert wird erst dann gelesen und konvertiert, wenn er in einer Nachfolgerschemaversion der Blätter des *dst* tatsächlich benötigt wird. Dabei werden jedoch einige Konvertierungen mehrfach gemacht (im Beispiel werden die Konvertierungsfunktionen $cf_{c,7\leftarrow 6}$, $cf_{c,6\leftarrow 5}$, ..., $cf_{c,2\leftarrow 1}$ jeweils doppelt ausgeführt, nämlich je einmal für sv_8 und für sv_9).
- Die vorangegangene Lösung kann verbessert werden, wenn die Ergebnisse der einzelnen Konvertierungsschritte temporär (bis die Rekursion beim Verlassen der Schachteln wieder denselben Knoten passiert) gespeichert werden. Im Beispiel wäre im Falle von sv_9 der durch sv_7 sichtbare Objektwert bei der Ausführung der Konvertierung für sv_8 (als Zwischenergebnis) temporär gespeichert worden, so daß die Konvertierungsfunktionen $cf_{c,7\leftarrow 6}$, ..., $cf_{c,2\leftarrow 1}$ nur einmal ausgeführt werden müssen.

Der später vorzustellende Algorithmus realisiert die zuletzt genannte Alternative.

7.2.3.2.2 Verändern einer Objektversion

Wir zeigen nun wie die Veränderung von Objekt o aus Schemaversion sv_4 in der von Abbildung 7.9 dargestellten Situation durchgeführt wird. Werte müssen dabei lediglich bis zu den Blättern des m-Propagationsbaumes $mst(sv_4.c)$ propagiert werden. Das untere Ende von $mst(sv_4.c)$ ist in den Abbildungen 7.9 und 7.10 durch eine Linie markiert.

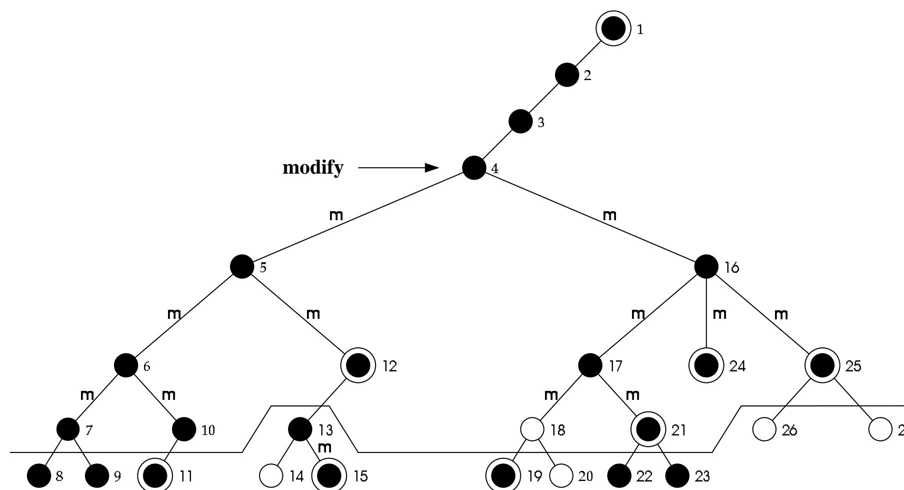


Abbildung 7.9: Der m-Propagationsbaum von $sv_4.c$ vor der Veränderung von $sv_4.o$.

Hierbei ist zu beachten, daß in einer Schemaversion sv aus $mst(sv.c)$, die das Objekt o nicht in ihrem Zugriffsbereich hat, nichts verändert werden muß, da wir uns bei der Festlegung der Semantik des Modifikationsflags entschieden haben, Objekte (z.B. in sv_{18}) in solchen Fällen nicht erneut zu erzeugen. Daher wird die Änderung nicht in alle Nachfolgerschemaversionen von sv propagiert, auch wenn diese zu $mst(sv.c)$ gehören, da keine Löcher traversiert werden. Aus

diesem Blickwinkel könnte man sagen, daß Modifikationen nicht so transitiv sind wie Löschungen, da sie an Löchern enden.

Abbildung 7.10 zeigt das physikalische Ergebnis der Veränderung.

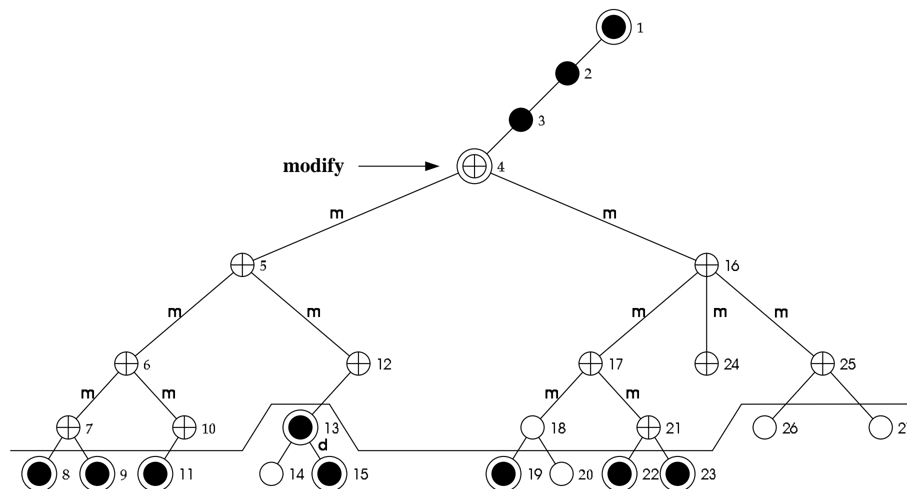


Abbildung 7.10: Der m-Propagationsbaum von $sv_4.c$ nach der Veränderung von $sv_4.o$.

7.2.3.3 Die Objektpropagationsalgorithmen

In diesem Abschnitt werden wir einige Details derjenigen Algorithmen vorstellen, die der Propagationsmanager zur Realisierung der verzögerten Propagation einsetzt. Die entsprechenden Methoden des Propagationsmanagers stellen wir im folgenden als Pseudo-Code dar, wobei wir u.a. darauf verzichten, Variablen zu deklarieren oder die Typen von Ein- und Rückgabeparametern formal anzugeben.

Ein Objekt gehört zwar immer derselben Klasse an, es kann jedoch in zahlreichen Versionen vorliegen, deren Typen den verschiedenen Versionen der Klasse des Objektes entsprechen. Aufgrund der verzögert arbeitenden Algorithmen wurden in einem Objekt i.Allg. nicht alle zuvor logisch durchgeführten Propagationen auch bereits physikalisch realisiert, wenn eine Applikation auf dieses Objekt zugreift. Daher müssen Applikationen zum Lesen (**PMread**), Schreiben (**PMwrite**) und Löschen (**PMdelete**) von Objekten (d.h. von Objektversionen) spezielle Methoden verwenden. Diese stellen sicher, daß alle von den in einem zugegriffenen Objekt bisher nicht physikalisch durchgeführten, nun aber notwendig werdenden Propagationsschritte nachgeholt werden, bevor tatsächlich auf eine der Objektversionen lesend, ändernd oder löschend zugegriffen wird. Die Durchführung von Propagationsschritten kann zum einen dadurch notwendig werden, daß ansonsten ein falscher Objektwert geliefert würde (beim Lesen) oder zum anderen auch dadurch, daß ansonsten später für die Propagation in andere Objektversionen benötigte Werte überschrieben würden und damit nicht mehr zur Verfügung stünden (beim Schreiben und Löschen).

Wir stellen zunächst zwei elementare Methoden vor (Abschnitt 7.2.3.3.1), bevor wir auf die eigentlichen, von den Applikationen direkt zu benutzenden Zugriffsmethoden **PMcreate**, **PMread**, **PMwrite** und **PMdelete** eingehen (Abschnitt 7.2.3.3.2). Schließlich beschreiben wir die Methode **PMpropagate**, die die eigentliche Propagation durchführt (Abschnitt 7.2.3.3.3).

7.2.3.3.1 Grundlegende Methoden

- **PM_convert** (sv_source, sv_drain, o)

Diese Methode führt die Konvertierung einer Objektversion der Klasse c von Schemaversion sv_{source} nach Schemaversion sv_{drain} entlang des ermittelten Pfades (1) schrittweise durch, indem sie die entsprechenden Konvertierungsfunktionen ($cf_{c,drain \leftarrow source}$) ausführt (2), und liefert den berechneten Objektwert zurück (3).

```
    method PM_convert (sv_source, sv_drain, o) {
       // liefert einen Objektwert (ovalue) zurück
       ovalue = OMcreateOv (sv_source, o);
(1)    (sv_1, ..., sv_n) = SMgetPath (csv (o), sv_drain, class (o))
       ∀i = 1, ..., (n − 1) {
            sv_source = sv_i;
            sv_drain = sv_{i+1};
(2)         ovalue = cf_{c,drain←source}(ovalue)
            }
(3)    return (ovalue)
       }
```

Die Methode **PM_convert**.

- **PM_o_visible** (sv_drain, o)

  Diese Methode prüft, ob das Objekt o in der Schemaversion sv_{drain} sichtbar ist. Dazu darf das Objekt zum einen in Schemaversion sv nicht gelöscht worden sein (1), zum anderen müssen die Propagationsflags auf dem Pfad von der Erzeugerschemaversion von o zu sv_{drain} (2) entsprechend gesetzt sein. Dabei kommt es auf die Richtung der Propagation (3) und auf die zeitliche Reihenfolge zwischen der Erzeugung des Objektes o und der der Schemaversion sv an (4).

```
     method PM_o_visible (sv_drain, o) {
        // liefert einen Wahrheitswert (boolean) zurück
        if sv_drain ∈ deletion_list (o)
(1)         then return (false)
            else {
(2)             (sv_1, ..., sv_n) = SMgetPath (csv (o), sv_drain, class (o))
                ∀i = 1, ..., (n − 1) {
                    sv_source = sv_i;
                    sv_drain = sv_{i+1};
(3)                 if sv_source ∈ dsucc (sv_drain) {
                        if c-flag ∉ cpo (source, drain, c)
                            then return (false)
                        }
                    else {
(4)                     if oct (o) > dt (sv_drain) AND
(4)                         c-flag ∉ cpo (source, drain, c)
(4)                             then return (false);
(4)                     if oct (o) < dt (sv_drain) AND
(4)                         s-flag ∉ cpo (source, drain ,c)
(4)                             then return (false);
                        }
                    }
                }
        return (true);
        }
```

Die Methode **PM_o_visible**.

7.2.3.3.2 Die Zugriffsmethoden

Die Methode **PMcreate** legt ein neues Objekt der Klasse *c* in Schemaversion *sv* an und initialisiert es mit dem übergebenen Wert.

```
method PMcreate (sv, c, ovalue) {
   // liefert ein Objekt (object, o) zurück
   o := OMcreate (sv, c); // reserviert Speicherplatz und oid für das neue Objekt
   OMwrite (sv, o, ovalue);
   return (o);
   }
```

Die Methode **PMcreate**.

Beim lesenden Zugriff auf eine Objektversion muß zunächst deren Sichtbarkeit geprüft werden (1). Ist diese gegeben, so propagiert **PMread** ggf. zunächst einen veralteten, aber physikalisch vorhandenen Wert, sofern dieser in einer anderen Schemaversion noch benötigt wird (2). Die dazu benutzte Methode **PMpropagate** werden wir in Abschnitt 7.2.3.3.3 näher vorstellen. Daraufhin wird der aktuelle Wert von *sv.o* ermittelt (3), physikalisch gespeichert (4) und schließlich zurückgegeben (5).

```
    method PMread (sv, o) {
       // liefert einen Objektwert (ovalue) zurück
(1)    if PM_o_visible (sv, o) {
(2)        PMpropagate (sv, o, ∅);
(3)        ovalue = PMread_direct (sv, o);
(4)        OMwrite (sv, o, ovalue)
(5)        return (ovalue);
           }
       else ERROR[127]("Objekt o nicht im Zugriffsbereich von sv")
       }
```

Die Methode **PMread**.

Der von uns relativ häufig erwartete Fall lokalen Zugriffsverhaltens (mehrere sequentielle Zugriffe auf ein Objekt kommen von Applikationen derselben Schemaversion) könnte zur Verringerung der Zugriffszeit speziell behandelt werden. Dazu wäre **PMread** vor (1) um zwei schnell durchführbare Prüfungen zu erweitern. Zum einen müßte festgestellt werden, ob eine Objektversion *sv.o* für Schemaversion *sv* gespeichert ist und zum anderen, ob diese den aktuellsten Zeitstempel aller Objektversionen von *o* hat.[128] Sind beide Bedingungen erfüllt, so kann der gespeicherte Wert von *o* ohne weitere Schritte zurückgeliefert werden, weil es sich dabei zumindest um den zweiten hintereinanderfolgenden Zugriff auf *o* aus *sv* handelt.

Die Methode **PMread_direct** wird von **PMread** benutzt, um den aktuellen Wert einer Objektversion *sv.o* zu ermitteln, ohne die physikalische Datenstruktur des Objektes *o* zu verändern. Würden solche Veränderungen nämlich gemacht, so zöge das die Aktualisierung zahlreicher Objektversionen nach sich, was den von unserem Propagationsmechanismus erreichten Grad an Verzögerung erheblich reduzieren würde.

Existiert für eine Schemaversion *sv* eine physikalisch gespeicherte Objektversion *ov* (1), die aktuell ist (2), so kann diese direkt verwendet werden (3). Ansonsten muß unter den gespeicherten

[127] Wir verwenden hier die Methode **ERROR** um anzudeuten, daß der beschriebene Fehler aufgetreten ist. Auf die technische Realisierung einer solchen Methode gehen wir hier nicht ein.

[128] Zur Ermittlung des maximalen Zeitstempels aller Versionen eines Objektes könnten diese nach Zeitstempeln sortiert abgelegt oder das jeweilige Maximum im Objekt *o* selbst gespeichert werden.

Objektwerten der aktuellste gefunden werden, der nach *sv* propagiert werden kann (4). Auf die dazu verwendete Methode **PM_FindOv** gehen wir hier allerdings nicht näher ein. Sie ist in [Eig97] ausführlich beschrieben.

```
    method PMread_direct (sv, o) {
       // liefert einen Objektwert (ovalue) zurück
       ovalue = OMread (sv, o);
(1)    if (ovalue != nil) {
(2)        if (ovct (sv.o) == t_max (o))t_max liefert den maximalen Zeitstempel,
                                        der für das Objekt o existiert
(3)            then return ovalue;
(4)            else return PM_FindOv (sv, o, t_max (o));
           }
(4)    else return PM_FindOv (sv, o, t_max (o));
       }
```

Die Methode **PMread_direct**.

Wie beim lesenden, so muß auch beim schreibenden Zugriff auf eine Objektversion zunächst propagiert werden (1), bevor der zu schreibende Wert in die physikalische Datenstruktur eingetragen werden kann (2), wobei ein zuvor bereits existierender Objektwert überschrieben wird.

```
    method PMwrite (sv, o, ovalue) {
       if PM_o_visible (sv, o) {
(1)        PMpropagate (sv, o, ∅);
(2)        OMwrite (sv, o, ovalue);
           }
       else ERROR^127 ("Objekt o nicht im Zugriffsbereich von sv")
       }
```

Die Methode **PMwrite**.

Das Löschen einer Objektversion *sv.o* erstreckt sich logisch auf den gesamten *d*-Propagationsbaum von *sv*. Da sich der Speicherplatzbedarf einer physikalischen Datenbank beim Löschen gespeicherter Objektversionen reduziert, propagieren wir das Löschen einer Objektversion — im Gegensatz zu deren Veränderung durch **PMwrite** — nicht nur logisch, sondern auch physikalisch sofort. Ein weiterer Grund für diese unterschiedliche Vorgehensweise ist, daß bei der Propagation von Löschungen keine Konvertierungsfunktionen auszuführen sind, was die benötigte Ausführungszeit verkürzt.

Invariante 7.3 {**Löschung**}
Ein Objekt ist in keiner Version eines d-Propagationsbaumes mehr physikalisch gespeichert, nachdem es in dessen Wurzel gelöscht wurde.

```
    method PMdelete (sv, o) {
       if PM_o_visible (sv, o) {
           PMdelete_direct (sv, o, ∅);
           }
       else ERROR^127 ("Objekt o nicht im Zugriffsbereich von sv")
       }
```

Die Methode **PMdelete**.

Die Methode **PMdelete_direct** führt die rekursive Propagation einer Löschung durch. Auch hier muß der bisherige Objektwert zunächst propagiert werden (1), da er von anderen Schemaversionen, die nicht im *d*-Propagationsbaum von *sv* enthalten sind, noch benötigt werden könnte. Nach der tatsächlichen Löschung der physikalischen Objektversion *sv.o* durch den Objektmanager (2) wird die Löschung in die anderen Schemaversionen des *d*-Propagationsbaumes von *sv* propagiert (3). Um die Propagation nur in Richtung von *sv* weg durchzuführen und weiterhin die Terminierung der rekursiven Methode garantieren zu können, hat **PMdelete_direct** den zusätzlichen Parameter **sv_visited**, der stets die bereits besuchten Schemaversionen enthält.

```
 method PMdelete_direct (sv, o, sv_visited) {
(1) PMpropagate (sv, o, ∅);
(2) OMdelete (sv, o)
(3) ∀ sv' ∈ (dsucc(sv) ∪ dpred(sv)) - sv_visited {
(3)     if d-flag ∈ cpo (sv, sv', c) then PMdelete_direct (sv', o, sv_visited ∪ {sv})
        }
    }
```

Die Methode **PMdelete_direct**.

7.2.3.3.3 Die Propagation

PMread, **PMwrite** und **PMdelete** nutzen die Methode **PMpropagate**, die sicherstellt, daß die Objektversion *sv.o* physikalisch aktuell ist und durch den Objektmanager direkt gelesen, geschrieben und gelöscht werden kann, ohne daß ggf. später noch benötigte Informationen verloren gehen. **PMwrite** und **PMdelete** können auf ähnliche Art und Weise behandelt werden, da das Schreiben und Löschen einer Objektversion dahingehend denselben Effekt haben, daß der alte Wert der Objektversion anschließend verloren ist. **PMread** ermittelt aufgrund der vorliegenden Versionen eines Objektes den aktuellen Wert der zu lesenden Objektversion *sv.o* und speichert diesen für zukünftige Zugriffe auch in *sv.o* ab, wobei ein ggf. bereits zuvor existierender, inzwischen für *sv* veralteter Wert genau wie bei **PMwrite** und **PMdelete** überschrieben wird. Daher müssen alle drei genannten Methoden (durch Aufruf von **PMpropagate**) zunächst sicherstellen, daß der alte Wert von *sv.o* in die Zugriffsbereiche aller Vorgänger- und Nachfolgerversionen von *sv* propagiert wird, wo er eventuell noch benötigt werden könnte. Das Propagieren eines Objektwertes zu einer Zielschemaversion überschreibt jedoch dort ggf. vorhandene, in Zugriffsbereichen dritter Schemaversionen noch benötigte Objektwerte, so daß sich die Propagation rekursiv fortsetzen muß. Der dritte Parameter der Methode **PMpropagate** wird benutzt, um zu speichern, welche Schemaversionen im Laufe der Rekursion bereits besucht wurden, so daß zum einen Werte nicht entgegen der korrekten Richtung wieder zurückpropagiert werden und daß zum zweiten die Rekursion terminiert.

Die Methode **PMpropagate** ist vergleichsweise kompliziert, da sie zahlreiche Fälle zu unterscheiden hat:

- Die zu propagierende Objektversion (*source object version*) kann oder kann nicht physikalisch gespeichert sein.
- Die Objektversion einer Vorgänger- oder Nachfolgerschemaversion, in die ggf. propagiert werden muß (*drain object version*), kann oder kann nicht physikalisch gespeichert sein.
- Wenn sowohl Quell- als auch Zielobjektversion physikalisch gespeichert sind, kann der Zeitstempel der Quellobjektversion älter, gleich alt oder neuer als der der Zielobjektversion sein.

- Die verschiedenen Flags für die Propagation von der Quell- in die Zielklassenversion können ein- oder ausgeschaltet sein.

- Die Quellklassenversion kann ein Vorgänger oder ein Nachfolger der Zielklassenversion im Klassenkonvertierungsgraphen der Klasse c sein, d.h. es ist entsprechend eine Rückwärts- oder eine Vorwärtskonvertierungsfunktion zu verwenden (siehe zu nutzende Konvertierungsfunktion in Abbildung 7.11).

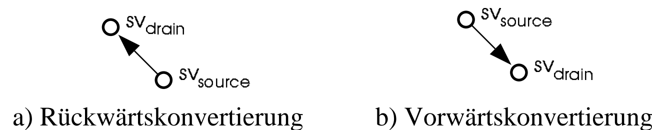


Abbildung 7.11: Zwei wichtige Fälle, die zu unterscheiden sind.

- Die Erzeugerschemaversion eines Objektes, d.h. diejenige Schemaversion in der das Objekt ursprünglich angelegt wurde, kann sich im Klassenkonvertierungsgraphen der Klasse c oberhalb, unterhalb oder neben Quell- und Zielschemaversion befinden.

Wir ergänzen die physikalischen Invarianten, insbesondere Invariante 7.2 hier um eine Bedingung, die nicht vom Objektmanager, sondern nur vom Propagationsmanager sichergestellt werden kann.

Invariante 7.4 {Lokalität}
Der Wert eines Objektes kann nur in solchen Schemaversionen gespeichert werden, deren Zugriffsbereiche das Objekt enthalten.

Invariante 7.4 drückt aus, daß ein Objekt o in einer Schemaversion sv sichtbar ist, wenn ein Wert von o für sv existiert. Die Invariante besagt allerdings nicht, daß der gespeicherte Objektwert auch der aktuell korrekte für Schemaversion sv sein muß. Wir können lediglich festhalten, daß der physikalisch gespeicherte Objektwert mit einem früheren logischen Objektwert derselben Schemaversion übereinstimmt. Weiterhin wird keine Aussage darüber gemacht, ob der gespeicherte Objektwert von einer anderen Schemaversion benötigt wird, oder nicht. Wenn er noch gespeichert ist so kann das auch daran liegen, daß unsere Propagationsalgorithmen aufgrund ihrer verzögerten Vorgehensweise seine Überflüssigkeit nicht feststellen können. Dies bringt natürlich einen Nachteil bezüglich des Speicherplatzbedarfes mit sich, der allerdings durch den hohen Grad an Verzögerung bedingt ist und der durch die Vorteile der Verzögerung zumindest aufgewogen wird.

Wir hatten in Abschnitt 7.2.1.1 bereits vorweggenommen, daß niemals mehr physikalische Versionen eines Objektes gespeichert werden, als logisch sichtbar sind. Diese Aussage wird nun durch die obige Invariante 7.4 belegt.

Die im folgenden zunächst dargestelle Methode **PMprop_conv_write** wird mehrmals von **PMpropagate** aufgerufen. Sie konvertiert eine Objektversion unter Benutzung der angegebenen Konvertierungsfunktion und speichert das Ergebnis in der Zielobjektversion. Ein dort bereits gespeicherter Objektwert muß ggf. vor dem Überschreiben selbst propagiert werden, weswegen **PMprop_conv_write** rekursiv **PMpropagate** aufruft.

```
method PMprop_conv_write (sv_source, sv_drain, sv_visited, o) {
   PMpropagate (sv_drain, o, sv_visited);
   OMwrite (sv_drain, o, PM_convert (sv_source, sv_drain, o));
   }
```

Die Methode **PMprop_conv_write**.

Wir werden nun näher auf die Methode **PMpropagate** eingehen. Der interessanteste Aspekt dabei ist die Untersuchung des aktuellen Zustandes eines Objektes bezüglich des Anteiles der Propagation, der aufgrund der verzögerten Strategie zu einem gewissen Zeitpunkt noch nicht durchgeführt wurde. Zur Verbesserung der Zugriffsgeschwindigkeit werden hierbei zunächst solche Informationen ausgewertet, die sofort aus der physikalischen Darstellung eines Objektes entnommen werden können. Dazu gehört, ob eine Objektversion physikalisch gespeichert ist, ob sie bereits gelöscht wurde, etc.

Beispiel 7.2 Abbildung 7.12 zeigt ein Beispiel für die Entwicklung eines Objektes sowohl bezüglich seiner logischen als auch bezüglich seiner physikalischen Objektversionen. Es handelt sich dabei um dasselbe Objekt, dessen physikalische Speicherstruktur wir bereits in Abbildung 7.4 dargestellt hatten. Der Einfachheit halber betrachten wir wieder nur eine einzelne Klasse c, die in allen existierenden Schemaversionen enthalten sein soll. Die Propagationsflags für die Vorwärts- und Rückwärtskonvertierung sind nahe der jeweiligen Zielschemaversion dargestellt, d.h. für die Vorwärtskonvertierungsfunktion $fcf_{c,3\leftarrow 1}$ sind c- und m-Flag eingeschaltet, für die entsprechende Rückwärtskonvertierungsfunktion $bcf_{c,1\leftarrow 3}$ ist das d-Flag ausgeschaltet ($\overline{d}$). In der Abbildung sind nur die für das Beispiel bedeutungsvollen Flags dargestellt.

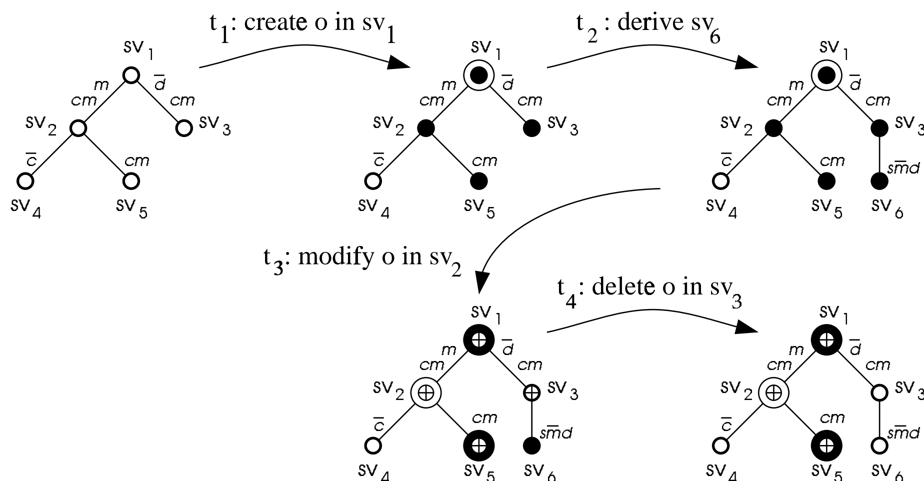


Abbildung 7.12: Beispiel der Entwicklung eines Objektes bezüglich seines logischen und seines physikalischen Wertes.

Zum Zeitpunkt t_1 wird das Objekt o in Schemaversion sv_1 erzeugt und wird damit auch in jeder anderen Schemaversion sichtbar (dargestellt als ausgefüllter Kreis •) außer in sv_4, wo das c-Flag ausgeschaltet ist. Ein physikalischer Wert wird dabei zunächst jedoch ausschließlich für sv_1 gespeichert (dargestellt als äußerer Ring um •).

Zum Zeitpunkt t_2 wird eine neue Schemaversion sv_6 von sv_3 abgeleitet, die das s-Flag für die Klasse c eingeschaltet hat. Daher wird das Objekt o auch in sv_6 sichtbar. Wie in Abbildung 7.12 ersichtlich, muß zu $dt(sv_6) = t_2$, also während der Ableitung von sv_6, keine physikalische Objektversion angelegt werden, obwohl das Objekt nun auch in der neuen Schemaversion sichtbar ist.

Zum Zeitpunkt t_3 wird das Objekt o durch eine Applikation von sv_2 modifiziert. Der neue Wert (dargestellt als $\oplus$) muß im Zugriffsbereich $IAS(sv_2)$ von sv_2 gespeichert werden und wird nach sv_1, nach sv_3 und nach sv_5 propagiert, weil das m-Flag für $bcf_{c,1\leftarrow 2}$, $fcf_{c,3\leftarrow 1}$ und für $fcf_{c,5\leftarrow 2}$ eingeschaltet ist. Da das m-Flag für $fcf_{c,6\leftarrow 3}$ ausgeschaltet ist, behält sv_6 den alten Objektwert. Hierbei ist insbesondere bemerkenswert, daß in sv_1 nun der veränderte Objektwert sichtbar ($\oplus$ im inneren Kreis) ist, aber der ursprüngliche Wert aufgrund der verzögerten Propagation weiterhin physikalisch (ausgefüllter äußerer Ring) gespeichert bleibt.

Schließlich wird das Objekt o zum Zeitpunkt t_4 in sv_3 gelöscht und diese Löschung wird vorwärts nach sv_6 propagiert, aber nicht rückwärts nach sv_1. □

Wir betrachten nun die Implementierung der Methode **PMpropagate**. Die im Quellcode markierten Abschnitte (1) bis (7) werden im Anschluß näher erläutert. Zur Unterscheidung von Vorwärts- und Rückwärtsflags verwenden wir hier die Symbole fs, fc, fm bzw. bc und bm anstelle der oben verwendeten, genaueren Darstellung $flag \in cpo(sv_{source}, sv_{drain}, c)$ ($flag \in \{fs, fc, fm\}$ für $fcf_{c,drain\leftarrow source}$) bzw. $flag \in cpo(sv_{source}, sv_{drain}, c)$ ($flag \in \{bc, bm\}$ für $bcf_{c,drain\leftarrow source}$).

Innerhalb der Abschnitte (3) bis (7) wird jeweils die oben erwähnte Methode **PMprop&conv&-write** aufgerufen. Vor jedem dieser Aufrufe muß sichergestellt werden, daß in die richtige Richtung propagiert wird. Sei beispielsweise sv_3 von sv_2 abgeleitet und diese sei wiederum von sv_1 abgeleitet worden (siehe Abbildung 7.13). Weiterhin sei ein Objekt in sv_1 angelegt und damit auch nach sv_2 und sv_3 propagiert worden. Dieses Objekt sei jedoch bisher nicht in $IAS(sv_2)$ gespeichert, da es noch von keiner Applikation von sv_2 benutzt wurde. In dieser Situation darf die Objektversion $sv_3.o$ nach einer Änderung oder Löschung von $sv_3.o$ nicht von sv_3 nach sv_2 zurückpropagiert werden, weil $sv_2.o' = bcf_{c,2\leftarrow 3}(sv_3.o) = bcf_{c,2\leftarrow 3} \circ fcf_{c,3\leftarrow 2} \circ fcf_{c,2\leftarrow 1}(sv_1.o)$ nicht notwendigerweise mit der korrekten Objektversion $sv_2.o = fcf_{c,2\leftarrow 1}(sv_1.o)$ identisch sein muß.

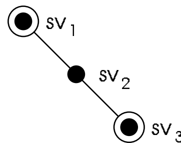


Abbildung 7.13: Bei einer Änderung von $sv_3.o$ darf der alte Objektwert ($\bullet$) nicht nach sv_2 propagiert werden, da dieser Wert von $csv(sv_3.) = sv_1$ kommt.

```
 method PMpropagate (sv_source, o, sv_visited) {
    sv_visited += sv_source; // um die Terminierung der Rekursion sicherzustellen
(1) ov_new = PMread_direct (sv_source, o)
    if sv_source ∈ ov_list (o)
       then ov_old = sv_source.o
       else ov_old = ov_new;
(2) ∀ sv_drain ∈ (dpred(sv_source) ∪ dsucc(sv_source)) - del_list (o) - sv_visited {
       if sv_drain ∉ ov_list (o) {
          if sv_source < sv_drain { // Rückwärtspropagation, siehe Abbildung 7.11a
             if csv (o) ≤ sv_source { // Erzeugerschemaversion unterhalb
(3)             if bc {
(3)                if bm {
(3)                   if origin (ov_new) ≤ sv_source // Ursprungsschemaversion unterhalb
(3)                      then PMprop_conv_write (sv_source, sv_drain, sv_visited, ov_new);
(3)                   }
(3)                else { // ¬bm
(3)                   if origin (ov_old) ≤ sv_source // Ursprungsschemaversion unterhalb
```

```
(3)                 then PMprop_conv_write (sv_source, sv_drain, sv_visited, ov_old);
(3)               }
(3)             }
(3)           else { } // bc̄, o nicht sichtbar in sv_drain
              }
            else { // Erzeugerschemaversion nicht unterhalb
(4)           if bm {
(4)             if origin (ov_new) ≤ sv_source // Ursprungsschemaversion unterhalb
(4)               then PMprop_conv_write (sv_source, sv_drain, sv_visited, ov_new);
(4)             }
(4)           else { // b̄m̄
(4)             if origin (ov_old) ≤ sv_source // Ursprungsschemaversion unterhalb
(4)               then PMprop_conv_write (sv_source, sv_drain, sv_visited, ov_old);
(4)             }
              }
            }
          else { // sv_source > sv_drain, Vorwärtspropagation, siehe Abbildung 7.11b
            if csv (o) < sv_drain { // Erzeugerschemaversion unterhalb, o ∈ IAS(sv_drain)
(5)           if fm {
(5)             if NOT origin (ov_new) ≤ sv_drain
(5)               then PMprop_conv_write (sv_source, sv_drain, sv_visited, ov_new);
(5)             }
(5)           else { // f̄m̄
(5)             if NOT origin (ov_old) ≤ sv_drain
(5)               then PMprop_conv_write (sv_source, sv_drain, sv_visited, ov_old);
(5)             }
              }
            else { // Erzeugerschemaversion nicht unterhalb
(6)           if dt (sv_drain) < ovct (ov_old) < ovct (ov_new) {
(6)             if fc { // o sichtbar
(6)               if fm {
(6)                 if NOT origin (ov_new) ≤ sv_drain
(6)                   then PMprop_conv_write (sv_source, sv_drain, sv_visited, ov_new);
(6)                 }
(6)               else { // f̄m̄
(6)                 if NOT origin (ov_old) ≤ sv_drain
(6)                   then PMprop_conv_write (sv_source, sv_drain, sv_visited, ov_old);
(6)                 }
(6)               }
(6)             }
(6)           else if ovct (ov_old) < dt (sv_drain) < ovct (ov_new) {
(6)             if fs { // o sichtbar
(6)               if fm {
(6)                 if NOT origin (ov_new) ≤ sv_drain
(6)                   then PMprop_conv_write (sv_source, sv_drain, sv_visited, ov_new);
(6)                 }
(6)               else { // f̄m̄
(6)                 if NOT origin (ov_old) ≤ sv_drain
(6)                   then PMprop_conv_write (sv_source, sv_drain, sv_visited, ov_old);
(6)                 }
(6)               }
(6)             }
(6)           else { // ovct (ov_old) < ovct (ov_new) < dt (sv_drain)
(6)             if fs { // o sichtbar
(6)               if NOT origin (ov_new) ≤ sv_drain
(6)                 then PMprop_conv_write (sv_source, sv_drain, sv_visited, ov_new);
(6)               }
```

```
(6)             }
              }
            }
          }
        else { // sv_drain ∈ ov_list (o)
          if ovct (sv_drain.o) < ovct (ov_new) {
(7)         if // Rückwärtspropagation
(7)           (origin (ov_new) ≤ sv_source < sv_drain AND bm)
(7)         OR // Vorwärtspropagation
(7)           (sv_drain < sv_source AND NOT (origin (ov_new) ≤ sv_drain) AND fm)
(7)           then PMprop_conv_write (sv_source, sv_drain, sv_visited, ov_new);
            }
          }
        }
      }
```

Die Methode **PMpropagate**.

Als Beispiel für die folgenden Erklärungen betrachten wir Abbildung 7.12 zum Zeitpunkt t_3, wo **PMwrite (sv$_2$, o, ovalue)** aufgerufen wird, weil eine Applikation von sv_2 das Objekt o modifiziert. Wie oben gezeigt, wird **PMwrite** die Methode **PMpropagate (sv$_2$, o, $\emptyset$)** aufrufen. Wir werden nun untersuchen, wie die verzögerte Propagation von sv_2 in andere Schemaversionen arbeitet.

(1) **PMread_direct** berechnet die Objektversion $sv_source.o$ durch Anwendung der benötigten Konvertierungsfunktionen. Anders als bei **PMread** werden von **PMread_direct** keine Objektversionen verändert. Daher kann diese Methode ohne vorherige Propagation aufgerufen werden.

(2) Nun müssen die direkten Vorgänger und Nachfolger von sv_source, in denen das Objekt nicht gelöscht wurde, betrachtet werden.

(3) Die Methode **csv (o)** gibt diejenige Schemaversion zurück, in der das Objekt o (mit Hilfe des **new**-Operators) erzeugt wurde. Wenn das c-Flag eingeschaltet ist, muß eventuell (falls $o \in IAS(sv_drain)$) **PMprop_conv_write** aufgerufen werden, um in Abhängigkeit von den Flags den alten (ov_old) oder den neuen Objektwert (ov_new) zu propagieren.

(4) Hier gilt $o \in IAS(sv_drain)$ auf jeden Fall und wiederum kann ein Aufruf von **PMprop_conv_write** notwendig sein.

    In dem Beispiel muß der alte Wert von Objekt o in sv_1 gespeichert werden, da in dieser Richtung mit sv_6 tatsächlich eine Schemaversion existiert, die diesen Wert benötigt, weil o aufgrund des s-Flags in $IAS(sv_6)$ sichbar ist, die Veränderung allerdings nicht nach sv_6 propagiert werden darf ($\overline{m}$-Flag). Allerdings ist der alte Wert bereits in sv_1 gespeichert, so daß zum Zeitpunkt t_3 keine Propagation von sv_2 nach sv_1 durchgeführt werden muß.

(5) In Abhängigkeit von den Flags muß ggf. propagiert werden.

(6) Hier müssen die Zeitstempel von sv_drain, ov_old und ov_new verglichen werden, um herauszufinden ob das s- oder das c-Flag vor dem Aufruf von **PMprop_conv_write** geprüft werden muß. Wir erinnern daran, daß das s-Flag entscheidend ist, wenn das Objekt vor dem Ableiten der neuen Schemaversion (sv_drain) angelegt wurde. Ansonsten hängt es vom c-Flag ab, ob ein in sv_source erzeugtes Objekt in eine existierende Schemaversion sv_drain propagiert wird.

    In dem Beispiel werden hier die Schemaversionen sv_4 und sv_5 betrachtet. Nach sv_4 darf nicht propagiert werden, weil das c-Flag ausgeschaltet ist. Eine Propagation nach sv_5 ist

nicht notwendig, da das m-Flag eingeschaltet ist und sv_5 damit den alten Wert von o nicht benötigt. Die Propagation nach sv_5 wird hier jedoch trotzdem durchgeführt, weil es sein könnte (was im Beispiel allerdings nicht der Fall ist), daß Nachfolgerversionen von sv_5 existieren, die den alten Wert von o benötigen.

An dem Beispiel ist zu erkennen, daß eventuell nicht nur überflüssige Propagationen durchgeführt werden, sondern daß möglicherweise auch überflüssige Objektwerte gespeichert werden. Darunter hatten wir solche Objektwerte verstanden, die für keine Schemaversion benötigt werden. Damit wird deutlich, daß die hier vorgestellten Propagationsalgorithmen die Durchführung der Propagation sehr stark verzögern und damit das Verändern eines Objektwertes mit einem sehr geringen Aufwand ermöglichen. Eine physikalische Invariante, die das Vorhandensein überflüssiger Objektversionen verbietet, war damit nicht notwendig gewesen, denn sie hätte hier eine aufwendigere Propagation erfordert und damit den Grad an erwünschter Verzögerung reduziert.

(7) Hier existiert die Zielobjektversion bereits und es muß anhand eines Vergleichs der Zeitstempel festgestellt werden, ob diese überschrieben werden muß.

7.2.4 Implementierung von ODL-Parser und ODL-Generator

Für die Implementierung des ODL-Parsers wurden die Werkzeug flex++ und bison++ verwendet. Diese sind Nachfolger von lex und yacc [LMB92], die C++-Code erzeugen und damit in der COAST-Umgebung einfacher zu verwenden sind.

Die Implementierung des ODL-Generators ist sehr flexibel gehalten und geht in zwei Schritten vor. Zunächst wird die Information des vorliegenden Schemas komplett in eine eigene Datenstruktur übernommen, die geeignet ist, beliebige Sprachen zu repräsentieren. Im zweiten Schritt wird aus dieser Datenstruktur die gewünschte Ausgabe generiert. Dieser Generationsschritt ist regelbasiert, d.h. die Syntax der Ausgabe kann in Form von BNF-Ausdrücken spezifiziert werden. Desweiteren kann die Formatierung der Ausgabe durch spezielle Ausdrücke in den Regeln festgelegt werden. Eingebaut sind bereits vier solche Regelsätze, die die in der Aufzählung von Abschnitt 7.1.2.3 vorkommenden Ausgaben realisieren:

- Das gesamte Schema wird als Menge isolierter Schemaversionen unter Beschränkung auf die erzeugende ODL ausgegeben. Hierbei wird auf die Ausgabe des optionalen Schlüsselwortes **create** verzichtet, um die Transparenz für Applikationsentwickler zu erhöhen.

- Das gesamte Schema wird inklusive der darin enthaltenen Ableitungsbeziehungen ausgegeben. Hierbei wird das optionale Schlüsselwort **create** stets angegeben. Wir hatten zuvor selbst das Anlegen der ersten benutzerdefinierten Schemaversion als Schemaänderung interpretiert, indem wir sie als Ableitung von der systemdefinierten Wurzelschemaversion sv_0 angesehen haben. Diese Betrachtungsweise der Erzeugung von Schemakomponenten als Veränderung des Gesamtschemas wird durch die Ausgabe des Schlüsselwortes **create** unterstützt.

- Sämtliche für eine Schemaversion benötigten Defaultkonvertierungsfunktionen werden ausgegeben.

- Einzelne Schemaversionen werden in Form von C++-Headerdateien, die der Applikationsanbindung an COAST dienen, ausgegeben.

7.3 Zusammenfassung und Bewertung

In diesem Kapitel haben wir eine prototypische Implementierung der zuvor entwickelten Mechanismen vorgestellt. Der von Wöhrle [Wöh96] unternommene Versuch, dazu auf einem existierenden OODBMS aufzusetzen hat gezeigt, daß die erforderlichen Erweiterungen für die Handhabung versionierter Objekte sowohl auf Schema- als auch auf Datenbankebene den Kern des Datenbanksystems selbst betreffen und sehr weitreichend sind. Dabei ging insbesondere die Transparenz unserer Mechanismen verloren. Daher haben wir hier den Weg gewählt, die wesentlichen Eigenschaften eines OODBMS selbst zu entwickeln, wobei stets die mit den Konzepten der Schemaversionierung in Zusammenhang stehenden Aspekte bevorzugt betrachtet wurden.

Nach der Vorstellung der Architektur des COAST-OODBMS haben wir dessen zentrale Module näher vorgestellt, wobei wir insbesondere auf den Propagationsmanager und dessen Algorithmen zur Realisierung der verzögerten Propagation nach dem technischen Teilziel 3.15 der Effizienz eingingen. Für die Beschreibung von Konsistenzbedingungen bezüglich der Speicherung von Objekten haben wir wiederum Invarianten, diesmal auf der physikalischen Ebene, verwendet.

Die Realisierung eines vollständigen, praktisch einsetzbaren Datenbanksystems ginge weit über die innerhalb des gegebenen Rahmens erreichbaren Ziele hinaus und konnte unmöglich erfüllt werden. Stattdessen haben wir vorrangig das Ziel verfolgt, die Realisierbarkeit der Schemaversionierung tatsächlich nachweisen zu können. Dazu haben wir einige Komponenten entwickelt und implementiert, die in anderen Systemen entweder gar nicht oder mit stark eingeschränkter Funktionalität vorzufinden sind. Hier ist zunächst die Schemabeschreibungs- und -änderungssprache COAST-ODL zu nennen, die über den Funktionsumfang anderer Sprachen dieser Kategorie hinausgehend Konstrukte für die Verwaltung versionierter Schemata sowie für die Steuerung der Propagation auf Objektebene anbietet. Als Grundlage dient der Objektmanager, der auch Versionen desselben Objektes liest und schreibt, die verschiedenen Typen entsprechen. Der COAST-Schemamanager verwaltet in Erweiterung entsprechender Komponenten anderer Objektdatenbanksysteme auch versionierte Schemata und sorgt bei Schemaänderungen für die Einhaltung der Invarianten. Schließlich erlaubt erst der Propagationsmanager, ein sonst nirgends anzutreffendes Modul des Datenbankkerns, durch die Ausführung der Konvertierungsfunktionen den Einsatz der Schemaversionierung auch bei nichtleeren Datenbanken. Durch die Verwendung verzögerter Mechanismen wird dabei eine erhebliche Verringerung des Laufzeit- und Speicherplatzbedarfes erreicht.

Die Realisierungsarbeiten werden im Rahmen des COAST-Projektes durchgeführt, das im Juli 1996 an der Universität Frankfurt am Main initiiert wurde. Das COAST-Projekt umfaßt derzeit insgesamt neun Diplomarbeiten, von denen einige inzwischen abgeschlossen sind [Dol99, Eig97, Gro00, Her99, Pri98, Wöh96, Wöl98]. Die Ergebnisse wurden 1998 und 1999 auf der CeBIT [BL98, Lau99b] am Rechner vorgeführt und 1999 auf der ECOOP in Form eines Posters vorgestellt [LADH99].

Obwohl die Implementierung des COAST-Objektdatenbanksystems noch nicht vollständig abgeschlossen ist [Apo00, Haa00], läßt sich zusammenfassend bereits sagen, daß die Realisierbarkeit und Einsetzbarkeit der hier entwickelten Konzepte erfolgreich nachgewiesen werden konnte.

Kapitel 8

Validierung des COAST-Modells

In diesem Kapitel soll eine Bewertung der in dieser Arbeit erreichten Ergebnisse vorgenommen werden. Eine solche Bewertung kann sich auf verschiedene Aspekte beziehen, wobei jeweils auch von anderen Maßstäben ausgegangen werden kann.

- In Kapitel 4 hatten wir die in der Literatur beschriebenen Konzepte und Systeme anhand der zuvor vorgestellten technischen Teilziele bewertet. Eine entsprechende Bewertung der hier entwickelten Konzepte bietet sich auf jeden Fall an. Damit kann analysiert werden, in wie weit die gesteckten Ziele erreicht wurden. Gleichzeitig wird eine Vergleichbarkeit unserer Konzepte mit denen der vorgestellten Literatur gegeben.

- Eine weitere Möglichkeit bestünde in der Bewertung der prototypischen Implementierung im Vergleich zu den für andere Ansätze der Literatur entwickelten Prototypen oder im Vergleich zu kommerziell erhältlichen Datenbanksystemen. Eine solche Bewertung kann jedoch aufgrund mangelnder Verfügbarkeit der zu vergleichenden Systeme hier nicht durchgeführt werden.

  Zum einen sind nicht alle in der Literatur beschriebenen Konzepte auch implementiert, so daß bei einem Vergleich existierender Systeme viele Konzepte gar nicht berücksichtigt werden könnten. Weiterhin sind Prototypen, sofern sie denn existieren, nicht immer öffentlich zugänglich. Auch die Berücksichtigung kommerzieller Systeme kommt wegen der damit verbundenen Anschaffungskosten hier nicht in Frage. Deren Untersuchung verspräche ohnehin nicht viele Erkenntnisse, da wir bereits bei der Literaturübersicht festgestellt haben, daß die Handhabung der Schemaevolution von den auf dem Markt angebotenen Systemen nur sehr eingeschränkt unterstützt wird.

  Zum anderen dient der nach den Konzepten dieser Arbeit an der Universität Frankfurt entwickelte Prototyp im wesentlichen dem Zweck, die Realisierbarkeit der vorgestellten Konzepte zu belegen. Er stellt jedoch kein vollwertiges Datenbanksystem dar, das praktisch in größerem Umfang einsetzbar wäre. Dies liegt in der Beschränkung der Implementierung auf die für die Schemaevolution erforderlichen oder interessanten Aspekte begründet. Eine vollständige Realisierung sämtlicher für den praktischen Einsatz von Datenbanksystemen erforderlicher Leistungsmerkmale war nicht beabsichtigt und innerhalb des gegebenen Rahmens in keinster Weise möglich.

- Eine sehr praktische Bewertung könnte durch den tatsächlichen Einsatz verschiedener Systeme in einem konkreten Anwendungsbereich erreicht werden. Dabei ließe sich sogar der für die Durchführung verschiedener Schemaänderungen erforderliche Aufwand ungefähr bestimmen und mit der Häufigkeit einzelner Schemaänderungen, wie sie beispielsweise von

Sjøberg [Sjø93a, Sjø93b] ermittelt wurde, verknüpfen. Neben der bereits dargelegten mangelnden Verfügbarkeit praktisch einsetzbarer Systeme scheitert ein solcher Vergleich an dem damit verbundenen immensen Aufwand.

Aufgrund der obigen Erörterung führen wir die Validierung im wesentlichen anhand der technischen Teilziele aus Abschnitt 3.2 durch. Ergänzend führen wir jedoch mitunter auch direkte Vergleiche unserer Ansätze mit denen der Literatur durch, insbesondere wenn von Ansätzen der Literatur Eigenschaften angeboten werden, die über das von den technischen Teilzielen geforderte Maß hinaus gehen.

8.1 Flexibilität bei der Durchführung von Schemaänderungen

Der COAST-Schemamanager bietet Methoden nicht nur zur Spezifikation neuer, sondern auch zur Veränderung gegebener Schemata und Schemaversionen. Die damit realisierten Schemaänderungsoperationen gestatten eine komfortable Durchführung notwendiger Änderungen, ohne daß dabei das gesamte Schema neu spezifiziert werden muß (Teilziel 3.1). Dabei wird die Konsistenz des Schemas zu jedem Zeitpunkt sichergestellt.

Da bei einer Änderung eines vorliegenden Schemas eine neue Schemaversion angelegt wird, besteht eine Unabhängigkeit von dem bisherigen Zustand, so daß sich insbesondere keine Einschränkungen der Durchführbarkeit von Schemaänderungen aufgrund wechselseitiger Abhängigkeiten ergeben (Teilziel 3.2). Die zur Durchführung von Schemaänderungen angebotene ODL integriert Anweisungen zur Schemabeschreibung und -änderung und macht damit Unterschiede zwischen dem Anlegen einer neuen Schemaversion und dem Ändern einer vorhandenen transparent.

Die Durchführung von Änderungen geschieht zumindest aus der Sicht des Benutzers auf Schemaebene (Teilziel 3.3). Dabei wird jeweils eine Liste von Schemaänderungsprimitiven auf eine ausgewählte Schemaversion angewendet, was in einer neuen Schemaversion resultiert, welche wiederum durch die Anwendung von Primitiven fortentwickelt werden kann. Jede Schemaversion enthält eine konsistente Menge von Klassenversionen, womit für den Applikationsentwickler die Notwendigkeit einer Konfigurationsverwaltung entfällt.

In Abschnitt 5.1 wurde dargelegt, der mit der Schemaversionierung verbundene Mehraufwand sei ggf. in manchen Fällen nicht notwendig. Diese Befürchtung wird jedoch durch zwei Aspekte gemildert. Zum einen ermöglichen die in Abschnitt 5.4.2 dargestellten Konzepte, auf die Erzeugung einer neuen Schemaversion zu verzichten, und stattdessen selbst eine gefrorene Schemaversion nachträglich zu verändern, wenn dabei ausschließlich kapazitätserweiternde Schemaänderungen verwendet werden. Zum anderen werden bei der Ableitung einer neuen Schemaversion mehrere Schemaänderungen zusammengefaßt, weswegen nicht für jede einzelne Änderung eine neue Schemaversion angelegt werden muß. Stattdessen repräsentiert jede Schemaversion eine aus Benutzersicht natürliche, korrekte und vollständige Modellierung der betrachteten Diskurswelt und unterscheidet sich damit in erheblichem Umfang von anderen Schemaversionen.

Unser Ansatz ist dem der Datenbankversionen nach Jomier (siehe Abschnitt 4.5.3.1) insofern analog, als wir konzeptionell das gesamte Schema versionieren (wie die Datenbank bei Jomier), physikalisch aber nur Klassen (wie Objekte bei Jomier). Diese Unterscheidung von logischer bzw. konzeptioneller Sicht und physikalischer Implementierung geschieht in beiden Fällen, um die Laufzeit- und Speicherplatzeffizienz zu verbessern und dort wie hier wird damit eine einfachere Verwaltung der Versionen (des Schemas bzw. der Datenbank) aus Benutzersicht erreicht.

Der ODL-Parser ermöglicht die Durchführung externer Schemaänderungen (Teilziel 3.4). In einem ersten Schritt verifiziert er, daß die in einer eingelesenen ODL-Datei enthaltenen Schemaänderungsprimitive vollständig ausgeführt werden können und die Konsistenz des vorliegenden

Schemas dabei nicht verletzt wird. Der dabei eingesetzte Transaktionsmechanismus garantiert die Atomizität der in einer ODL-Datei enthaltenen Schemaänderungsblöcke, deren Listen von Primitiven jeweils ganz oder gar nicht ausgeführt werden. Nur im Erfolgsfalle werden die in der ODL-Datei spezifizierten Schemaänderungen im zweiten Schritt tatsächlich durchgeführt. Der Schemaassistent kann zu jeder Zeit für die Analyse einer beliebigen Schemaversion und die Erstellung einer verbesserten Kopie davon eingesetzt werden. Die damit verbundene Untersuchung von Ähnlichkeiten mit anderen Schemaversionen bietet sich jedoch gerade nach der Durchführung externer Schemaänderungen an, um die dabei nicht spezifizierten Ableitungsbeziehungen zwischen Klassenversionen zu ergänzen.

Bei der Durchführung von Schemaänderungen werden neue Versionen abgeleitet. Bestehende Versionen bleiben dabei unverändert und stehen somit für den Gebrauch weiterhin zur Verfügung. Dies schließt Dokumentationszwecke ebenso ein, wie die Verwendung durch (alte und neue) Applikationen und für die Ableitung weiterer Schemaversionen. Wenn der Schemaentwickler eine gemachte Änderung verwerfen will, so kann er stets auf einen beliebigen, früheren Zustand zurücksetzen (Teilziel 3.5).

Auch die Ableitung alternativer Schemaversionen (Teilziel 3.6) ist nach dem vorgestellten Modell jederzeit problemlos möglich. Wenn eine durchgeführte Schemaänderung nicht die gewünschten Ergebnisse bringt, dann kann damit nicht nur auf historische Vorgängerschemaversionen zurückgesetzt werden, sondern die Entwicklung kann auch bei bereits bestehenden Alternativen fortgesetzt werden.

Für die Integration von Schemaversionen nach Teilziel 3.7 stehen spezielle Integrationsprimitive zur Verfügung. Diese erlauben die Übernahme bestehender Klassen in neue Schemaversionen, d.h. sie arbeiten auf Klassenebene. Hieraus ergibt sich allerdings kein Widerspruch zu Teilziel 3.3 (Durchführung von Änderungen auf Schemaebene), da die Integration nur ein Teilschritt beim Anlegen einer neuen Schemaversion darstellt und somit auf derselben Ebene wie die Erzeugung neuer Klassen angesiedelt ist. Die Attributebene, als mögliche, nächst feinere Granularität wurde für die Integration nicht berücksichtigt. Eine dahingehende Möglichkeit wäre bei einer Beschränkung der Betrachtung auf das Schema eventuell wünschenswert, da sich so beispielsweise die Verschmelzung zweier Klassenversionen durchführen ließe. Solche Möglichkeiten werden durch unseren Ansatz jedoch absichtlich nicht geboten, da sie die Komplexität unserer Mechanismen deutlich erhöhen und letztlich in einem sehr verschiedenen Modell resultieren würden. Neben der Notwendigkeit, explizit Attributversionen zu verwalten, müßten auf die Attributebene zugeschnittene Propagationsflags angeboten werden, wobei zwischen den Flags von Attributen derselben Quell- oder Zielklassenversion sicher zusätzlich Konsistenzbedingungen zu betrachten wären. Für die Abbildung einer Verschmelzung auf die Datenbank wäre ein Verschmelzen von Objekten notwendig. Dazu ist jedoch die Sichtbarkeit des Objektes in beiden betroffenen Klassenversionen Voraussetzung, was die Flexibilität bei der Propagationssteuerung begrenzen würde. Daher haben wir uns hier für die Wahrung der Atomizität von Objekten bzw. von Objektversionen entschieden. Gibt man die Beschränkung auf atomare Objekte jedoch auf, so lassen sich im Umfeld eines Rollenmodells sog. *Kontexte* einführen, die bezüglich ihrer Granularität zwischen einzelnen Attributen und vollständigen Klassen anzusiedeln sind. In [Lau99a] haben wir bereits festgestellt, daß sich diese Kontexte als feinere Ebene sowohl für die Evolution und Integration von Komponenten auf Schemaebene als auch für die Propagation auf Objektebene eignen.

Die durch die Verwendung des Integrationsprimitives etablierte Ableitungsbeziehung wird explizit verwaltet (Teilziel 3.8). Dabei wird zusätzlich zwischen den Klassenableitungsbäumen und dem Schemaableitungsgraph unterschieden, wobei letzterer als Vereinigung der ersteren verstanden werden kann.

Für komplexe Schemaänderungen (von Lerner *compound type changes* genannt), von denen einige in Abschnitt 4.3.4.4 kurz aufgezählt wurden, stehen zum gegenwärtigen Zeitpunkt keine Operationen zur Verfügung. Solche ließen sich auf Schemaebene mit vergleichbarer Semantik sehr einfach ergänzen und könnten auch durch das Werkzeug entsprechend unterstützt werden. Offene Probleme ergeben sich jedoch bei der Spezifikation deren Auswirkungen auf die Objektebene. Hier gilt analog zu dem bei der obigen Besprechung der Integration gesagten, daß die Flexibilität bei der Propagationssteuerung auf Objektebene zugunsten erweiterter Möglichkeiten bei der Durchführung von Änderungen auf Schemaebene in Kauf genommen werden müßten. Dies haben wir absichtlich nicht gemacht, da die flexible Propagationssteuerung ein besonders wichtiges Alleinstellungsmerkmal unseres Ansatzes ist, welches wir demzufolge nicht von vornherein wieder einschränken möchten. Trotzdem sehen wir unter Berücksichtigung gewisser, noch zu spezifizierender Randbedingungen durchaus Möglichkeiten sowohl für eine feinere Granularität bei der Integration als auch für die Spezifikation komplexer Schemaänderungsoperationen.

Labib und Saunders (siehe Abschnitt 4.5.3.5) definieren neben den Schemaänderungen noch eine zweite Kategorie von Primitiven, die eher der Datenbanktransformation als der Schemaänderung dienen. Diese Kategorie umfaßt beispielsweise Operationen zum Reskalieren von Attributwerten oder zum Ändern von Wertebereichen. Diese Operationen wurden hier nicht berücksichtigt, da sie keine Schemaänderungen im eigentlichen Sinne darstellen.

8.2 Berücksichtigung von Applikationen

Ein wesentlicher Vorteil der Schemaversionierung im Vergleich zum Ansatz der direkten Schemaevolution besteht in der Erhaltung bisheriger Schemazustände und insbesondere in deren Nutzbarkeit durch bestehende Applikationen (Teilziel 3.9). Damit entfällt die Notwendigkeit zur sofortigen Anpassung sämtlicher bestehender Applikationen nach jeder Schemaänderung.

Allein die Möglichkeit zur unterbrechungsfreien Ausführung von Applikationen auf bisherigen Versionen eines Schemas genügt jedoch nicht für die Erreichung einer sinnvollen Form der Unterstützung evolutionärer Schemaänderungsprozesse. Entsprechend ihrer Definition müssen alle Datenbanksysteme Möglichkeiten zur Verwaltung gemeinsam genutzter Datenbestände offerieren. Zahlreiche Applikationen greifen dabei gleichzeitig auf dieselben Informationen zu, wobei das Datenbanksystem Gewähr für die Wahrung der Konsistenz der gesamten Datenbank bietet. In dem hier behandelten Szenario fortgesetzter evolutionärer Schemaänderungsprozesse basieren die kooperierenden Applikationen jedoch auf verschiedenen Versionen eines Schemas. Durch umfangreiche Propagationsmechanismen, die eine einzige Datenbank zur selben Zeit den Vorgaben verschiedener Schemaversionen folgend anzubieten vermögen und dabei insbesondere die Konvertierung von Objekten leisten, wird das technische Teilziel der Kooperation 3.10 voll erfüllt.

8.3 Flexibilität bei der Propagation von Schemaänderungen auf die Objektebene

Wenn die Kooperation zwischen Applikationen verschiedener Versionen eines Schemas so möglich sein soll, wie zwischen Applikationen eines unversionierten Schemas, so ist eine vollständige Propagation erforderlich (Teilziel 3.11). Dies bedeutet eine Propagation von jeder Schemaversion in jede beliebige andere Schemaversion und wird durch die vorgestellten Propagationsmechanismen erreicht. Jedes Objekt gehört genau einer Klasse direkt an, und diese Klasse kann in verschiedenen Versionen eines Schemas enthalten sein. Zwischen den Versionen dieser Klasse wird durch die Verwendung des Integrationsprimitives eine Ableitungsbeziehung etabliert, die stets eine Baumstruktur besitzt. Demzufolge ist eine Propagation vollständig und eindeutig,

wenn sie genau entlang der Ableitungskanten durchgeführt wird. Die ergänzende Einführung von Extrakonvertierungsfunktionen gestattet weitere Propagationspfade, wobei jedoch nachgewiesen wurde, daß die Eindeutigkeit dadurch nicht verletzt wird.

Die Durchführung einer vollständigen Propagation beinhaltet die uneingeschränkte Weitergabe von Datenbankzuständen zur Ableitungszeit einer neuen Schemaversion sowie von späteren Objekterzeugungen, -modifikationen und -löschungen. Diese Weitergabe späterer Änderungen geschieht stets insbesondere von der Schemaversion, in der die Änderung durch eine Applikation durchgeführt wurde, in alle anderen Schemaversionen. Das Ziel 3.12 der flexiblen Propagationssteuerung soll es dem Schemaentwickler gestatten, diese allumfassende Propagation gezielt einschränken zu können. Dies wird durch die Einführung der Propagationsflags erreicht, mit denen Einschränkungen speziell für bestimmte Klassenversionen und für verschiedene Typen von Änderungen vorgenommen werden können.

Ist ein Objekt in verschiedenen Schemaversionen sichtbar, die Propagation von Änderungen dazwischen jedoch abgeschaltet, so ergibt sich bei jeder folgenden Änderung zwangsläufig eine Situation, in der das Objekt in verschiedenen Schemaversionen in verschiedenen Zuständen vorliegt. Teilziel 3.13 fordert, daß eine solche Situation als natürlich eingestuft und entsprechend behandelt werden muß. Der Einsatz von Versionierungskonzepten für die Objekte der Datenbank gestattet die Handhabung verschiedener Datenbankzustände für verschiedene Schemaversionen.

Nach Teilziel 3.14 sollten semantische Beziehungen zwischen beliebigen Versionen eines Schemas etablierbar sein. Dies wird den vorgestellten Konzepten folgend durch zweierlei Mechanismen erreicht. Zum einen können bei der Ableitung einer neuen Schemaversion beliebige Klassen bestehender Versionen desselben Schemas integriert werden. Zum anderen besteht durch die Spezifikation von Extrakonvertierungsfunktionen die Möglichkeit, zusätzliche Querbeziehungen zu etablieren, entlang derer ein erhöhtes Maß an Semantik propagierbar ist.

Die Vorschläge von Kim und Chou zur Realisierung eines Schemaversionierungskonzeptes für ORION (siehe Abschnitt 4.5.3.7.2) hatten den Anstoß für einige unserer Überlegungen gegeben. In Ergebnis erreichen wir in COAST bei der Steuerung der Propagation jedoch eine Flexibilität, die bezüglich Propagationsrichtung, -granularität und -zeitpunkt weit über die der Mechanismen von Kim und Chou hinausgeht. Insbesondere kann der für ORION vorgeschlagene Mechanismus von COAST simuliert werden.

8.4 Effizienz und Handhabung der Mechanismen

Bei der Bewertung der Effizienz unserer Konzepte (Teilziel 3.15) müssen wir zwischen Speicherplatz- und Laufzeitbedarf unterscheiden. Der Speicherplatzbedarf ist ähnlich hoch wie bei der Technik des Screening, welche bei Systemen mit direkter Schemaevolution (beispielsweise bei O_2, siehe Abschnitt 4.3.4.2) zur Realisierung verzögerter Konvertierungsmechanismen verwendet wird. Die dort gespeicherten Objektzustände entsprechen bezüglich des Speicherplatzbedarfes den Objektversionen. Da wir für die Schemaversionierung ebenfalls einen verzögert wirkenden Propagationsmechanismus einsetzen, erreichen wir eine ähnliche Speicherplatzersparnis wie bei der direkten Schemaevolution. Diese ergibt sich in beiden Fällen daraus, daß der Speicherplatz für einen Objektzustand bzw. für eine Objektversion physikalisch erst dann belegt wird, wenn darauf zum ersten Mal zugegriffen wird. Von diesem Grundsatz weichen die in Abschnitt 7.2.3.3 vorgestellten Algorithmen zur Vereinfachung und Beschleunigung später zu erwartender Konvertierungen allerdings manchmal ab. Ein erheblicher Unterschied besteht jedoch darin, daß die Applikationen bei der direkten Schemaevolution nicht auf die logisch ausgeblendeten Werte eines Objektes zugreifen können, obwohl diese physikalisch vorhanden sind. Dies führt dort letzt-

endlich dazu, daß sämtliche Applikationen bei einer Schemaänderung sofort angepaßt werden müssen.

Durch die verzögerte Propagation wird neben der genannten Speicherplatzersparnis insbesondere auch eine erhebliche Verringerung der Laufzeitanforderungen erreicht. Zum einen wird nicht nur der Speicherplatz, sondern auch die Zeit für die erstmalige Konvertierung eines Objektes in eine bestimmte Schemaversion erst dann benötigt, wenn tatsächlich eine Applikation auf das Objekt zugreift.[129] Zum anderen verringert sich die benötigte Laufzeit im Vergleich zu einem System mit sofortiger Propagation von Objektänderungen durch eine anzunehmende Lokalität der Objektzugriffe ganz erheblich. Dabei gehen wir davon aus, daß eine Applikation oder zumindest Applikationen derselben Schemaversion mehrmals auf ein Objekt zugreifen, bevor dieses Objekt wieder von einer Applikation einer anderen Schemaversion benötigt wird. Erstreckt sich ein solch lokales Verhalten auf n aufeinanderfolgende Zugriffe auf ein Objekt durch dieselbe Schemaversion, so erreicht der verzögerte Mechanismus eine Einsparung von $n-1$ der ansonsten notwendigen n Konvertierungen. Diese Einsparung war der ausschlaggebende Aspekt bei der Entscheidung für den Einsatz eines verzögerten Mechanismus, da sie als sehr erheblich einzustufen ist. Damit erscheint auch die im Vergleich etwa zu ORION deutlich höhere Komplexität der Algorithmen aus Abschnitt 7.2.3.3 gerechtfertigt.

Eine weitere deutliche Verbesserung der Effizienz sowohl bezüglich des Platz- als auch bezüglich des Laufzeitbedarfes wird durch die in Abschnitt 5.4.2 erläuterte Reduzierung der Anzahl notwendiger Schemaversionen erreicht. Damit sinkt nicht nur der Verwaltungsaufwand für das versionierte Schema, sondern auch die Zahl der in der Datenbank abgelegten und bei der Propagation zu berücksichtigenden Objektversionen. Wenn eine neue Schemaversion allerdings mit dem Ziel angelegt wird, die Propagation einzuschränken, dann kommt die Möglichkeit einer späten Modifikation einer bereits existierenden Schemaversion nicht in Betracht.

Die Spezifikation der Propagation wird dem Schemaentwickler durch verschiedene Aspekte erleichtert (Teilziel 3.16). Zum einen wird durch das Angebot automatisch generierter, die durchgeführten Schemaänderungen berücksichtigender Defaultkonvertierungsfunktionen ein Vorschlag unterbreitet, der in vielen Fällen bereits ausreichend sein wird. Anderenfalls kann der Vorschlag verändert und somit trotzdem noch als Grundlage einer manuellen Spezifikation dienen. Auf jeden Fall ist durch die Defaultkonvertierungsfunktionen sichergestellt, daß jede notwendige Konvertierung auch durchgeführt werden kann. Zum zweiten verringert sich die Zahl der insgesamt notwendigen Konvertierungsfunktionen durch die Einführung der Klassenkonvertierungsgraphen. Entlang derer Kanten werden Konvertierungsfunktionen nämlich bei Bedarf hintereinander ausgeführt, um transitive Propagationen zu bewerkstelligen. Darin manifestiert sich auch eine Form der Wiederbenutzung vorhandener Spezifikationen. Schließlich wird der Schemaentwickler durch das Werkzeug unterstützt, das die manuell modifizierten Konvertierungsfunktionen daraufhin prüfen kann, ob jedem Attribut der Zielklassenversion ein Wert zugewiesen wird. Alternativ hätte sich angeboten, die Defaultkonvertierungsfunktionen wie bei O_2 auf jeden Fall auszuführen und deren Resultat in einem zweiten Schritt durch ggf. vorhandene, benutzerdefinierte Konvertierungsfunktionen modifizieren zu lassen.

Das Teilziel 3.17 der Lokalität bei der Spezifikation neuer Schemazustände wird dadurch erreicht, daß neue Schemaversionen normalerweise nur von einer oder wenigen Quellschemaversionen abgeleitet werden. Selbst wenn der Schemaentwickler keine vollständige Übersicht über alle vorhandenen Versionen eines zu modifizierenden Schemas besitzt, so kann der Schemaassistent durch Navigationshilfen das Auffinden von Klassenversionen, die der gewünschten Spezifikation ähneln, erleichtern. Aber auch nach der Spezifikation einer neuen Klassenversion können noch Vergleiche angestellt und ggf. Vorschläge für Verbesserungen unterbreitet werden. Auch

[129] Auch hier gelten die bei der obigen Analyse der Speicherplatzeffizienz bereits erwähnten, durch die in Abschnitt 7.2.3.3 vorgestellten Algorithmen bedingten Einschränkungen.

bei der Propagation wird ein Art von Lokalität erreicht, indem diese nicht zu allen Schemaversionen spezifiziert werden muß, sondern nur zu denjenigen, von denen eine neue Schemaversion abgeleitet wurde. Somit erbringt die automatische, transitive Propagation entlang eindeutiger Propagationspfade auch für das Teilziel der Lokalität einen Nutzen.

Die Schemaänderungsprimitive sind so ausgelegt, daß sie die Konsistenz des Schemas stets erhalten (Teilziel 3.18). Diesem Vorteil stünde allerdings die ggf. etwas aufwendigere Spezifikation von Schemaversionen gegenüber. Beispielsweise können zwei Klassen a und b, die sich gegenseitig referenzieren (z.B. Firma und Angestellter), nicht nacheinander angelegt werden, da sich nach der Spezifikation der einen Klasse ein unvollständiges Schema ergäbe, weil die referenzierte zweite Klasse zu diesem Zeitpunkt noch nicht existierte. Daher muß eine der Klassen eigentlich zunächst ohne die Referenz spezifiziert werden. Daraufhin kann die zweite Klasse dann vollständig spezifiziert werden und schließlich muß die Referenz der ersten Klasse auf die zweite nachträglich hinzugefügt werden. Für Spezifikationen in der ODL konnten wir diese aufwendigere Vorgehensweise durch ein mehrschrittiges Vorgehen des Parsers vermeiden und damit denselben Effekt erzielen, den die Einführung eines Transaktionsmechanismus auf Schemaebene erbracht hätte.

Auf der Ebene der Datenbank wird die Konsistenz eines an eine zugreifende Applikation gelieferten Objektes zumindest bezüglich seines Typs gewährleistet. Ein Objekt wird einer Applikation Dank des Propagationsmechanismus stets in dem von der Applikation entsprechend ihrer Schemaversion erwarteten Typ geliefert. Außerdem berücksichtigen die Defaultkonvertierungsfunktionen die Semantik der durchgeführten Schemaänderungen und erreichen damit eine Form der Konsistenz zwischen den verschiedenen Versionen eines Objektes. Weiterhin ist sichergestellt, daß der Propagationsmechanismus keine Referenzen auf in der Zielschemaversion nicht sichtbare Objekte propagiert.

Der Schemaversionierungsmechanismus macht die Evolution eines Schemas für die Gruppe der Applikationsentwickler weitestgehend transparent (Teilziel 3.19). Jeder Applikationsentwickler sieht nämlich jeweils nur eine Schemaversion, die der Schemaentwickler für ihn entsprechend den Anforderungen aussucht und die sich für ihn genau wie ein Schema eines herkömmlichen Datenbanksystems ohne Schemaversionierung darstellt. Insbesondere muß er keine Konfigurationsverwaltung zwischen Klassenversionen betreiben. Die Transparenz ist lediglich dann eingeschränkt, wenn eine vorliegende Applikation an eine neue Schemaversion anzupassen ist. Dies tritt jedoch notwendigerweise bei jeder Evolution des Schemas ein, nur haben die Applikationsentwickler hier mit zwei Schemaversionen anstelle zweier Schemata zu tun. Ihre Arbeit erleichtert sich außerdem durch die Erreichung von Teilziel 3.14, da sie den semantischen Zusammenhang zwischen den beiden Schemaversionen anhand der durchgeführten Schemaänderungen erkennen können.

8.5 Zusammenfassung und Bewertung

Das COAST-Modell unterstützt die evolutionäre Anpassung von Datenbankschemata an sich ändernde Geschäftsfelder und -prozesse der Diskurswelt. Wir konnten dies im einzelnen anhand der technischen Teilziele sehr detailliert nachweisen. An dieser Stelle gehen wir zusammenfassend nochmals kurz auf die wesentlichen Aspekte ein.

Die Anpassung von Schemata ist bereits in Programmiersprachen ohne Persistenz erforderlich und kann dort problemlos durchgeführt werden, da keine externen Abhängigkeiten zu solchen Schemata existieren. Die im Datenbankbereich verwendeten Schemata werden jedoch für eine Menge kooperierender Applikationen entworfen und stellen gleichsam eine Schnittstelle zwischen den persistenten Objekten der Datenbank und den darauf zugreifenden Applikationen dar. Demzufolge ergeben sich bei Schemaänderungen Konsequenzen auf diesen beiden Seiten. Die Behandlung persistenter Objekte ist eine Aufgabe, mit der sich die Hersteller von Daten-

banksystemen in den letzten Jahren zu beschäftigen begannen. Dies unterstreicht die Bedeutung von Schemaänderungen und die Notwendigkeit entsprechend weitreichender Unterstützung im industriellen Umfeld. Gänzlich unberücksichtigt blieb bislang allerding die Seite der Applikationen. Hier behandelt das Versionierungsmodell von COAST neue Aspekte und leistet damit einen Beitrag zur Verbesserung der Tauglichkeit von Datenbanksystemen zur Modellierung dynamischer Umgebungen.

In COAST werden einmal verwendete Schemazustände in Form von Schemaversionen aufbewahrt und bleiben somit unbegrenzt für Applikationen nutzbar. Die Ableitung neuer Schemaversionen hat keinerlei Konsequenzen für an bestehende Schemaversionen gebundene Applikationen. Damit wird die Notwendigkeit des Reengineering bestehender Applikationen aufgehoben.

Auch die Kooperation von Applikationen verschiedener Schemaversionen ist Dank des Propagationsmechanismus möglich. Darüber hinaus wird durch die Propagationssteuerung das gesamte Spektrum von isolierten Schemaversionen einerseits bis hin zu einer den Sichtenmechanismen ähnlichen Kooperation aller Schemaversionen auf jeweils nur einer Version jedes Objektes andererseits abgedeckt. Damit leistet das COAST-Modell eine Vervollständigung der Betrachtung von Konsequenzen bei der Durchführung von Schemaänderungen für Objekte und Applikationen.

Kapitel 9

Zusammenfassung und Ausblick

Gegenstand dieses Kapitels ist zunächst die Zusammenfassung der in dieser Arbeit erreichten Ergebnisse im Hinblick auf die ursprüngliche Zielsetzung, also die Unterstützung von Schemaevolution in objektorientierten Datenbanksystemen. Anschließend folgen Überlegungen, welche der erzielten Ergebnisse zur Lösung von Problemen in anderen Arbeitsbereichen herangezogen werden können und auf welche Weise dies geschehen kann. Die Arbeit schließt mit einem Ausblick auf weitere Arbeiten im von uns bearbeiteten Themengebiet.

9.1 Erreichte Ziele

Ausgangspunkt unserer Arbeit war die Beobachtung, daß die Evolution von Datenbankschemata, welche zur Anpassung an sich ändernde funktionale und nicht-funktionale Anforderungen[130] der Diskurswelt benötigt wird, durch die gegenwärtig verfügbaren Modelle und Systeme nicht adäquat unterstützt wird. Wir stellten daraufhin die Hypothese auf, daß ein Modell auf der Basis der Versionierung von Schemata mit einer entsprechenden Abbildung der Änderungen auf die Objektebene dies leisten kann. Wir konnten in dieser Abhandlung zeigen, daß das nach diesen Gesichtspunkten entworfene COAST-Modell die daran gestellten Erwartungen erfüllt und Schemaänderungen in Gegenwart existierender Objekte und Applikationen erfolgreich realisierbar sind. Die einzelnen Arbeitsschritte und Ergebnisse ergaben sich dabei wie folgt:

- *Problemanalyse und grober Modellentwurf:* Die Notwendigkeit einer Unterstützung für Schemaevolutionsprozesse ergab sich aus der Beobachtung, daß vor allem moderne Anwendungsbereiche eine flexible Anpassung an ständig veränderliche Umstände erfordern. Während des Betriebs einer Datenbank aufkommende Änderungsanforderungen lassen sich zur Entwurfszeit nicht vorhersehen und sind im Rahmen fest vorgegebener Datenbankschemata nur schwerlich adäquat umsetzbar. Wir konnten in diesem Zusammenhang bei den vorhandenen Systemen zur Unterstützung der Schemaevolution das Fehlen einer Berücksichtigung im Betrieb befindlicher Datenbankapplikationen feststellen. Gleichzeitig blieben Anforderungen an flexibel koppelbare Datenbankzustände für verschiedene Schemaausprägungen bisher unberücksichtigt.

  Das an dieser Stelle grob skizzierte Modell zur Lösung des Problems beruhte folgerichtig auf dem Einsatz von Versionierungskonzepten auf der Ebene der Datenbankschemata. Solche Versionierungskonzepte hatten ihre Fähigkeiten zur Unterstützung von Evolutionsprozes-

[130] Funktionale Anforderungen rühren zumeist von den zu unterstützenden Geschäftsprozessen her, während Aspekte wie Performanz und Verteilung zu den nicht-funktionalen Anforderungen zählen.

sen insbesondere auch im Zusammenhang mit Entwurfsaufgaben bereits zuvor sowohl auf der Ebene kompletter Datenbanken als auch auf der einzelner Objekte nachgewiesen.

- *Untersuchung bestehender Lösungsansätze:* Wir konnten für das Lösungsmodell vier elementare Aspekte identifizieren: Durchführung von Änderungen auf Schemaebene unter Verwendung von Versionierungskonzepten, Erzeugung und Verwaltung der Abhängigkeitsbeziehungen zwischen den Schemaversionen, Abbildung der Änderungen auf die Objektebene sowie flexible Konzepte zur Steuerung und Durchführung der Objektpropagation. Da kein Modell existierte, das all diese Punkte berücksichtigt, mußten wir uns in der Literaturrecherche auf Ansätze beschränken, die sich mit einzelnen Aspekten unserer Aufgabenstellung befassen. Aus dieser Perspektive stellt sich unser Modell zu einem Teil als Integration früherer Arbeiten dar. Zum anderen Teil beruht unser Modell in den grundlegenden Aspekten der Behandlung von Ableitungsbeziehungen sowie der Steuerung und Durchführung der Objektpropagation auf gänzlich neuen Ansätzen. Die wesentliche Erkenntnis dieses Arbeitsschrittes ist somit die Feststellung, daß einerseits verschiedene Konzepte bestehender Ansätze übernommen werden konnten, obwohl keine Arbeit alle Anforderungen an unser Modell erfüllte, und andererseits einige Aspekte bislang nahezu vollständig ignoriert wurden.

- *Detaillierte Modellbildung:* Aufgrund der Erkenntnisse über bestehende Arbeiten entwarfen wir in diesem Arbeitsschritt COAST als Modell zur Durchführung von Schemaänderungen in Anwesenheit von Objekten und Applikationen. Grundbestandteile unseres Ansatzes sind die Schemaversionen, die vergleichbar einer Konfigurationsverwaltung semantisch in Zusammenhang stehende Klassenversionen zu konsistenten Teilstrukturen zusammensetzen.

  Für die Durchführung von Schemaänderungen haben wir zwei grundsätzlich verschiedene Wege analysiert und resultierende Konsequenzen studiert. Dem internen Ansatz folgend wird eine von dem jeweiligen Einsatzgebiet des Systems unabhängige, fest vorgegebene und konzeptionell vollständige Taxonomie von Schemaänderungsprimitiven bereitgestellt, deren Semantik a priori bekannt ist und demzufolge bei allen weiteren Schritten auf Schema- und Objektebene berücksichtigt werden kann. Der externe Ansatz, als Alternative, erlaubt die Durchführung applikationsspezifischer Schemaänderungen und erhöht damit die Flexibilität. Der Prozeß des Einbringens extern erstellter Schemaversionen in das System kann dabei in vielfältiger Weise unterstützt werden.

  Bemerkenswerterweise fanden sich die auf Schemaebene angewandten Konzepte analog auf Instanzenebene wieder und zwar bei den Objektversionen, deren Zusammenhang sich in Form von Propagationsgraphen widerspiegelt ähnlich den Ableitungsbeziehungen auf Schemaebene. Die Steuerung der Objektpropagation sowohl zum Zeitpunkt der Schemaänderung als auch später, als Reaktion auf verändernde Datenbankzugriffe ist im behandelten Umfeld mit Sicherheit einzigartig.

- *Validierung des Modells:* Aussagen zur Tauglichkeit des Modells im Hinblick auf seine Konzeptionsziele konnten wir durch eine Evaluierung anhand unserer sehr detailliert beschriebenen Vorgaben erhalten. Damit ist gleichzeitig ein Vergleich mit bisherigen Lösungswegen insgesamt und mit einzelnen Vertretern davon gegeben. Die Evaluierung konnte in allen Aspekten belegen, daß das COAST-Modell zur Unterstützung von Schemaänderungen in Benutzung befindlicher Datenbanksysteme gut geeignet ist.

  Wir möchten an dieser Stelle nochmals betonen, daß COAST in vielerlei Hinsicht einfach erweitert werden kann und im Vergleich zu bisherigen Systemen durch erheblich flexiblere Möglichkeiten der Einflußnahme und eine verbesserte Tauglichkeit ausgezeichnet wird. Zunächst sind sowohl die hier vorgestellte Schemaänderungstaxonomie als auch die damit verbundene Propagationssprache vielfältig um komplexe Operationen erweiterbar. Weiterhin

kann die Menge verwendbarer Propagationsflags insbesondere mit Blick auf das Verhalten bei komplexen Schemaänderungen hin ergänzt werden. Aber bereits die hier dargestellten Möglichkeiten der Propagationssteuerung decken das gesamte Spektrum von isolierten Schemaversionen am einen Ende bis hin zur kompletten Propagation am anderen Ende ab. In Erweiterung der Konzepte auf der Basis von Sichtenmechanismen können durch die Objektversionierung beliebige Änderungen des Datenbankzustandes durchgeführt werden. Damit wird ein erheblicher Beitrag für die Transparenz der Schemaevolution für die Applikationsentwickler geleistet.

Die für die Tauglichkeit wichtigste Eigenschaft besteht zweifelsfrei in der Möglichkeit, bestehende Applikationen auch noch nach der Durchführung von Schemaänderungen ohne Anpassung weiterverwenden zu können. Damit erweitert sich der Einsatzbereich von Schemaänderungen auf sehr große, komponenten-basierte Systeme auf der Basis zahlreicher Einzelapplikationen.

- *Hinweise für den Datenbankentwurf:* Um den Schemaversionierungmechanismus adäquat einsetzen zu können, erweisen sich Aussagen über den Schemaänderungsprozeß als notwendig. Daher haben wir diesen Prozeß systematisch in Teilschritte zerlegt, die nacheinander betrachtet werden können. Bereits während der Modellbildung haben wir dort, wo sich einem Schemaentwickler Alternativen im Umgang mit COAST bieten, diese aufgezeigt und ihre jeweiligen Konsequenzen untersucht. Im Ergebnis resultiert für den Schemaentwickler im Vergleich zu unversionierten Systemen ein zusätzlicher Spezifikationsaufwand. Dies ist jedoch für die Erreichung unserer Ziele unvermeidbar und der Gesamtaufwand für die Durchführung einer Schemaänderung reduziert sich dem Versionierungskonzept folgend erheblich, nicht zuletzt, weil aus der Sicht der Applikationsentwickler die volle Transparenz gewährleistet wird.

- *Realisierungsbetrachtungen:* Um Erkenntnisse über den Realisierungsaufwand sowie die zu erwartenden Leistungsmerkmale zu erhalten, haben wir zweierlei Ansätze für eine prototypische Realisierung untersucht. Zum einen haben wir einen Schemaversionierungsmechanismus als Aufsatz auf dem kommerziellen Objektdatenbanksystem O_2 implementiert. Auch wenn sich dies konzeptionell als möglich erwiesen hat, so haben sich dort an mehreren Stellen erhebliche Einbußen bezüglich der Transparenz gezeigt [Wöh96]. Daher haben wir uns verstärkt mit dem zweiten, deutlich aufwendigeren Weg beschäftigt: die Eigenentwicklung eines kompletten, objektorientierten Datenbankmanagementsystems, bei dem die Konzepte von COAST transparent und von Anfang an im Kern integriert werden konnten.

  Die Realisierung der verzögerten Propagation kann bei realistischen Zugriffsprofilen mit einer gewissen Lokalität bezüglich der Schemaversionen größenordnungsmäßig mit unversionierten Systemen vergleichbare Laufzeiten erreichen. Konzeptionell bedingt muß zwar ein größerer Platzbedarf als bei einem statischen System (ohne Schemaänderungen) in Kauf genommen werden. Im Vergleich zu anderen Konzepten der Schemaevolution, etwa den Ansätzen auf der Basis von isolierten Datenbanken oder mit materialisierten Sichten tritt aber kein Mehraufwand auf. In all diesen Varianten wird, ähnlich wie bei Puffern absichtlich Platz zugunsten einer erhöhten Zugriffsgeschwindigkeit investiert. Je ähnlicher sich verschiedene Schemaversionen sind und je intensiver die Propagation zwischen ihnen demnach ausfällt, desto besser sind die Voraussetzungen für platzsparende Mechanismen auf der Basis von mehreren Schemaversionen gemeinsam genutzter Objektversionen. In diesem Zusammenhang konnten wir eine Reihe von Verbesserungsmöglichkeiten identifizieren, die den COAST-Prototyp Systemen auf der Basis von Sichtenmechanismen gleichstellen würden.

9.2 Übertragbarkeit der Ergebnisse

Die Tragfähigkeit der Konzepte des COAST-Modells für die Unterstützung evolutionärer Schemaänderungen haben wir erfolgreich belegen können. Im folgenden sprechen wir noch einige Möglichkeiten an, die in dieser Abhandlung gewonnenen Erkenntnisse auch im Zusammenhang mit anderen Konzepten einzusetzen.

- Abschnitt 2.2 hatte allgemeine Versionierungskonzepte vorgestellt, wie sie typischerweise auf Objekte angewendet werden, um deren inhaltliche Evolution abzubilden. Wir haben dieselben Konzepte in der vorliegenden Arbeit auf Schemata als Instanzen der Metaebene angewendet, um deren evolutionäre Entwicklung zu unterstützen. Dabei hat sich die Verwendung versionierter Datenbankobjekte ebenfalls als sehr hilfreich erwiesen. Die Versionen eines Objektes bei der Schemaversionierung repräsentieren ein Objekt in verschiedenen Datentypen. Von Situationen abgesehen, in denen das Modifikationsflag abgeschaltet ist und Änderungen daher gewollt nicht propagiert werden, repräsentieren die Versionen eines Objektes denselben logischen Objektwert. Damit unterscheiden sich die hier verwendeten Objektversionen von denen der klassischen Objektversionierung. Letztere stellen nämlich verschiedene logische Objektwerte dar, die allerdings alle demselben Datentyp entsprechen.

  Die vorangegangene, vergleichende Betrachtung zeigt einerseits, daß die beiden Formen der Versionierung unterschiedliche Ziele verfolgen, und andererseits legt sie die Vermutung nahe, daß es sich um orthogonale und damit gewinnbringend kombinierbare Ansätze handelt. Verschiedene Typen zur Darstellung desselben logischen Objektwertes hier stehen verschiedenen Objektwerten desselben Typs dort gegenüber.

  Tatsächlich läßt sich unser Ansatz um Mechanismen zur Objektversionierung erweitern, indem jede unserer Objektversionen nun durch verschiedene klassische Objektversionen ersetzt wird. Damit entsteht ein zweidimensionaler Raum von Objektversionen: entlang der einen Dimension liegt jeweils ein logischer Objektwert in verschiedenen Typen, entlang der anderen Dimension liegen jeweils verschiedene logische Zustände eines Objektes, die durch denselben Typ repräsentiert werden.

  Um die Kombination der beiden Versionierungskonzepte zu erreichen, sind allerdings noch einige Fragen näher zu untersuchen. Diese beschäftigen sich beispielsweise mit dem Zusammenhang zwischen den Objektversionen entlang der beiden Dimensionen und mit der Propagation versionierter Objekte. Hier ist beispielsweise zu klären, ob alle, oder wenn nicht welche der logischen Objektwerte eines Objektes in andere Schemaversionen zu propagieren sind. Schließlich bietet die Realisierung des integrierten Gesamtkonzeptes zahlreiche Ansatzpunkte für technische Optimierungen und erfordert diese auch, um sowohl den Zeitaufwand für die Propagation als auch den Platzbedarf für die Speicherung der zweidimensional versionierten Objekte zu reduzieren.

- Wir waren in der Literaturrecherche auf das Sichtenkonzept als Grundlage zur Simulation von Schemaänderungen eingegangen und hatten dabei einige Defizite bei der Lösung der hier betrachteten Aufgabenstellung identifiziert. Dies impliziert jedoch keine Aussage über die Tragfähigkeit von Sichten in dem Umfeld, für das sie ursprünglich konzipiert worden waren. Aufgrund der mit COAST erzielten Transparenz, die eine Schemaversion nach außen hin wie ein Schema eines unversionierten Systems erscheinen läßt, kann ein Sichtenkonzept auf dem COAST-Modell aufgesetzt werden. Konzeptionelle Schwierigkeiten sind durch die Kombination von Sichten und Schemaversionen nicht zu erwarten: Beim Ableiten neuer Schemaversionen können auf den Vorgängern definierte Sichten bei Bedarf mitintegriert werden und das Anlegen, Ändern und Löschen von Sichten kann durch die Primitive des Sichtenmechanismus erfolgen. In einem beide Konzepte integrierenden Sys-

tem kann entsprechend der gestellten Anforderungen entschieden werden, ob diese besser durch Anlegen einer neuen Sicht oder durch Ableiten einer neuen Schemaversion erfüllt werden.

- Einen Schritt über die Integration eines separaten Sichtensystems hinaus geht die Überlegung, ob Sichten nicht sogar durch Schemaversionen simuliert werden können. Damit wäre dann auch eine vollständig homogene Integration beider Konzepte in einem System erreicht. Um diese Überlegung zu verfolgen, betrachten wir die konzeptionellen Komponenten eines Sichtensystems und analysieren kurz, wie diese auf die Konzepte von COAST abgebildet werden können.

  Die für den Schemaentwickler zu verwendende Schnittstelle eines Sichtensystems ist durch die Sichtendefinitionssprache gegeben. Die dort zur Verfügung stehenden Konstrukte dienen zunächst der Definition des Sichtschemas und sind insoweit durch die Primitive der COAST-ODL abgedeckt. Darüber hinaus bestimmt eine Sichtendefinition die Extension der Sichtklassen durch Angabe je einer Anfrage, wobei das Ergebnis dieser Anfrage dem Schema der definierten Sichtklasse entsprechen muß bzw. dieses implizit erst bestimmt. Um diesen Teil eines Sichtenkonzeptes zu simulieren, sind in COAST zwei Erweiterungen notwendig. Zum einen wird eine Anfragesprache benötigt. Diese wäre für die Vervollständigung von COAST sowieso erforderlich und könnte sich konzeptionell sehr stark an bestehenden objektorientierten Anfragesprachen orientieren. Ein Anfragesystem muß zu jeder Anfrage zunächst das Schema ihres Resultates ermitteln und dieses könnte dann mit den Primitiven der COAST-ODL erstellt werden. Daraufhin muß das Anfragesystem die eigentliche Durchführung der Anfrage auf der Datenbank erledigen. Dies beschreibt gleichzeitig die zweite in COAST erforderliche Erweiterung. Für die Durchführung der Anfrage und insbesondere der darin ggf. enthaltenen Selektion von Objekten müßte eine entsprechende Erweiterung der Propagationssprache von COAST vorgenommen werden.

  Änderungen von Objekten in Sichtklassen sind aufgrund des Sichtenänderungsproblems i.Allg. nicht durchführbar, da in der Sichtendefinitionssprache keine Möglichkeit besteht, die Auswirkungen einer solchen Änderung auf die Objekte des Basisschemas zu spezifizieren. Daher bieten einige Konzepte Erweiterungen an, die man in COAST durch Verwendung von Rückwärtskonvertierungsfunktionen bereits hat. Durch die Vorwärts- und Rückwärtskonvertierungsfunktionen können beide Richtungen von Abbildungen zwischen (simuliertem) Basisschema und (simuliertem) Sichtschema sogar homogen durch dasselbe Konzept spezifiziert werden.

  Die Propagationsflags wären zur Simulation alle eingeschaltet und durch die Verwendung der verzögerten Propagationsmechanismen von COAST liefert die Simulation von Sichten durch Schemaversionen zusätzlich ein optimiertes Konzept der Materialisierung von Sichten.

- Das Konzept der direkten Schemaevolution hatte sich bei der Anwendung in dem hier beschriebenen Einsatzgebiet als zu restriktiv erwiesen. Nichtsdestotrotz kann die direkte Schemaevolution in Einzelfällen für die Durchführung von Schemaänderungen genügen, insbesondere solange noch keine Applikationen für eine Schemaversion implementiert sind. Folgerichtig können Situationen entstehen, wo selbst eingefrorene Schemaversionen noch in eingeschränktem Umfang änderbar wären, auch wenn dies i.Allg. nicht der Fall ist. Daher haben wir eine Kombination der direkten Schemaänderung mit dem Versionierungsansatz auf der Basis des Datenmodells von O_2 untersucht [FL96] (siehe Abschnitt 5.4.2). Dort konnten wir durch eine Klassifikation der Schemaänderungsprimitive feststellen, ob den im Einzelfall gegebenen Umständen zufolge die Ableitung einer neuen Schemaversion erforderlich ist oder nicht. Auf diesem Wege kann die Zahl entstehender Schemaversionen und damit auch der sich ergebende Verwaltungsaufwand reduziert werden.

- Die in Datenbanksystemen benötigten Änderungsoperationen lassen sich drei elementaren Kategorien zuordnen:
  - Zunächst sind die typischen Änderungsoperationen in Datenbanken zu nennen, die jeweils nur den Zustand eines Objektes, also die Werte seiner Attribute betreffen, z.B. die Änderung des Gehalts eines Angestellten. Derartige Änderungen werden von allen Datenbanksystemen unterstützt.
  - Die Kategorie der Schemaänderungen, die i.Allg. alle Objekte einer veränderten Klasse betreffen, hatten wir in dieser Arbeit ausführlich untersucht. Als Beispiel sei hier die Ergänzung der Modellierung der Angestellten um ihre Abteilungsnummer genannt.
  - Schließlich bestehen Änderungsanforderungen, die die Struktur und das Verhalten einzelner Objekte betreffen. Die Migration eines Studenten in einen Angestellten bei Abschluß des Studiums und Aufnahme eines Beschäftigungsverhältnisses ist ein typisches Beispiel für diese Kategorie. So geartete Änderungsanforderungen treten insbesondere in objektorientierten Datenbanksystemen in Erscheinung, weil nur diese das Konzept der Objektidentitäten explizit unterstützen. Die Studiums- und die Beschäftigungssituation werden als Beispiele für *Rollen* angesehen, die ein Objekt — auch gleichzeitig — spielen und dynamisch annehmen und aufgeben kann. Entsprechend flexible Rollenmodelle werden von existierenden Datenbanksystemen praktisch nicht unterstützt.

  Ein Modell, das Änderungsoperationen für all diese Kategorien anbietet, stellt nun offensichtlich ein besonders erstrebenswertes Ziel dar. Auf dem Wege dorthin stellt sich zuallererst die Frage, ob sich das Ziel durch eine Integration von Konzepten der Schemaevolution mit denen eines Rollenmodells erreichen läßt. In [Lau99a] haben wir gezeigt, daß diese Frage positiv beantwortet werden kann. Dort gelang uns die Kombination aus dem hier vorgestellten Schemaversionierungsmodell und dem Rollenmodell OPAQUE [Lan95, LL98, Lau93]. Diese Integration wurde bisher jedoch noch nicht abschließend untersucht und fand daher hier keine weitergehende Berücksichtigung.

9.3 Ausblick

Die vorliegende Abhandlung kann in generalisierender Weise als Beitrag verstanden werden, der die Sicherstellung fortwährender Kongruenz in rasch veränderlichen Anwendungsgebieten zum Ziel hat. Häufig sich ändernde Diskurswelten verlangen entsprechend flexible Möglichkeiten der Anpassung bestehender Modellierungen. Dabei stellt die Erhaltung der Betriebsfähigkeit früherer Modellierungen einen an Bedeutung zunehmenden Aspekt dar.

Für uns steht daher zunächst der Ausbau des Ansatzes im Vordergrund. Zu den konkret anstehenden Arbeiten gehört die Evaluierung in verschiedenen Anwendungsszenarien, um die Erreichbarkeit der gesteckten Ziele durch die entwickelten Konzepte auch praxisnah belegen zu können. Zum anderen bilden die Vervollständigung des Schemaassistenten und die Entwicklung weiterer Werkzeuge entsprechend den Vorschlägen aus Abschnitt 7.1.2.2 sowie weitere Effizienzverbesserungen wichtige Anknüpfungspunkte in Bezug auf die Prototyprealisierung.

Als über Realisierungsaufgaben hinausgehendes Ziel bietet sich an, das COAST-Modell um Konzepte zur Durchführung komplexer Schemaänderungen zu erweitern. Dazu wäre neben dem Angebot komplexer Schemaänderungsoperationen insbesondere zu untersuchen, welche Konsequenzen sich hieraus für die Spezifikation von Konvertierungsfunktionen und Propagationssteuerung sowie für die Algorithmen zur verzögerten Durchführung ergeben. Klar ersichtlich ist, daß diese und andere denkbare Erweiterungen weitreichende Folgen in Bezug auf die Konzeption

des Modells haben und von daher selbst wieder Gegenstand sorgfältiger und systematischer Untersuchungen sein müssen. Bezüglich der beschriebenen Möglichkeiten der Übertragung unserer Ergebnisse auf angrenzende Arbeitsbereiche haben wir derartige Untersuchungen bereits begonnen. Die dabei erreichten Zwischenergebnisse waren so erfolgversprechend, daß sie uns in naher Zukunft weitere nutzbringende Erweiterungen erwarten lassen.

Anhang A

Übersicht: Definitionen, Invarianten und Regeln

Dieses Kapitel des Anhangs gibt einen Überblick über die verwendeten Definitionen, Invarianten und Regeln. Diese sind jeweils tabellarisch zusammengestellt und (bei Invarianten innerhalb der Kategorien) dem Namen nach alphabetisch sortiert.

A.1 Definitionen

Formel	Name der Definition	Nr.	Seite
$dt(sv)$	Ableitungszeit einer Schemaversion	5.22	141
db	Datenbank	5.20	138
	Datenbank (referenzielle Integrität)	5.21	138
$csv(c)$	Erzeugerschemaversion einer Klasse	5.28	147
$csv(o)$	Erzeugerschemaversion eines Objektes	6.2	183
$oct(o)$	Erzeugungszeit eines Objektes	6.2	183
$ovct(o)$	Erzeugungszeit einer Objektversion	6.2	183
$xcf_{c,y\leftarrow x}$	Extrakonvertierungsfunktion	6.13	220
$oId, sId, svId, cId,$	Herkunft eines Merkmals einer Klasse	5.16	127
$oId, sId, svId, cId,$ $aId, mId, pId, dbId$	Identifikatoren	5.1	117
c	Klasse	5.8	123
$classes(s)$, $(classes(sv))$	Klassen eines Schemas (einer Schemaversion)	5.15	127
CDT: $(sv(c), <^1_{sv,c})$	Klassenableitungsbaum einer Klasse	5.30	147
CCT_c: $(sv(c), <^1_{cc})$	Klassenkonvertierungsbaum einer Klasse	6.6	205
CCG: $(sv(c), <^1_{cc})$	Klassenkonvertierungsgraph einer Klasse	6.14	222
CIG: $(classes(sv), <^1_c)$	Klassenvererbungsgraph einer Schemaversion	5.9	124
$sv.c$	Klassenversion	5.26	146
$cv(c)$	Klassenversionen einer Klasse	5.27	146
$scp_c(sv_x, sv_y)$	kleinste gemeinsame Vorgängerschemaversion	6.8	207
$cf_{c,z\leftarrow y} \circ cf_{c,y\leftarrow x}$	Komposition von Konvertierungsfunktionen		
$\{s, c, m, d\} \circ \{s, c, m, d\}$	Komposition von Propagationsflags		
$cf_{c,y\leftarrow x}$	Konvertierungsfunktion	6.3	184
	Konvertierungspfad	6.7	205

Formel	Name der Definition	Nr.	Seite
$succ(sv.c)$ $(dsucc(sv.c))$	(direkte) Nachfolgerklassenversionen	5.31	148
$succ(sv)$ $(dsucc(sv))$	(direkte) Nachfolgerschemaversionen	5.24	144
$NS: \ldots Names$	Namen	5.1	117
$name(sv.c)$	Name einer Klassenversion		
$pred(c)$ $(dpred(c))$	(direkte) Oberklassen	5.10	125
$T_v \leq_t T_u$	Ober- und Untertypen	5.7	122
o	Objekt (unversioniert)	5.4	120
o	Objekt (versioniert)	6.2	183
$sv.o$	Objektversion	6.2	183
	Objektwerte (logische)	6.4	189
	Objektversion (überflüssig, ableitbar, benötigt)	7.1	252
$cst(sv.c)$, $mst(sv.c)$, $dst(sv.c)$	Propagationsbäume	6.11	213
s, c, m, d	Propagationsflags		
$cpo(sv, sv', c)$	Propagationsflags einer Klasse	6.5	193
	regulärer Konvertierungspfad	6.9	209
$bcf_{c,u \leftarrow v}$	Rückwärtskonvertierungsfunktion	6.3	184
s_{struct}	strukturelles Schema	5.11	125
	Konsistenz strukturelles Schema	5.17	137
s_{behav}	verhaltensmäßiges Schema	5.14	126
	Konsistenz verhaltensmäßiges Schema	5.18	137
s	unversioniertes Schema	5.15	127
	Konsistenz unversioniertes Schema	5.19	137
s	versioniertes Schema	5.25	145
	Konsistenz versioniertes Schema	5.32	150
SDG: $(sv(s), <^1_{sv})$	Schemaableitungsgraph eines Schemas	5.23	143
sv	Schemaversion	5.22	141
$sv(c)$	Schemaversionen einer Klasse	5.29	147
$sv(s)$	Schemaversionen eines Schemas	5.25	145
sig	Signatur	5.12	126
$depth(xcf_{c,y \leftarrow x})$	Tiefe einer Extrakonvertierungsfunktion	6.13	220
$depth_c(sv)$	Tiefe einer Klasse in einer Schemaversion	6.12	219
$level_c$	Tiefe eines Konvertierungspfades	6.15	223
$t \in T^{cId}$	Typ	5.5	120
$succ(c)$ $(dsucc(c))$	(direkte) Unterklassen	5.10	125
$origin(ov)$	Ursprungsschemaversion einer Objektversion	6.2	183
$pred(sv.c)$ $(dpred(sv.c))$	(direkte) Vorgängerklassenversionen	5.31	148
$pred(sv)$ $(dpred(sv))$	(direkte) Vorgängerschemaversionen	5.24	144
$fcf_{c,v \leftarrow u}$	Vorwärtskonvertierungsfunktion	6.3	184
$v \in V^{oId}$	Wert	5.3	119
$dom_{_} \ldots$	Wertebereich (atomar)	5.2	118
$dom^{cId'}(t)$	Wertebereich eines Typs t über $cId' \subseteq cId$	5.6	121
	Wohldefiniertheit von Schemaableitungsgraphen	6.10	212
c_o, **Object**	Wurzelklasse jedes Klassenvererbungsgraphen	5.8, 5.9	123, 124
sv_0, **RootSV**	Wurzelschemaversion jedes Schemaableitungsgraphen	5.22, 5.23	141, 143
$IAS(sv)$, $IAS(sv.c)$	Zugriffsbereich einer Schema- bzw. Klassenversion	6.1	182
$state(sv)$	Zustand einer Schemaversion	5.22	141

A.2 Invarianten

Kategorie	Name der Invarianten	Nr.	Seite
Eindeutigkeit	Eindeutigkeit der Herkunft	5.2	128
	Eindeutigkeit der Namen	5.1	128
physikalische Speicherstruktur	Lokalität	7.4	263
	Löschung	7.3	261
	Objektstruktur	7.2	246
	Sichtbarkeit	7.1	245
Propagationsflags	Erzeugungsflag, c	6.2	194
	Löschungsflag, d	6.4	200
	Modifikationsflag, m	6.3	195
	Schnappschußflag, s	6.1	193
(Mehrfach-) Vererbung	Klassenvererbungsgraph, CIG	5.4	129
	strukturelle Typkonformität der Klassenvererbung	5.7	130
	Mehrfachvererbung	5.8	133
	Minimalität Klassenvererbungsgraph	5.5	129
	Vollständige Vererbung	5.6	129
Verhalten	Überladen von Methoden	5.9	134
	verhaltensmäßige Typkonformität der Klassenvererbung	5.10	136
versionierte Schemata	Eindeutigkeit der Klassenversionen	5.12	149
	Eindeutigkeit der Schemaversionen	5.11	149
	Minimalität Schemaableitungsgraph	5.14	150
	Vollständigkeit des Schemas	5.13	149
Vollständigkeit	Vollständigkeit des Schemas	5.3	128

A.3 Regeln

Name der Regel	Nr.	Seite
Ableiten einer Schemaversion	5.1	158
Definieren von Konvertierungsfunktionen bei Schemaableitung	6.1	188
Definieren von Extrakonvertierungsfunktionen bei Schemaableitung	6.3	223
Einfrieren und Auftauen einer Schemaversion	5.4	160
Löschen einer Schemaversion	5.3	159
Propagationsflags von Vorwärts- und Rückwärtskonvertierungsfunktionen	6.2	191
Propagationsflags von Extrakonvertierungsfunktionen	6.4	224
Verändern einer Schemaversion	5.2	159

Anhang B

Die COAST-ODL

Im Verlaufe der Arbeit wurden an zahlreichen Stellen insbesondere in Abschnitt 5.5 einzelne Primitive der COAST-ODL vorgestellt. Um einen besseren Überblick zu geben, werden diese hier gesammelt in einer erweiterten BNF aufgeführt. Weitergehende Ausführungen zur ODL finden sich in [Her99].

```
ODL_EXPR ::= SCHEMA_CHANGE

SCHEMA_CHANGE ::=   SCHEMA_CREATE
                  | SCHEMA_MODIFY
                  | SCHEMA_RENAME
                  | SCHEMA_DELETE

SCHEMA_CREATE  ::= [create]  schema sname { SV_CREATE* };
SCHEMA_MODIFY  ::= modify    schema sname { SV_CHANGE* };
SCHEMA_RENAME  ::= rename    schema sname to sname';
SCHEMA_DELETE  ::= delete    schema sname;

SV_CHANGE ::=   SV_CREATE
              | SV_MODIFY
              | SV_RENAME
              | SV_DELETE
              | SV_FREEZE
              | SV_UNFREEZE

SV_CREATE    ::= [create]   schemaversion svname { CLASS_CREATE* };
SV_MODIFY    ::= modify     schemaversion svname
                            { SV_CONTENTS_CHANGE* };
SV_RENAME    ::= rename     schemaversion svname to svname';
SV_DELETE    ::= delete     schemaversion svname;
SV_FREEZE    ::= freeze     schemaversion svname;
SV_UNFREEZE  ::= unfreeze   schemaversion svname;

SV_CONTENTS_CHANGE ::=   CLASS_CHANGE
                       | DERIVATION_CHANGE
```

```
CLASS_CHANGE  ::=    CLASS_CREATE
                   | CLASS_MODIFY
                   | CLASS_RENAME
                   | CLASS_DELETE
                   | CLASS_INTEGRATE

CLASS_CREATE     ::= [create]   class classname {
                          [inherit { INHERIT_CREATE* } ]
                          [attributes { ATTRIBUTE_CREATE* } ]
                          [methods { METHOD_CREATE* } ] };
CLASS_MODIFY     ::= modify     class classname {
                          [inherit { INHERIT_CHANGE* } ]
                          [attributes { ATTRIBUTE_CHANGE* } ]
                          [methods { METHOD_CHANGE* } ] };
CLASS_RENAME     ::= rename     class classname to classname';
CLASS_DELETE     ::= delete     class classname;
CLASS_INTEGRATE  ::= integrate  [sub[classes]] [comp[classes]]
                                class classname from svname
                          [FWD_PROPAGATION]
                          [BWD_PROPAGATION];

INHERIT_CHANGE  ::=    INHERIT_CREATE
                     | INHERIT_DELETE

INHERIT_CREATE  ::= [create]  inheritname;
INHERIT_DELETE  ::= delete    inheritname;

ATTRIBUTE_CHANGE  ::=    ATT_CREATE
                       | ATT_OVERWRITE
                       | ATT_DELETE_OVERWRITE
                       | ATT_RETYPE
                       | ATT_RENAME
                       | ATT_DELETE

ATT_CREATE            ::= [create]          attributename TYPE;
ATT_OVERWRITE         ::= overwrite         attributename to TYPE;
ATT_DELETE_OVERWRITE  ::= delete overwrite  attributename;
ATT_RETYPE            ::= retype            attributename to TYPE;
ATT_RENAME            ::= rename            attributename
                                            to attributename';
ATT_DELETE            ::= delete            attributename;

METHOD_CHANGE  ::=    METHOD_CREATE
                    | METHOD_RENAME
                    | METHOD_DELETE

METHOD_CREATE  ::= [create]  TYPE methodname
                             (TYPE paraname_1, ..., TYPE paraname_n);
METHOD_RENAME  ::= rename    TYPE methodname
                             (TYPE paraname_1, ..., TYPE paraname_n)
                             to methodname';
METHOD_DELETE  ::= delete    TYPE methodname
                             (TYPE paraname_1, ..., TYPE paraname_n);
```

```
TYPE ::=  ATOMIC_TYPE
        | CONTAINER_TYPE
        | TUPLE_TYPE
        | CLASS_REF_TYPE

ATOMIC_TYPE ::=  bool
               | int
               | real
               | char
               | string
               | date

CONTAINER_TYPE ::=  list   ( TYPE )
                  | set    ( TYPE )
                  | bag    ( TYPE )
                  | array  ( TYPE )
                  | dag    ( TYPE TYPE )

TUPLE_TYPE ::= tuple ( TYPE attributename_1, ..., TYPE attributename_n )

CLASS_REF_TYPE ::= classname

DERIVATION_CHANGE ::=  DERIVATION_CREATE
                     | DERIVATION_MODIFY
                     | DERIVATION_DELETE

DERIVATION_CREATE ::= [create] derivation for class classname
                               to class classname
                               in schemaversion svname
                               [FWD_PROPAGATION]
                               [BWD_PROPAGATION]
                               [FWD_CONVERSION]
                               [BWD_CONVERSION];
DERIVATION_MODIFY ::= modify   derivation for class classname
                               [FWD_PROPAGATION]
                               [BWD_PROPAGATION]
                               [FWD_CONVERSION]
                               [BWD_CONVERSION];
DERIVATION_DELETE ::= delete   derivation for class classname;

FWD_PROPAGATION ::= forward propagation
                            S +|- C +|- M +|- D +|-
BWD_PROPAGATION ::= backward propagation
                            C +|- M +|- D +|-
FWD_CONVERSION  ::= forward conversion
                            { Konvertierungsanweisungen }
BWD_CONVERSION  ::= backward conversion
                            { Konvertierungsanweisungen }
```

Literaturverzeichnis

[AA93] Reda Alhajj und M. Erol Arkun. An Object Algebra for Object-Oriented Database Systems. *Database*, 24(3):13–22, August 1993.

[AAA+94] Serge Abiteboul, Michel E. Adiba, Jim Arlow, Pasquale Armenise, Sergio Bandinelli, Luciano Baresi, Philippe Brèche, Frank Buddrus, Christine Collet, P. Corte, Thierry Coupaye, Claude Delobel, Wolfgang Emmerich, Guy Ferran, Fabrizio Ferrandina, Alfonso Fuggetta, Carlo Ghezzi, Sven-Eric Lautemann, Luigi Lavazza, Joëlle Madec, Mark Phoenix, Sabine Sachweh, Wilhelm Schäfer, Cassio Souza dos Santos, G. Tigg und Roberto Zicari. The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes. In *Proc. of the 1st Asian Pacific Software Engineering Conf.*, Seiten 10–19, Tokio, Japan, 1994. IEEE Computer Society Press.

[AAA+96] Serge Abiteboul, Michel E. Adiba, Jim Arlow, Pasquale Armenise, Sergio Bandinelli, Luciano Baresi, Philippe Brèche, J. Brunsman, Frank Buddrus, Christine Collet, P. Corte, Thierry Coupaye, Claude Delobel, Wolfgang Emmerich, Guy Ferran, Fabrizio Ferrandina, Alfonso Fuggetta, Carlo Ghezzi, Sven-Eric Lautemann, Luigi Lavazza, Joëlle Madec, Mark Phoenix, Sabine Sachweh, Wilhelm Schäfer, Cassio Souza dos Santos, G. Tigg und Roberto Zicari. The GOODSTEP Project Final Report. In *Datenbank Rundbrief*, Seiten 14–40. Gesellschaft für Informatik, Mai 1996.

[AB91] Serge Abiteboul und Anthony Bonner. Objects and Views. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seiten 238–247, Denver, Colorado, USA, Mai 1991. ACM Press. SIGMOD Record, Band 20, Nr. 2, Juni 1991.

[ABDW+89] Malcolm P. Atkinson, François Bancilhon, David J. De Witt, Klaus R. Dittrich, David Maier und Stanley Zdonik. The Object-Oriented Database System Manifesto. In Won Kim, Jean-Marie Nicolas und Shojiro Nishio, Hrsg., *Proc. of the 1st Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Seiten 40–57, Kyoto, Japan, Dezember 1989. Elsevier Science Publishers B.V.

[ABGS91] Rakesh Agrawal, S. J. Buroff, N. H. Gehani und D. Shasha. Object Versioning in Ode. In *Proc. of the 7th Int'l Conf. on Data Engineering (ICDE)*, Seiten 446–455, Kobe, Japan, April 1991. IEEE, IEEE Computer Society Press.

[ADHP00] Malcolm P. Atkinson, Misha Dmitriev, Craig Hamilton und Tony Printezis. Scalable and Recoverable Implementation of Object Evolution for the PJama Platform. In *Proc. of the 9th Int'l Workshop on Persistent Object Systems*, Lillehammer, Norwegen, September 2000. Springer-Verlag.

[AFY98] Paul Allen, Stuart Frost und Edward Yourdon. *Component Based Development for Enterprise Systems: Applying the Select Perspective.* Managing Object Technology Series, Band 12. Cambridge University Press, Juni 1998.

[AG89] Rakesh Agrawal und Narain H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In James Clifford, Bruce Lindsay und David Maier, Hrsg., *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seiten 36–45, Portland, Oregon, Juni 1989. ACM, ACM Press. SIGMOD Record, Band 18, Nr. 2, Juni 1989.

[AHLPZ96] Eric Andonoff, Gilles Hubert, Annig Le Parc und Gilles Zurfluh. Integrating Versions in the OMT Model. In Bernhard Thalheim, Hrsg., *Proc. of the 15th Int'l Conf. on Conceptual Modeling (ER)*, Seiten 472–487, Cottbus, Deutschland, Oktober 1996. Springer-Verlag. Lecture Notes in Computer Science Nr. 1157.

[AJ89] Rakesh Agrawal und H. V. Jagadish. On Correctly Configuring Versioned Objects. In Peter M. G. Apers und Gio Wiederhold, Hrsg., *Proc. of the 15th Int'l Conf. on Very Large Databases (VLDB)*, Seiten 367–374, Amsterdam, Niederlande, August 1989. Morgan Kaufmann Publishers.

[AJ96] Petia Assenova und Paul Johannesson. Improving Quality in Conceptual Modelling by the Use of Schema Transformations. In Bernhard Thalheim, Hrsg., *Proc. of the 15th Int'l Conf. on Conceptual Modeling (ER)*, Seiten 277–291, Cottbus, Deutschland, Oktober 1996. Springer-Verlag. Lecture Notes in Computer Science Nr. 1157.

[AJEFL95] Lina Al-Jadir, Thibault Estier, Gilles Falquet und Michel Léonard. Evolution Features of the F2 OODBMS. In Tok Wang Ling und Yoshifumi Masunaga, Hrsg., *Proc. of the 4th Int'l Conf. on Database Systems for Advanced Applications (DASFAA)*, Seiten 284–291, Singapur, April 1995. World Scientific Publishing Co. Pte Ltd. In Advanced Database Research and Development Series, Band 5.

[AJFL93] Lina Al-Jadir, Gilles Falquet und Michel Léonard. Context Versions in an Object-Oriented Model. In Vladimír Mařík, Jiří Lažanský und Roland R. Wagner, Hrsg., *Proc. of the 4th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Seiten 24–35, Prag, Tschechische Republik, September 1993. Springer-Verlag. Lecture Notes in Computer Science Nr. 720.

[AL90] Pierre America und Frank van der Linden. A Parallel Object-Oriented Language with Inheritance and Subtyping. In Norman Meyrowitz, Hrsg., *Proc. of the 5th Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), gemeinsam veranstaltet mit 4th European Conf. on Object-Oriented Programming (ECOOP)*, Seiten 161–168, Ottawa, Kanada, Oktober 1990. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 25, Nr. 10, Oktober 1990.

[ALP91] José Andany, Michel Léonard und Carole Palisser. Management Of Schema Evolution In Databases. In Guy M. Lohman, Amílcar Sernades und Rafael Camps, Hrsg., *Proc. of the 17th Int'l Conf. on Very Large Databases (VLDB)*, Seiten 161–170, Barcelona, Spanien, September 1991. Morgan Kaufmann Publishers.

[AN91] Rafi Ahmed und Shamkant B. Navathe. Version Management of Composite Objects in CAD Databases. In James Clifford und Roger King, Hrsg., *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seiten 218–227, Denver, Colorado, USA, Mai 1991. ACM, ACM Press. SIGMOD Record, Band 20, Nr. 2, Juni 1991.

[Apo00] Sabbas Apostolidis. Der Schema-Evolutions-Assistent (SEA): Ein Werkzeug zur Unterstützung evolutionärer Schemaänderungen in einem objektorientierten Datenbankmanagementsystem mit Schemaversionierung. Diplomarbeit, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, Juni 2000. In Vorbereitung.

[Ari91] Gad Ariav. Temporally oriented data definitions: Managing schema evolution in temporally oriented databases. *Data & Knowledge Engineering*, 6:451–467, 1991.

[Ban93] François Bancilhon. Object Database Systems: Functional Architecture. In Shojiro Nishio und Akinori Yonezawa, Hrsg., *Proc. of Object Technologies for Advanced Software*, Seiten 163–175, Kanazawa, Japan, November 1993. Springer-Verlag. Lecture Notes in Computer Science Nr. 742.

[Bar91] Gilles Barbedette. Schema Modifications in the LISPO_2 Persistent Object-Oriented Language. In Pierre America, Hrsg., *Proc. of the 5th European Conf. on Object-Oriented Programming (ECOOP)*, Seiten 77–96, Genf, Schweiz, Juli 1991. Springer-Verlag. Lecture Notes in Computer Science Nr. 512.

[BB88] Anders Björnerstedt und Stefan Britts. AVANCE: An Object Management System. In Norman Meyrowitz, Hrsg., *Proc. of the 3rd Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 206–221, San Diego, Kalifornien, September 1988. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 23, Nr. 11, November 1988.

[BB95] Mohamed Bouneffa und Nacer Boudijlida. Managing Schema Changes in Object-Relationship Databases. In Michael P. Papazoglou, Hrsg., *Proc. of the 14th Int'l Conf. on Object-Oriented and Entity-Relationship Modeling (OOER)*, Seiten 113–122, Gold Coast, Australien, Dezember 1995. Springer-Verlag. Lecture Notes in Computer Science Nr. 1021.

[BBP96] Nacer Boudjlida, M. A. Bouneffa und Oliver Perrin. Object and Schema Versioning and Restructuring in Databases. In *Proc. of the 16th Annual Database Conference (DATASEM)*, Seiten 159–169, Brünn, Tschechische Republik, Oktober 1996.

[BCG+87] Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, Nathaniel Ballou und Hyoung-Joo Kim. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, Januar 1987.

[BDK92] François Bancilhon, Claude Delobel und Paris Kanellakis, Hrsg. *Building an Object-Oriented Database System — The Story of O_2*. Morgan Kaufmann Publishers, San Mateo, Kalifornien, 1992.

[Bel95] Tarik Beldjilali. Managing Schema Evolution in an Object Oriented Database Populated by Instances. In Norman Revell und A Min Tjoa, Hrsg., *Proc. of the 6th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Seiten 214–222, London, Großbritannien, September 1995. Workshop Proceedings.

[Ber92] Elisa Bertino. A View Mechanism for Object-Oriented Databases. In Alain Pirotte, Claude Delobel und Georg Gottlob, Hrsg., *Proc. of the 3rd Int'l Conf. on Extending Database Technology (EDBT)*, Seiten 136–151, Wien, Österreich, März 1992. Springer-Verlag. Lecture Notes in Computer Science Nr. 580.

[Ber94] Paul L. Bergstein. *Managing the Evolution of Object-Oriented Systems*. Dissertation, College of Computer Science, Northeastern University, Juni 1994. 151 Seiten.

[Ber97] Paul L. Bergstein. Maintenance of Object-Oriented Systems During Structural Schema Evolution. *Theory And Practice of Object Systems (TAPOS)*, 3(3):185–212, Juli 1997.

[BF95] François Bancilhon und Guy Ferran. The ODMG Standard for Object Databases. In Tok Wang Ling und Yoshifumi Masunaga, Hrsg., *Proc. of the 4th Int'l Conf. on Database Systems for Advanced Applications (DASFAA)*, Seiten 273–283, Singapur, April 1995. World Scientific Publishing Co. Pte Ltd. In Advanced Database Research and Development Series, Band 5.

[BFK95] Philippe Brèche, Fabrizio Ferrandina und Martin Kuklok. Simulation of Schema Change using Views. In Norman Revell und A Min Tjoa, Hrsg., *Proc. of the 6th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Seiten 247–258, London, Großbritannien, September 1995. Springer-Verlag. Lecture Notes in Computer Science Nr. 978.

[BGL97] Frank Buddrus, Heino Gärtner und Sven-Eric Lautemann. First Steps to a Formal Framework for Multilevel Database Modifications. In Abdelkader Hameurlain und A Min Tjoa, Hrsg., *Proc. of the 8th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Seiten 240–251, Toulouse, Frankreich, September 1997. Springer-Verlag. Lecture Notes in Computer Science Nr. 1308.

[BH89] Anders Björnerstedt und Christer Hultén. Version Control in an Object-Oriented Architecture. In *[KL89]*, Kapitel 18, Seiten 451–485. Addison Wesley, 1989.

[BH93] Paul L. Bergstein und Walter L. Hürsch. Maintaining Behavioral Consistency during Schema Evolution. In Shojiro Nishio und Akinori Yonezawa, Hrsg., *Proc. of Int'l Symposium on Object Technologies for Advanced Software*, Seiten 176–193, Kanazawa, Japan, November 1993. Springer-Verlag. Lecture Notes in Computer Science Nr. 742.

[BK93] Peter J. Barclay und Jessie B. Kennedy. Viewing Objects. In Michael F. Worboys und A. F. Grundy, Hrsg., *Proc. of the 11th British National Conf. on Databases (BNCOD)*, Seiten 93–110, Keele, Großbritannien, Juli 1993. Springer-Verlag. Lecture Notes in Computer Science Nr. 696.

[BKK88] Jay Banerjee, Won Kim und Kyung-Chang Kim. Queries in Object-Oriented Databases. In *Proc. of the 4th Int'l Conf. on Data Engineering (ICDE)*, Seiten 31–38, Los Angeles, Kalifornien, 1988. IEEE, IEEE Computer Society Press.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim und Henry F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In Umeshwar Dayal und Irv Traiger, Hrsg., *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seiten 311–322, San Francisco, Kalifornien, Mai 1987. ACM, ACM Press. SIGMOD Record, Band 16, Nr. 3, Dezember 1987.

[BL98] Frank Buddrus und Sven-Eric Lautemann. Internet, Java, Datenbanken. In *Kooperationspartner in Forschung und Innovation*, Seiten 11–12. Hessisches Ministerium für Wissenschaft und Kunst, März 1998. CeBIT Messebroschüre.

[BLB97] Frank Buddrus, Sven-Eric Lauteman und Marco Bellavia. Making O_2 become a WWW Server. In Maria E. Orlowska und Roberto Zicari, Hrsg., *Proc. of the 4th Int'l Conf. on Object-Oriented Information Systems (OOIS)*, Seiten 168–178, Brisbane, Australien, November 1997. Springer-Verlag.

[BM88] David Beech und Brom Mahbod. Generalized Version Control in an Object-Oriented Database. In *Proc. of the 4th Int'l Conf. on Data Engineering (ICDE)*, Seiten 14–22, Los Angeles, Kalifornien, 1988. IEEE, IEEE Computer Society Press.

[BM93a] Elisa Bertino und Lorenzo Martino. *Object-Oriented Database Systems: Concepts and Architectures*. International Computer Science Series. Addison Wesley, 1993.

[BM93b] Kwang June Byeon und Dennis McLeod. Towards the Unification of Views and Versions for Object Databases. In Shojiro Nishio und Akinori Yonezawa, Hrsg., *Proc. of Object Technologies for Advanced Software*, Seiten 220–236, Kanazawa, Japan, November 1993. Springer-Verlag. Lecture Notes in Computer Science Nr. 742.

[BMO+89] Robert Bretl, David Maier, Allen Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams und Monty Williams. The GemStone Data Management System. In *[KL89]*, Kapitel 12, Seiten 283–308. Addison Wesley, 1989.

[Boo95] Grady Booch. *Object Solutions: Managing the Object-Oriented Project*. Object Technology Series. Addison Wesley, 1995.

[BOS91] Paul Butterworth, Allen Otis und Jacob Stein. The GemStone Object Database Management System. *Communications of the ACM*, 34(10):64–77, Oktober 1991.

[BP94] Alexandros Biliris und Euthimios Panagos. *EOS User's Guide*. AT&T Bell Laboratories, Murray Hill, NJ 07974, Oktober 1994. Release 2.2, 51 Seiten.

[BP95] Nacer Boudjlida und Oliver Perrin. An Open Approach for Data Integration. In *Proc. of the 2nd Int'l Conf. on Object Oriented Information Systems (OOIS)*, Seiten 94–98, Dublin, Großbritannien, Dezember 1995. Springer-Verlag.

[BP96] Alexandros Biliris und Euthimios Panagos. The BeSS Object Storage Manager: Architecture Overview. *SIGMOD Record*, 25(3):53–58, September 1996.

[Bra92] Svein Erik Bratsberg. Unified Class Evolution by Object-Oriented Views. In Günther Pernul und A Min Tjoa, Hrsg., *Proc. of the 11th Int'l Conf. on the Entity-Relationship Approach (ER)*, Seiten 423–439, Karlsruhe, Deutschland, Oktober 1992. Springer-Verlag. Lecture Notes in Computer Science Nr. 645.

[Bra93] Svein Erik Bratsberg. *Evolution and Integration of Classes in Object-Oriented Databases*. Dissertation, The Norwegian Institute of Technology, University of Trondheim, Juni 1993.

[Bre90] Yuri Breitbart. Multidatabase Interoperability. *SIGMOD Record*, 19(3):53–60, September 1990.

[Brè96] Philippe Brèche. Advanced Primitives for Changing Schemas of Object Databases. In Panos Constantopoulos, John Mylopoulos und Yannis Vassiliou, Hrsg., *Proc. of the 7th Conf. on Advanced Information Systems Engineering (CAiSE)*, Seiten 476–495, Kreta, Griechenland, Mai 1996. Springer-Verlag. Lecture Notes in Computer Science Nr. 1080.

[BWJ96] Marie-Jo Bellosta, Robert Wrembel und Geneviève Jomier. Management of Schema Versions and Versions of Schema Instance in a Multiversion Database. Vorläufige Version, 23 Seiten, 1996.

[BZ87] Toby Bloom und Stanley B. Zdonik. Issues in the Design of Object-Oriented Database Programming Languages. In Norman Meyrowitz, Hrsg., *Proc. of the 2nd Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 441–451, Orlando, Florida, Oktober 1987. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 22, Nr. 12, Dezember 1987.

[Car84] Luca Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D. B. MacQueen und G. Plotkin, Hrsg., *Proc. of the Int'l Symposium on Semantics of Data Types*, Seiten 51–72, Sophia-Antipolis, Frankreich, Juni 1984. Springer-Verlag. Lecture Notes in Computer Science Nr. 173.

[Cas95] Eduardo Casais. Managing Class Evolution in Object-Oriented Systems. In Oscar Nierstrasz und Dennis Tsichritzis, Hrsg., *Object-Oriented Software Composition*, Kapitel 8, Seiten 201–244. Prentice Hall, Dezember 1995.

[Cat96] R. G. G. Cattell, Hrsg. *The Object Database Standard: ODMG-93 (Release 1.2)*. Data Management Systems. Morgan Kaufmann Publishers, San Francisco, Kalifornien, 1996. Mit Beiträgen von Tom Atwood, Douglas Barry, Joshua Duhl, Jeff Eastman, Guy Ferran, David Jordan, Mary Loomis und Drew Wade.

[CB98] R. G. G. Cattell und Douglas K. Barry, Hrsg. *The Object Database Standard: ODMG 2.0.* Data Management Systems. Morgan Kaufmann Publishers, San Francisco, Kalifornien, Februar 1998. Mit Beiträgen von Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland und Drew Wade.

[CBB+00] R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda und Fernando Vélez, Hrsg. *The Object Data Standard: ODMG 3.0.* Data Management Systems. Morgan Kaufmann Publishers, San Francisco, Kalifornien, Januar 2000.

[CCC+99] Fabiano Cattaneo, Francesco Coda, Edoardo Corsetti, Alfredo Femminella, Wolfgang Gillrath, Hannes Händel, Claus Herold, Sven-Eric Lautemann, Stefano Panfilis, Thomas Plambeck, Wolfgang Rothe, Marcus Scholten, Joachim Seifert, Guido Weissbrich, Antonella Zanzi und Roberto Zicari. Industrial Use of the GOODSTEP Technology. In *Proc. of the 6th Int'l Conf. on Empirical Assessment & Evaluation in Software Engineering (EASE)*, Seiten 1–10, Staffordshire, Großbritannien, April 1999. Band 2: Experience Papers.

[CDWN93] Michael J. Carey, David J. De Witt und Jeffrey F. Naughton. The OO7 Benchmark. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seiten 12–21, Washington, DC, USA, Mai 1993.

[CFMS94] Silvana Castano, Maria Grazia Fugini, Giancarlo Martella und Pierangela Samarati. *Database Security.* Addison Wesley, September 1994.

[Che76] Peter Pin-Shan Chen. The Entity-Relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, März 1976. Auch in [Sto88].

[Chr75] Nicos Christofides. *Graph Theory.* Academic Press, 1975.

[CJ90] Wojciech Cellary und Geneviève Jomier. Consistency of Versions in Object-Oriented Databases. In Dennis McLeod, Ron Sacks-Davis und Hans-Jörg Schek, Hrsg., *Proc. of the 16th Int'l Conf. on Very Large Databases (VLDB)*, Seiten 432–441, Brisbane, Australien, August 1990. Morgan Kaufmann Publishers.

[CJ92] Wojciech Cellary und Geneviève Jomier. Consistency of Versions in Object-Oriented Databases. In *[BDK92]*, Kapitel 19, Seiten 447–462. Morgan Kaufmann Publishers, 1992.

[CJ94] Wojciech Cellary und Geneviève Jomier. Apparent Versioning and Concurrency Control in Object-Oriented Databases. In *Proc. of the Int'l Conf. on Computing and Information (ICCI)*, Peterborough, Ontario, Kanada, 1994.

[CJK91] Wojciech Cellary, Geneviève Jomier und Tomasz Koszlajda. Formal Model of an Object-Oriented Database with Versioned Objects and Schema. In *Proc. of the 2nd Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Seiten 239–244, Berlin, Deutschland, 1991. Springer-Verlag.

[CJR98] Kajal Claypool, Jin Jing und Elke A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. Technischer Bericht WPI-CS-TR-98-9, Worcester Polytechnic Institute, Worcester, Massachusetts 01609, Juni 1998.

[CK86] Hong-Tai Chou und Won Kim. A Unifying Framework for Version Control in a CAD Environment. In *Proc. of the 12th Int'l Conf. on Very Large Databases (VLDB)*, Seiten 336–344, Kyoto, Japan, August 1986. Morgan Kaufmann Publishers.

[CK88] Hong-Tai Chou und Won Kim. Versions and Change Notification in an Object-Oriented Database System. In *Proc. of the 25th Design Automation Conf.*, Seiten 275–281, Anaheim, Kalifornien, Juni 1988. ACM/IEEE, IEEE Computer Society Press.

[Cla92] Stewart M. Clamen. Type Evolution and Instance Adaption. Technischer Bericht CMU-CS-92-133R, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Juni 1992.

[Cla94] Stewart M. Clamen. Schema Evolution and Integration. *Distributed and Parallel Databases: An International Journal*, 2(1):101–126, Januar 1994.

[CM84] George P. Copeland und David Maier. Making Smalltalk a Database System. In Beatrice Yormark, Hrsg., *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seiten 316–325, Boston, Massachusetts, Juni 1984. ACM, ACM Press. SIGMOD Record, Band 14, Nr. 2, Juni 1984.

[CM95] Helen Campbell und Simon R. Monk. Changing the Database Paradigm - An Annotated Bibliography on the Adoption of Object-Oriented Technology in Database Systems. Technischer Bericht CO-95-04-02, Department of Computing, University of Central Lancashire, Preston, PR1 2TQ, Großbritannien, 1995.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, Juni 1970. Auch in [Sto88].

[Cod79] E. F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4):397–434, Dezember 1979. Auch in [Sto88].

[Con86] Dan Conde. Bibiliography on Version and Configuration Management. *ACM SIGSOFT Software Engineering Notes*, 11(3):81–84, Juli 1986.

[Coo89] William R. Cook. A Proposal for Making Eiffel Type-Safe. In S. Cook, Hrsg., *Proc. of the 3rd European Conf. on Object-Oriented Programming (ECOOP)*, Seiten 57–70, Nottingham, Großbritannien, Juli 1989. Cambridge University Press. Auch in The Computer Journal, Band 32(4), Seiten 290-340, British Computer Society, August 1989.

[CPLZ92a] Alberto Coen-Poresini, Luigi Lavazza und Roberto Zicari. Assuring Type-Safety of Object Oriented Languages. Technischer Bericht 4/92, Fachbereich Informatik, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, 1992.

[CPLZ92b] Alberto Coen-Poresini, Luigi Lavazza und Roberto Zicari. Overview and Progress Report of the ESSE Project: Supporting Object-Oriented Database Schema Analysis and Evolution. Technischer Bericht 6/92, Fachbereich Informatik, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, 1992.

[CPLZ92c] Alberto Coen-Poresini, Luigi Lavazza und Roberto Zicari. Static Type Checking of an Object-Oriented Database Schema. Technischer Bericht 5/92, Fachbereich Informatik, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, 1992.

[CPLZ92d] Alberto Coen-Poresini, Luigi Lavazza und Roberto Zicari. The ESSE Project: An Overview. In Qiming Chen, Yahiko Kambayashi und Ron Sacks-Davis, Hrsg., *Proc. of the 2nd Far-East Workshop on Future Database Systems*, Seiten 28–37, Kyoto, Japan, April 1992. World Scientific. Auch als Technischer Bericht 91-077, Politecnico di Milano, Italien, Dipartimento di Elettronica, Dezember 1991.

[CTR96] Viviane Crestana-Taube und Elke A. Rundensteiner. Consistent View Removal in Transparent Schema Evolution Systems. In *Proc. of the Int'l Workshop on Research Issues in Data Engineering – Interoperability of Nontraditional Database Systems (RIDE–NDS)*, Seiten 138–147, New Orleans, Louisiana, Februar 1996. IEEE Computer Society Press.

[DA99] Misha Dmitriev und Malcolm P. Atkinson. Evolutionary Data Conversion in the PJama Persistent Language. In Ana Moreira und Serge Demeyer, Hrsg., *Proc. of the 13th European Conf. on Object-Oriented Programming (ECOOP)*, Seiten 211–212, Lissabon, Portugal, Juni 1999. Springer-Verlag. Workshop Reader. Lecture Notes in Computer Science Nr. 1743. 1st ECOOP Workshop on Object-Oriented Databases.

[Dar1859] Charles Darwin. *The Origin of Species by Means of Natural Selection.* The works of Charles Darwin, Band 15. New York University Press [1988], 1859.

[Dea91] O. Deux et al. The O_2 System. *Communications of the ACM*, 34(10):34–48, Oktober 1991.

[DFG+97] Anne Doucet, Marie-Christine Fauvet, Stéphane Gançarski, Geneviève Jomier und Sophie Monties. Using Database Versions to Implement Temporal Integrity Constraints. In *Proc. of the 2nd Int'l Workshop on Constraints and Databases (CP)*, Seiten 219–233, Cambridge, Massachusetts, USA, August 1997.

[DGJM96a] Anne Doucet, Stéphane Gançarski, Geneviève Jomier und Sophie Monties. Integrity Constraints and Versions. In *Proc. of the 6th Int'l Workshop on Foundations of Models and Languages for Data and Objects: Integrity in Databases*, Dagstuhl, Deutschland, September 1996.

[DGJM96b] Anne Doucet, Stéphane Gançarski, Geneviève Jomier und Sophie Monties. Integrity Constraints in Multiversion Databases. In Ronald Morrison und Jessie B. Kennedy, Hrsg., *Proc. of the 14th British National Conf. on Databases (BNCOD)*, Seiten 56–73, Edinburgh, Großbritannien, Juli 1996. Springer-Verlag. Lecture Notes in Computer Science Nr. 1094.

[DL88] Klaus R. Dittrich und Raymond A. Lorie. Version Support for Engineering Database Systems. *ACM Transactions on Software Engineering*, 14(4):429–437, April 1988.

[Dol99] Alexander Doll. Der COAST-Schemamanager: Verwaltung und konsistenzerhaltende Änderung der versionierten Schemata eines Objektdatenbanksystems. Diplomarbeit, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, Dezember 1999.

[dSAD94] Cássio Souza dos Santos, Serge Abiteboul und Claude Delobel. Virtual Schemas and Bases. In Matthias Jarke, Janis Bubenko und Keith Jeffery, Hrsg., *Proc. of the 4th Int'l Conf. on Extending Database Technology (EDBT)*, Seiten 81–94, Cambridge, Großbritannien, März 1994. Springer-Verlag. Lecture Notes in Computer Science Nr. 779.

[DV95] Antonia Dattolo und Loia Vincenzo. Hypertext Version Management in an Actor-based Framework. In Juhani Iivari, Kalle Lyytinen und Matti Rossi, Hrsg., *Proc. of the 7th Conf. on Advanced Information Systems Engineering (CAiSE)*, Jyväskylä, Finnland, Juni 1995. Springer-Verlag. Lecture Notes in Computer Science Nr. 932.

[DZ91] Christine Delcourt und Roberto Zicari. The Design of an Integrity Consistency Checker (ICC) for an Object Oriented Database System. In Pierre America, Hrsg., *Proc. of the 5th European Conf. on Object-Oriented Programming (ECOOP)*, Seiten 97–117, Genf, Schweiz, Juli 1991. Springer-Verlag. Lecture Notes in Computer Science Nr. 512.

[Eig97] Patricia Eigner. Objektpropagation in einem Schemaversionierungsansatz: Entwicklung und prototypische Implementierung in einem objektorientierten Datenbanksystem. Diplomarbeit, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, Juli 1997.

[ES94] Susan Even und Markku Sakkinen. The safe use of polymorphism in the O_2C database language. Technischer Bericht 5/94, Fachbereich Informatik, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, Juni 1994. 27 Seiten.

[FAB+89] Daniel H. Fishman, Jurgen Annevelink, David Beech, E. C. Chow, Tim Connors, J. W. Davis, Waqar Hasan, C. G. Hoch, William Kent, S. Leichner, Peter Lyngbaek, Brom Mahbod, Marie-Anne Neimat, Tore Risch, Ming-Chien Shan und W. Kevin Wilkinson. Overview of the Iris DBMS. In *[KL89]*, Kapitel 10, Seiten 219–250. Addison Wesley, 1989.

[FBC+87] Daniel H. Fishman, David Beech, H. P. Cate, E. C. Chow, Tim Connors, J. W. Davis, N. Derrett, C. G. Hoch, William Kent, Peter Lyngbaek, Brom Mahbod, Marie-Anne Neimat, T. A. Ryan und Ming-Chien Shan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, Januar 1987.

[FD95] Edi Fontana und Yves Dannebouy. Schema Evolution by using Timestamped Versions and Lazy Strategy. In *Proc. of the 11th Conf. on Advanced Databases*, Seiten 43–60, Nancy, Frankreich, September 1995.

[FJL+88] Steve Ford, John Joseph, David E. Langworthy, David F. Lively, Girish Pathak, Edward R. Perez, Robert W. Peterson, Dinan M. Sparacin, Satish M. Thatte, David L. Wells und Sanjive Agarwala. ZEITGEIST: Database Support for Object-Oriented Programming. In Klaus R. Dittrich, Hrsg., *Advances in Object-Oriented Database Systems, 2nd Int'l Workshop on Object-Oriented Database Systems*, Seiten 23–42, Bad Münster, Deutschland, September 1988. Springer-Verlag. Lecture Notes in Computer Science Nr. 334.

[FL96] Fabrizio Ferrandina und Sven-Eric Lautemann. An Integrated Approach to Schema Evolution in Object Databases. In Dilip Patel, Yuan Sun und Shushma Patel, Hrsg., *Proc. of the 3rd Int'l Conf. on Object Oriented Information Systems (OOIS)*, Seiten 280–294, London, Großbritannien, Dezember 1996. Springer-Verlag.

[FMZ94a] Fabrizio Ferrandina, Thorsten Meyer und Roberto Zicari. Correctness of Lazy Database Updates for an Object Database System. In Malcolm P. Atkinson, David Maier und Véronique Benzaken, Hrsg., *Proc. of the 6th Int'l Workshop on Persistent Object Systems*, Seiten 284–301, Tarascon, Frankreich, September 1994. Springer-Verlag.

[FMZ94b] Fabrizio Ferrandina, Thorsten Meyer und Roberto Zicari. Implementing Lazy Database Updates for an Object Database System. In Jorge Bocca, Matthias Jarke und Carlo Zaniolo, Hrsg., *Proc. of the 20th Int'l Conf. on Very Large Databases (VLDB)*, Seiten 261–272, Santiago, Chile, September 1994. Morgan Kaufmann Publishers.

[FMZ95a] Fabrizio Ferrandina, Thorsten Meyer und Roberto Zicari. Measuring the Performance of Immediate and Deferred Updates in Object Database Systems. In *Proceedings of the OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*, Austin, Texas, Oktober 1995.

[FMZ+95b] Fabrizio Ferrandina, Thorsten Meyer, Roberto Zicari, Guy Ferran und Joëlle Madec. Schema and Database Evolution in the O_2 Object Database System. In Dayal Umeshwar, Peter M. D. Gray und Nishio Shojiro, Hrsg., *Proc. of the 21st Int'l Conf. on Very Large Databases (VLDB)*, Seiten 170–181, Zürich, Schweiz, September 1995. Morgan Kaufmann Publishers.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns*, Kapitel 4, Seiten 139–150. Professional Computing Series. Addison Wesley, 1995.

[GJ94] Stéphane Gançarski und Geneviève Jomier. Managing Entity Versions within their Contexts: A Formal Approach. In Dimitris Karagiannis, Hrsg., *Proc. of the 5th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Seiten 400–409. Springer-Verlag, September 1994. Lecture Notes in Computer Science Nr. 856.

[GJS96] James Gosling, Bill Joy und Guy Steele. *The Java Language Specification.* The Java Series. Addison Wesley, 1996.

[GJZ95] Stéphane Gançarski, Geneviève Jomier und Michel Zamfiroïu. A Framework for the Manipulation of a Multiversion Database. In Norman Revell und A Min Tjoa, Hrsg., *Proc. of the 6th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Seiten 247–256, London, Großbritannien, September 1995. Workshop Proceedings.

[GK88] Jorge F. Garza und Won Kim. Transaction Management in an Object-Oriented Database System. In Haran Boral und Per-Åke Larson, Hrsg., *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seiten 37–45, Chicago, Illinois, USA, September 1988. ACM Press. SIGMOD Record, Band 17, Nr. 3, Juni 1988.

[GKL94] David Garlan, Charles W. Krueger und Barbara Staudt Lerner. TransformGen: Automating the Maintenance of Structure-Oriented Environments. *ACM Transactions on Programming Languages and Systems*, 16(3):727–774, Mai 1994.

[GPZ88] Georg Gottlob, Paolo Paolini und Roberto Zicari. Properties and Update Semantics of Consistent Views. *ACM Transactions on Database Systems*, 13(4):486–524, Dezember 1988. Auch als Technischer Bericht 88-002, Politecnico di Milano, Italien, Dipartimento di Elettronica.

[GR83] Adele Goldberg und David Robson. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, Reading, Massachusetts, USA, 1983.

[Gri91] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance.* Dissertation, Computer Science Department, University of Washington, August 1991.

[Gro00] Michael Großmann. Der COAST-Schemaeditor: Eine Internet-basierte, plattformunabhängige graphische Benutzungsoberfläche für die Erstellung und Modifikation versionierter Schemata einer objektorientierten Datenbank. Diplomarbeit, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, Februar 2000.

[GT95] The GOODSTEP-Team. The GOODSTEP Technical Report Series. Technischer Bericht 1-28, Fachbereich Informatik, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, 1993-1995.

[GTC+90] Simon Gibbs, Dennis Tsichritzis, Eduardo Casais, Oscar Nierstrasz und Xavier Pintado. Class Management for Software Communities. *Communications of the ACM*, 33(9):90–103, September 1990.

[GZ94] Massimo Gentile und Roberto Zicari. Updating Views in Object Oriented Database Systems. In Shunsuke Uemura und Masatoshi Yoshikawa, Hrsg., *Proc. of the Int'l Symposium on Advanced Database Technologies and Their Integration*, Seiten 168–175, Nara, Japan, Oktober 1994.

[Haa00] Jan Haase. Objektpropagation in einem objektorientierten Datenbanksystem mit Schemaversionierung. Diplomarbeit, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, August 2000. In Vorbereitung.

[Har72] Frank Harary. *Graph Theory.* Addison Wesley, 1972.

[Her99] Detlef Herchen. Schnittstellen zur textuellen Beschreibung versionierter Schemata in objektorientierten Datenbankmanagementsystemen. Diplomarbeit, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, März 1999.

[Heu97] Andreas Heuer. *Objektorientierte Datenbanken: Konzepte, Modelle, Standards und Systeme.* Addison Wesley, 2. Auflage, 1997.

[HKV94] Christian Huemer, Gerti Kappel und Stefan Vieweg. Management of Data Model Evolution in Object-Oriented Database Systems. In Shunsuke Uemura und Masatoshi Yoshikawa, Hrsg., *Proc. of the Int'l Symposium on Advanced Database Technologies and Their Integration*, Seiten 39–46, Nara, Japan, Oktober 1994.

[HR83] Theo Härder und Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, Dezember 1983.

[HS95] Wolfgang Hiestand und Christian Schüller. TDF User's Manual. Technischer Bericht 25 in [GT95], Universität Frankfurt, April 1995. 16 Seiten.

[HS96] Walter L. Hürsch und Linda M. Seiter. Automating the Evolution of Object-Oriented Systems. In Kokichi Futatsugi und Satoshi Matsuoka, Hrsg., *Proc. of Object Technologies for Advanced Software*, Seiten 2–21, Kanazawa, Japan, März 1996. Springer-Verlag. Lecture Notes in Computer Science Nr. 1049.

[HS97] Wolfgang Hiestand und Christian Schüller. TDF – eine Methode zur statischen Typüberprüfung einer objektorientierten Datenbanksprache. Diplomarbeit, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, März 1997.

[Hür95] Walter L. Hürsch. *Maintaining Consistency and Behaviour of Object-Oriented Systems during Evolution.* Dissertation, College of Computer Science, Northeastern University, August 1995. 331 Seiten.

[HVZ90] Gilbert Harrus, Fernando Vélez und Roberto Zicari. Implementing Schema Updates in an Object-Oriented Database System: a Cost Analysis. Technischer Bericht, GIP Altaïr, 1990.

[HY95] Eunji Hong und Suk I. Yoo. A Schema Evolution Mechanism. In Norman Revell und A Min Tjoa, Hrsg., *Proc. of the 6th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Seiten 97–104, London, Großbritannien, September 1995. Workshop Proceedings.

[HZ90] Sandra Heiler und Stanley B. Zdonik. Object Views: Extending the Vision. In *Proc. of the 6th Int'l Conf. on Data Engineering (ICDE)*, Seiten 86–93, Los Angeles, Kalifornien, Februar 1990. IEEE, IEEE Computer Society Press.

[IBE95] IBEX Corporation SA, F-74160 Archamps, Frankreich. *ITASCA: Distributed Object Database Management System. Technische Zusammenfassung, Version 2.3.5*, 1995.

[JF88] Ralph E. Johnson und Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming (JOOP)*, 1(2):22–35, Juni 1988.

[JTTW89] John Joseph, Satish Thatte, Craig Thompson und David Wells. Report on the Object-Oriented Database Workshop. *SIGMOD Record*, 18(3):78–101, September 1989.

[Kat90] Randy H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, Dezember 1990.

[KBC+87] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza und Darrell Woelk. Composite Object Support in an Object-Oriented Database System. In Norman Meyrowitz, Hrsg., *Proc. of the 2nd Conf. on Object-Oriented Programming Systems,*

Languages, and Applications (OOPSLA), Seiten 118–125, Orlando, Florida, Oktober 1987. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 22, Nr. 12, Dezember 1987.

[KBC+89] Won Kim, Nathaniel Ballou, Hong-Tai Chou, Jorge F. Garza und Darrell Woelk. Features of the ORION Object-Oriented Database System. In *[KL89]*, Kapitel 11, Seiten 251–282. Addison Wesley, 1989.

[KBG+88] Won Kim, Nathaniel Ballou, Jorge F. Garza, Darrell Woelk und Jay Banerjee. Integrating an Object-Oriented Programming System with a Database System. In Norman Meyrowitz, Hrsg., *Proc. of the 3rd Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 142–152, San Diego, Kalifornien, September 1988. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 23, Nr. 11, November 1988.

[KBG89] Won Kim, Elisa Bertino und Jorge F. Garza. Composite Objects Revisited. In James Clifford, Bruce Lindsay und David Maier, Hrsg., *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seiten 337–347, Portland, Oregon, Juni 1989. ACM, ACM Press. SIGMOD Record, Band 18, Nr. 2, Juni 1989.

[KBGW91] Won Kim, Nathaniel Ballou, Jorge F. Garza und Darrell Woelk. A Distributed Object-Oriented Database System Supporting Shared and Private Databases. *ACM Transactions on Information Systems*, 9(1):31–51, Januar 1991.

[KC88] Won Kim und Hong-Tai Chou. Versions of Schema for Object-Oriented Databases. In François Bancilhon und David J. De Witt, Hrsg., *Proc. of the 14th Int'l Conf. on Very Large Databases (VLDB)*, Seiten 148–159, Los Angeles, USA, August 1988. Morgan Kaufmann Publishers.

[KCB86] Randy H. Katz, Ellis Chang und Rajiv Bhateja. Version Modeling Concepts for Computer-Aided Design Databases. In Carlo Zaniolo, Hrsg., *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seiten 379–386, Washington, DC, USA, Mai 1986. ACM, ACM Press. SIGMOD Record, Band 15, Nr. 2, Juni 1986.

[KCB87] Won Kim, Hong-Tai Chou und Jay Banerjee. Operations and Implementation of Complex Objects. In *Proc. of the 3rd Int'l Conf. on Data Engineering (ICDE)*, Seiten 626–633, Los Angeles, Kalifornien, Februar 1987. IEEE, IEEE Computer Society Press.

[Ken89] William Kent. Panel: An Overview of the Versioning Problem. In James Clifford, Bruce Lindsay und David Maier, Hrsg., *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seiten 5–7, Portland, Oregon, Juni 1989. ACM, ACM Press. SIGMOD Record, Band 18, Nr. 2, Juni 1989.

[KGBW90] Won Kim, Jorge F. Garza, Nathaniel Ballou und Darrell Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, März 1990.

[Kho93] Setrag Khoshafian. *Object-Oriented Databases.* Wiley Professional Computing. John Wiley & Sons, 1993.

[Kim91] Won Kim. *Introduction to Object-Oriented Databases.* Computer Systems. MIT Press, Cambridge, Massachusetts, 2. Auflage, 1991.

[KL89] Won Kim und Frederick H. Lochovsky, Hrsg. *Object-Oriented Concepts, Databases, and Applications.* ACM Frontier Series. Addison Wesley, 1989.

[KLMP84] Won Kim, Raymond Lorie, Dan McNabb und Wil Plouffe. A Transaction Mechanism for Engineering Design Databases. In *Proc. of the 10th Int'l Conf. on Very Large Databases (VLDB)*, Seiten 355–362, Singapur, August 1984. Morgan Kaufmann Publishers.

[KM94] Alfons Heinrich Kemper und Guido Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[KR95] Harumi A. Kuno und Elke A. Rundensteiner. Materialized Object-Oriented Views in MultiView. In Omran A. Bukhres, M. Tamer Özsu und Ming-Chien Shan, Hrsg., *Proc. of the Int'l Workshop on Research Issues in Data Engineering – Distributed Object Management (RIDE–DOM)*, Seiten 78–85, Taipeh, Taiwan, März 1995. IEEE Computer Society Press.

[KR96a] Harumi A. Kuno und Elke A. Rundensteiner. The MultiView OODB View System: Design and Implementation. *Theory And Practice of Object Systems (TAPOS)*, 2(3):202–225, Juli 1996.

[KR96b] Harumi A. Kuno und Elke A. Rundensteiner. Using Object-Oriented Principles to Optimize Update Propagation to Materialized Views. In Stanley Y. W. Su, Hrsg., *Proc. of the 12th Int'l Conf. on Data Engineering (ICDE)*, Seiten 310–317, New Orleans, Louisiana, Februar 1996. IEEE, IEEE Computer Society Press.

[KS92] Wolfgang Käfer und Harald Schöning. Mapping a Version Model to a Complex-Object Data Model. In Forouzan Golshani, Hrsg., *Proc. of the 8th Int'l Conf. on Data Engineering (ICDE)*, Seiten 348–357, Tempe, Arizona, Februar 1992. IEEE, IEEE Computer Society Press.

[KSW86] Peter Klahold, Gunter Schlageter und Wolfgang Wilkes. A General Model for Version Management in Databases. In *Proc. of the 12th Int'l Conf. on Very Large Databases (VLDB)*, Seiten 319–327, Kyoto, Japan, August 1986. Morgan Kaufmann Publishers.

[KWG+87] Won Kim, Darrell Woelk, Jorge F. Garza, Hong-Tai Chou, Jay Banerjee und Nathaniel Ballou. Enhancing the Object-Oriented Concepts for Database Support. In *Proc. of the 3rd Int'l Conf. on Data Engineering (ICDE)*, Seiten 291–292, Los Angeles, Kalifornien, Februar 1987. IEEE, IEEE Computer Society Press.

[LADH99] Sven-Eric Lautemann, Sabbas Apostolidis, Alexander Doll und Jan Haase. Dynamic Schema Changes with COAST. In Ana Moreira und Serge Demeyer, Hrsg., *Proc. of the 13th European Conf. on Object-Oriented Programming (ECOOP)*, Seiten 378–383, Lisbon, Portugal, Juni 1999. Springer-Verlag. Workshop Reader. Lecture Notes in Computer Science Nr. 1743. Poster Demonstration.

[Lan95] Stefan M. Lang. *Ein objektbasiertes Rollenmodell für situationsbedingtes Verhalten.* Dissertation, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation (IPD), Am Fasanengarten 5, D-76128 Karlsruhe, Deutschland, 1995. Fortschrittberichte VDI, Reihe 20: Rechnerunterstützte Verfahren, Nr. 177, VDI-Verlag, Düsseldorf.

[Lau91] Sven-Eric Lautemann. Versionierung in Smalltalk. Studienarbeit, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation (IPD). Am Fasanengarten 5, D-76128 Karlsruhe, Deutschland, April 1991.

[Lau93] Sven-Eric Lautemann. OPAQUE: Definition eines Objektmodells auf der Basis von Kontexten. Diplomarbeit, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation (IPD), Am Fasanengarten 5, D-76128 Karlsruhe, Deutschland, Januar 1993.

[Lau96a] Sven-Eric Lautemann. An Introduction to Schema Versioning in OODBMS. In Roland R. Wagner und C. Helmut Thoma, Hrsg., *Proc. of the 7th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Seiten 132–139, Zürich, Schweiz, September 1996. IEEE Computer Society Press. Workshop Proceedings.

[Lau96b] Sven-Eric Lautemann. Integration of Populated Schema Versions. In Jaroslav Pokorný, Hrsg., *Proc. of the 16th Annual Database Conference (DATASEM)*, Seiten 147–158, Brünn, Tschechische Republik, Oktober 1996. CS-Compex, Tiskárna Šmahel-Print.

[Lau97a] Sven-Eric Lautemann. A Propagation Mechanism for Populated Schema Versions. In Keith Jeffrey und Elke A. Rundensteiner, Hrsg., *Proc. of the 13th Int'l Conf. on Data Engineering (ICDE)*, Seiten 67–78, Birmingham, Großbritannien, April 1997. IEEE, IEEE Computer Society Press.

[Lau97b] Sven-Eric Lautemann. Schema Versions in Object-Oriented Database Systems. In Rodney Topor und Katsumi Tanaka, Hrsg., *Proc. of the 5th Int'l Conf. on Database Systems for Advanced Applications (DASFAA)*, Seiten 323–332, Melbourne, Australien, April 1997. World Scientific. In Advanced Database Research and Development Series, Band 6.

[Lau99a] Sven-Eric Lautemann. Change Management with Roles. In Arbee L. P. Chen und Frederick H. Lochovsky, Hrsg., *Proc. of the 6th Int'l Conf. on Database Systems for Advanced Applications (DASFAA)*, Seiten 291–300, Hsinchu, Taiwan, April 1999. IEEE.

[Lau99b] Sven-Eric Lautemann. Dynamische Schemaänderungen mit COAST. In *Kooperationspartner in Forschung und Innovation*, Seiten 17–18. Hessisches Ministerium für Wissenschaft und Kunst, März 1999. CeBIT Messebroschüre.

[Lau00] Sven-Eric Lautemann. The COAST Project Homepage. http://www.coast.uni-frankfurt.de/, 1995-2000.

[Ler94] Barbara Staudt Lerner. Type Evolution Support for Complex Type Changes. Technischer Bericht UM-CS-1994-071, Computer Science Department, University of Massachusetts, Amherst, Oktober 1994.

[Ler96] Barbara Staudt Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. Technischer Bericht UM-CS-1996-044, Computer Science Department, University of Massachusetts, Amherst, Juni 1996.

[Ler97] Barbara Staudt Lerner. TESS: Automated Support for the Evolution of Persistent Types. In *Proc. of the Int'l Conf. on Automated Software Engineering (ASE)*, Incline Village, Nevada, November 1997. IEEE.

[Ler00] Barbara Staudt Lerner. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems*, 25(1):83–127, März 2000.

[LEW97] Sven-Eric Lautemann, Patricia Eigner und Christian Wöhrle. The COAST Project: Design and Implementation. In François Bry, Raghu Ramakrishnan und

Kotagiri Ramamohanarao, Hrsg., *Proc. of the 5th Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Seiten 229–246, Montreux, Schweiz, Dezember 1997. Springer-Verlag. Lecture Notes in Computer Science Nr. 1341.

[LH89] Karl J. Lieberherr und Ian M. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Software*, 6(5):38–48, September 1989.

[LH90] Barbara Staudt Lerner und A. Nico Habermann. Beyond Schema Evolution to Database Reorganization. In Norman Meyrowitz, Hrsg., *Proc. of the 5th Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), gemeinsam veranstaltet mit 4th European Conf. on Object-Oriented Programming (ECOOP)*, Seiten 67–76, Ottawa, Kanada, Oktober 1990. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 25, Nr. 10, Oktober 1990.

[LHR88] Karl J. Lieberherr, Ian Holland und Arthur J. Riel. Object-Oriented Programming: An Objective Sense of Style. In Norman Meyrowitz, Hrsg., *Proc. of the 3rd Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 323–334, San Diego, Kalifornien, September 1988. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 23, Nr. 11, November 1988.

[LL95] Stefan M. Lang und Peter C. Lockemann. *Datenbankeinsatz.* Springer-Verlag, 1995.

[LL98] Stefan M. Lang und Peter C. Lockemann. Beviorally adaptive Objects. *Theory And Practice of Object Systems (TAPOS)*, 4(3):169–182, Juli 1998.

[LLOW91] Charles W. Lamb, Gordon Landis, Jack A. Orenstein und Daniel L. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, Oktober 1991.

[LMB92] John R. Levine, Tony Mason und Doug Brown. *Lex & Yacc.* O'Reilly, Sebastopol, Kalifornien, 2. Auflage, 1992.

[LRV88] Christophe Lécluse, Philippe Richard und Fernando Vélez. O_2, an Object-Oriented Data Model. In Stanley B. Zdonik und David Maier, Hrsg., *Readings in Object-Oriented Database Systems*, Seiten 227–236. Morgan Kaufmann Publishers, 1988. Auch in SIGMOD Record, Band 17, Nr. 3, Seiten 424-433, Juni 1988.

[LRV90] Christophe Lécluse, Philippe Richard und Fernando Vélez. O_2, an Object-Oriented Data Model. In François Bancilhon und Peter Buneman, Hrsg., *Advances in Database Programming Languages*, Kapitel 15, Seiten 257–276. ACM Press, New York, 1990.

[LS93] Gamal Labib und Dave Saunders. A Class Versioning Approach for OODBS. Technischer Bericht QMW-DCS-1993-651, Queen Mary and Westfield College, Department of Computer Science, Oktober 1993.

[LS94] Gamal Labib und Dave Saunders. A Class Versioning Approach for OODBS. In David S. Bowers, Hrsg., *Proc. of the 12th British National Conf. on Databases (BNCOD)*, Seiten 9–12, Guildford, Großbritannien, Juli 1994. Poster Summaries.

[LV96] Georg Lausen und Gottfried Vossen. *Objekt-orientierte Datenbanken: Modelle und Sprachen.* R. Oldenbourg, 1996.

[LZHL97] Ling Liu, Roberto Zicari, Walter L. Hürsch und Karl J. Lieberherr. The Role of Polymorphic Reuse Mechanisms in Schema Evolution in an Object-Oriented Database. *IEEE Transactions on Knowledge and Data Engineering*, 9(1):50–67, Januar 1997.

[MBJ96] Claudia Bauzer Medeiros, Marie-Jo Bellosta und Geneviève Jomier. Managing Multiple Representations of Georeferenced Elements. In Roland R. Wagner und C. Helmut Thoma, Hrsg., *Proc. of the 7th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Seiten 364–370, Zürich, Schweiz, September 1996. IEEE Computer Society Press. Workshop Proceedings.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, 1992.

[MJ94] Claudia Bauzer Medeiros und Geneviève Jomier. Using Versions in GIS. In Dimitris Karagiannis, Hrsg., *Proc. of the 5th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Seiten 465–474. Springer-Verlag, September 1994. Lecture Notes in Computer Science Nr. 856.

[MM91] Jean-Claude Mamou und Claudia Bauzer Medeiros. Interactive Manipulation of Object-oriented Views. In *Proc. of the 7th Int'l Conf. on Data Engineering (ICDE)*, Seiten 60–69, Kobe, Japan, April 1991. IEEE, IEEE Computer Society Press.

[MMW94] Alberto O. Mendelzon, Tova Milo und Emmanuel Waller. Object-Migration. In *Proc. of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Seiten 232–242, Minneapolis, Mai 1994. ACM, ACM Press.

[MNK92] Magdi M. Morsi, Shamkant B. Navathe und Hyoung-Joo Kim. An Extensible Object-Oriented Database Testbed. In Forouzan Golshani, Hrsg., *Proc. of the 8th Int'l Conf. on Data Engineering (ICDE)*, Seiten 150–157, Tempe, Arizona, Februar 1992. IEEE, IEEE Computer Society Press.

[MNS94] Magdi M. A. Morsi, Shamkant B. Navathe und John Shilling. On Behavioural Schema Evolution in Object-Oriented Databases. In Matthias Jarke, Janis Bubenko und Keith Jeffery, Hrsg., *Proc. of the 4th Int'l Conf. on Extending Database Technology (EDBT)*, Seiten 173–186, Cambridge, Großbritannien, März 1994. Springer-Verlag. Lecture Notes in Computer Science Nr. 779.

[Mon93] Simon R. Monk. *A Model for Schema Evolution in Object-Oriented Database Systems*. Dissertation, Computing Department, Lancaster University, Februar 1993. 182 Seiten.

[Mot87] Amihai Motro. Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering*, 13(7):785–798, Juli 1987.

[MP96] Renate Motschnig-Pitrik. Requirements and Comparison of View Mechanisms for Object-Oriented Databases. *Informations Systems*, 21(3):229–252, 1996.

[MS92] Simon R. Monk und Ian Sommerville. A Model for Versioning of Classes in Object-Oriented Databases. In P. M. D. Gray und R. J. Lucas, Hrsg., *Proc. of the 10th British National Conf. on Databases (BNCOD)*, Seiten 42–58, Aberdeen, Schottland, Juli 1992. Springer-Verlag. Lecture Notes in Computer Science Nr. 618.

[MS93] Simon R. Monk und Ian Sommerville. Schema Evolution in OODBs Using Class Versioning. *SIGMOD Record*, 22(3):16–22, September 1993.

[MSOP86] David Maier, Jacob Stein, Allen Otis und Alan Purdy. Development of and Object-oriented DBMS. In Norman Meyrowitz, Hrsg., *Proc. of the 1st Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 472–482, Portland, Oregon, September 1986. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 21, Nr. 11, November 1986.

[MZ92] Guido Moerkotte und Andreas Zachmann. Multiple Substitutability Without Affecting the Taxonomy. In Alain Pirotte, Claude Delobel und Georg Gottlob, Hrsg., *Proc. of the 3rd Int'l Conf. on Extending Database Technology (EDBT)*, Seiten 120–135, Wien, Österreich, März 1992. Springer-Verlag. Lecture Notes in Computer Science Nr. 580.

[MZ93] Guido Moerkotte und Andreas Zachmann. Towards More Flexible Schema Management in Object Bases. In *Proc. of the 9th Int'l Conf. on Data Engineering (ICDE)*, Seiten 174–181, Wien, Österreich, April 1993. IEEE, IEEE Computer Society Press.

[Nol76] Hartmut Noltemeier. *Graphentheorie.* Walter De Gruyter, 1976.

[NR89] Gia Toan Nguyen und Dominique Rieu. Schema evolution in object-oriented database system. *Data & Knowledge Engineering*, 4(1):43–67, 1989.

[O2 96a] O2 Technology, Versailles Cedex, Frankreich. *O_2C Reference Manual, Version 4.6*, April 1996.

[O2 96b] O2 Technology, Versailles Cedex, Frankreich. *System Administration Guide, Version 4.6*, April 1996.

[O2 96c] O2 Technology, Versailles Cedex, Frankreich. *Version Management Reference Manual, Version 4.6*, April 1996.

[Obj96] ObjectDesign, Inc., Burlington, MA. *ObjectStore Documentation, Version 4.0.2*, Mai 1996.

[Odb92] Erik Odberg. A Framework for Managing Schema Versioning in Object-Oriented Databases. In *Proc. of the 3rd Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Valencia, Spanien, September 1992.

[Odb94a] Erik Odberg. A Global Perspective of Schema Modification Management for Object-Oriented Databases. In Malcolm P. Atkinson, David Maier und Véronique Benzaken, Hrsg., *Proc. of the 6th Int'l Workshop on Persistent Object Systems*, Seiten 479–502, Tarascon, Frankreich, September 1994. Springer-Verlag.

[Odb94b] Erik Odberg. Category Classes: Flexible Classification and Evolution in Object-Oriented Databases. In Gerhard Wijers, Sjaak Brinkkemper und Tony Wassermann, Hrsg., *Proc. of the 6th Conf. on Advanced Information Systems Engineering (CAiSE)*, Seiten 406–420, Utrecht, Niederlande, Juni 1994. Springer-Verlag. Lecture Notes in Computer Science Nr. 811.

[Odb95] Erik Odberg. *MultiPerspectives: Object Evolution and Schema Modification Management for Object-Oriented Databases.* Dissertation, Department of Computer Systems and Telematics, Norwegian Institute of Technology, Februar 1995. 408 Seiten.

[OHK87] Patrick D. O'Brien, Daniel C. Halbert und Michael F. Kilian. The Trellis Programming Environment. In Norman Meyrowitz, Hrsg., *Proc. of the 2nd Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 91–102, Orlando, Florida, Oktober 1987. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 22, Nr. 12, Dezember 1987.

[Ont91] Ontologic Inc. *ONTOS Developer's Guide, Version 2.0*, Februar 1991.

[ÖPB+93] M. Tamer Özsu, Randal Peters, Irani Bomann, Anna Lipka, Adriana Munoz und Duane Szafron. TIGUKAT Object Management System: Initial Design and Current Directions. In *Proc. of the Centre for Advanced Studies Conf. (CASCON)*, Seiten 595–611, Oktober 1993.

[Opd92] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks.* Dissertation, Computer Science Department, University of Illinois, Mai 1992.

[Osb89] Sylvia L. Osborn. The Role of Polymorphism in Schema Evolution in an Object-Oriented Database. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):310–317, September 1989.

[Per89] Barbara Pernici. Objects with Roles. In Dennis Tsichritzis, Hrsg., *Object-Oriented Development*, Seiten 75–100. Université de Genève, Schweiz, Juli 1989.

[PÖ95] Randal J. Peters und M. Tamer Özsu. Axiomatization of Dynamic Schema Evolution in Objectbases. In Arbee L. P. Chen Philip S. Yu, Hrsg., *Proc. of the 11th Int'l Conf. on Data Engineering (ICDE)*, Taipeh, Taiwan, März 1995. IEEE, IEEE Computer Society Press.

[Poe95] Poet Software Corporation, San Mateo, Kalifornien. *Poet Reference Guide, Version 3.0*, Januar 1995.

[Pri98] Manfred Prien. Entwurf und Implementierung eines Schema-Managers für ein objektorientiertes Datenbanksystem mit Schemaversionierung. Diplomarbeit, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, April 1998.

[PS87] D. Jason Penney und Jacob Stein. Class Modification in the GemStone Object-Oriented DBMS. In Norman Meyrowitz, Hrsg., *Proc. of the 2nd Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 111–117, Orlando, Florida, Oktober 1987. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 22, Nr. 12, Dezember 1987.

[RC96] Jarogniew Rykowski und Wojciech Cellary. Using Multiversion Object-Oriented Databases in CAD/CIM Systems. In Roland R. Wagner und C. Helmut Thoma, Hrsg., *Proc. of the 7th Int'l Conf. on Database and Expert Systems Applications (DEXA)*, Seiten 1–10, Zürich, Schweiz, September 1996. Springer-Verlag. Lecture Notes in Computer Science Nr. 1134.

[RCR93] John F. Roddick, Noel G. Craske und Thomas J. Richards. A Taxonomy for Schema Versioning Based on the Relational and Entity Relationship Models. In Ramez Elmasri, Vram Kouramajian und Bernhard Thalheim, Hrsg., *Proc. of the 12th Int'l Conf. on the Entity-Relationship Approach (ER)*, Seiten 137–148, Arlington, Texas, USA, Dezember 1993. Springer-Verlag. Lecture Notes in Computer Science Nr. 823.

[RKR+96] Elke A. Rundensteiner, Harumi A. Kuno, Young-Gook Ra, Viviane Crestana-Taube, Matthew C. Jones und P. J. Marron. The MultiView Project: Object-Oriented View Technology and Applications. In H. V. Jagadish und Inderpal Singh Mumick, Hrsg., *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seite 555, Montreal, Quebec, Kanada, Juni 1996. ACM, ACM Press. SIGMOD Record, Band 25, Nr. 2, Juni 1996.

[Rod92a] John F. Roddick. Schema Evolution in Database Systems – An Annotated Bibliography. *SIGMOD Record*, 21(4):35–40, Dezember 1992.

[Rod92b] John F. Roddick. SQL/SE - A Query Language Extension for Databases Supporting Schema Evolution. *SIGMOD Record*, 21(3):10–16, September 1992.

[RR95] Young-Gook Ra und Elke A. Rundensteiner. A Transparent Object-Oriented Schema Change Approach Using View Evolution. In Philip S. Yu und Arbee L. P. Chen, Hrsg., *Proc. of the 11th Int'l Conf. on Data Engineering (ICDE)*, Seiten 165–172, Taipeh, Taiwan, März 1995. IEEE, IEEE Computer Society Press.

[RR96] R. Ramakrishnan und D. Janaki Ram. Modeling Design Versions. In T. M. Vijayaraman, Alejandro Buchmann, C. Mohan und Nandlal L. Sarda, Hrsg., *Proc. of the 22nd Int'l Conf. on Very Large Databases (VLDB)*, Seiten 556–566, Bombay, Indien, September 1996. Morgan Kaufmann Publishers.

[RS91] Joel Richardson und Peter Schwarz. Aspects: Extending Objects to Support Multiple, Independent Roles. In James Clifford und Roger King, Hrsg., *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seiten 298–307, Denver, Colorado, USA, Mai 1991. ACM, ACM Press. SIGMOD Record, Band 20, Nr. 2, Juni 1991.

[RS99] Awais Rashid und Peter Sawyer. Evaluation for Evolution: How Well Commercial Systems Do. In Ana Moreira und Serge Demeyer, Hrsg., *Proc. of the 13th European Conf. on Object-Oriented Programming (ECOOP)*, Seiten 210–211, Lissabon, Portugal, Juni 1999. Springer-Verlag. Workshop Reader. Lecture Notes in Computer Science Nr. 1743. 1st ECOOP Workshop on Object-Oriented Databases.

[Run92] Elke A. Rundensteiner. Multiview: a Methodology for Supporting Multiple Views in Object-Oriented Databases. In Li-Yan Yuan, Hrsg., *Proc. of the 18th Int'l Conf. on Very Large Databases (VLDB)*, Seiten 187–198, Vancouver, Kanada, August 1992.

[Sak88a] Markku Sakkinen. Comments on “the Law of Demeter” and C++. *SIGPLAN Notices*, 23(12):38–44, Dezember 1988.

[Sak88b] Markku Sakkinen. On the darker side of C++. In Stein Gjessing und Kristen Nygaard, Hrsg., *Proc. of the 2nd European Conf. on Object-Oriented Programming (ECOOP)*, Seiten 162–176, Oslo, Norwegen, August 1988. Springer-Verlag. Lecture Notes in Computer Science Nr. 322.

[Sak92] Markku Sakkinen. A critique of the inheritance principles of C++. *Computing Systems*, 5(1):69–110, 1992. Corrigendum: Computing Systems 5(3): 361 (1992).

[SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Killian und Carrie Wilpolt. An Introduction to Trellis/Owl. In Norman Meyrowitz, Hrsg., *Proc. of the 1st Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 9–16, Portland, Oregon, September 1986. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 21, Nr. 11, November 1986.

[Sch93] Bernhard Schiefer. *Eine Umgebung zur Unterstützung von Schemaänderungen und Sichten in objektorientierten Datenbanksystemen.* Dissertation, Universität Karlsruhe, Karlsruhe, Deutschland, Dezember 1993.

[Sci91a] Edward Sciore. Multidimensional Versioning for Object-Oriented Databases. In Claude Delobel, Michael Kifer und Yoshifumi Masunaga, Hrsg., *Proc. of the 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Seiten 167–188, München, Deutschland, Dezember 1991. Springer-Verlag. Lecture Notes in Computer Science Nr. 566.

[Sci91b] Edward Sciore. Using Annotations to Support Multiple Kinds of Versioning in an Object-Oriented Database System. *ACM Transactions on Database Systems*, 16(3):417–438, September 1991.

[Sci94] Edward Sciore. Versioning and Configuration Management in an Object-Oriented Data Model. *VLDB Journal*, 3(1):77–106, 1994.

[SGD93] Stefan Scherrer, Andreas Geppert und Klaus R. Dittrich. Schema Evolution in NO^2. Technischer Bericht ifi-93.12, Department of Computer Science, University of Zurich, April 1993.

[Sjø93a] Dag I. K. Sjøberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–44, Januar 1993.

[Sjø93b] Dag I. K. Sjøberg. *Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems.* Dissertation, University of Glasgow, Juli 1993.

[SK91] Michael Stonebraker und Greg Kemnitz. The Postgres Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–92, Oktober 1991.

[SLR+92] Marc H. Scholl, Christian Laasch, C. Rich, Hans-Jörg Schek und Markus Tresch. The COCOON object model. Technischer Bericht 193, Department of Computer Science, ETH Zürich, Dezember 1992. Auch als Technischer Bericht 93-02, Universität Ulm, Deutschland, Department of Computer Science, Februar 1993.

[SLT91] Marc H. Scholl, Christian Laasch und Markus Tresch. Updateable Views in Object-Oriented Databases. In Claude Delobel, Michael Kifer und Yoshifumi Masunaga, Hrsg., *Proc. of the 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Seiten 189–207, München, Deutschland, Dezember 1991. Springer-Verlag. Lecture Notes in Computer Science Nr. 566.

[SRL+90] Michael Stonebraker, Lawrence A. Rowe, Bruce Lindsay, James Gray, Michael Carey, Michael Brodie, Philip Bernstein und David Beech. Third-Generation Database System Manifesto. *SIGMOD Record*, 19(3):31–44, September 1990.

[SS89] John J. Shilling und Peter F. Sweeney. Three Steps to Views: Extending the Object-Oriented Paradigm. In Norman Meyrowitz, Hrsg., *Proc. of the 4th Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 353–361, New Orleans, Louisiana, Oktober 1989. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 24, Nr. 10, Oktober 1989.

[SS94a] Sabine Sachweh und Wilhelm Schäfer. Version Management for tightly integrated Software Engineering Environments. Technischer Bericht 22 in [GT95], Universität Dortmund, Juli 1994. 14 Seiten.

[SS94b] Peter Schwarz und Kurt Shoens. Managing Change in the Rufus System. In *Proc. of the 10th Int'l Conf. on Data Engineering (ICDE)*, Seiten 170–179, Houston, Texas, Februar 1994. IEEE, IEEE Computer Society Press.

[ST94] Marc H. Scholl und Markus Tresch. Evolution towards, in, and beyond Object Databases. In Kai von Luck und Heinz Marburger, Hrsg., *Proc. of Management and Processing of Complex Data Structures, Third Workshop on Information Systems and Artificial Intelligence*, Seiten 64–82, Hamburg, Deutschland, Februar 1994. Springer-Verlag. Lecture Notes in Computer Science Nr. 777.

[Sto88] Michael Stonebraker. *Readings in Database Systems.* Morgan Kaufmann Publishers, San Mateo, Kalifornien, 1988. 2. Auflage, 1993.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 3. Auflage, 1997.

[SZ86] Andrea H. Skarra und Stanley B. Zdonik. The Management of Changing Types in an Object-Oriented Database. In Norman Meyrowitz, Hrsg., *Proc. of the 1st Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 483–495, Portland, Oregon, September 1986. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 21, Nr. 11, November 1986.

[SZ87] Andrea H. Skarra und Stanley B. Zdonik. Type Evolution in an Object-Oriented Database. In Bruce Shriver und Peter Wegner, Hrsg., *Research Directions in Object-Oriented Programming*, Seiten 393–415. MIT Press, 1987.

[TG92] Vassilis J. Tsotras und B. Gopinath. Optimal Versioning of Objects. In Forouzan Golshani, Hrsg., *Proc. of the 8th Int'l Conf. on Data Engineering (ICDE)*, Seiten 358–365, Tempe, Arizona, Februar 1992. IEEE, IEEE Computer Society Press.

[TK89] Lichao Tan und Takuya Katayama. Meta Operations for Type Management in Object-Oriented Databases - A Lazy Mechanism for Schema Evolution. In Won Kim, Jean-Marie Nicolas und Shojiro Nishio, Hrsg., *Proc. of the 1st Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Seiten 241–258, Kyoto, Japan, 1989.

[TOC93] G. Talens, Chabane Oussalah und Marie Francoise Colinas. Versions of Simple and Composite Objects. In Rakesh Agrawal, Seán Baker und David Bell, Hrsg., *Proc. of the 19th Int'l Conf. on Very Large Databases (VLDB)*, Seiten 62–72, Dublin, Irland, August 1993. Morgan Kaufmann Publishers.

[Tre95] Markus Tresch. *Evolution in Objekt-Datenbanken: Anpassung und Integration bestehender Informationssysteme*, Jgg. 10 von *Teubner-Texte zur Informatik.* Teubner, 1995.

[TS92] Markus Tresch und Marc H. Scholl. Meta Object Management and its Application to Database Evolution. In Günther Pernul und A Min Tjoa, Hrsg., *Proc. of the 11th Int'l Conf. on the Entity-Relationship Approach (ER)*, Seiten 299–321, Karlsruhe, Deutschland, Oktober 1992. Springer-Verlag. Lecture Notes in Computer Science Nr. 645.

[TS93a] Chris Thieme und Arno Siebes. Schema Integration in Object-Oriented Databases. In Colette Rolland, François Bodart und Corine Cauvet, Hrsg., *Proc. of the 5th Conf. on Advanced Information Systems Engineering (CAiSE)*, Seiten 54–70, Paris, Frankreich, Juni 1993. Springer-Verlag. Lecture Notes in Computer Science Nr. 685.

[TS93b] Chris Thieme und Arno Siebes. Schema refinement and schema integration in object-oriented databases. Technischer Bericht CS-R9354, CWI Amsterdam, 1993.

[TS93c] Markus Tresch und Marc H. Scholl. Schema Transformation without Database Reorganization. *SIGMOD Record*, 22(1):21–27, März 1993.

[TT93] Karl Teille und Diedrich Thomas. Evaluationsbericht zu GemStone – Version 3.0.1. In *Datenbank Rundbrief*, Seiten 77–80. Gesellschaft für Informatik, Mai 1993.

[TYF86] Toby Teorey, Dongqing Yang und James P. Fry. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationshop Model. *ACM Computing Surveys*, 18(2):197–222, Juni 1986.

[TYH+91] Kazuyuki Tsuda, Kensaku Yamamoto, Masahito Hirakawa, Minoru Tanaka und Tadao Ichikawa. MORE: An Object-Oriented Data Model with a Facility for Changing Object Structures. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):444–460, Dezember 1991.

[VDDW+90] Fernando Vélez, Vineeta Darnis, David J. De Witt, Philippe Futtersack, Gilbert Harrus, David Maier und Michel Raoux. Implementing the O_2 Object Manager: Some Lessons. In Alan Dearle, Gail M. Shaw und Stanley B. Zdonik, Hrsg., *Implementing Persistent Object Bases*, Seiten 131–138, Massachusetts, USA, September 1990. Morgan Kaufmann Publishers.

[Ver98] Versant Corporation, 4500 Bohannon Drive Menlo Park, Kalifornien 94025. *Versant User Manual, Version 5*, 1998.

[Vos91] Gottfried Vossen. Bibliography on Object-Oriented Database Management. *SIGMOD Record*, 20(1):24–46, März 1991.

[Vos94] Gottfried Vossen. Formalization of OODB Models. In Franz Baader, Martin Buchheit, Manfred A. Jeusfeld und Werner Nutt, Hrsg., *Proc. of the 1st Workshop on Reasoning about Structured Objects: Knowledge Representation Meets Databases (KRDB)*, Saarbrücken, Deutschland, September 1994.

[Wal91] Emmanuel Waller. Schema Updates and Consistency. In Claude Delobel, Michael Kifer und Yoshifumi Masunaga, Hrsg., *Proc. of the 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Seiten 167–188, München, Deutschland, Dezember 1991. Springer-Verlag. Lecture Notes in Computer Science Nr. 566.

[WC95] Jennifer Widom und Stefano Ceri, Hrsg. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Data Management Series. Morgan Kaufmann Publishers, 1995.

[WCL97] Raymond K. Wong, H. Lewis Chau und Frederick H. Lochovsky. A Data Model and Semantics of Objects with Dynamic Roles. In Keith Jeffrey und Elke A. Rundensteiner, Hrsg., *Proc. of the 13th Int'l Conf. on Data Engineering (ICDE)*, Seiten 402–411, Birmingham, Großbritannien, April 1997. IEEE, IEEE Computer Society Press.

[WdJ91] Roelf Johannes Wieringa und Wiebren de Jonge. The Identification of Objects and Roles. Technischer Bericht IR-267, Faculty of Mathematics and Computer Science, University of Vrije, Amsterdam, Niederlande, Dezember 1991.

[Wed94] Hartmut Wedekind. Are the Terms "Version" and "Variant" Orthogonal to One Another? - A Critical Assessment of the STEP Standardization. *SIGMOD Record*, 23(4):3–7, Dezember 1994.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In Norman Meyrowitz, Hrsg., *Proc. of the 2nd Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seiten 168–182, Orlando, Florida, Oktober 1987. ACM, ACM Press. Sonderausgabe von SIGPLAN Notices, Band 22, Nr. 12, Dezember 1987.

[Weg90] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):7–87, August 1990. Erweiterte Fassung des OOPSLA '89 Keynote Talk.

[Wie90] Roelf Johannes Wieringa. *Algebraic Foundations for Dynamic Conceptual Models*. Dissertation, Faculty of Mathematics and Computer Science, University of Vrije, Amsterdam, Niederlande, 1990.

[WK87] Darrell Woelk und Won Kim. Multimedia Information Management in an Object-Oriented Database System. In Peter M. Stocker und William Kent, Hrsg., *Proc. of the 13th Int'l Conf. on Very Large Databases (VLDB)*, Seiten 319–329, Brighton, England, September 1987. Morgan Kaufmann Publishers.

[WKL86] Darrell Woelk, Won Kim und Willis Luther. An Object-Oriented Approach to Multimedia Databases. In Carlo Zaniolo, Hrsg., *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, Seiten 311–325, Washington, DC, USA, Mai 1986. ACM, ACM Press. SIGMOD Record, Band 15, Nr. 2, Juni 1986.

[Wöh96] Christian Wöhrle. Schemaversionierung in objektorientierten Datenbanksystemen. Diplomarbeit, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, April 1996.

[Wöl98] Kay Wölfle. Verwaltung und Propagation persistenter Objekte in einem Schemaversionierung unterstützenden objektorientierten Datenbanksystem. Diplomarbeit, Universität Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Deutschland, Mai 1998.

[ZCF+97] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian und Roberto Zicari. *Advanced Database Systems*. Data Management Systems. Morgan Kaufmann Publishers, 1997.

[Zdo86] Stanley B. Zdonik. Maintaining Consistency in a Database with Changing Types. In *Object-Oriented Programming Workshop*, Seiten 120–127. ACM, ACM Press, Oktober 1986. SIGPLAN Notices, Band 21, Nr. 10.

[Zdo90] Stanley B. Zdonik. Object-Oriented Type Evolution. In François Bancilhon und Peter Buneman, Hrsg., *Advances in Database Programming Languages*, Kapitel 16, Seiten 277–288. Addison Wesley, New York, 1990.

[Zic89a] Roberto Zicari. A Framework for O_2 Schema Updates. Technischer Bericht 89-036, Politecnico di Milano, Mailand, Italien, Mai 1989.

[Zic89b] Roberto Zicari. Schema Updates in the O_2 Object Oriented Database System. Technischer Bericht 89-057, Politecnico di Milano, Mailand, Italien, Oktober 1989.

[Zic91a] Roberto Zicari. A Framework for Schema Updates In An Object-Oriented Database System. In *Proc. of the 7th Int'l Conf. on Data Engineering (ICDE)*, Seiten 2–13, Kobe, Japan, April 1991. IEEE, IEEE Computer Society Press.

[Zic91b] Roberto Zicari. The ESSE Project: An Overview. Technischer Bericht 91-077, Politecnico di Milano, Mailand, Italien, Dezember 1991.

[Zic92] Roberto Zicari. A Framework for Schema Updates in an Object-Oriented Database System. In *[BDK92]*, Kapitel 7, Seiten 146–182. Morgan Kaufmann Publishers, 1992.

[Zic97] Roberto Zicari. Schema and Database Evolution in Object Database Systems. In *[ZCF+97]*, Kapitel 6, Seiten 411–495. Morgan Kaufmann Publishers, 1997.

[ZM91] Stanley B. Zdonik und Gail Mitchell. An Object-Oriented Approach to Database Modelling and Querying. *Data Engineering Bulletin*, 14(2):53–57, 1991.

[ZW88] Stanley B. Zdonik und Peter Wegner. Language and Methodology for Object-Oriented Database Environments. In Malcolm P. Atkinson, Peter Buneman und R. Morrison, Hrsg., *Data Types and Persistence*, Kapitel 11, Seiten 155–171. Springer-Verlag, 1988.

Index

A

Ableitung von Schemaversionen 151
  (SÄP) 158
Ableitungsgraph . . . . *siehe* Schemaableitungsgraph
Ableitungszeit (Definition) 141
Ad-hoc Anfragemöglichkeit 14
Änderung . . . . *siehe* Veränderung
Anlegen . . . . *siehe* Erzeugung
Applikationsanbindung 14
  an Schemaversionen 153
Attribut
  erzeugen (SÄP) 170
  löschen (SÄP) 170
  überschreiben (SÄP) 170
  Überschreibung löschen (SÄP) 170
  umbenennen (SÄP) 170
  umtypisieren (SÄP) 170
Auftauen von Schemaversionen (SÄP) . . 160
Ausfallsicherheit 14
Auswählen von Schemaversionen für Applikationen 155

B

Baum
  gerichtet (Definition) 20

C

Cellary, Wojciech 85
Chou, Hong-Tai 103
Clamen, Steward 93
CLOSQL 90
COAST
  Objektmodell 117, 138
  ODL . . . . *siehe* ODL
  Realisierung 233
COCOON 80

D

Datenabstraktion 10
Datenbank
  (Definition) 138
  referenzielle Integrität (Definition) . 138
Datenbankobjekt . . . . *siehe* Objekt
Datenbankschema . . . . *siehe* Schema
Defaultkonvertierungsfunktion 186, 240
Definitionen (Übersicht) 287
Demeter 69
Designtransaktionen 14
DOOR 21

E

Eindeutigkeit
  Klassenversionen (Invariante) 149
  Schemaversionen (Invariante) 149
Einfrieren von Schemaversionen (SÄP) . 160
ENCORE 86
Entfernung von Oberklassen (SÄP) 167
Erweiterbarkeit 13
Erzeugende ODL 175
Erzeugerschemaversion
  einer Klasse (Definition) 147
  eines Objektes (Definition) 183
Erzeugung
  von Attributen (SÄP) 170
  von Klassen 165
    (SÄP) 165
  von Methoden (SÄP) 171
  von Schemata (SÄP) 157
  von Schemaversionen (SÄP) 158
Erzeugungsflag . . . . . *siehe* Propagationsflags
Extension . . . . *siehe* Klassenextension
Extrakonvertierungsfunktion . . . . *siehe* Konvertierungsfunktion

F

Ferrandina, Fabrizio 67

G

GemStone 66
GOODSTEP 21

Graph
gerichtet
azyklisch (Definition) 19
(Definition) 18
verwurzelt (Definition) 20
Teilg.
(Definition) 19
partiell (Definition) 19
vollständig (Definition) 19
zusammenhängend (Definition) 19

H

Habermann, Nico 71
Halbordnung . . . . *siehe* Ordnungsrelation
Herkunft eines Klassenmerkmals
(Definition) 127
Eindeutigkeit (Invariante) 128
Hierarchie . . . . *siehe* Ordnungsrelation
Hintergrundspeicher-Management 13
Hinzufügung von Oberklassen (SÄP) . . . 167
Hülle
reflexiv (Definition) 19
reflexiv und transitiv
(Definition) 20
(Lemma) 20
transitiv (Definition) 19

I

Identifikator (Definition) 117
Integration
von Klassen 160, 176
(SÄP) 161
von Schemata 111, 176
Invarianten 128, 149, 289
Eindeutigkeit 128, 149
(Invariante) 128
Methoden überladen (Invariante) . . . 134
Methoden überschreiben (Invariante) 136
Minimalität (Invariante) 129, 150
(Übersicht) 289
Vererbung 129
(Invariante) 129
Verhalten 133
Vollständigkeit 128
(Invariante) 128, 149
ITASCA 103

J

Jomier, Geneviève 85

K

Kim, Won 103
Klasse 8
erzeugen 165
(SÄP) 165
integrieren 176
(SÄP) 161
löschen 165
(SÄP) 165
umbenennen 165
(SÄP) 165
unversioniert (Definition) 123
verändern 165
(SÄP) 165
versioniert
(Definition) 146
Tiefe (Definition) 219
Klassenableitungsbaum
(Definition) 147
Zusammenhang (Lemma) 149
Klassenextension 8, 246
Klassenintegration . . . . *siehe* Integration
Klassenkonvertierungsbaum (Definition) 205
Klassenkonvertierungsgraph (Definition) 222
Klassenvererbung . . . . *siehe* Vererbung
Klassenversion
(Definition) 146
Eindeutigkeit (Invariante) 149
Konsistenz
von Datenbanken 45, 251
von Schemata 45
von unversionierten Schemata 127
von versionierten Schemata 149
Konvertierung
Wohldefiniertheit (Lemma) 225
Konvertierungsfunktion
Defaultk. 186, 240
Extrak.
definieren (Regel) 223
(Definition) 220
Propagationsflags (Regel) 224
Tiefe (Definition) 220
Vorwärts- und Rückwärtsk.
definieren (Regel) 188
(Definition) 184
Propagationsflags (Regel) 191
Konvertierungspfad
Bestimmung 206, 223
(Definition) 205
Eindeutigkeit 209, 225
Löcher 214
regulär (Definition) 209
Tiefe (Definition) 223

L

Löcher in Konvertierungspfaden........214
Labib, Gamal........................95
Lieberherr, Karl........................69
Löschung
  von Attributen (SÄP).............170
  von Klassen......................165
    (SÄP)..........................165
  von Methoden (SÄP).............171
  von Schemata (SÄP)...............157
  von Schemaversionen.............227
    (SÄP)..........................159
Löschungsflag...... *siehe* Propagationsflags
Lösungsmodell..........................46

M

Mehrfachvererbung........ *siehe* Vererbung
Merkmale eines Objektes / einer Klasse .. 8
Methode
  (Definition)........................126
  erzeugen (SÄP)....................171
  löschen (SÄP)....................171
  spätes Binden......................12
  überladen..........................12
    (Invariante).....................134
  überschreiben......................12
  umbenennen (SÄP)................171
  verändern (SÄP).................171
Methodensignatur.......... *siehe* Signatur
Minimalität
  Klassenvererbungsgraph (Invariante) 129
  Schemaableitungsgraph (Invariante) 150
Modifikationsflag... *siehe* Propagationsflags
Monk, Simon..........................90
MultiView............................80

N

Nachfolger
  (Definition)........................19
  einer Knotenmenge (Definition).....20
Nachfolgerklassenversion (Definition)...148
Nachfolgerschemaversion (Definition)...144
Name
  (Definition)........................117
  Eindeutigkeit (Invariante)..........128
Nebenläufigkeit........................11

O

O_2..................................67
Oberklasse
  (Definition)........................125
  entfernen (SÄP)..................167
  hinzufügen (SÄP).................167
Obertyp (Definition)..................122
Objekt.................................7
  (Definition)........................120
  versioniert (Definition).............183
Objektmanager..................235, 244
Objektversion
  ableitbar
    (Definition).....................252
    (Lemma).........................253
  benötigt (Definition)...............252
  (Definition)........................183
  überflüssig (Definition).............252
Objektwert
  logisch (Definition)................189
Odberg, Erik...........................97
ODL............................155, 291
  erzeugende.......................175
  Generator....................242, 268
  Parser........................242, 268
  Taxonomie.......................156
  (Übersicht).......................291
Ordnungsrelation........... *siehe* Relation
ORION..........................61, 103

P

Penney, Jason..........................66
Persistenz.............................11
`PM_convert`..........................259
`PM_o_visible`........................259
`PMcreate`............................260
`PMdelete`............................261
`PMdelete_direct`.....................262
`PMprop_conv_write`..................264
`PMpropagate`.........................265
`PMread`..............................260
`PMread_direct`.......................261
`PMwrite`.............................261
Propagationsalgorithmen...............258
Propagationsbaum (Definition).........213

Propagationsflags
einer Klasse (Definition) 193
Erzeugungsflag (Invariante) 194
Kombination von 201
Löschungsflag (Invariante) 200
Modifikationflag (Invariante) 195
Schnappschußflag (Invariante) 193
von Extrakonvertierungs-
funktionen (Regel) 224
von Vorwärts- und Rückwärtskon-
vertierungsfunktionen (Regel) . . 191
Propagationsmanager 236, 249
Propagationssprache 202

Q

Quellschemaversion 152

R

Ra, Young-Gook 80
Regeln (Übersicht) 289
Relation
abgeleitet (Definition) 18
(Definition) 17
Eigenschaften (Definition) 18
Ordnungsr.
(Definition) 18
Zyklenfreiheit (Lemma) 18
Rückwärtskonvertierungsfunktion . . . . . *siehe* Konvertierungsfunktion
Rundensteiner, Elke 80

S

Saunders, Dave 95
Schema
erzeugen (SÄP) 157
integrieren 176
löschen (SÄP) 157
strukturell
(Definition) 125
konsistent (Definition) 137
umbenennen (SÄP) 157
unversioniert
(Definition) 127
konsistent (Definition) 137
verändern (SÄP) 157
verhaltensmäßig
(Definition) 126
konsistent (Definition) 137
versioniert
(Definition) 145
konsistent (Definition) 150
Vollständigkeit (Invariante) . . . 128, 149
Schemaableitungsgraph
(Definition) 143
Minimalität (Invariante) 150
Wohldefiniertheit (Definition) 212
Schemaänderungen
direkt 60
durch isolierte Datenbanken 58
durch Sichten 73
durch Versionierung 82
ohne persistente Objekte 57
Schemaänderungsprimitiv . . . . *siehe* ODL
Schemaassistent 237
Schemabeschreibungsprimitiv . . . *siehe* ODL
Schemabeschreibungssprache . . . . *siehe* ODL
Schemaeditor 237
Schemaintegration . . . . *siehe* Integration
Schemainvariante . . . . *siehe* Invarianten
Schemamanager 236, 247
Schemaversion
ableiten 151
(Regel) 158
(SÄP) 158
Ableitungszeit (Definition) 141
Anzahl reduzieren 152
Applikationsanbindung 153
auftauen
(Regel) 160
(SÄP) 160
auswählen für Applikation 155
(Definition) 141
Eindeutigkeit (Invariante) 149
einer Klasse (Definition) 147
einfrieren
(Regel) 160
(SÄP) 160
erzeugen (SÄP) 158
löschen 227
(Regel) 159
(SÄP) 159
umbenennen
(Regel) 159
(SÄP) 159
verändern
(Regel) 159
(SÄP) 159
Zustand (Definition) 141
Schnappschußflag . . *siehe* Propagationsflags
Scholl, Marc 80
SEA . . . . *siehe* Schemaassistent
Sichten 73
Signatur (Definition) 126

Skarra, Andrea 86
Sommerville, Ian 90
Speicherstruktur
  Lokalität (Invariante) 263
  Löschung (Invariante) 261
  Objektstruktur (Invariante) 246
  Sichtbarkeit (Invariante) 245
Staudt Lerner, Barbara 71
Stein, Jacob 66

T

Taxonomie
  von COAST 156
  von ORION 61
Teilgraph ..... *siehe* Graph
Teilziele 30
Tess 71
Tiefe
  einer Extrakonvertierungsfunktion (Definition) 220
  einer Klasse (Definition) 219
  eines Konvertierungspfades (Definition) 223
Tresch, Markus 80
TSE 80
Typ (Definition) 120
Typhierarchie (Definition) 122
Typisierung
  streng 10
Typkonformität
  Vererbung
    strukturell (Invariante) 130
    verhaltensmäßig (Invariante) 136

U

Überschreiben von Attributen (SÄP) ... 170
Überschreibung von Attributen löschen (SÄP) 170
Umbenennung
  von Attributen (SÄP) 170
  von Klassen 165
    (SÄP) 165
  von Methoden (SÄP) 171
  von Schemata (SÄP) 157
  von Schemaversionen (SÄP) 159
Umtypisierung von Attributen (SÄP) .. 170
Unterklasse (Definition) 125
Untertyp (Definition) 122
Ursprungsschemaversion eines Objektes (Definition) 183

V

Veränderung
  von Klassen 165
    (SÄP) 165
  von Methoden (SÄP) 171
  von Schemata (SÄP) 157
  von Schemaversionen (SÄP) 159
Vererbung 8
  Mehrfachv. (Invariante) 133
  Typkonformität
    strukturell (Invariante) 130
    verhaltensmäßig (Invariante) 136
  Vollständigkeit (Invariante) 129
Vererbungsgraph
  (Definition) 124
  (Invariante) 129
  Minimalität (Invariante) 129
Version 14
  Klassenv. (Definition) 146
  Objektv. (Definition) 183
  Schemav.
    (Definition) 141
    Quellschemav. 152
    Wurzelschemav. (Definition) 143
    Zielschemav 152
Versionierung 15, 115
Verteilung 14
Vollständigkeit 13
  Schema (Invariante) 128, 149
  Vererbung (Invariante) 129
Vorgänger
  (Definition) 19
  einer Knotenmenge (Definition) 20
Vorgängerklassenversion (Definition) ... 148
Vorgängerschemaversion
  (Definition) 144
  kleinste gemeinsame (Definition) ... 207
Vorwärtskonvertierungsfunktion ...... *siehe* Konvertierungsfunktion

W

Wald
  gerichtet (Definition) 20
Weg
  (Definition) 18
  einfach (Definition) 19
  elementar (Definition) 19
  geschlossen (Definition) 19
Wert (Definition) 119

Wertebereich
atomar (Definition) 118
eines Typs (Definition) 121
Wohldefiniertheit
Konvertierung (Lemma) 225
Schemaableitungsgraph (Lemma) . . 212
Wurzelklasse (Definition) 124
Wurzelschemaversion (Definition) 143

Z

Zdonik, Stanley 86
Zicari, Roberto 67
Zielschemaversion 152
Zugriffsbereich (Definition) 182
Zustand einer Schemaversion 159
(Definition) 141
Zyklus (Definition) 19

Lebenslauf

Zur Person

	Sven-Eric Lautemann
*	1.12.1967 in Karlsruhe
Mutter:	Berit Lautemann, geb. Jönsson (schwedisch)
Vater:	Dieter Lautemann (Kaufmann, deutsch)
Staatsangehörigkeit:	deutsch
Familienstand:	ledig
Adresse:	August-Bebel-Str. 43 D-65479 Raunheim
Tel.:	+49/69/798-28434
Fax:	+49/69/798-28353
email:	lautemann@dbis.informatik.uni-frankfurt.de
URL:	http://www.dbis.informatik.uni-frankfurt.de/~lauteman/

Ausbildung

7/74 – 6/78	Grundschule Wörth a. Rh.
7/78 – 5/87	Gymnasium Wörth a. Rh.
5/87	Abitur (Prüfungsfächer: Mathematik, Physik, Sozialkunde, Englisch)
7/87 – 9/87	Praktikum bei Daimler Benz, Wörth a. Rh. (Abt. Systemgestaltung)
10/87 – 9/89	Vorstudium Informatik, TU Karlsruhe, Nebenfach Mathematik Akademische Lehrer (u.a.): Prof. Dr. W. Zorn — Informatik Prof. Dr. A. Schmitt — Technische Informatik
10/89 – 1/93	Hauptstudium Informatik, TU Karlsruhe, Nebenfach BWL Akademische Lehrer: Prof. Dr. P. Lockemann — Informationssysteme (Vertiefungsfach und Diplomarbeit) Prof. Dr. H. Wettstein — Systemstrukturen und Benutzerschnittstellen (Vertiefungsfach) Prof. Dr. H. G. Gemünden — Unternehmensführung und Organisation (Ergänzungsfach) und (u.a.) Prof. Dr. J. Clamet, Prof. Dr. S. Jähnichen, Prof. Dr. W. Menzel, Prof. Dr. H. H. Nagel, Prof. Dr. U. Rembold, Prof. Dr. D. Schmid
1/93	Abschluß Diplom-Informatiker (Diplomarbeit [Lau93])

seit 9/93	Promotion Informatik, Universität Frankfurt a. M. [Lau96a, Lau96b, FL96, Lau97b, Lau97a, BGL97, BLB97, Lau99a] Akademische Lehrer: Prof. Dott. Ing. R. Zicari

Projekte

1/92 – 8/93	**IRIS**: Integrated Refinery Information System Mobil Marketing und Raffinerie GmbH, Wörth a. Rh.
1/93 – 2/96	**GOODSTEP**: General Object-Oriented Database for SofTware Engineering Processes Universität Frankfurt a. M. (ESPRIT-III Projekt 6115) [AAA+94, AAA+96]
1/98 – 4/99	**DOOR**: Design Object-Oriented applications Rapidly Universität Frankfurt a. M. (ESPRIT-IV Projekt 23768) [CCC+99]
1/96 – 6/00	**COAST**: Complex Object And Schema Transformation Universität Frankfurt a. M. [LEW97, BL98, Lau99b, Lau99a, LADH99]

Sonstiges

- Mitglied der Programm-Komitees von
  - Int'l Conference on Object-Oriented Information Systems (OOIS), 1998
  - Int'l Conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe), 1999 und 2000