Objektpropagation in einem objektorientierten Datenbanksystem mit Schemaversionierung

Diplomarbeit
von
Jan Haase

Johann Wolfgang Goethe-Universität Frankfurt am Main
Fachbereich 20: Informatik
Datenbanken und Informationssysteme (DBIS)

04.08.2000

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 2

1.2 Zielsetzung 2

1.3 Gliederung dieser Arbeit 2

2 Grundlagen **5**

2.1 Schema einer Datenbank 5

2.2 Objektorientierte Datenbanken 5

2.2.1 Objekte 6

2.2.2 Klassen 6

2.2.3 Klassenbeziehungen 6

2.2.4 Vererbung 7

2.2.5 Polymorphie 7

2.2.6 Kapselung 7

2.2.7 Datenbanken 8

3 Schemaevolution **9**

3.1 Probleme bei Schemaänderungen 10

3.2 Lösungsansätze 10

3.2.1 Komplette Neuerstellung 11

3.2.2 Kopie der Datenbank 11

3.2.3 Sichten 12

3.2.4 Schemaversionierung 13

3.3 Zusammenfassung und Bewertung 15

4 Schemaänderungen und Konvertierungsfunktionen **17**

4.1 Konsequenzen von Schemaänderungen auf Schemaebene 17

4.1.1 Einfache Schemaänderungen 18

4.1.2 Komplexe Schemaänderungen 19

4.2 Konsequenzen von Schemaänderungen auf Objektebene 20

4.2.1 Konvertierungsfunktionen 21
4.2.1.1 Default-Konvertierungsfunktionen 21
4.2.1.2 Einfache Konvertierungsfunktionen 23
4.2.1.3 Komplexe Konvertierungsfunktionen 24
4.2.2 Konsequenzen von Schemaänderungen für die Objektebene 24
4.2.2.1 Einfache Schemaänderungen 24
4.2.2.2 Komplexe Schemaänderungen 25
4.3 Typische Schemaänderungen 26
4.3.1 Einfache Schemaänderungsoperationen, bei denen Default-Konvertierungsfunktionen ausreichen 27
4.3.1.1 Umbenennen von Klassen 27
4.3.1.2 Löschen von Attributen 28
4.3.2 Einfache Schemaänderungsoperationen, bei denen Default-Konvertierungsfunktionen nicht ausreichen 29
4.3.2.1 Aufsplitten von Attributen 29
4.3.2.2 Verbinden von Attributen 31
4.3.2.3 Einfügen redundanter Attribute 32
4.3.2.4 Einfügen von Attributen und Belegung mit Startwerten . . 33
4.3.2.5 Typänderungen 34
4.3.3 Komplexe Schemaänderungsoperationen, bei denen Default-Konvertierungsfunktionen ausreichen 35
4.3.3.1 Kopieren von Attributen über Referenzen 36
4.3.3.2 Einfügen von Referenzen 37
4.3.3.3 Aufbrechen von Klassen 38
4.3.3.4 Zusammenführen von Klassen über Referenzen 41
4.3.3.5 Zusammenführen von Klassen über Wertevergleiche 43
4.3.3.6 Verschieben von Attributen entlang der Aggregationskanten 44
4.3.3.7 Umkehrung der Aggregationsreihenfolge 45
4.3.4 Komplexe Schemaänderungsoperationen, bei denen Default-Konvertierungsfunktionen nicht ausreichen 47
4.3.4.1 Kopieren von Attributen über Wertevergleiche 47
4.3.4.2 Verschieben von Attributen in eine Oberklasse 49
4.3.4.3 Verschieben von Attributen in eine Unterklasse 52
4.4 Migration 53
4.5 Zusammenfassung und Bewertung 56

5 Propagation **59**

5.1 Ziel 59

5.2 Konzepte . . . 60
5.3 Ausgangssituation . . . 60
5.4 Die Strategie der verzögerten Propagation . . . 60
5.5 Propagationsflags . . . 62
5.6 Der Propagationsgraph . . . 62
5.6.1 Propagationsgraph für einfache Konvertierungsfunktionen . . . 63
5.6.2 Propagationsgraph für komplexe Konvertierungsfunktionen . . . 63
5.6.2.1 Von zwei Quellklassenversionen zu einer Zielklassenversion 64
5.6.2.2 Von einer Quellklassenversion zu zwei Zielklassenversionen 67
5.6.3 Zurücklaufende Propagation . . . 70
5.7 Transitive Propagation . . . 71
5.8 Zusammenfassung und Bewertung . . . 72

6 Eine Sprache für Konvertierungsfunktionen 75
6.1 Funktionale Ziele . . . 75
6.2 Konzeptionelle Ziele . . . 77
6.3 Einfache Konvertierungsfunktionen . . . 79
6.3.1 Schreibzugriff auf Zielattribute . . . 79
6.3.2 Lesezugriff auf Quellattribute . . . 79
6.3.3 Zugriff auf Variablen mit komplexen Typen . . . 80
6.3.4 Berechnung von Werten . . . 80
6.3.5 Lokale Variablen . . . 81
6.3.6 Globale Konstanten . . . 81
6.3.7 Kommentare . . . 82
6.3.8 Bedingungen . . . 82
6.3.9 Schleifen . . . 82
6.4 Komplexe Konvertierungsfunktionen . . . 83
6.4.1 Zugriff auf andere Klassen . . . 83
6.4.2 Erstellen und Dereferenzieren von Referenzen . . . 83
6.4.3 Suchen von Objekten . . . 84
6.4.4 Erzeugen und Löschen von Objekten . . . 84
6.4.5 Existenzprüfung von Objekten . . . 85
6.4.6 Zusammenfassung . . . 85
6.5 Datenerhaltende Konvertierungsfunktionen . . . 86
6.6 Syntax der Konvertierungssprache in BNF . . . 87
6.7 Zusammenfassung und Bewertung . . . 88

7 Implementierung **89**

7.1 Funktionsweise von COAST 89

7.2 Architektur des COAST-Projekts 90

7.2.1 Objektmanager 90

7.2.2 Schemamanager 90

7.2.3 Propagationsmanager 92

7.2.4 ODL-Parser und ODL-Generator 93

7.2.5 Schemaeditor 94

7.2.6 Schematool SEA 94

7.2.7 COAST Runtime 94

7.3 Realisierungskonzepte 94

7.3.1 Entwicklungsumgebung 94

7.3.2 Vorgefundene Situation 95

7.3.3 Die Konfigurationsdatei .coastrc 95

7.3.4 Wann muss eine Propagation erfolgen? 95

7.4 Speicherung von Konvertierungsfunktionen 96

7.5 Speicherung der Historie von Schemaänderungen 98

7.6 Effizienzbetrachtungen 99

7.6.1 Optimierung des Programmablaufs 99

7.6.2 Optimierung von Schemaänderungsoperationen 100

7.7 Zusammenfassung und Bewertung 101

8 Zusammenfassung und Ausblick **103**

8.1 Zusammenfassung 103

8.2 Ausblick 104

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe.

Frankfurt am Main, den

(Einsetzen der Kopie der Prüfungsamtsbestätigung über die Vergabe der Diplomarbeit!)

Danksagung

Ich möchte mich zunächst bei Herrn Prof. Dott. Ing. Zicari für die Vergabe des interessanten Diplomarbeitsthemas bedanken. Mein besonderer Dank gilt meinem Betreuer Herrn Dipl.-Inform. Sven-Eric Lautemann für viele anregende Diskussionen und fortwährende konstruktive Kritik. Des Weiteren danke ich dem gesamten COAST-Team für Tipps und Teamgeist, besonders eingeschlossen den Systemadministrator Rainer Konrad.

Für viele gemeinsame Prüfungsvorbereitungen und gute Zusammenarbeit das ganze Studium hindurch danke ich Thorsten Dröge und Kai Dušek, letzterem insbesondere auch für Tipps und Kniffe im Zusammenhang mit LaTeX.

Ein besonderer Dank gilt auch Herrn Dipl.-Inform. Andreas Bleck, der im Verlauf des Studiums immer wieder eine Anlaufstelle bei Problemen oder auch nur beim Berichten von der jeweils gerade bestandenen Prüfung war und den ich gewissermaßen als meinen Mentor angesehen habe.

Anmerkung

Diese Diplomarbeit wurde nach der seit 1. August 1998 geltenden deutschen Rechtschreibung erstellt.

Kapitel 1

Einleitung

Der Bereich objektorientierter Datenbanken führt zurzeit noch ein Nischendasein, die meisten Datenbanken sind relational organisiert. Dabei bietet gerade die Objektorientierung eine hohe Flexibilität bei der Beschreibung und Modellierung, eine deutliche Verbesserung gegenüber dem Konzept der relationalen Datenbanken.
Im Allgemeinen werden objektorientierte Datenbanken in den großen Firmen jedoch nur als „nett“, aber nicht konkurrenzfähig angesehen, weil man annimmt, dass sie im Vergleich zu relationalen Datenbanken, die seit über 30 Jahren verwendet werden, mit ihrer kurzen Entwicklungszeit von nur etwa 10 Jahren nicht ausgereift sein können. Zugegeben, bei anspruchslosen Massendatenbanken wie Telefonbüchern stellt das relationale Konzept die bessere Alternative dar, weil es schneller ist und keine komplexen Beziehungen modelliert werden müssen. Aber gerade bei komplexen Sachverhalten können objektorientierte Datenbanken ihre Stärken zeigen.

Die Modellierung der Diskurswelt in Datenbanken zeigt sich im Datenbankschema. Ändern sich die beschriebenen Objekte oder Sachverhalte, muss auch das Schema angepasst werden. Dieser Vorgang nennt sich Schemaevolution. Da solche Änderungen häufig erfolgen müssen, ist für die Entwicklung einer performanten Datenbank eine besondere Untersuchung von Schemaänderungen lohnenswert.
Schmemaänderungen ziehen immer eine Reihe von Konsequenzen nach sich: Die in der Datenbank abgelegten Daten müssen an das neue Schema angepasst werden und die Applikationen, die mit der Datenbank arbeiten, müssen umprogrammiert und neu kompiliert werden. Oftmals ist der Quellcode von alten Applikationen in inzwischen ungebräuchlichen Sprachen wie COBOL programmiert oder auch einfach nicht mehr auffindbar. Die Erstellung neuer Applikationen erfordert viel Zeit und zusätzlich Schulungsmaßnahmen für die Benutzer.
Schemaänderungen sind damit aufwändig und teuer. Aus diesem Grund werden Schemaänderungen i.A. nur dann durchgeführt, wenn es unumgänglich ist, erfahrungsgemäß in Abständen von mehreren Monaten oder Jahren.

Der Prototyp COAST (Complex Object and Schema Transformation), der an der Professur für Datenbanken und Informationssystem (DBIS) der J.W.G.-Universität Frankfurt am Main entwickelt wird, bietet Unterstützung für die Durchführung von Schemaänderungen, indem mehrere *Schemaversionen* parallel verwaltet und benutzt werden können. Die Objektorientierung, die die größten Modellierungsmöglichkeiten bietet und daher in COAST verwendet wird, bietet das Konzept der Vererbung. Dieses wird bei der Schemaversionierung genutzt: Durch eine Schemaänderung wird eine neue Schemaversion von einer bereits existierenden Schemaversion abgeleitet und steht dadurch mit ihr in einer Beziehung. Damit enthält die neue Schemaversion zunächst den gleichen Aufbau und wird

dann an die neuen Erfordernisse angepasst.
Die Existenz verschiedener Schemaversionen wird auf die Datenobjekte der Datenbank übertragen, so dass diese ebenfalls in mehreren *Objektversionen* vorliegen – die durch die Beziehung zwischen den Schemaversionen ebenfalls in Beziehung zueinander stehen. Die Daten in den Objektversionen können durch Ausnutzung dieser Beziehungen automatisch aktualisiert werden – und zwar zur Laufzeit. Dieser Vorgang nennt sich *Propagation*.
Der Vorteil von mehreren Schemaversionen liegt darin, dass Applikationen, die von der Schemaänderung nicht betroffen sind, auch nicht angepasst werden müssen, sondern einfach weiterverwendet werden können. Durch die automatische Konvertierung von Daten zwischen verschiedenen Schemaversionen kann sowohl die alte wie auch die neue Schemaversion parallel genutzt werden. Dem Schemaentwickler bietet sich so die Möglichkeit, auch in kürzeren Abständen Schemaänderungen vorzunehmen.
Abgesehen davon ist so eine Modellierung der Diskurswelt aus verschiedener Sicht möglich: Soll beispielsweise in einer Firma ein Produkt sowohl aus Sicht der Fertigung als auch aus Sicht des Vertriebs modelliert werden, bietet die Schemaversionierung eine geeignete Basis, um beide Beschreibungsformen miteinander in Beziehung zu setzen. Auch hier ist damit die automatische Propagation von Zustandsänderungen der Daten möglich.

1.1 Motivation

In der Praxis reichen einfache Schemaänderungen nicht aus. Diese bieten nur Änderungen innerhalb jeweils einer Klasse, beispielsweise das Hinzufügen oder Löschen eines Attributs. Eine Erweiterung auf komplexe Schemaänderungen, die auch aus mehr als einer Quellklasse Attribute in mehr als eine Zielklasse übernehmen können, wurde bereits für andere Systeme vorgeschlagen.
Die Idee ist nun, komplexe Schemaänderungen auch in COAST einzuführen und damit auf das leistungsfähigste Konzept, eben dem der Schemaversionierung, zu übertragen.

Für mich stellte sich das mir angebotene Diplomarbeitsthema als hochinteressant dar, zumal ein Mechanismus wie die Schemaversionierung mit der Propagation ein Novum im Bereich der Datenbanken ist.

1.2 Zielsetzung

In dieser Diplomarbeit wird COAST durch die Einführung komplexer Schemaänderungen erweitert. Diese werden dazu zunächst spezifiziert und ihre Auswirkungen auf die Propagation analysiert. Die Analyse führt dazu, dass die Propagation ebenfalls erweitert wird, um komplexe Schemaänderungen nachbilden zu können. Für die Beschreibung der Propagation auf Objektebene wird eine Sprache für Konvertierungsfunktionen entwickelt.

1.3 Gliederung dieser Arbeit

Diese Diplomarbeit besteht aus den folgenden Kapiteln:

2. **Grundlagen**
   In Kapitel 2 werden einige Grundbegriffe und Konzepte erläutert, die im Umfeld der Diplomarbeit wichtig sind.

3. **Schemaevolution**
   Schemaänderungen haben nicht nur Auswirkungen auf das Schema einer Datenbank, sondern auch auf die darin gespeicherten Daten und die auf sie zugreifenden Applikationen. Das Konzept der Schemaversionierung bietet sich als die im Vergleich leistungsfähigste Möglichkeit zur Unterstützung von evolutionären Schemaänderungen an. Ein Kern der Schemaversionierung ist die Propagation, um die es in dieser Diplomarbeit geht.
   In Kapitel 3 werden Schemaänderungen vorgestellt und auf Schwierigkeiten bei deren Umsetzung eingegangen. Es werden verschiedene Lösungsansätze skizziert und dabei besonderes Augenmerk auf das Konzept der Schemaversionierung, das auch dem COAST-Projekt zugrundeliegt, gelegt.

In den folgenden Kapiteln werden die in dieser Diplomarbeit neu eingeführten komplexen Schemaänderungsoperationen beschrieben und deren Auswirkung auf die Schema- und die Objektebene untersucht. Es schließt sich eine Analyse der daraus resultierenden nötigen Erweiterungen an, gefolgt von der Einführung einer neuen Art von Propagationskanten und der Entwicklung einer Sprache zur Spezifikation der Propagation.

4. **Schemaänderungen und Konvertierungsfunktionen**
   In Kapitel 4 werden verschiedene typische Schemaänderungen analysiert und klassifiziert. Dabei wird festgestellt, dass einige davon nicht mehr mit den bereits vorhandenen *einfachen* Schemaänderungsoperationen auskommen. Aus dem Grunde wird das Konzept um *komplexe* Schemaänderungsoperationen erweitert. Diese erhalten eine Syntax gemäß der Schemabeschreibungssprache ODL (Object Definition Language, s. [Her99]). Des Weiteren wird angegeben, welche Default-Konvertierungsfunktionen bei den vorgestellten Operationen durch das System erzeugt werden sollen. Diese werden in der Syntax der in Kapitel 6 entwicklten Konvertierungssprache angegeben. Außerdem wird auf das Konzept der Migration von Objekten innerhalb einer Schemaversion eingegangen.

5. **Propagation**
   Schemaänderungen haben Auswirkungen auf die Objektebene, es müssen dort entsprechend ebenfalls Änderungen durchgeführt werden. Die automatische Propagation von Zustandsänderungen wird in Kapitel 5 beschrieben und ein besonderes Augenmerk auf das Konzept der verzögerten Propagation gelegt.
   Durch die Einführung von komplexen Schemaänderungsoperationen muss das Konzept von Propagationskanten erweitert werden. Die dafür nötigen kombinierten Propagationskanten werden in ebenfalls in diesem Kapitel entwickelt und vorgestellt.

6. **Eine Sprache für Konvertierungsfunktionen**
   Man benötigt Konvertierungsfunktionen, um die Propagation zu spezifizieren. Die dafür im Rahmen dieser Diplomarbeit entwickelte Sprache wird in Kapitel 6 vorgestellt.

7. **Implementierung**
   In Kapitel 7 wird eine Übersicht über den Prototyp COAST gegeben und auf Besonderheiten bei der Implementierung der Konzepte aus den Kapiteln 4, 5 und 6 eingegangen. Weiter werden Möglichkeiten zur Effizienzverbesserung beschrieben.

8. **Zusammenfassung und Ausblick**
   In Kapitel 8 wird die Diplomarbeit zusammengefasst und auf weitere zu diskutierende Probleme im Zusammenhang mit der Schemaversionierung hingewiesen.

Kapitel 2

Grundlagen

In diesem Kapitel werden einige verwendete Begriffe und Konzepte aus dem Umfeld von Datenbanken und objektorientierten Programmiersprachen erklärt. Die eingeführten Begriffe sollen dem fachlich „entfernteren“ Leser das Verständnis dieser Arbeit erleichtern, das Kapitel kann von anderen übersprungen werden.

2.1 Schema einer Datenbank

Man versucht, in einer Datenbank die Diskurswelt – oder zumindest die für die jeweilige Anwendung relevanten Teile davon – abzubilden. Diese Modellierung der Diskurswelt nennt man *Schema*. Ein Schema in objektorientierten Datenbanken besteht aus mehreren Klassen und den Zusammenhängen zwischen ihnen. Eine Klasse besitzt *Attribute*, die den Zustand von Objekten beschreiben und *Methoden*, die das Verhalten beschreiben. Attribute haben einen Namen und einen Datentyp, also beispielsweise `string`, `integer`, `real` oder `date`.

Eine Klasse „Angestellter“ aus einem einfachen Beispielschema, die die Attribute `Vorname` vom Typ `string`, `Name` vom Typ `string`, `Geburtsdatum` vom Typ `date` und `Jahresgehalt` vom Typ `real` besitzt, könnte wie folgt aussehen:

```
CLASS Angestellter {
  ATTRIBUTES {
    Vorname STRING
    Name STRING
    Geburtsdatum DATE
    Jahresgehalt REAL
  }
}
```

2.2 Objektorientierte Datenbanken

Die Möglichkeiten zur Modellierung komplexer Zusammenhänge sind bei objektorientierten Datenbanken weitaus größer als bei relationalen Datenbanken. In dieser Diplomarbeit wird der Ansatz objektorientierter Datenbanksystemen verfolgt. Daher ist es notwendig, in diesem Bereich einige Konzepte einzuführen.

2.2.1 Objekte

Ein Objekt wird in [Dud93] folgendermaßen definiert:

„Ein Objekt ist ein Informationsträger, der einen (zeitlich veränderbaren) Zustand besitzt und für den definiert ist, wie er auf bestimmte eingehende Mitteilungen an das Objekt zu reagieren hat.“

Der Zustand eines Objektes ist durch die Werte seiner Attribute festgelegt. Änderungen können durch Zugriffe auf die Methoden des Objektes erfolgen – dadurch werden also die Reaktionen und das Verhalten eines Objektes beschrieben. Methoden sind objektinterne Funktionen, die zum einen interne Zustandsänderungen ermöglichen, zum anderen aber auch die Schnittstelle zur Außenwelt darstellen. Objekte kommunizieren über Methodenaufrufe miteinander.
Jedes Objekt ist eindeutig, es besitzt einen Objekt-Identifikator (*oid*).

2.2.2 Klassen

Objekte gleicher Struktur werden zu Klassen zusammengefasst. In [Boo94] heißt es:

„Eine Klasse ist eine Menge von Objekten, die eine gemeinsame Struktur und ein gemeinsames Verhalten aufweisen.“

Aus einer anderen Perspektive betrachtet kann man sagen, dass eine Klasse eine abstrakte Beschreibung eines Objekts darstellt, wovon beliebig viele gleichartige Objekte *instanziiert*, also nach diesem Bauplan erstellt werden können. Auf diese Weise muss nicht jedes Objekt einzeln in seiner Struktur und seinen Verhaltensweisen beschrieben werden, sondern es genügt, diese Beschreibung einmalig für die Klasse vorzunehmen. Diese Beschreibung wird als *Klassenintension*, die Menge aller Objekte einer Klasse als *Klassenextension* bezeichnet.

2.2.3 Klassenbeziehungen

Klassen können in Beziehungen zueinander stehen und beispielsweise Referenzen aufeinander haben. Man unterscheidet im wesentlichen drei Arten von Beziehungen:

- **Aggregation**
  Stellen Objekte einer Klasse Komponenten von Objekten einer anderen Klasse dar, liegt also eine „is-part-of“-Beziehung vor, handelt es sich um eine Aggregationsbeziehung. Ein Beispiel dafür ist eine Klasse „Blatt“, die als Komponente der Klasse „Baum“ referenziert wird.

- **Spezialisierung / Generalisierung**
  Man spricht von der Spezialisierung, wenn Objekte der einen Klasse Spezialfälle der Objekte der anderen Klasse sind – es liegt dann eine „is-a“-Beziehung vor. Die umgekehrte Sichtweise wird als Generalisierung bezeichnet. Ein Beispiel dafür ist eine Klasse „Nadelbaum“ als eine Spezialisierung der Klasse „Baum“, und damit wäre die Klasse „Baum“ eine Generalisierung der Klasse „Nadelbaum“.

- **Assoziation**
  Kann eine Klassenbeziehung weder als Aggregation noch als Spezialisierung bzw. Generalisierung eingeordnet werden, spricht man von einer Assoziationsbeziehung. Ein Beispiel dafür ist eine Beziehung zwischen den Klassen „Gärtnereibetrieb“ und

„Baum“, denn weder ist ein Baum Teil eines Gärtnereibetriebes oder andersherum, noch ist ein Baum eine Spezialisierung oder Generalisierung eines Gärtnereibetriebes.

Es ist möglich, dass Klassen gleichzeitig zu mehreren anderen Klassen in gleichen oder verschiedenen Beziehungen stehen, wie in den obigen Beispielen mit der Klasse „Baum“ zu sehen.

2.2.4 Vererbung

Ein Ziel der Objektorientierung ist die Wiederverwendbarkeit von Code. Die Vererbung ist ein Konzept, das dieser Forderung Rechnung trägt: Ähnliche Klassen können voneinander abgeleitet werden. Dabei erbt eine Klasse (die *Unterklasse*) von einer anderen Klasse (der *Oberklasse*) all ihre Attribute und Methoden. Anschließend können weitere (lokal definierte) Attribute oder Methoden hinzugefügt werden. Da Unterklassen alle Attribute und Methoden ihrer Oberklassen besitzen, zählen sie auch zur Klassenextension der Oberklasse. Es ist möglich, dass eine Klasse mehrere Oberklassen hat, also von mehreren Klassen deren Attribute und Methoden erbt.
Die Vererbung ist in der Objektorientierung die Umsetzung des Konzepts der Spezialisierung. Für objektorientierte Datenbanken und insbesondere COAST ist die Vererbung insofern wichtig, als dass neben Klassen auch Schemata voneinander abgeleitet werden können und automatisch alle Eigenschaften erben.

2.2.5 Polymorphie

Ein weiteres wichtiges Konzept in der Objektorientierung ist die Polymorphie, die Vielgestaltigkeit von Klassen. Durch die Polymorphie wird ermöglicht, die Methoden einer geerbten Klasse noch weiter zu verändern. Zum einen kann eine Methode lokal redefiniert werden (*Überschreiben* oder engl. *Override*), zum anderen kann sogar eine Methode mehrere Implementierungen erhalten. Je nach Art der mit dem Methodenaufruf übergebenen Parameter wird entschieden, welche Methodenimplementierung Verwendung findet (*Überladen* oder engl. *Overload*).

2.2.6 Kapselung

Ist eine Änderung des Zustands eines Objekts nur durch Aufruf seiner Methoden möglich, spricht man von *Kapselung*. Dadurch ist gewährleistet, dass die Auswirkungen von Änderungen am Verhalten oder an der Struktur einer Klasse auf die betroffene Klasse beschränkt bleiben.
Die Wartbarkeit verbessert sich ebenfalls, denn auf diese Weise kann eine Anpassung der Klasse an neue Umstände erfolgen, indem die interne Implementierung modifiziert wird. So ändert sich zwar das ausgegebene Ergebnis, nicht jedoch die Art und Weise, wie das Objekt angesprochen wird und auch nicht der Rückgabetyp. Der Programmierer kann punktgenau eingreifen. Ein Objekt wirkt damit wie eine „Black Box“ für Applikationen: Es kann von außen über Methodenaufrufe angesprochen werden, aber wie es die Methodenaufrufe realisiert, ist von außen unsichtbar.
Im Softwareentwurf ist eine Forderung die Modularität, diese wird durch die Kapselung realisiert.

2.2.7 Datenbanken

Wenn das Konzept der Objektorientierung auf Datenbanken übertragen wird, bieten sich dem Datenbankenprogrammierer im Vergleich zu relationalen Datenbanken neue Möglichkeiten, komplexe Zusammenhänge zu modellieren [ABDW$^+$89, Heu97, STS97, BCG$^+$87]. So ist es beispielsweise möglich, eigene komplexe Typen wie `set`, `list`, `graph`, `bag`, `tuple`, etc. zu erstellen.

In relationalen Datenbanken kann nur mit Tabellen (Listen von gleichartigen Tupeln) gearbeitet werden, und es können andere Tabellen in Schlüsselattributen referenziert werden. Vererbung ist nicht möglich. Demgegenüber bieten objektorientierte Datenbanken zusätzlich noch Methoden in den Datenobjekten an, d.h. die Verhaltensweise kann logisch dort bereitgestellt werden, wo sie verwendet wird.

Kapitel 3

Schemaevolution

Ein zentrales Problem im Bereich von Datenbanken besteht darin, dass Schemaänderungen in den meisten Fällen auch Änderungen an den darauf zugreifenden Applikationen und vor allem den in der Datenbank bereits vorhandenen Daten zur Folge haben. Eine sehr elegante Lösung dieses Problems ist das Konzept der Schemaversionierung. Der Kern dieses Verfahrens ist die Propagation, die einer der Schwerpunkte dieser Diplomarbeit ist, und die ab dem nächsten Kapitel ausführlich behandelt wird.

Um ein Gefühl für die Idee der Schemaversionierung und einen Eindruck zu bekommen, welchen Schwierigkeiten der Schemaentwickler gegenübersteht, wird in diesem Kapitel auf diese Probleme eingegangen und mehrere Lösungsvorschläge aus der Literatur diskutiert.

Der Ablauf der Neuentwicklung einer Datenbank besteht aus vier Phasen:

1. Die Analyse der Erfordernisse für die zu erstellende Datenbank.
2. Die Modellierung der gewonnenen Erkenntnisse, beispielsweise mit Hilfe des Entity-Relationship- (ER-) Modells.
3. Die Umsetzung von der abstrakten Modellierungsebene in ein Datenbankschema.
4. Die Testphase, in der die Datenbank genutzt und Fehler und Schwächen entdeckt werden sollen.

Es wird nicht zu vermeiden sein, dass im Verlauf der Entwicklung immer wieder Fehler ausgemerzt oder auch zu Beginn noch nicht bekannte Anforderungen eingeflochten werden müssen. Daher ist immer wieder eine nachträgliche Änderung des Schemas nötig, bis die Datenbank für den täglichen Einsatz beim Anwender tauglich ist. Der Vorgang ist dann i.A. derselbe wie bei der Neuentwicklung, die Gewichtung verschiebt sich dann nur etwas mehr zu den Phasen 3 und 4.

Aber auch bereits im Einsatz befindliche Datenbanken können nötigen Modifikationen unterworfen sein: Es kommen Veränderungen der äußeren Gegebenheiten vor, die in der Datenbank modelliert werden sollen (beispielsweise Währungsumstellungen von DM auf Euro), Erweiterungen der vorhandenen Klassen um neue Attribute, die Forderung nach neuen Klassen oder neuen Spezialisierungen existierender Klassen. Mit zunehmender Lebensdauer steigt die Wahrscheinlichkeit, dass Änderungen erforderlich werden. Es können Fehler im Design der vorhandenen Datenbank entdeckt werden oder sogar ein ganzer Bereich der Datenbank überflüssig werden, beispielsweise wenn in einer Firma die Herstellung eines Produkts eingestellt wird.

3.1 Probleme bei Schemaänderungen

Eine Schemaänderung zieht weitere notwendige Änderungen mit sich: Die Applikationen, die auf das Schema zugreifen, müssen an die neue Situation angepasst werden. Werden beispielsweise Attribute gelöscht oder haben sich die Namen von Attributen geändert, würde eine unveränderte Applikation „ins Leere greifen“. Wurden neue Attribute oder Klassen hinzugefügt, sollen diese durch die Applikationen auch mit Daten versehen werden und abgefragt werden können.
Dieser zusätzliche Aufwand führt im laufenden Betrieb oft dazu, dass Schemaänderungen nur in großen Zeitabständen vorgenommen werden, beispielsweise, wenn sich viele Änderungswünsche angesammelt haben oder wenn die alte Datenbank aus zwingenden Gründen nicht mehr weiter verwendet werden kann. Bis dahin wird versucht, sich mit den vorhandenen Strukturen zu arrangieren. Ein lohnendes Ziel wäre sicherlich, Schemaänderungen jederzeit ohne Produktivitätseinbußen durchführen zu können.

Weiter kann es sein, dass das Schema der Datenbank zur Laufzeit nicht modifiziert werden kann, es müssen also für den Umstellungsvorgang alle Applikationen, die auf die Datenbank zugreifen, beendet und die Datenbank selber heruntergefahren werden. In dieser Zeit ist die Datenbank nicht benutzbar, der Produktivitätsausfall sollte also möglichst kurz gehalten werden.
Die Umstellung kostet aber gerade bei großen Datenbeständen viel Zeit, vor allem, wenn viele Datensätze manuell konvertiert werden müssten. Hier wäre es hilfreich, wenn die Konvertierung aller Daten automatisch durchgeführt werden könnte.

Noch sorgfältiger sollte die Entscheidung, ein Schema zu verändern und die Datenbank umzustellen, überdacht werden, wenn durch die Schemaänderung Daten verloren gehen. Das kann beispielsweise durch Löschen von Attributen oder Klassen oder auch durch Veränderung von Datentypen in Typen mit geringerer Genauigkeit (z.B. von Fließkomma-(`real`-)Werten zu ganzzahligen (`integer`-)Werten) geschehen. Die dann einmal verlorenen Informationen sind nur noch durch Einspielen eines Backups wiederherstellbar, so dass sichergestellt werden muss, dass kein Fehler bei der Schemaänderung passiert, der die Daten unwiederbringlich löscht oder zumindest viel Zeit und Arbeit kostet, wenn der Vorgang rückgängig gemacht werden muss.

3.2 Lösungsansätze

Aus den genannten Problematiken ergeben sich die folgenden Forderungen an das Datenbanksystem:

- Eine Schemaänderung sollte zur Laufzeit möglich sein.
- Eine Schemaänderung sollte mit möglichst geringem Aufwand und damit insbesondere in kürzeren zeitlichen Abständen durchführbar sein.
- Die betroffenen Daten sollten automatisch konvertiert werden.
- Es sollten möglichst wenige Änderungen an den Applikationen, die auf die Datenbank zugreifen, nötig sein.
- Daten, die bei der Schemaänderung gelöscht werden, sollten im Notfall weiterhin verfügbar sein.

Es gibt verschiedene Strategien, die Schemaänderungen zu realisieren, um dabei möglichst viele der genannten Forderungen zu erfüllen.

3.2.1 Komplette Neuerstellung

Die einfachste Methode ist die komplette Neuerstellung der Datenbank, dabei werden sämtliche Strukturen neu entwickelt und erzeugt. Allerdings sind keine Beziehungen zwischen den Daten der alten Datenbank und den im weiteren Verlauf in die neue Datenbank eingegebenen Daten vorhanden. Eine automatisierte Konvertierung der Daten (beispielsweise über spezielle Applikationen, die Daten aus der alten Datenbank abfragen und in die neue Datenbank eintragen) ist zumeist schwierig oder sogar unmöglich.

Es ist nicht gesichert, dass nach einer Neuerstellung alle Daten in die neue Datenbank übernommen werden können. Die Applikationen, die die Datenbank verwenden, müssen ebenfalls neu erstellt oder mit erheblichem Aufwand angepasst werden, eventuell unter Wiederverwendung von Teilen der alten Quellcodes.

3.2.2 Kopie der Datenbank

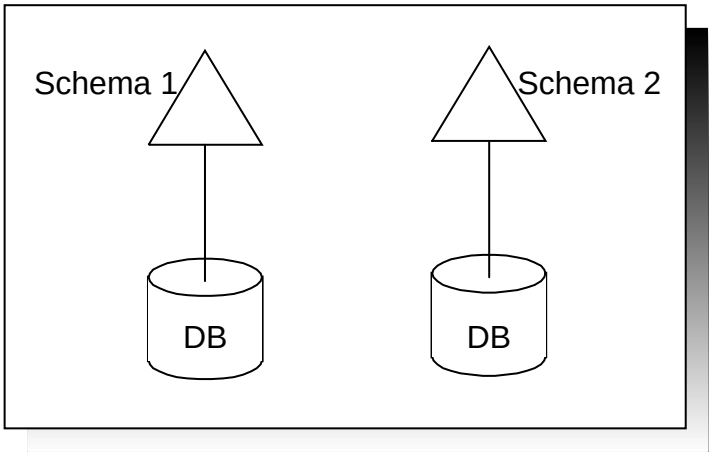

Abbildung 3.1: Komplette Kopie einer Datenbank und des Schemas

Alternativ zur kompletten Neuerstellung der Datenbank bietet sich an, eine Kopie der Datenbank anzulegen und an der neuen Kopie die nötigen Schemaänderungen vorzunehmen (s. Abb. 3.1). Dies kann beispielsweise durch Schemaänderungsoperationen oder durch direkte Manipulation der Schemabeschreibung geschehen. Applikationen, die von den Änderungen nicht berührt werden, können weiter auf der alten Datenbank laufen, neue Applikationen können das neue Schema verwenden und je nach Implementierung vielleicht sogar auf beide Schemata zugreifen.
Der Nachteil dieser Vorgehensweise besteht in der Redundanz der Daten: Alle Daten liegen doppelt vor und werden ohne regelmäßigen Abgleich nach und nach immer weiter voneinander abweichen. Ein solcher Abgleich ist allerdings sehr aufwändig. Er kann i.A. nur durch speziell dafür entwickelte Software erfolgen, die Änderungen in der einen Datenbank sucht und in der anderen nachholt. Eine solcher Vorgang braucht viel Zeit, da die gesamte Datenbank durchgegangen werden muss. Eine Aktualisierung wird daher nur in grösseren Abständen wie beispielsweise jede Nacht oder noch seltener durchgeführt werden.

Dieser Ansatz, eine Schemaänderung zu ermöglichen, ist zwar zur Laufzeit möglich, stellt jedoch einen hohen Aufwand dar, wenn die Daten kopiert und an das neue Schema angepasst werden. Der nachträglich immer wieder notwendige Abgleich der redundant vorliegenden Daten ist aufwändig und teuer. Es müssen nicht alle Applikationen verändert

werden, sondern nur diejenigen, die von der Schemaänderung betroffen sind. Durch die Schemaänderung gelöschte Daten liegen weiterhin in der alten Datenbank vor.

3.2.3 Sichten

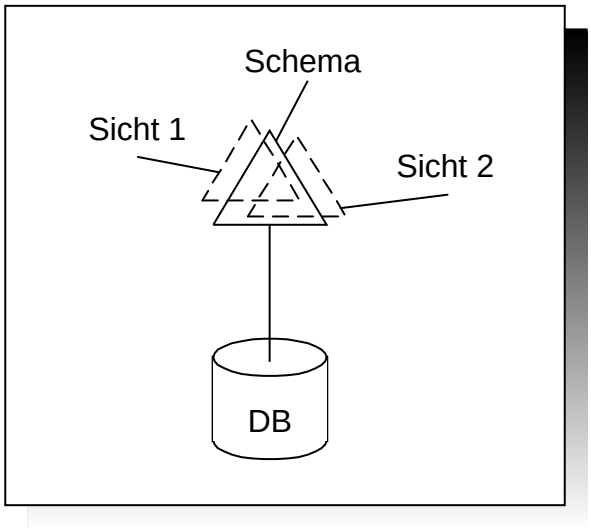

Abbildung 3.2: Sichten auf einem Schema

Eine Weiterentwicklung des Schemas kann auch durch Sichten (engl. views) simuliert werden [BFK95, RR95]. Die Idee ist, ein einziges großes Basisschema zu haben, das unverändert bleibt, aber statt einer Schemaänderung eine neue Sicht anzulegen, die nur die jeweils gewünschten Klassen und Attribute zeigt (s. Abb. 3.2).
Applikationen können auf Sichten zugreifen und so unabhängig davon, wie die Daten tatsächlich in der Datenbank vorliegen, arbeiten. Abgesehen davon ist es auf diese Weise möglich, eine geplante Schemaänderung vorab zu testen, bevor die Daten wirklich umgestellt werden. Allerdings ist es nicht möglich, Daten nur in einer einzigen Sicht zu ändern. Änderungen werden durch die Tatsache, dass die Daten nur einmalig vorliegen, automatisch überall sichtbar.
Weiter gibt es Probleme, wenn beispielsweise eine Sicht erzeugt wird, die die beiden Klassen „Arbeiter“ und „Angestellter“ auf eine nur in der Sicht des Benutzers existente Klasse „Person“ abbildet und der Benutzer dann einen neuen Datensatz in dieser Klasse „Person“ anlegt. Das System kann in diesem Fall nicht entscheiden, ob dieser Datensatz zur Klasse „Arbeiter“ oder zur Klasse „Angestellter“ gehören soll. Abgesehen davon ist die Anzahl der möglichen Schemaänderungsoperationen begrenzt: Beispielsweise kann das Löschen oder Umbenennen von Attributen problemlos modelliert werden, während Neuanlegen oder Typänderungen von Attributen nicht direkt möglich sind. Man muss in diesem Fall die neuen Attribute im Basisschema hinzufügen und die entsprechenden Sichten anpassen.

Eine Schemaänderung ist zur Laufzeit möglich, der Aufwand hält sich ebenfalls in Grenzen. Allerdings sind Applikationen, die auf neue Sichten zugreifen sollen, auch anzupassen. Eine Datenkonvertierung ist nicht notwendig, da für die Daten selbst nur eine einzige Datenbasis existiert. Die in den neuen Sichten nicht mehr verfügbare Daten sind aber weiterhin in der Datenbank vorhanden und können (auf anderem Wege) trotzdem erreicht werden.

3.2.4 Schemaversionierung

Bei der Schemaversionierung wird das Konzept der Versionierung, das auch in anderen Bereichen wie Textverarbeitung, Programmierung im Team, etc. (s. auch [CJ90]) und auch auf Objektebene [CK86, DL88, Kat90, Sci91, TOC93], schon länger eingesetzt wird, auf ein Datenbankschema übertragen.

Bei einer Datenbank mit Schemaversionierung resultiert die Durchführung einer Schemaänderung in der Erzeugung einer neuen *Schemaversion*. Der bisherige Zustand des Schemas bleibt dabei als Schemaversion in unveränderter Form erhalten. Somit existieren nach Schemaänderungen mehrere Schemaversionen, die sich beispielsweise in Typen von einzelnen Attributen oder ganzen Klassen unterscheiden.

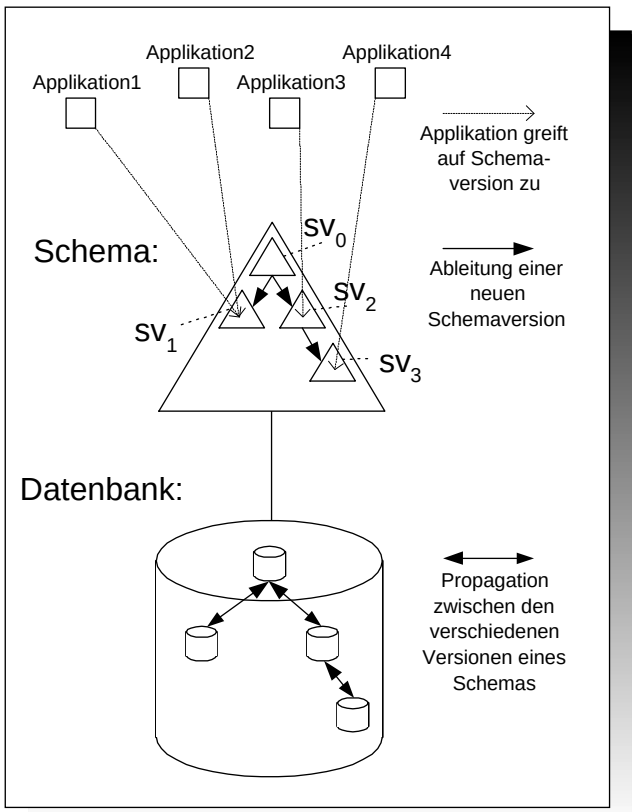

Abbildung 3.3: Schemaversionierung: Mehrere Applikationen greifen auf verschiedene Versionen des Schemas zu. Die Datenobjekte liegen in Zugriffsbereichen, die der jeweiligen Schemaversion zugeordnet sind.

Jede Applikation ist für genau eine Schemaversion entwickelt, über die sie auf die Datenbank zugreift. Daher können verschiedene Applikationen die unterschiedlichen Typen von Attributen oder Klassen nutzen.
Wird eine Schemaänderung durchgeführt, hat dies zumeist auch Auswirkungen auf die vorhandenen Objekte: Sie müssen entsprechend der Struktur der neuen Schemaversion konvertiert werden. Damit liegen sie ebenfalls in mehreren Versionen, den *Objektversionen* vor. Wird einer Klasse beispielsweise ein Attribut hinzugefügt, so müssen von allen Objekten dieser Klasse neue Versionen angelegt werden, die Speicherplatz für den Wert des neuen Attributs haben.

Durch vorhandene Schemaversionen sind Objekte in verschiedenen Typen sicht- und zugreifbar. Eventuell ist sogar die Menge der zugreifbaren Objekte unterschiedlich, da nicht

jedes Objekt in jeder Schemaversion sichtbar sein muss. Daher wird im Folgenden auch vom Zugriffsbereich einer Schemaversion gesprochen; damit ist die Menge aller für Applikationen dieser Schemaversion zugreifbaren Objekte gemeint. Um eine Kooperation zwischen Applikationen verschiedener Schemaversionen zu ermöglichen, ist ein Informationsaustausch zwischen den durch die verschiedenen Schemaversionen sichtbaren Zugriffsbereichen nötig. Diese Weitergabe von Informationen an Objekte anderer Schemaversionen heißt *Propagation*. Wird die Erzeugung oder Modifikation eines Objektes propagiert, so kann eine Umwandlung des Objekttyps entsprechend der Klasse in der Ziel-Schemaversion erforderlich sein. Dies ist eine *Konvertierung*.

Die Propagation muss bei jeder Änderung der Daten durchgeführt werden, und zwar i.A. in beiden Richtungen: d.h. vom Zugriffsbereich der alten Schemaversion zu dem der neuen Schemaversion hin, wie auch umgekehrt. Eine Propagation ist in zwei Fällen erforderlich, und zwar zunächst dann, wenn eine neue Schemaversion abgeleitet wird. In dem Moment muss der Zugriffsbereich dieser neuen Schemaversion mit passenden Versionen der Objekte befüllt werden. Später ergibt sich Propagationsbedarf, wenn eine Applikation ein Objekt im Zugriffsbereich ihrer Schemaversion anlegt, ändert oder löscht. Diese Änderung muss an die Zugriffsbereich der anderen Schemaversionen weitergegeben werden.

Abb. 3.3 zeigt ein versioniertes Schema mit vier Schemaversionen sv_0, sv_1, sv_2 und sv_3, die alle auf derselben Datenbasis arbeiten. Diese Datenbasis enthält die Datenobjekte, die ihrerseits in verschiedenen Objektversionen vorliegen. Die Propagation der Daten von einer in eine andere Objektversion wird für den Anwender transparent durchgeführt, d.h. der Anwender muss sich nicht darum kümmern, mit welcher Schemaversion seine Applikation arbeitet.

Es gibt zwei Arten, die Propagation zu realisieren. Zum einen kann man alle Daten nach einer Änderung durch eine Applikation sofort in alle anderen Objektversionen propagieren bzw. beim Ableiten einer neuen Schemaversion sofort alle Objekte anlegen und durch Konvertierung mit Werten befüllen. Zum anderen kann diese Propagation aber auch erst zum Zeitpunkt des Zugriffs erfolgen, d.h. erst dann, wenn die propagierten Daten wirklich benötigt werden. Die erste Methode wird als *sofortige Propagation* (engl. immediate propagation), die zweite als *verzögerte Propagation* (engl. lazy propagation) bezeichnet.

Die sofortige Propagation hat den Nachteil, dass alle Daten, ungeachtet dessen, ob auf sie in der anderen Schemaversion überhaupt zugegriffen wird, propagiert und gespeichert werden. Abgesehen davon ergibt sich beispielsweise beim Erstellen einer neuen Schemaversion ein Mehraufwand für das Kopieren und Konvertieren der Daten, wenn nicht nur einzelne, sondern alle Daten auf einmal übertragen werden. Dies ist bei der verzögerten Propagation nicht notwendig. Darüber hinaus hat diese zudem noch den Vorteil, dass unnötige Propagationsvorgänge eingespart werden, wenn zwischen zwei Objekt-Änderungen durch Applikationen einer Schemaversion nicht in einer anderen Schemaversion auf das Datenobjekt zugegriffen wurde.

Die verzögerte und sofortige Propagation werden nochmals gesondert in Abschnitt 5.4 behandelt.

Die folgenden fünf Forderungen werden in [Lau96] an ein Datenbanksystem gestellt:

1. Schemaevolution muss unterstützt werden.
2. Schemaversionen müssen unterstützt werden.
3. Schemaevolution soll auch durch das Vorhandensein von Daten nicht verhindert werden, d.h. vorhandene Daten müssen erhalten bleiben und in das neue Format konvertiert werden.

4. Anwendungen, die mit einer alten Schemaversion arbeiten, sollten auch weiterhin verwendet werden können.

5. Anwendungen sollen auch bei Schemaversionierung kooperieren können, d.h. selbst Anwendungen, die auf unterschiedlichen Versionen eines Schemas aufsetzen, sollen mit denselben Daten arbeiten können.

Ein Datenbanksystem, das diese Forderungen erfüllt, ist COAST (Complex Object and Schema Transformation). COAST bietet die Möglichkeit, eine neue Schemaversion aus einer bestehenden Schemaversion abzuleiten und Modifikationen durchzuführen (1.). Die Unterstützung von Schemaversionierung ist damit gegeben (2.). Vorhandene Daten werden von einer Schemaversion in andere Schemaversionen propagiert, damit sind auch bereits in der Datenbank vorhandene Daten in einer neuen Schemaversion weiterverwendbar (3.). Durch die Tatsache, dass die alte Schemaversion beibehalten wird, können Applikationen, die auf diese zugreifen und von den erforderlichen Änderungen nicht berührt werden (weil sie beispielsweise nur auf Klassen zugreifen, die in der alten und der neuen Schemaversion gleich sind), unverändert weiterverwendet werden (4.). Der Propagationsmechanismus ermöglicht die automatische Konvertierung von Daten zwischen zwei Schemaversionen in beiden Richtungen. Damit haben Daten, die durch eine auf die alte Schemaversion zugreifende Applikation geändert wurden, auch in der neuen Schemaversion den geänderten Wert und umgekehrt (5.).

Die Architektur des COAST-Prototypen wird in Abschnitt 7.2 näher beschrieben.

3.3 Zusammenfassung und Bewertung

Dieses Kapitel hat in die Herausforderungen der Schemaevolution eingeführt. Dann wurden die besonderen Anforderungen beschrieben, die sich bei der Änderung des Schemas einer in Benutzung befindlichen Datenbank ergeben. Diese resultieren aus der Existenz gespeicherter Daten und darauf zugreifender Applikationen. Bei einer naiven Schemaänderung des Schemas, das ja genau die Schnittstelle zwischen den Applikationen und der Datenbank darstellt, ergeben sich zu berücksichtigende Konsequenzen auf beiden Seiten.

Die beschriebenen Probleme sind schwierig und bisher nicht ausreichend gelöst. Die Schemaversionierung zeigt sich als vielversprechendster Ansatz, der hier weiter verfolgt werden soll.

Kapitel 4

Schemaänderungen und Konvertierungsfunktionen

Nachdem im vorangegangenen Kapitel auf die Notwendigkeit von Schemaänderungen und die Methoden, diese durchzuführen, eingegangen wurde, wird in diesem Kapitel ein besonderes Augenmerk auf Schemaänderungen an sich gelegt. Ein Ziel dieser Diplomarbeit war, die schon vorhandenen *einfachen* Schemaänderungen um *komplexe* Schemaänderungen zu erweitern.

Die Vorgehensweise bei der Entwicklung ist die, typische Schemaänderungen aufzuführen und zu prüfen, ob sie mit einfachen Schemaänderungsoperationen durchführbar sind. Ist dies nicht der Fall, wird die jeweilige Operation zerlegt und der Teil identifiziert, der nicht mit einfachen Schemaänderungsoperationen zu bewerkstelligen ist. Dabei wird sowohl die Schema- als auch die Objektebene betrachtet. Auf Objektebene wird untersucht, durch welche Konvertierungsfunktionen die Daten am besten, d.h. unter möglichst verlustfreier Einbeziehung aller Daten aus der alten in die jeweilige neue Objektversion, übernommen werden können. Entsprechend wird zu jeder der aufgeführten typischen Schemaänderungen eine Default-Konvertierungsfunktion angegeben, die das System beim Durchführen der entsprechenden Schemaänderung automatisch erzeugt.
Bei der Untersuchung wurde zum einen die Beobachtung gemacht, dass es Schemaänderungen gibt, die nicht aus einfachen Schemaänderungsoperationen aufgebaut werden können. Zum anderen können Schemaänderungen identifiziert werden, bei denen die Erzeugung einer sinnvollen Default-Konvertierungsfunktion nicht ohne Eingriff des Schemaentwicklers möglich ist. Daher werden alle typischen Schemaänderungen in die aus den Beobachtungen resultierenden vier Kategorien eingeteilt.

Abschließend wird auf ein elegantes Konzept zur Durchführung einer speziellen Operation eingegangen, die sich mit den beschriebenen Konvertierungsfunktionen nur umständlich lösen lässt. Dieses Konzept wird in [FMZ+95] eingeführt und Migration genannt.

4.1 Konsequenzen von Schemaänderungen auf Schemaebene

Bei Schemaänderungen werden Schemata beispielsweise durch Einfügen von Attributen in Klassen, Erzeugung neuer Klassen, Änderung von Attributstypen oder auch Löschen von Klassen modifiziert. Im Folgenden werden Grundoperationen zur Durchführung von Schemaänderungen vorgestellt. Man unterscheidet hierbei zwischen einfachen und komplexen Schemaänderungen. Aus diesen Grundoperationen lassen sich andere Schemaänderungsoperationen aufbauen, wie sie beispielsweise in Abschnitt 4.3 beschrieben werden.

4.1.1 Einfache Schemaänderungen

Unter *einfachen* Schemaänderungen versteht man nach [KC88] Änderungen, die entweder die Attribute einer einzelnen Klasse beeinflussen, einen Knoten (Klasse) im Vererbungsgraphen der Klassen erzeugen, löschen oder umbenennen oder eine Kante (Vererbungsbeziehung) im Vererbungsgraphen verändern. Dazu genügen die folgenden Operationen (s. [Dol99]):

S1.1 **Einfügen eines Attributs**
Beim Einfügen eines Attributs in eine Klasse ist zu beachten, dass das Attribut durch die Vererbungsmechanismen auch in allen Unterklassen der Klasse erscheinen wird. Dadurch kann es zu Konflikten kommen, falls dort bereits ein Attribut dieses Namens existiert.

S1.2 **Löschen eines Attributs**
Ebenso wie beim Einfügen eines Attributs wirkt sich auch ein Löschen eines Attributs aus einer Klasse auf alle Unterklassen aus. Sowohl ein geerbtes als auch ein dort lokal redefiniertes Attribut werden mitgelöscht.

S1.3 **Umbenennen eines Attributs**
Eine Attributsumbenennung kann auf Schemaebene prinzipiell auch durch Löschen des ursprünglichen Attributs und Einfügen eines Attributs mit dem neuen Namen geschehen. Allerdings tritt ein Problem auf, wenn die bearbeitete Klasse eine Unterklasse hat, in der ein geerbtes Attribut lokal redefiniert ist. Beim Löschen in der Oberklasse würde die Redefinition des Attributs in der Unterklasse gelöscht und beim Neueinfügen des Attributs nicht wieder hergestellt. Nach dem Neueinfügen hätte man dann in der Unterklasse das Attribut mit dem neu eingefügten Typ modifiziert, obwohl eigentlich nur in der Oberklasse etwas geändert werden sollte.

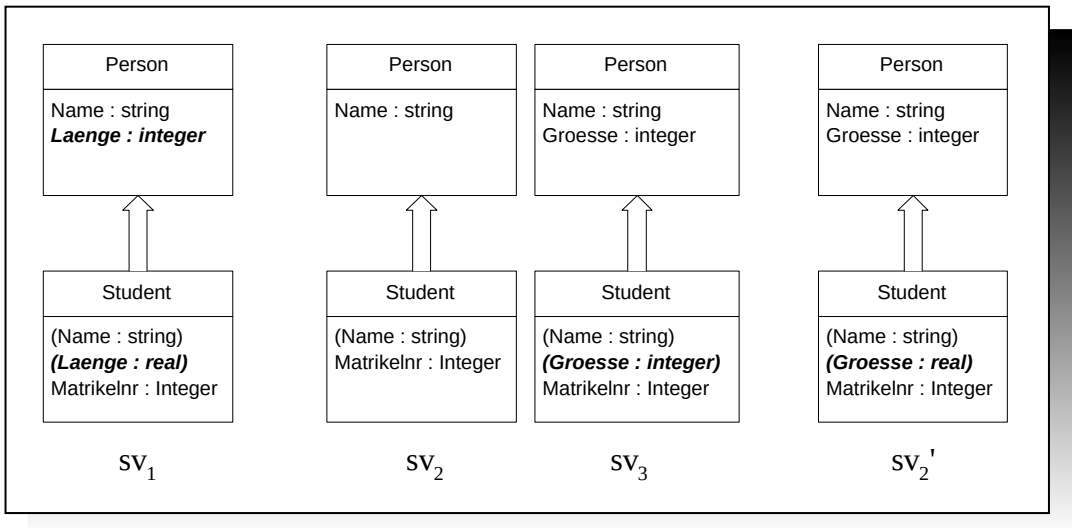

Abbildung 4.1: Umbenennen eines Attributs im Vergleich zum Löschen und Neueinfügen

Beispiel 4.1.1 *Im Beispiel (s. Abb. 4.1) soll in der Klasse „Person“ das Attribut* `Laenge` *aus* sv_1 *in* `Groesse` *umbenannt werden. Der Ansatz, das Attribut erst zu löschen und dann neu einzufügen (*sv_2 *und* sv_3*), erreicht zwar für die Klasse „Person“ den gewünschten Effekt, nicht jedoch für die Unterklasse „Student“ – hier ist*

der Typ des umbenannten Attributs von `real` *nach* `integer` *geändert worden. Besser ist die reine Umbenennung des Attributs (sv'_2), wobei die Typredefinitionen in den Unterklassen erhalten bleiben.*

Durch Löschen und Neuanlegen von Attributen gehen also Redefinitionen dieses Attributs in Unterklassen verloren, die beim Umbenennen jedoch erhalten bleiben müssen. Aus diesem Grund wird eine spezielle Operation benötigt.

S1.4 **Ändern des Typs eines Attributs**
Das Ändern des Typs eines Attributs wirkt sich ebenfalls auf alle Unterklassen aus. Allerdings kann diese Auswirkung dort durch lokale Redefinitionen wieder rückgängig gemacht werden.

S2.1 **Erzeugen einer Klasse**
Eine neu erzeugte Klasse erbt automatisch alle Attribute und Methoden ihrer Oberklassen. Im Folgenden werden jedoch nur die geerbten Attribute betrachtet, da COAST zurzeit keine Methoden unterstützt.

S2.2 **Löschen einer Klasse**
Im COAST-Projekt ist das Löschen von Klassen aus der „Mitte“ einer Vererbungshierarchie, d.h. das Löschen von Klassen, die noch Unterklassen haben, nicht erlaubt. In diesem Falle würden aus Konsistenzgründen (s. [Dol99]) automatisch alle Unterklassen entfernt werden. Daher bezieht sich die Operation „Löschen einer Klasse“ nur auf das Löschen von Klassen ohne Unterklassen.

S2.3 **Umbenennen einer Klasse**
Das Umbenennen einer Klasse ist als eine Operation aufgeführt, weil nicht gewährleistet werden kann, dass nach dem Löschen und Neueinfügen einer Klasse alle Nachfolgerklassen dasselbe Aussehen haben wie nach einer Umbenennung:

Beispiel 4.1.2 *Beim Löschen aus der „Mitte“ einer Vererbungsstruktur können eventuelle Redefinitionen verloren gehen. Dies wird in Abb. 4.2 klar: Versucht man, die Klasse B in B' umzubenennen, indem man zunächst die Klasse B löscht (sv_2), wird C eine direkte Unterklasse von A. Es existiert dann nach dem Neueinfügen von B' in sv_3 keine Vererbungsbeziehung zwischen B' und C mehr. Eventuelle Redefinitionen von Attributen in B gehen bei dieser Methode verloren.*
Der einzig mögliche Weg ist die direkte Umbenennung der Klasse (sv'_2).

In COAST ist ein Löschen von Klassen aus einer Vererbungshierarchie ohnehin nicht möglich, da durch den Schemamanager automatisch auch alle Unterklassen mitgelöscht würden, um die Konsistenz des Schemas zu erhalten (s.[Dol99]). So ist die Umbenennung in COAST der einzige Weg, um den gewünschten Effekt zu erzielen.

Wie bei Attributen ist es auch bei Klassen nicht möglich, eine Umbenennung durch Löschen und neu Anlegen mit dem veränderten Namen zu erreichen. In beiden Fällen liegt dies an korrigierenden Maßnahmen (s. [Dol99]), die beim Löschen ergriffen werden, um die Konsistenz des Schemas zu erhalten und die beim Anlegen nicht wiederhergestellt werden können. Daher sind Schemaänderungsoperationen zum Umbenennen von Attributen und Klassen notwendig.

4.1.2 Komplexe Schemaänderungen

Von *komplexen* Schemaänderungen spricht man, wenn mehr als nur eine Quellklasse involviert ist. Typische Beispiele für komplexe Schemaänderungen sind das Verschieben von

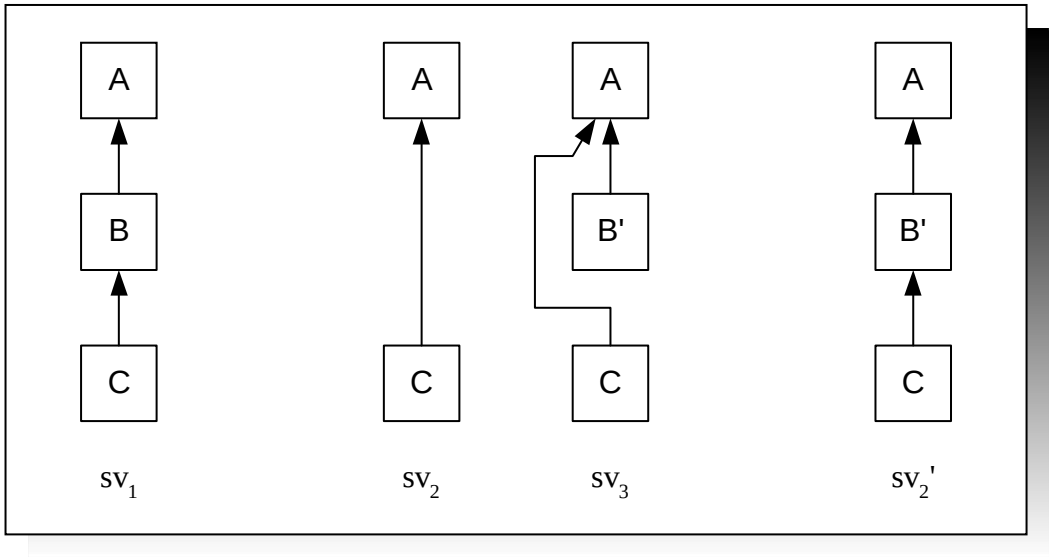

Abbildung 4.2: Umbenennen einer Klasse im Vergleich zum Löschen und Neueinfügen

Attributen in Unterklassen oder auch die Erzeugung neuer Klassen, die dann in einer anderen Klasse referenziert werden.

Hier sind die folgenden Operationen nötig:

S3.1 **Kopieren von Attributen von einer Klasse in eine andere Klasse**
Ein Attribut liegt nach dem Kopieren sowohl in der Quell- wie auch in der Zielklasse vor. Dies ist mit den einfachen Schemaänderungsoperationen nicht möglich, da diese immer nur zwischen zwei Versionen derselben Klasse agieren können.

S3.2 **Erstellen von Referenzen auf eine andere Klasse**
Eine Referenz ist ein Attribut, das die Objekt-ID des referenzierten Objektes enthält.

S3.3 **Hinzufügen von Vererbungskanten**
Beim Hinzufügen von Vererbungskanten zwischen zwei Klassen ist zu beachten, dass dadurch alle Attribute (und Methoden) der Oberklasse auch in jeder Unterklasse auftauchen. Sollten bereits gleichnamige Attribute vorhanden sein, gibt es doppelte Attribute, die durch eine automatische Umbenennung wieder eindeutig gemacht werden müssen.

S3.4 **Löschen von Vererbungskanten**
Durch das Löschen einer Vererbungskante werden die Attribute (und Methoden) der Oberklasse aus der Unterklasse und aller von ihr abgeleiteten Klassen entfernt.

4.2 Konsequenzen von Schemaänderungen auf Objektebene

Schemaänderungen müssen auf Objektebene gesondert betrachtet werden. Jede Schemaänderung hat Auswirkungen auf die Objekte der Datenbank. Es sind neue Objektversionen zu erzeugen, die die Schemaänderung nachbilden.

Wenn beispielsweise durch eine Operation `add attribute PLZ:string` ein Attribut hinzugefügt wird, soll in die neue Klassenversion zusätzlich zu den schon vorhandenen Attributen ein neues Attribut `PLZ` vom Typ `string` eingetragen werden. Es würde wenig

Sinn machen, wenn durch diese Schemaänderung sämtliche zu dieser Klasse gehörenden Datenobjekte verworfen würden. Vielmehr sollen die schon vorhandenen Informationen übertragen und nur das neue Attribut mit einem leeren oder sogar vordefinierten Startwert belegt werden.

Beim Löschen oder Konvertieren eines Datenobjekts in eine neue Schemaversion benötigt das System eine Vorschrift, was genau mit den Daten zu geschehen hat, also wie die neue Objektversion mit Daten zu befüllen ist, ausgehend von der existierenden Objektversion. Eine solche Vorschrift heisst in dieser Diplomarbeit *Konvertierungsfunktion*.

Konvertierungsfunktionen werden vom Schemaentwickler im Zuge der Schemaänderung in der in Abschnitt 6.3 und 6.4 entwickelten Syntax angegeben. Das System kann in vielen Fällen eine Default-Konvertierungsfunktion vorgeben, die dem Schemaentwickler Schreibarbeit spart, indem sie für jede Konvertierung eine Standardvorgabe macht. Diese kann dann vom Schemaentwickler angepasst werden.

4.2.1 Konvertierungsfunktionen

Die Propagation von Zustandsänderungen zwischen verschiedenen Objektversionen spezifiziert man mit Hilfe soganannter *Konvertierungsfunktionen*.

Eine Konvertierungsfunktion cf ist eine klassenspezifische Funktion, die als Parameter das Ursprungsobjekt ov_i in der Schemaversion sv_i erhält und ein neues Objekt ov_j in der Schemaversion sv_j erzeugt:

$$ov_j = cf_{j \leftarrow i}(ov_i)$$

Sei die Schemaversion sv_j von sv_i abgeleitet, d.h. sie befindet sich im Ableitungsgraphen unter der Schemaversion sv_i. Eine Konvertierung in der Richtung von sv_i zu sv_j nennt man *Vorwärtskonvertierungsfunktion* ($fcf_{j \leftarrow i}$), eine Konvertierung in der umgekehrten Richtung von sv_j nach sv_i nennt man *Rückwärtskonvertierungsfunktion* ($bcf_{j \leftarrow i}$).

4.2.1.1 Default-Konvertierungsfunktionen

Um dem Schemaentwickler bei der Durchführung von Schemaänderungen möglichst viel Arbeit abzunehmen, wäre es sehr hilfreich, wenn bei Schemaänderungen die Konvertierung der vorhandenen Daten weitestgehend automatisiert ablaufen könnte, ohne dass der Schemaentwickler genauere Konvertierungsanweisungen geben muss.

Konkret gibt es viele sehr einfache Konvertierungen, allen vorweg die Konvertierung von Objekten und Attributen, die sich im Vergleich zur Vorgängerschemaversion nicht geändert haben. Für diese muss aber trotz allem eine Konvertierungsfunktion an das System übergeben werden – in diesem Falle eben die Identitätsfunktion, die den alten Wert in das neue Objekt übernimmt. Genau hier ist also ein großes Potenzial, dem Benutzer viel Schreibarbeit abzunehmen – er kann sich dann auf die tatsächlichen Änderungen konzentrieren.

Beim Anlegen von Attributen sind Default-Konvertierungsfunktionen ebenfalls hilfreich: Neu angelegte Attribute können automatisch mit einem definierten Startwert belegt werden. Mehr noch, das System kann sogar bei den geänderten Attributen sinnvolle Vorgaben liefern, beispielsweise bei Umbenennungen oder einfachen Typumwandlungen. So kann sich der Schemaentwickler beispielsweise die Angabe einer Konvertierungsfunktion für die Konvertierung eines Attributs vom Typ `integer` in den Typ `string` sparen – das System liefert bereits eine fertige Konvertierungsfunktion für diesen Fall, die dann vom Schemaentwickler übernommen oder notfalls editiert werden kann. Diese automatisch vorgegebenen Konvertierungsanleitungen nennt man *Default-Konvertierungsfunktionen*.

Default-Konvertierungsfunktionen dienen also dem Ziel der Einsparung von Spezifikationsaufwand. Sie werden ebenfalls textuell generiert, um dem Benutzer in übersichtlicher Form die Möglichkeit zu Änderungen zu geben.

```
CREATE SCHEMA Autohaendler {
     SCHEMAVERSION ErsteVersion {
               CLASS Auto {
                        ATTRIBUTES {
                                Marke: STRING;
                                Farbe: STRING;
                                PS: integer;
                                kW: integer;
                        };
               };
     };
};

MODIFY SCHEMA Autohaendler {
        CREATE SCHEMAVERSION NeueVersion {
                INTEGRATE CLASS Auto FROM ErsteVersion

                MODIFY CLASS Auto {
                        ATTRIBUTES {
                        DELETE PS;
                       }
                };
        };
};
```

Abbildung 4.3: Beispiel in ODL-Syntax

Sei folgendes Beispiel gegeben:

Beispiel 4.2.1 *Es liege eine Schemaversion „ErsteVersion“ mit einer Klasse „Autohändler“ mit den vier Attributen* `Marke`*,* `Farbe`*,* `PS` *und* `kW` *vor. Der Benutzer erstellt eine neue Schemaversion „NeueVersion“, in der die Klasse „Autohändler“ ebenfalls existieren soll und alle Attribute aus „ErsteVersion“ außer dem Attribut* `PS` *existieren sollen (s. Abb. 4.3 in ODL[1]-Syntax).*

Dem Anwender kann viel Arbeit abgenommen werden, indem die unveränderten Attribute automatisch unverändert übernommen werden. Dem Anwender steht natürlich weiterhin frei, andere Konvertierungsfunktionen anzugeben.
Das System würde im Beispiel die Attribute `Marke`*,* `Farbe` *und* `kW` *als default einfach unverändert übernehmen. Es erkennt, dass die Attribute* `Marke`*,* `Farbe` *und* `kW` *keine Änderungen gegenüber der ersten Schemaversion erfahren haben und kann damit die Werte der Attribute übertragen.*
Falls der Typ der beiden Attribute identisch ist, wird einfach der Wert aus der alten Objektversion in die neue Objektversion kopiert. Ansonsten wird je nach beteiligten Typen eine Konvertierungsfunktion vorgeschlagen, die der Schemaentwickler ggf. abändern kann.

In diesem Beispiel wird also die folgende Konvertierungsfunktion[2] generiert (die alte Objektversion sei als `old`*, die neue Objektversion als* `new` *im System bekannt):*

[1] Object Definition Language, s. [Her99]
[2] Die Syntax einer Konvertierungsfunktion wird in den Abschnitten 6.3 und 6.4 entwickelt.

```
new.Marke = old.Marke
new.Farbe = old.Farbe
new.kW = old.kW
```

4.2.1.2 Einfache Konvertierungsfunktionen

In dieser Diplomarbeit werden einfache und komplexe Konvertierungsfunktionen unterschieden. *Einfache Konvertierungsfunktionen* sind Konvertierungsfunktionen, die zu einfachen Schemaänderungen gehören. Daher kommen sie zum Einsatz, wenn eine Objektversion mit Werten aus einer anderen Objektversion desselben Objekts gefüllt wird. In vielen Fällen ist eine solche Konvertierung auch semantisch einfach: Wenn beispielsweise nur ein Attribut hinzugefügt worden ist, können alle bereits in der Quellobjektversion existierenden Attribute mit der Identitätsfunktion übernommen werden, das neue Attribut bekommt einen Standardvorgabewert. Es steht dem Schemaentwickler allerdings frei, andere Konvertierungen vorzugeben.

Zu diesen semantisch einfachen Fällen gehören im Einzelnen:

- **gleichbleibende Attribute**
  Attribute, die sich im Vergleich zum Ursprungsobjekt nicht verändert haben, also weder eine Umbenennung noch eine Typänderung erfahren haben, können einfach identisch übernommen werden.

- **einfache Typänderungen**
  Zu einfachen Typänderungen gehören Typänderungen, die vom System durch bereits vorhandene interne Funktionen zu realisieren sind, also beispielsweise bei einer Konvertierung vom Typ `char` zum Typ `int`.

  Kay Wölfle stellt in [Wöl98] eine Tabelle von Typkonvertierungen vor. Darin wird eingeordnet, welche Typkonvertierungen automatisch vom System vorgenommen werden können und bei welchen der Schemaentwickler eingreifen muss, um die gewünschte Konvertierung zu spezifizieren. Es wird sowohl auf einfache Typen (`int`, `string`, `date`, `char`, `real`) wie auch auf komplexe Typen (`set`, `list`, `array`, `tuple`, `class_ref`) eingegangen.

  Bei komplexeren Konvertierungen von einfachen Typen muss der Schemaentwickler eine spezielle Konvertierungsfunktion entwerfen. Bei verlustbehafteten Konvertierungen wie vom Typ `string` zum Typ `char` kann beipielsweise nur ein Zeichen des alten Strings übernommen werden. Ob dieses Zeichen nun das erste oder auch ein zufälliges Zeichen des Strings sein soll oder vielleicht eine Kodierung, die die 8 Bits des Datentyps `char` ausnutzt, kann das System nicht von alleine erkennen.
  Auch bei der Konvertierung von komplexen Typen ist ein Eingriff des Schemaentwicklers nötig. Wenn beispielsweise vom Typ `list(int)` in den Typ `int` konvertiert werden soll, kann die Summe der Integerwerte aus der Liste als Zielwert gedacht sein, möglicherweise aber auch das erste oder letzte Element der Liste oder der Durchschnitt.

- **Attributsumbenennungen**
  Wenn das System erkennt, dass ein Attribut lediglich umbenannt wird, kann der bisherige Inhalt unverändert übernommen werden.

- **Neu angelegte Attribute**
  Hierfür können Initialwerte vergeben werden, beispielsweise 0 für Integerwerte oder

eine leere Zeichenkette für Strings. Dadurch ist sichergestellt, dass in jedem neuen Attribut ein definierter Wert steht.

4.2.1.3 Komplexe Konvertierungsfunktionen

Der Begriff *komplexe Konvertierungsfunktionen* wird in dieser Diplomarbeit für Konvertierungsfunktionen verwendet, die zu komplexen Schemaänderungen gehören. Dies ist dann der Fall, wenn mehr als eine neue und eine alte Objektversion involviert sind, beispielsweise beim Kopieren eines Attributs in eine andere Klasse (s. Abschnitt 4.3.3.1 unter „Typische Schemaänderungen“), wobei trotzdem der Inhalt des Attributs propagiert werden soll.

In diesem Falle wird es für das System schwieriger, eine Default-Konvertierungsfunktion anzubieten. Daher wird im Normalfall nur der einfache Teil der Konvertierung als Default-Konvertierungsfunktion, für die nicht offensichtlichen Konvertierungen aber nur ein Defaultwert erzeugt, der dann vom Schemaentwickler angepasst werden muss.

4.2.2 Konsequenzen von Schemaänderungen für die Objektebene

Schemaänderungen haben Auswirkungen auf die Objektebene. Die entsprechenden Objektversionen müssen analog zur entsprechenden Schemaversion angepasst werden. Die zur Umsetzung nötigen Operationen auf Objektebene werden ebenfalls in einfache und komplexe Schemaänderungen unterteilt.

4.2.2.1 Einfache Schemaänderungen

Unter *einfachen* Schemaänderungen versteht man auch auf der Objektebene Änderungen, die nur eine einzelne Klasse beeinflussen, sich also nur auf die Attribute innerhalb dieser einen bestimmten Klasse beziehen[3].

Die Konvertierung von Attributen von einer Quellobjektversion zu einer Zielobjektversion des gleichen Objekts ist mit den folgenden Operationen oder Kombinationen daraus möglich:

O1.1 **Füllen eines Attributs mit einem konstanten Wert**
Der Wert eines Attributs wird auf einen festen Vorgabewert gesetzt, der seinem Typ entspricht. Dazu gehören Nullwerte wie 0, NULL oder der Leerstring „“ ebenso wie Vorgaben, dass z.B. bei jedem neuen Mitarbeiter automatisch in einem Attribut `Wohnort` der Wert „Frankfurt“ einzutragen ist.

O1.2 **Füllen eines Attributs mit einem Wert aus einer anderen Objektversion**
Ein Attribut wird mit dem Wert aus einer anderen Objektversion des gleichen Objekts oder eines anderen Objekts gefüllt.

O1.3 **Berechnung eines Wertes im Objekt**
Ein Attribut wird mit einem berechneten Wert gefüllt, dazu gehören beispielsweise:

- Umrechnungen (DM in Euro, Meilen in Kilometer, etc.).
- Summenbildungen wie beispielsweise ein Attribut `Jahresgehalt`, das die Summe der Attribute `Monatsgehalt1` bis `Monatsgehalt12` enthält.

[3] Prinzipiell müssten auch Methoden betrachtet werden, diese werden allerdings von COAST (noch) nicht unterstützt.

– Differenzbildungen wie ein Attribut `Alter`, das aus der Differenz aus dem Geburtsdatum und der Schemaableitungszeit ermittelt wird[4].

Bei Typänderungen muss auf den Quell- und den Zieltyp geachtet werden. So kann es beispielsweise vorkommen, dass aus einem Attribut des Typs `integer` ein Attribut des Typs `set(string)` werden soll. Es ist also nicht nur eine Umrechnung auf den neuen einfachen Typ zu erledigen, sondern in bestimmten Fällen sogar aus einem einzelnen Typ ein komplexer Typ oder umgekehrt zu erzeugen. Ein komplexer Typ ist mit Typkonstruktoren wie `set` (Menge), `list` (Liste), `tuple` (Datensatz), `array` (Tabelle), `graph` (Graph) etc. aus einfachen Typen aufgebaut.

4.2.2.2 Komplexe Schemaänderungen

Wenn mehr als eine Klasse beeinflusst wird, handelt es sich um eine *komplexe* Schemaänderung. Im Gegensatz zur Schemaebene, wo komplexe Schemaänderungen durch Hintereinanderausführung mehrerer einfacher Schemaänderungen simuliert werden können, ist es auf Objektebene nötig, spezielle Operationen einzuführen, um Mehrdeutigkeiten zu vermeiden.
Letztlich kann auf Schemaebene jede Schemaänderung durch einfaches Löschen aller Klassen der Quellschemaversion und Einfügen der Klassen der Zielschemaversion realisiert werden.
In diesem Falle hätte das System aber keinerlei Information über die Entstehung der neuen Schemaversion und könnte keine Beziehungen zwischen einzelnen Attributen in den Schemaversionen herstellen. Auf Objektebene hieße das, dass sämtliche Attribute aller Objektversionen der neuen Schemaversion mit NULL-Werten belegt würden, weil alle Attribute neu erzeugt wurden.

Für komplexe Schemaänderungen wird also die Menge der nötigen Operationen erweitert.

Komplexe Schemaänderungen können auf Objektebene durch Aneinanderreihung mehrerer der folgenden und der vorangegangenen (einfachen) Operationen vollzogen werden.

O2.1 **Selektieren eines Objekts über eine vorhandene Referenz**
Das Selektieren eines Objekts über eine vorhandene Referenz ist genau genommen keine Operation. Diese Operation wurde trotzdem aufgeführt, um Unterschiede bei Schemaänderungsoperationen verdeutlichen zu können, die über Wertevergleiche oder über Referenzen auf andere Objekte zugreifen.

O2.2 **Selektieren eines Objekts mittels Suche über Wertevergleich**
Die Suche erfolgt über Vergleiche von Attributen, beispielsweise wenn zu einem vorhandenen Objekt ein weiteres Objekt gefunden soll, das den gleichen Wert im Attribut `Name` hat.

O2.3 **Setzen einer Referenz auf ein anderes Objekt**
Eine Referenz auf ein Objekt wird in einem Attribut eines anderen Objekts gespeichert.

[4]Es darf keine Differenzbildung mit der aktuellen Systemzeit erfolgen, da durch die verzögerte Propagation nicht geklärt ist, wann der Wert tatsächlich konvertiert wird. Sonst könnte es bei diesem Beispiel sein, dass die verzögerte Propagation der Änderung des Geburtsdatums erst ein Jahr später erfolgt - und damit für das Alter ein anderes Ergebnis liefern würde als die sofortige Propagation. S. auch den Konzeptionsgrundsatz K10 in Abschnitt 6.2.

O2.4 **Erzeugen eines neuen Objekts**
Es kann nötig sein, auf Objektebene neue Objekte zu erzeugen. Dies ist vor allem dann der Fall, wenn beispielsweise durch eine Schemaänderungsoperation wie das „Aufbrechen von Klassen“ (s. Abschnitt 4.3.3.3) aus einer Klasse zwei Klassen entstehen. Bei dieser Operation werden einzelne Attribute in ein neu anzulegendes Komponentenobjekt ausgelagert und eine Referenz darauf einfügt.

O2.5 **Löschen eines Objekts**
Im Fall, dass durch eine Schemaänderungsoperation eine Objektversion nicht mehr benötigt wird, muss diese auch gelöscht werden können.

4.3 Typische Schemaänderungen

Ziel dieser Diplomarbeit war auch, die Propagation in COAST um komplexe Schemaänderungen zu erweitern. Die Vorgehensweise bei der Entwicklung war die, typische Schemaänderungen zusammenzustellen und zu prüfen, ob sie mit den schon vorhandenen (also den *einfachen*) Schemaänderungsoperationen durchführbar sind. Des Weiteren war ein Ziel, zu jeder dieser Operationen auch eine Default-Konvertierungsfunktion angeben zu können, die das System automatisch generiert.

In diesem Abschnitt werden also Schemaänderungen vorgestellt, die für den Betrieb einer Datenbank typisch sind. Diese sind in vier Kategorien unterteilt:

1. Schemaänderungen, die mit einfachen Schemaänderungsoperationen und Default-Konvertierungsfunktionen auskommen

2. Schemaänderungen, die mit einfachen Schemaänderungsoperationen auskommen, bei denen aber benutzerdefinierte Konvertierungsfunktionen notwendig sind

3. Schemaänderungen, die komplexe Schemaänderungsoperationen benötigen, bei denen ansonsten aber Default-Konvertierungsfunktionen ausreichend sind

4. Schemaänderungen, die komplexe Schemaänderungsoperationen und benutzerdefinierte Konvertierungsfunktionen benötigen

Bei einer größeren Umstellung des Schemas kann es sein, dass mehrere der folgenden Schemaänderungen für die komplette Schemaänderung nötig sind. Die Einordnung in die vier Gruppen stimmt für die einzelnen Teile der kompletten Schemaänderung immer noch, allerdings ist nur dann kein Eingriff des Schemaentwicklers in die Konvertierungsfunktionen notwendig, wenn alle Schemaänderungen, die durchgeführt werden, aus den Kategorien 1 und 3 stammen.

In den folgenden Abschnitten 4.3.1 bis 4.3.4 wird für jede der beschriebenen Schemaänderungen ein Beispielfall, eine Liste von notwendigen Operationen auf Schemaebene, eine Liste von notwendigen Operationen auf Objektebene, die Syntax des neuen Befehls und eine Default-Konvertierungsfunktion in der in dieser Diplomarbeit entwickelten Syntax für Konvertierungsfunktionen (s. Abschnitt 6.6) angegeben.
Dabei kann eine stilisierte SQL-Abfrage (sql_query(„...“)) auftauchen. Hier wäre jeweils die entsprechende Abfrage einzusetzen, um aus den Suchbegriffen, die sich in den entsprechenden Quellattributen befinden, die Zielklasse zu finden. Das System kann diese SQL-Abfrage automatisch generieren, da ihm alle nötigen Informationen bekannt sind, die Ausformulierung der Abfragen sind im Folgenden aber der Übersichtlichkeit halber und

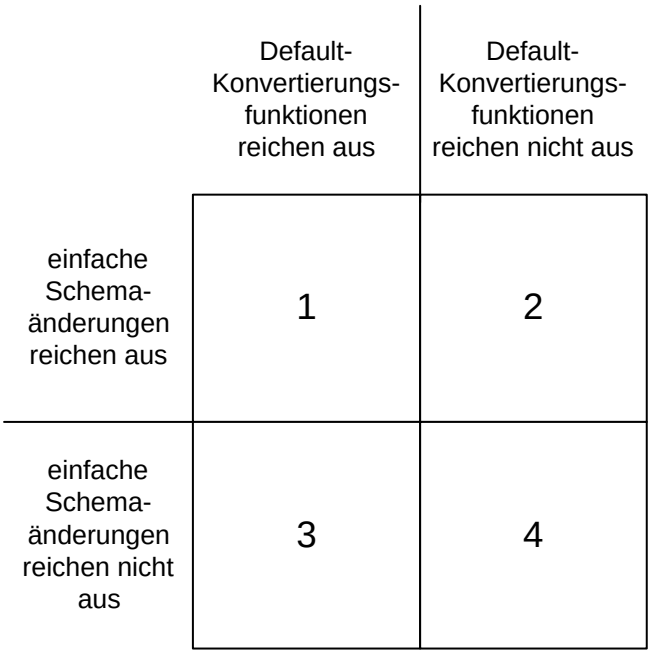

Abbildung 4.4: Schemaänderungen können in vier Kategorien eingeordnet werden

weil sie für das Verständnis keinen weiteren Wert haben, dementsprechend vereinfacht worden.

Bei den komplexen Schemaänderungsoperationen wird vereinfachend immer der Fall angenommen, dass nur zwei Quell- oder Zielklassen involviert sind. Die Operationen können jedoch prinzipiell auf mehr als zwei Quell- oder Zielklassen erweitert werden, indem die entsprechende Operation mehrfach hintereinander ausgeführt oder die Funktionalität entsprechend ausgebaut wird.

4.3.1 Einfache Schemaänderungsoperationen, bei denen Default-Konvertierungsfunktionen ausreichen

In diesem Abschnitt werden typische Schemaänderungen aufgeführt, die sich mit den bereits vorhandenen einfachen Schemaänderungsoperationen bewerkstelligen lassen, und bei denen auch ohne Eingriff des Schemaentwicklers eine sinnvolle Default-Konvertierungsfunktion generiert werden kann.

Die aufgeführten Schemaänderungen sind das Umbenennen von Klassen und das Löschen von Attributen.

4.3.1.1 Umbenennen von Klassen

Wenn eine Klasse umbenannt wird, sollten natürlich ihre Objekte unverändert erhalten bleiben, d.h. die Attribute müssen nach der Umbenennung der Klasse noch dieselben Werte haben.

Beispielsituation: siehe Abb. 4.5

Vorgehensweise:

- auf Schemaebene
  Umbenennen der Klasse (Operation S2.3):

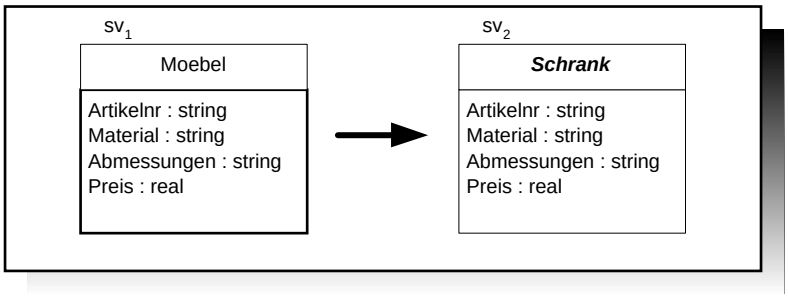

Abbildung 4.5: Beispiel: Umbenennen von Klassen

```
RENAME CLASS Moebel TO Schrank
```

- auf Objektebene
  Es ist keine weitere Nachbehandlung notwendig.

Syntax:
`RENAME CLASS` Klassenname `TO` Neuer_Klassenname

Syntax für dieses Beispiel:
`RENAME CLASS Moebel TO Schrank`

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf)
```
new.Artikelnr = old.Artikelnr
new.Material = old.Material
new.Abmessungen = old.Abmessungen
new.Preis = old.Preis
```

- Rückwärtskonvertierungsfunktion (bcf)
```
new.Artikelnr = old.Artikelnr
new.Material = old.Material
new.Abmessungen = old.Abmessungen
new.Preis = old.Preis
```

4.3.1.2 Löschen von Attributen

Das Löschen von Attributen ist wie das Umbenennen von Klassen ein Sonderfall, in dem auf Objektebene kein weiterer Handlungsbedarf besteht. Diese Operation wird aber trotzdem aus Vollständigkeitsgründen hier mit aufgenommen.

Beispielsituation: siehe Abb. 4.6

Vorgehensweise:

- auf Schemaebene
  Löschen des Attributs (Operation S1.2):
  `DELETE ATTRIBUTE Religion`

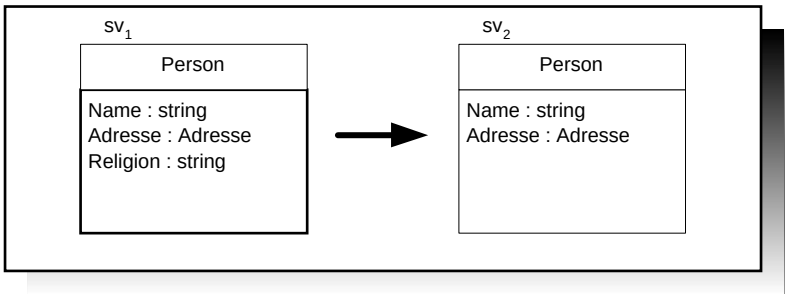

Abbildung 4.6: Beispiel: Löschen von Attributen

- auf Objektebene
  Es ist keine weitere Nachbehandlung notwendig.

Syntax:
`DELETE ATTRIBUTE` Attributname

Syntax für dieses Beispiel:
`DELETE ATTRIBUTE Religion`

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf)
  ```
  new.Name = old.Name
  new.Adresse = old.Adresse
  ```

- Rückwärtskonvertierungsfunktion (bcf)
  ```
  new.Name = old.Name
  new.Adresse = old.Adresse
  new.Religion = „\
  ```

4.3.2 Einfache Schemaänderungsoperationen, bei denen Default-Konvertierungsfunktionen nicht ausreichen

In diesem Abschnitt werden wie im vorigen typische Schemaänderungen aufgeführt, die sich mit den bereits vorhandenen einfachen Schemaänderungsoperationen bewerkstelligen lassen. Allerdings ist keine Erzeugung einer sinnvollen Default-Konvertierungsfunktion möglich, weil Fragen auftreten, die nur durch den Schemaentwickler geklärt werden können. Trotzdem werden Default-Konvertierungsfunktionen erstellt, die dann aber vom Schemaentwickler angepasst werden müssen.

Die aufgeführten Schemaänderungen sind das Aufsplitten von Attributen, das Verbinden von Attributen, das Einfügen redundanter Attribute, das Einfügen von Attributen und Belegung mit Startwerten und Typänderungen.

4.3.2.1 Aufsplitten von Attributen

Eine weitere in der Praxis erfahrungsgemäß häufig vorkommende Schemaänderung ist das Aufsplitten von Attributen, d.h. die Aufteilung der Information aus einem Attribut in

mehrere Attribute. Das Aufsplitten von Attributen ist hier nur mit Attributen vom Typ `string` möglich.
Bei dieser Operation ist ein Benutzereingriff notwendig, um die Position, an dem der Attributsinhalt getrennt werden soll, zu bestimmen.

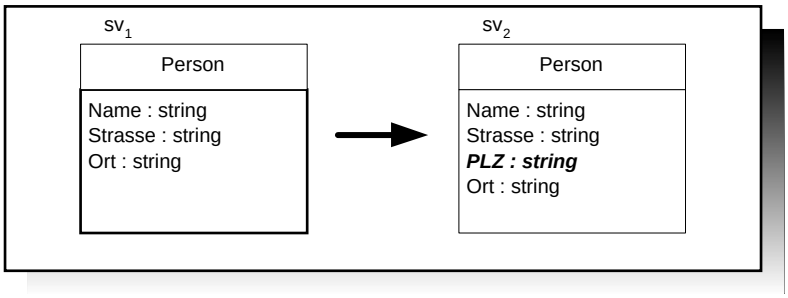

Abbildung 4.7: Beispiel: Aufsplitten von Attributen

Beispielsituation: siehe Abb. 4.7

Vorgehensweise:

- auf Schemaebene
  Einfügen des neuen Attributs (Operation S1.1):
  `CREATE ATTRIBUTE PLZ:string`

- auf Objektebene
  Füllen des neuen Attributs (Operation O1.3):
  `new.PLZ = left(old.Ort,5)`
  Ändern des Inhalts des alten Attributs (Operation O1.3):
  `new.Ort = right(old.Ort,length(old.Ort)-5)`

Syntax:
`SPLIT ATTRIBUTE` Attribut `TO` Attribut_links`,`Attribut_rechts

Die Vorgaben, an welcher Stelle und nach welchem System der String getrennt werden soll, tauchen nur in der Konvertierungsfunktion auf.

Syntax für dieses Beispiel:
`SPLIT ATTRIBUTE Ort TO PLZ,Ort`

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf)
  ```
  new.Name = old.Name
  new.Strasse = old.Strasse
  new.PLZ = left(old.Ort,length(old.Ort/2))
  new.Ort = right(old.Ort,length(old.Ort/2))
  ```
- Rückwärtskonvertierungsfunktion (bcf)
  ```
  new.Name = old.Name
  new.Strasse = old.Strasse
  new.Ort = concat(old.PLZ,old.Ort)
  ```

Das System kann an dieser Stelle nicht wissen, was der Schemaentwickler für eine Trennung im Sinn hat, daher wird in der Default-Konvertierungsfunktion eine Trennung in der Mitte

des Strings vorgegeben. Der Schemaentwickler würde jetzt sinnvollerweise eingreifen und die beiden Zeilen
`new.PLZ = left(old.Ort,length(old.Ort/2))` und
`right(old.Ort,length(old.Ort)/2)`
abändern in
`new.PLZ = left(old.Ort,5)` und
`right(old.Ort,length(old.Ort)-5)`,
wenn er die Trennung aus dem Beispiel (5. Stelle des Strings als Trennstelle) wünscht.

Denkbar wäre beispielsweise auch eine Trennung beim ersten Leerzeichen oder Komma oder an jeder beliebigen anderen Stelle.

Die Funktionen `right` und `left` stellen hier Funktionen dar, die den linken Teil des Strings Ort (also die Postleitzahl) und den rechten Teil (also den Ortsnamen) extrahieren sollen. Die Funktion `concat` fügt zwei Strings zu einem String zusammen, `length` ermittelt die Länge eines Strings.

4.3.2.2 Verbinden von Attributen

Die Umkehrung zur Operation „Aufsplitten von Attributen“ ist ebenfalls möglich.
Bei dieser Operation ist für die Vorwärtskonvertierungsfunktion kein Benutzereingriff notwendig, da die beiden Quellattribute (die beide vom Typ `string` sein müssen) einfach aneinandergehängt werden können. Letztlich wäre es aber hilfreich, wenn der Benutzer beispielsweise zusätzlich angibt, ob die beiden Strings direkt verbunden werden sollen, ein Trennzeichen zwischen den beiden Strings eingefügt werden soll oder vielleicht sogar eine komplexere Kombination gewünscht wird. Dies kann der Benutzer in der Konvertierungsfunktion angeben. Für die Rückwärtskonvertierungsfunktion muss der Benutzer analog zur im vorangegangenen Abschnitt 4.3.2.1 beschriebenen Operation angeben, wie er die Trennung des Strings wünscht.

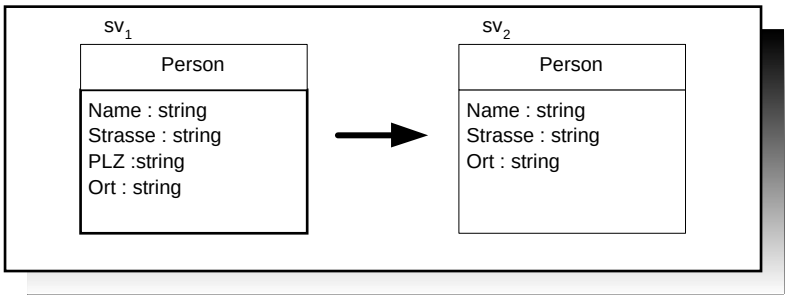

Abbildung 4.8: Beispiel: Verbinden von Attributen

Beispielsituation: siehe Abb. 4.8

Vorgehensweise:

- auf Schemaebene
  Löschen des überflüssigen Attributs (Operation S1.2):
  `DELETE ATTRIBUTE PLZ`

- auf Objektebene
  Ersetzen des Inhalts des Zielattributs (Operation O1.3):
  `new.Ort = concat(old.PLZ,old.Ort)`

Syntax:
`CONCAT ATTRIBUTE` Attribut_links`,`Attribut_rechts `TO` Zielattribut

Syntax für dieses Beispiel:
`CONCAT ATTRIBUTE PLZ,Ort TO Ort`

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf)
```
new.Name = old.Name
new.Strasse = old.Strasse
new.Ort = concat (old.PLZ, old.Ort)
```
- Rückwärtskonvertierungsfunktion (bcf)
```
new.Name = old.Name
new.Strasse = old.Strasse
new.PLZ = left(old.Ort,length(old.Ort)/2)
new.Ort = right(old.Ort,length(old.Ort)/2)
```

Wie zuvor bei der Operation „Aufsplitten von Attributen“ kann das System hier nicht wissen, an welcher Stelle die Teilung des Strings in der Rückwärtskonvertierungsfunktion erfolgen soll. In der Default-Konvertierungsfunktion wird daher als Vorgabewert die Teilung bei der halben Länge des Strings eingetragen. Der Schemaentwickler muss nun eingreifen und seine persönlichen Vorstellungen eintragen, indem er die beiden Zeilen entsprechend abändert.

4.3.2.3 Einfügen redundanter Attribute

Das Einfügen redundanter Attribute kann beispielsweise dann interessant werden, wenn Attribute, deren Werte aus dem Inhalt anderer Attribute ermittelt werden können, die Übersichtlichkeit erhöhen oder durch einfache Berechnungen dem Benutzer zusätzliche Information geboten werden kann.
Typische Beispiele wären ein Feld „Alter“, das sich aus der Differenz zwischen der Schemaableitungszeit und dem Geburtsdatum ermitteln ließe oder ein Feld, das den Preis eines Produktes zusätzlich in einer anderen Währung (z.B. in Euro) angibt.
Es ist zwar theoretisch möglich, dem Benutzer auch komplexe Berechnungen zu ermöglichen, im Normalfall reichen aber die vier Grundrechenarten aus.

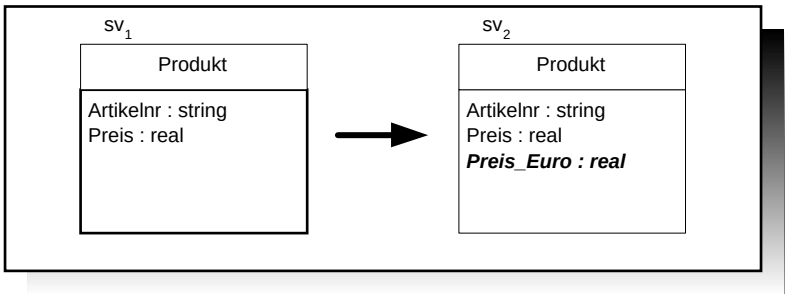

Abbildung 4.9: Beispiel: Einfügen redundanter Attribute

Beispielsituation: siehe Abb. 4.9

Vorgehensweise:

- auf Schemaebene
  Einfügen des neuen Attributs (Operation S1.1):
  `CREATE ATTRIBUTE Preis_Euro:real`

- auf Objektebene
  Füllen des neuen Attributs (Operation O1.3):
  `new.Preis_Euro = old.Preis * Eurokonstante`

Syntax:
`CREATE ATTRIBUTE` Attribut`:`Typ

Syntax für dieses Beispiel:
`CREATE ATTRIBUTE Preis_Euro:real`

Die Konvertierungsfunktion enthält genauere Informationen darüber, wie das neue Attribut zu befüllen ist, also beispielsweise eine Berechnungsvorschrift.

Standardmäßig weist die Default-Konvertierungsfunktion dem neuen Attribut einen NULL-Wert zu. Dieser ist dann vom Schemaentwickler entsprechend anzupassen. Die Berechnungsformel muss dabei eine gültige Formel aus Attributen (vom Typ `integer` oder `real`), Konstanten und den Symbolen +, -, * und / sein. Es obliegt dem Schemaentwickler, zu gewährleisten, dass nur Werte aus dem Definitionsbereich des Typs des Attributs zugewiesen werden können.

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf)
  `new.Artikelnr = old.Artikelnr`
  `new.Preis = old.Preis`
  `new.Preis_Euro = 0`

- Rückwärtskonvertierungsfunktion (bcf)
  `new.Artikelnr = old.Artikelnr`
  `new.Preis = old.Preis`

4.3.2.4 Einfügen von Attributen und Belegung mit Startwerten

Ist ein neues Attribut einzufügen, für das es nicht möglich ist, den Wert aus einem anderen Attribut oder aus der Vorgängerobjektversion zu kopieren oder berechnen, kann auch ein Wert vorgegeben werden.

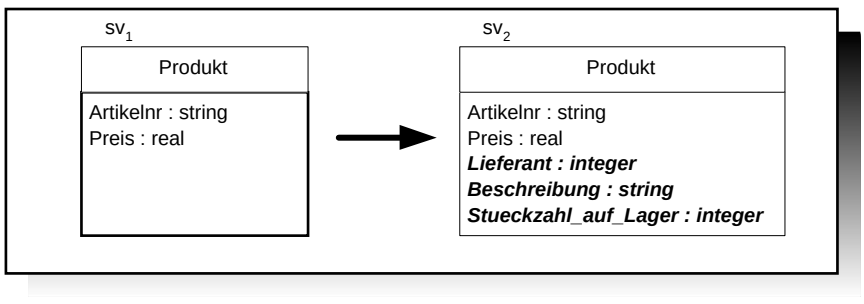

Abbildung 4.10: Beispiel: Einfügen von Attributen und Belegung mit Startwerten

Beispielsituation: siehe Abb. 4.10

Vorgehensweise:

- auf Schemaebene
  Einfügen des neuen Attributs (Operation S1.1):
  `CREATE ATTRIBUTE Lieferant:integer`
  `CREATE ATTRIBUTE Beschreibung:string`
  `CREATE ATTRIBUTE Stueckzahl_auf_Lager:integer`
- auf Objektebene
  Füllen der Attribute mit Konstanten (Operation O1.1):
  `new.Lieferant = 37`
  `new.Beschreibung = „Bleistift\`
  `new.Stueckzahl_auf_Lager = 1000`

Syntax:
`CREATE ATTRIBUTE` Attribut:Typ

Syntax für dieses Beispiel:
```
CREATE ATTRIBUTE Lieferant:integer
CREATE ATTRIBUTE Beschreibung:string
CREATE ATTRIBUTE Stueckzahl_auf_Lager:integer
```

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf)
  ```
  new.Artikelnr = old.Artikelnr
  new.Preis = old.Preis
  new.Lieferant = 0
  new.Beschreibung = „\
  new.Stueckzahl_auf_Lager = 0
  ```
- Rückwärtskonvertierungsfunktion (bcf)
  ```
  new.Artikelnr = old.Artikelnr
  new.Preis = old.Preis
  ```

Da das System nicht wissen kann, welche Vorgabewerte der Schemaentwickler wünscht, gibt es in der Default-Konvertierungsfunktion NULL-Werte vor. Der Schemaentwickler muss also an dieser Stelle eingreifen und die gewünschten Startwerte eintragen.

4.3.2.5 Typänderungen

Die Änderung des Typs eines Attributs gehört auch zu den häufiger benötigten Schemaänderungen. Hier ist zu beachten, dass es Quelltyp-Zieltyp-Paare gibt, die nur eine informationsverlustbehaftete oder gar keine Konvertierung zulassen.
Es kann auch Typänderungen geben, die von einfachen (atomaren) Typen auf komplexe Typen (Mengen-Typen, beispielsweise `list(integer)` oder `set(integer)`) wechseln, oder umgekehrt. Insofern ist es nötig, jede mögliche Kombination von Quell- und Zieltyp genauer zu untersuchen. Diese Untersuchung wurde in [Wöl98, S. 45ff] durchgeführt.

Beispielsituation: siehe Abb. 4.11

Vorgehensweise:

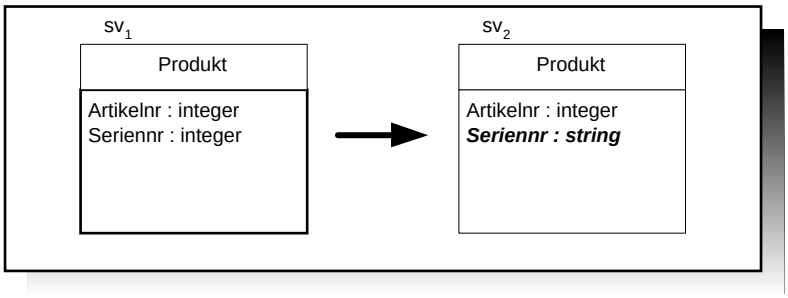

Abbildung 4.11: Beispiel: Typänderungen

- auf Schemaebene
  Ändern des Attributtyps (Operation S1.4):
  `RETYPE ATTRIBUTE Seriennr:string`

- auf Objektebene
  Berechnen des neuen Attributinhalts (Operation O1.3):
  `new.Seriennr = conv(old.Seriennr)`

Hierbei ist anzumerken dass „`conv`“ nur als Platzhalter für die entsprechende zu verwendende Typkonvertierung zu verstehen ist. Im Beispiel wird die C-Funktion „`itoa`“ verwendet, weil vom Typ `integer` zum Typ `string` konvertiert werden soll.

Syntax:
`RETYPE ATTRIBUTE` Attribut `TO` Typ

Syntax für dieses Beispiel:
`RETYPE ATTRIBUTE Seriennr TO string`

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf)
  `new.Artikelnr = old.Artikelnr`
  `new.Seriennr = itoa(old.Seriennr)`

- Rückwärtskonvertierungsfunktion (bcf)
  `new.Artikelnr = old.Artikelnr`
  `new.Seriennr = 0`

Da dem System nicht bekannt ist, wie die gewünschte Konvertierung aussieht, kann es in der Default-Konvertierungsfunktion nur Vorgaben eintragen. Diese Vorgaben können bei einfachen Konvertierungen (s. [Wöl98]) gleich die entsprechende Konvertierung (hier im Beispiel von `integer` nach `string` in der Vorwärtskonvertierungsfunktion) oder sonst ein NULL-Wert (hier im Beispiel von `string` nach `integer` in der Rückwärtskonvertierungsfunktion) sein.

4.3.3 Komplexe Schemaänderungsoperationen, bei denen Default-Konvertierungsfunktionen ausreichen

Im Gegensatz zu den in den beiden vorhergehenden Abschnitten vorgestellten typischen Schemaänderungen reichen bei den folgenden die *einfachen* Schemaänderungen zur Durch-

führung nicht mehr aus. Stattdessen ist die Verwendung von komplexen Schemaänderungsoperationen notwendig. Trotzdem ist die Erstellung von sinnvollen Default-Konvertierungsfunktionen ohne Eingriff des Schemaentwicklers möglich.

Die aufgeführten Schemaänderungen sind das Kopieren von Attributen über Referenzen, das Einfügen von Referenzen, das Aufbrechen von Klassen, das Zusammenführen von Klassen über Referenzen und über Wertevergleiche, das Verschieben von Attributen entlang der Aggregationskanten und die Umkehrung der Aggregationsreihenfolge. Dabei wird beim Aufbrechen von Klassen besonders auf zwei verschiedene Möglichkeiten zur Referenzierung von Komponentenobjekten eingegangen.

4.3.3.1 Kopieren von Attributen über Referenzen

Daten werden von einem durch eine Referenz zu selektierenden Objekt in das referenzierende Objekt kopiert.

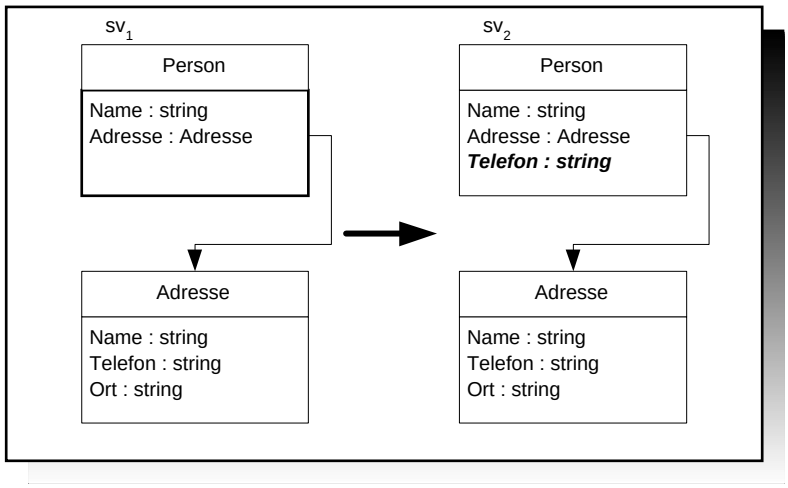

Abbildung 4.12: Beispiel: Kopieren von Attributen über Referenzen

Beispielsituation: siehe Abb. 4.12

Vorgehensweise:

- auf Schemaebene
  Kopieren eines Attributs (Operation S3.1):
  `COPY ATTRIBUTE Telefon FROM CLASS Adresse` (in der Umgebung Person)

- auf Objektebene
  Selektieren des zugehörigen Objektes (Operation O2.1):
  `_tmp_Adresse = deref(old.Person.Adresse)`[5]
  Überschreiben des neuen Attributs durch den Wert aus dem gefundenen Objekt (Operation O1.2):
  `new.Person.Telefon = _tmp_Adresse.Telefon`

Syntax:
`COPY ATTRIBUTE` Attribut `FROM CLASS` Quellklasse

[5]Die Operation `deref` gibt die Referenz auf ein Objekt zurück. Dazu wird ein Attribut, das diese Referenz enthält, ausgelesen.

Syntax für dieses Beispiel:
`COPY ATTRIBUTE Telefon FROM CLASS Adresse` (in der Umgebung Person)

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Person
  ```
  _tmp_Adresse = deref(old.Person.Adresse)
  new.Person.Name = old.Person.Name
  new.Person.Telefon = _tmp_Adresse.Telefon
  ```
- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Adresse
  ```
  new.Adresse.Name = old.Adresse.Name
  new.Adresse.Telefon = old.Adresse.Telefon
  new.Adresse.Ort = old.Adresse.Ort
  ```
- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Person
  ```
  new.Person.Name = old.Person.Name
  ```
- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Adresse
  ```
  new.Adresse.Name = old.Adresse.Name
  new.Adresse.Telefon = old.Adresse.Telefon
  new.Adresse.Ort = old.Adresse.Ort
  ```

4.3.3.2 Einfügen von Referenzen

Im Prinzip wird durch das Einfügen von Referenzen aus einer einfachen Verkettung der Objekte (über die vorhandene Referenz im ersten Objekt, die auf das zweite Objekt zeigt) eine doppelte Verkettung gemacht. Dazu wird im zweiten Objekt zusätzlich eine Referenz auf das erste Objekt eingefügt. Voraussetzung dafür ist, dass zwischen den beiden Klassen eine 1:1-Beziehung gilt. Für das folgende Beispiel bedeutet das, dass jeweils nur ein Arbeiter- mit einem Vereinsmitglied-Objekt in Beziehung steht. Andernfalls müsste man statt der einen neuen Referenz eine Liste (oder Menge) von Referenzen einfügen.
Barbara Staudt Lerner [Ler96] nennt diese Schemaänderung „Link Addition“.

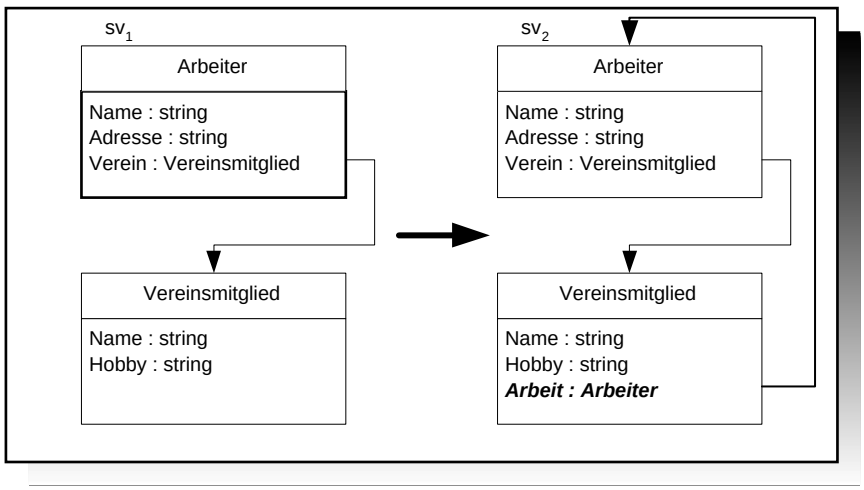

Abbildung 4.13: Beispiel: Einfügen von Referenzen

Beispielsituation: siehe Abb. 4.13

Vorgehensweise:

- auf Schemaebene
  Einfügen einer Referenz (Operation S3.2):
  `CREATE REFERENCE Arbeit:Arbeiter` (in der Umgebung Vereinsmitglied)
- auf Objektebene
  Selektieren des Unterobjektes (Operation O2.1):
  `_tmp_Vereinsmitglied = deref(old.Arbeiter.Verein)`
  Setzen der neuen Referenz (Operation O2.3):
  `new.Vereinsmitglied.Arbeit = ref(new.Arbeiter)`

Syntax:
`CREATE REFERENCE` Attribut`:` Zielklasse

Syntax für dieses Beispiel:
`CREATE REFERENCE Arbeit:Arbeiter`

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Arbeiter
  `new.Arbeiter.Name = old.Arbeiter.Name`
  `new.Arbeiter.Adresse = old.Arbeiter.Adresse`
  `new.Arbeiter.Verein = old.Arbeiter.Verein`
- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Vereinsmitglied
  `new.Vereinsmitglied.Name = old.Vereinsmitglied.Name`
  `new.Vereinsmitglied.Hobby = old.Vereinsmitglied.Hobby`
  `new.Vereinsmitglied.Arbeit = ref(new.Arbeiter)`
- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Arbeiter
  `new.Arbeiter.Name = old.Arbeiter.Name`
  `new.Arbeiter.Adresse = old.Arbeiter.Adresse`
  `new.Arbeiter.Verein = old.Arbeiter.Verein`
- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Vereinsmitglied
  `new.Vereinsmitglied.Name = old.Vereinsmitglied.Name`
  `new.Vereinsmitglied.Hobby = old.Vereinsmitglied.Hobby`

4.3.3.3 Aufbrechen von Klassen

Es wird ein neues Objekt samt einer Referenz darauf erzeugt und ein oder mehrere Attribute in dieses neue Objekt kopiert, anschließend werden die Quellattribute gelöscht.

Diese Schemaänderungsoperation kann beispielsweise auch dann benötigt werden, wenn eine Klasse „Auto“ vorliegt, die ein Attribut `Foto` enthält, in dem ein Foto (in guter Qualität und daher mit entsprechender Byteanzahl) abgelegt ist. In diesem Falle mag es sinnvoll erscheinen, das Attribut `Foto` in eine Komponentenklasse auszulagern, damit beim häufigen Zugriff auf die Klasse „Auto“ nicht immer auch das Foto mitgeladen werden muss, sondern nur dann ein Zugriff darauf erfolgt, wenn das Foto auch explizit gewünscht wird.

Beispielsituation: siehe Abb. 4.14

Vorgehensweise:

- auf Schemaebene Erzeugen einer neuen Klasse (Operation S2.1):
  `CREATE CLASS Adresse`

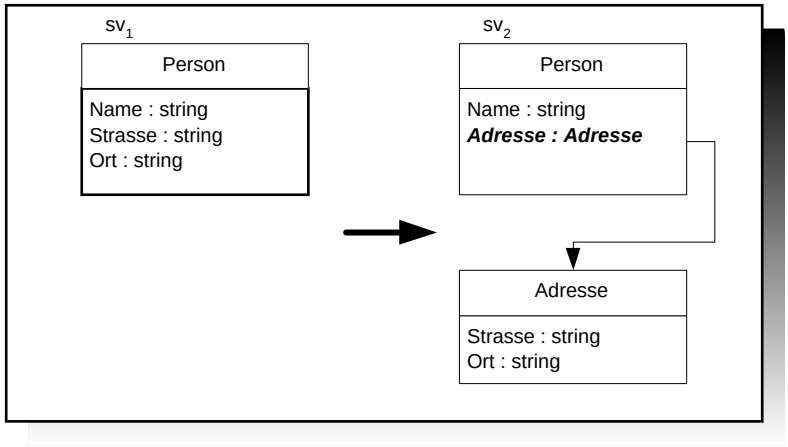

Abbildung 4.14: Beispiel: Aufbrechen von Klassen

Erzeugen der benötigten Referenz (Operation S3.2):
`CREATE REFERENCE Adresse:Adresse` (in der Umgebung Person)
Kopieren der benötigten Attribute in die neue Klasse (Operation S3.1):
`COPY ATTRIBUTE Strasse FROM CLASS Person` (in der Umgebung Adresse)
`COPY ATTRIBUTE Ort FROM CLASS Person` (in der Umgebung Adresse)
Löschen der nicht mehr benötigten Attribute in der Quellklasse (Operation S1.2):
`DELETE ATTRIBUTE Strasse` (in der Umgebung Adresse)
`DELETE ATTRIBUTE Ort` (in der Umgebung Adresse)

- auf Objektebene
  Erzeugen des neuen Objekts und setzen der Referenz (Operation O2.4 und O2.3):
  `new.Person.Adresse = ref(create object Adresse)`
  Überschreiben der neuen Attribute durch den Wert aus „Person“ (Operation O1.2):
  `new.Adresse.Strasse = old.Person.Strasse`
  `new.Adresse.Ort = old.Person.Ort`

Syntax:
`CREATE CLASS` Neue_Klasse `AS REFERENCE` Attribut `IN` Klasse `WITH (`a_1, ..., a_n`)`, wobei a_1 bis a_n die zu verschiebenden Attribute sind.

Syntax für dieses Beispiel:
`CREATE CLASS Adresse AS REFERENCE Adresse IN Person WITH (Strasse, Ort)`

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Person
  `new.Person.Name = old.Name`
  `new.Person.Adresse = ref(create object Adresse)`
- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Adresse
  `_tmp_Person = sql_query(„...\)`
  `new.Adresse.Strasse = _tmp_Person.Strasse`
  `new.Adresse.Ort = _tmp_Person.Ort`
- Rückwärtskonvertierungsfunktion (bcf)
  `_tmp_Adresse = old.Person.Adresse`
  `new.Name = old.Person.Name`
  `new.Strasse = _tmp_Adresse.Strasse`
  `new.Ort = _tmp_Adresse.Ort`

Einschub:

Bei der Operation „Aufbrechen von Klassen“ tritt das Problem auf, dass das System zwei Möglichkeiten hat, die referenzierten Objekte anzulegen (s. Abb. 4.15).

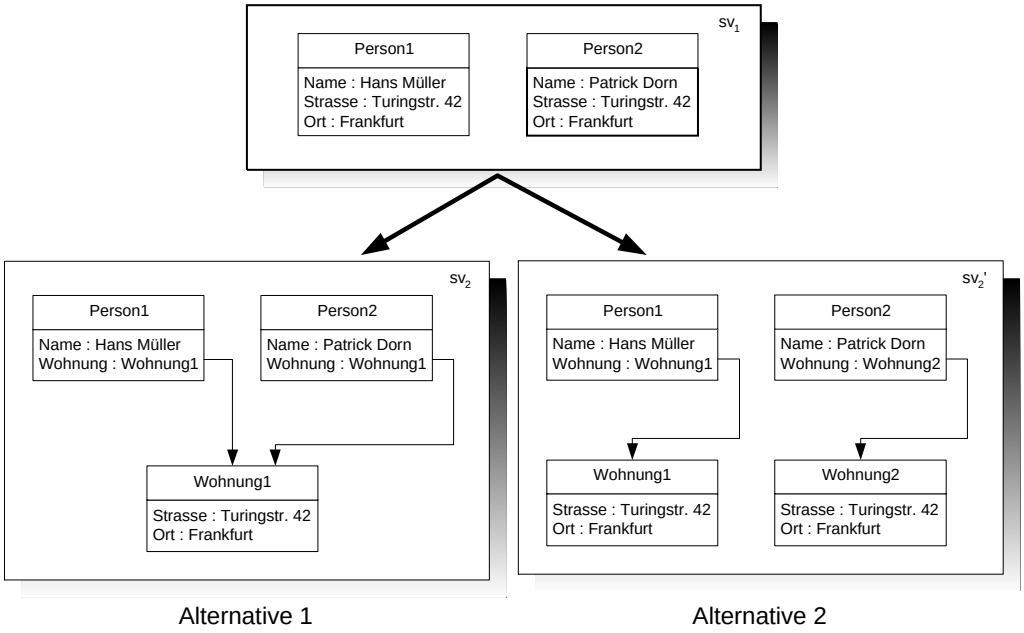

Abbildung 4.15: Zwei Alternativen für die Referenzierung

Alternative 1
Jedes der referenzierenden Objekte kann ein gemeinsames Objekt referenzieren, das nur einmal vorhanden ist. In diesem Falle spricht man von **grouping** ([GLG95]).

Alternative 2
Jedes der referenzierenden Objekte kann ein „eigenes“ referenziertes Objekt haben, ungeachtet der Tatsache, dass der Inhalt dieser referenzierten Objekte gleich ist.

Der Vorteil der ersten gegenüber der zweiten Alternative liegt auf der Hand: Es kann Platz eingespart werden, da die Speicherung redundanter Informationen vermieden wird. Allerdings erkauft man sich dafür ein anderes Problem, und zwar die Gefahr von Zeigern, die auf nicht mehr existente Objekte zeigen (engl. dangling pointer). Sollte im ersten Fall das Objekt „Person1“ gelöscht werden, würde automatisch das Adress-Objekt mitgelöscht, und das Objekt „Person2“ zeigte mit seiner Referenz auf ein nicht mehr vorhandenes Adress-Objekt.
Dieses Problem kann beispielsweise durch einen Zähler, der angibt, wieviele Referenzen auf das gemeinsam referenzierte Objekt zeigen, gelöst werden. In dem Falle wird bei der Erzeugung jeder Referenz, die auf das gemeinsam referenzierte Objekt zeigt, der Zähler erhöht und bei jeder Löschung einer solchen Referenz der Zähler erniedrigt. Erst bei Erreichen eines Zählerstandes von 0 darf das gemeinsame Objekt tatsächlich gelöscht werden – allerdings bedingt diese Methode damit auch erhöhten Verwaltungsaufwand.

Eine Implementierung kann folgendermaßen aussehen: Jedes Objekt erhält intern den genannten Zähler. Die Operation `create object` prüft auf die Existenz eines Objektes mit genau den gewünschen Eigenschaften und erhöht, falls es ein solches Objekt gibt, den Zähler dieses Objektes und gibt eine Referenz darauf zurück. Falls das Objekt nicht existiert, wird es angelegt, der Zähler auf 1 gesetzt und ebenfalls eine Referenz darauf zurückgegeben.
Die Operation `delete object` erniedrigt den Zähler um 1 und prüft, ob der Zählerwert

0 ist. Falls dies zutrifft, wird das Objekt tatsächlich gelöscht. In beiden Fällen wird eine erfolgreiche Löschung zurückgemeldet.

Eine Variante hiervon ist, dass ein Objekt auch mit einem Zählerwert von 0 nicht gelöscht wird, beispielsweise, wenn eine Adresse darin abgelegt ist und zu erwarten ist, dass das Objekt demnächst wieder referenziert werden wird. Hierbei ist allerdings nicht gewährleistet, dass ein Objekt, das gelöscht werden soll, auch wirklich nach dem Löschen nicht mehr existiert. Will also beispielsweise ein Benutzer Objekte löschen, um Speicherplatz zu sparen, dann kann es passieren, dass zwar alle Löschungen erfolgreich sind, physikalisch aber kein Platz freigeworden ist. Hierfür benötigt man eine zusätzliche explizite Löschoperation, etwa ein `delete! object`, das den Zähler auf 0 setzt und dann das Objekt löscht. Die Gefahr von Dangling Pointern ist damit natürlich wieder gegeben.
Der Datenbankenhersteller O2 hat einen ähnlichen Ansatz verfolgt. Nach wiederholten Nachfragen von Kunden bot er aber zum impliziten Löschoperator noch einen expliziten Löschoperator an (s. [O2 96]).

Weiter treten bei der Zähler-Variante Schwierigkeiten auf, wenn einer der Datensätze geändert werden soll. Ein Beispiel: Wenn Alternative 1 gewählt wurde – man also tatsächlich dasselbe Objekt referenziert – und sich die Adresse von Person1 geändert hat. Es wird notwendig, ein neues Adress-Objekt für das Objekt „Person1“ anzulegen, sonst würde die Adresse von Person2 auch verändert.
Damit muss Mehraufwand in Kauf genommen werden: Es ist nach einem Adress-Objekt zu suchen, das die neue Adresse enthält. Ist diese Suche erfolglos, muss ein neues Adress-Objekt angelegt werden und die neuen Adressdaten eingetragen werden. Außerdem ist die Referenz im Person-Objekt von Hans Müller auf das neue Adress-Objekt zu setzen.

Die zweite Alternative, bei der alle Unterobjekte redundant angelegt werden, ist einfach und unabhängig von den obigen Überlegungen. Allerdings lässt sich damit keine Platzersparnis erzielen und es ist auch keine Fehlervermeidung in Hinsicht auf Dangling Pointer möglich. Die Vermeidung solcher Probleme obliegt dem Datenbankenentwickler.

Für COAST wurde in dieser Diplomarbeit die zweite Alternative realisiert, um nicht nur eine Teilmenge der Möglichkeiten anderer Datenbankensysteme anzubieten. Gerade die Erfahrung, dass die Kunden auf eine – wenn auch gefährliche – explizite Löschoperation bestehen, führte zu dieser Entscheidung.

(Ende des Einschubs)

4.3.3.4 Zusammenführen von Klassen über Referenzen

Das Zusammenführen von Attributen ist gewissermaßen die Umkehrung der Operation „Aufbrechen von Klassen“. Es werden Attribute von der referenzierten Klasse in die referenzierende Klasse kopiert und dann die referenzierte Klasse gelöscht.
Barbara Staudt Lerner [Ler96] nennt diese Schemaänderung „Inlining“.

Beispielsituation: siehe Abb. 4.16

Vorgehensweise:

- auf Schemaebene
  Kopieren der neuen Attribute (Operation S3.1):
  `COPY ATTRIBUTE Strasse FROM CLASS Adresse` (in der Umgebung Person)
  `COPY ATTRIBUTE Ort FROM CLASS Adresse` (in der Umgebung Person)
  Löschen des überflüssigen Attributs (Operation S1.2):

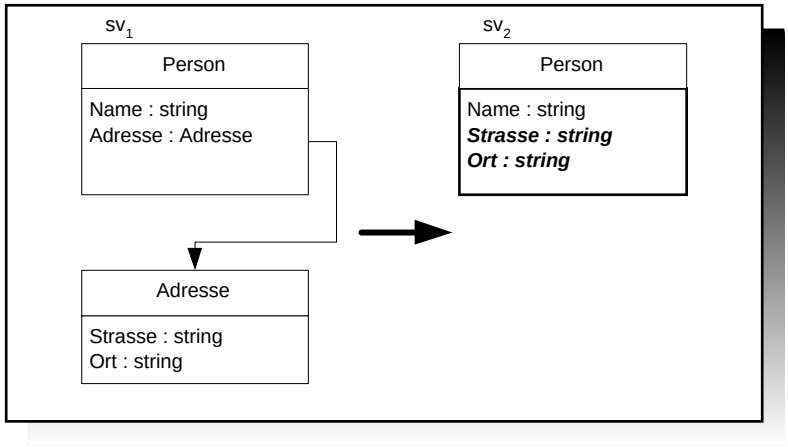

Abbildung 4.16: Beispiel: Zusammenführen von Klassen über Referenzen

`DELETE ATTRIBUTE Adresse` (in der Umgebung Person)
Löschen der überflüssigen Klasse (Operation S2.2):
`DELETE CLASS Adresse`

- auf Objektebene
  Selektieren des Unterobjektes (Operation O2.1):
  `_tmp_Adresse = deref(old.Adresse)`
  Überschreiben der neuen Attribute durch den Wert aus „Adresse“ (Operation O1.2):
  `new.Strasse = _tmp_Adresse.Strasse`
  `new.Ort = _tmp_Adresse.Ort`

Syntax:
`INLINE CLASS` Quellklasse1 `USING REFERENCE` Attribut

Syntax für dieses Beispiel:
`INLINE CLASS Person USING REFERENCE Adresse`

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf)
  ```
  _tmp_Adresse = deref(old.Person.Adresse)
  new.Name = old.Person.Name
  new.Strasse = _tmp_Adresse.Strasse
  new.Ort = _tmp_Adresse.Ort
  ```

- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Person
  ```
  _tmp_Adresse = sql_query(„...\)
  new.Person.Name = old.Name
  new.Person.Adresse = ref(_tmp_Adresse)
  ```

- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Adresse
  ```
  _tmp_Person = sql_query(„...\)
  new.Adresse.Strasse = _tmp_Person.Strasse
  new.Adresse.Ort = _tmp_Person.Ort
  ```

4.3.3.5 Zusammenführen von Klassen über Wertevergleiche

Beim „Zusammenführen von Klassen über Wertevergleiche“ werden wie beim „Zusammenführen von Klassen über Referenzen“ Attribute aus zwei Objekten in eines zusammengeführt. Allerdings existiert hier keine Referenz vom ersten auf das zweite Attribut, die man für den Zugriff auf das zweite Objekt nutzen könnte. Die Entscheidung, dass beide Objekte zusammengehören, erfolgt über den Vergleich eines oder mehrerer Attribute. Barbara Staudt Lerner [Ler96] nennt diese Schemaänderung „Merge“.

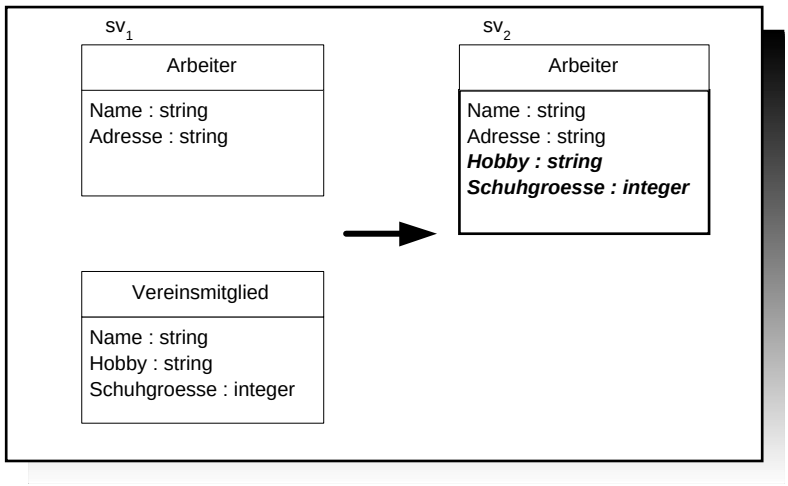

Abbildung 4.17: Beispiel: Zusammenführen von Klassen über Wertevergleiche

Beispielsituation: siehe Abb. 4.17

Vorgehensweise:

- auf Schemaebene
  Kopieren der gewünschten Attribute (Operation S3.1):
  `COPY ATTRIBUTE Hobby FROM CLASS Vereinsmitglied` (in der Umgebung Arbeiter)
  `COPY ATTRIBUTE Schuhgroesse FROM CLASS Vereinsmitglied` (in der Umgebung Arbeiter)
  Löschen der überflüssigen Klasse (Operation S2.2):
  `DELETE CLASS Vereinsmitglied`
- auf Objektebene
  Suchen des Unterobjektes (Operation O2.2):
  `_tmp_Vereinsmitglied = sql_query(„...\)`
  Überschreiben der neuen Attribute durch den Wert aus „Vereinsmitglied“ (Operation O1.2):
  `new.Hobby = _tmp_Vereinsmitglied.Hobby`
  `new.Schuhgroesse = _tmp_Vereinsmitglied.Schuhgroesse`

Syntax:
`INLINE CLASS` Quellklasse1`,`Quellklasse2 `TO` Zielklasse `USING (`$a_1 = b_1$, ... , $a_n = b_n$`)`, wobei a_1 bis a_n Attribute aus Quellklasse1 und b_1 bis b_n Attribute aus Quellklasse2 sind, die paarweise verglichen werden sollen.

Syntax für dieses Beispiel:

```
INLINE CLASS Arbeiter,Vereinsmitglied TO Arbeiter
USING (Arbeiter.Name=Vereinsmitglied.Name)
```

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf)
  ```
  _tmp_Vereinsmitglied = sql_query(„...\)
  new.Name = old.Name
  new.Adresse = old.Adresse
  new.Hobby = _tmp_Vereinsmitglied.Hobby
  new.Schuhgroesse = _tmp_Vereinsmitglied.Schuhgroesse
  ```
- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Arbeiter
  ```
  new.Name = old.Name
  new.Adresse = old.Adresse
  ```
- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Vereinsmitglied
  ```
  _tmp_Arbeiter = sql_query(„...\)
  new.Name = _tmp_Arbeiter.Name
  new.Hobby = _tmp_Arbeiter.Hobby
  new.Schuhgroesse = _tmp_Arbeiter.Schuhgroesse
  ```

4.3.3.6 Verschieben von Attributen entlang der Aggregationskanten

Hier wird anhand einer Referenz ein Objekt identifiziert und ein Attribut aus diesem referenzierten Objekt in das referenzierende Objekt – oder umgekehrt – kopiert und anschließend in der Quellklasse gelöscht.
Man kann diese Schemaänderung durch die Operation „Kopieren von Attributen“ mit anschließendem Löschen des überflüssigen Attributs im Unterobjekt durchführen.
Barbara Staudt Lerner [Ler96] nennt diese Schemaänderung „Moving using a structural relationship“.

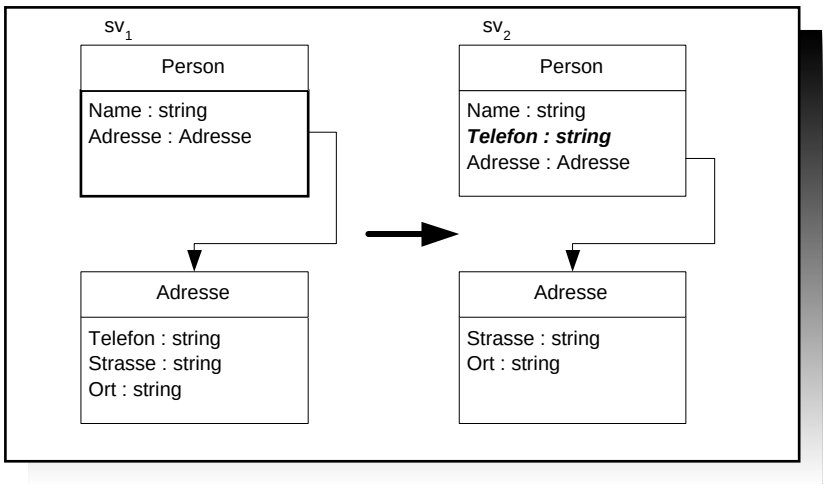

Abbildung 4.18: Beispiel: Verschieben von Attributen entlang der Aggregationskanten

Beispielsituation: siehe Abb. 4.18

Vorgehensweise:

- auf Schemaebene
  Kopieren des Attributs (Operation S3.1):
  `COPY ATTRIBUTE Telefon FROM CLASS Adresse` (in der Umgebung Person)

Löschen des überflüssigen Attributs (Operation S1.2).
`DELETE ATTRIBUTE Telefon` (in der Umgebung Adresse)

- auf Objektebene
  Überschreiben des neuen Attributs durch den Wert aus „Adresse“ (Operation O1.2):
  `new.Person.Telefon = old.Adresse.Telefon`

Syntax:
`MOVE ATTRIBUTE` Attribut `IN` Quellklasse `TO` Zielklasse `USING REFERENCE` Referenzattribut `IN` Zielklasse

Syntax für dieses Beispiel:

```
MOVE ATTRIBUTE Telefon IN Adresse TO Person USING REFERENCE
Adresse IN Person
```

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Person

```
_tmp_Adresse = old.Person.Adresse
new.Person.Name = old.Person.Name
new.Person.Telefon = _tmp_Adresse.Telefon
new.Person.Adresse = old.Person.Adresse
```

- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Adresse

```
new.Adresse.Strasse = old.Adresse.Strasse
new.Adresse.Ort = old.Adresse.Ort
```

- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Person

```
new.Person.Name = old.Person.Name
new.Person.Adresse = old.Person.Adresse
```

- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Adresse

```
_tmp_Person = sql_query(„...\)
new.Adresse.Telefon = _tmp_Person.Telefon
new.Adresse.Strasse = old.Adresse.Strasse
new.Adresse.Ort = old.Adresse.Ort
```

4.3.3.7 Umkehrung der Aggregationsreihenfolge

Bei der „Umkehrung der Aggregationsreihenfolge“ wird die Verkettungsreihenfolge, die durch eine Referenz im ersten Objekt auf das zweite Objekt gegeben ist, umgekehrt.
Im Prinzip entspricht diese Operation der Operation „Einfügen von Referenzen“ mit anschließendem Löschen der ursprünglichen Referenz.
Barbara Staudt Lerner [Ler96] nennt diese Schemaänderung „Link Reversal“.

Beispielsituation: siehe Abb. 4.19

Vorgehensweise:

- auf Schemaebene
  Einfügen des neuen Attributs (Operation S3.2):
  `CREATE REFERENCE Arbeit:Arbeiter` (in der Umgebung Vereinsmitglied)
  Löschen der überflüssigen Referenz (Operation S1.2):
  `DELETE ATTRIBUTE Verein` (in der Umgebung Arbeiter)

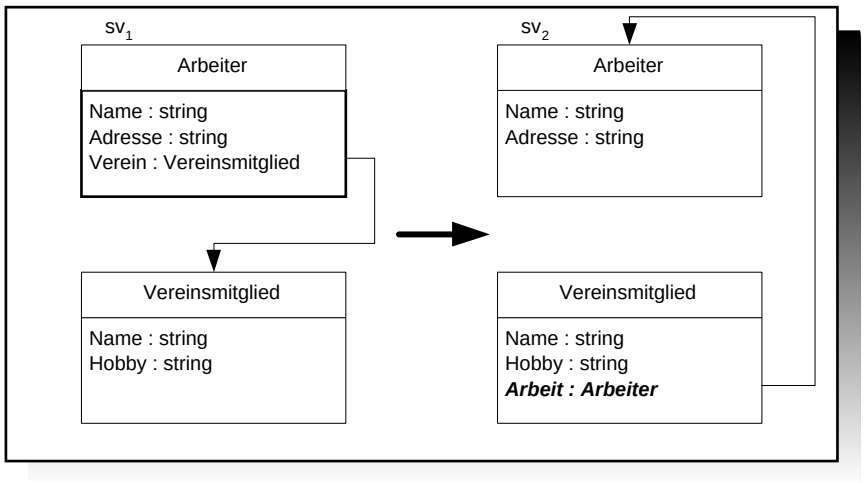

Abbildung 4.19: Beispiel: Umkehrung der Aggregationsreihenfolge

- auf Objektebene
  Setzen der neuen Referenz (Operation O2.3):
  `new.Vereinsmitglied.Arbeit = ref(new.Arbeiter)`

Syntax:
`SWAP REFERENCE` Referenzattribut_in_Klasse1`,`Zielreferenzattribut_in_Klasse2 `IN` Klasse1`,`Klasse2

Syntax für dieses Beispiel:
`SWAP REFERENCE Verein,Arbeit IN Arbeiter,Vereinsmitglied`

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Arbeiter
  ```
  new.Arbeiter.Name = old.Arbeiter.Name
  new.Arbeiter.Adresse = old.Arbeiter.Adresse
  ```

- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Vereinsmitglied
  ```
  _tmp_Arbeiter = sql_query(„...\)
  new.Vereinsmitglied.Name = old.Vereinsmitglied.Name
  new.Vereinsmitglied.Hobby = old.Vereinsmitglied.Hobby
  new.Vereinsmitglied.Arbeit = _tmp_Arbeiter
  ```

- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Arbeiter
  ```
  _tmp_Vereinsmitglied = sql_query(„...\)
  new.Arbeiter.Name = old.Arbeiter.Name
  new.Arbeiter.Adresse = old.Arbeiter.Adresse
  new.Arbeiter.Verein = _tmp_Vereinsmitglied
  ```

- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Vereinsmitglied
  ```
  new.Vereinsmitglied.Name = old.Vereinsmitglied.Name
  new.Vereinsmitglied.Hobby = old.Vereinsmitglied.Hobby
  ```

4.3.4 Komplexe Schemaänderungsoperationen, bei denen Default-Konvertierungsfunktionen nicht ausreichen

Die im Folgenden beschriebenen Schemaänderungen benötigen komplexe Schemaänderungsoperationen. Zudem tauchen Mehrdeutigkeiten bei der Erzeugung der jeweiligen Default-Konvertierungsfunktion auf, so dass dort ein Eingriff des Schemaentwicklers notwendig ist.

Die aufgeführten Schemaänderungen sind das Kopieren von Attributen über Wertevergleiche und das Verschieben von Attributen in eine Ober- oder Unterklasse. Dabei wird besonderes Augenmerk auf Konfliktmöglichkeiten bei der zweiten Operation gelegt.

4.3.4.1 Kopieren von Attributen über Wertevergleiche

Daten werden aus einem durch Wertevergleich zu findenden Objekt in ein anderes Objekt – oder umgekehrt – kopiert.
Barbara Staudt Lerner [Ler96] nennt diese Schemaänderung „Duplication based on value relationship“.

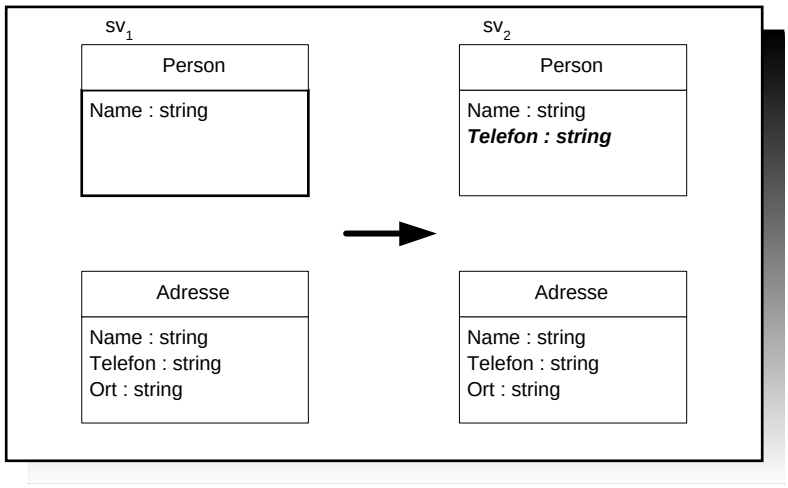

Abbildung 4.20: Beispiel: Kopieren von Attributen über Wertevergleiche

Beispielsituation: siehe Abb. 4.20

Vorgehensweise:

- auf Schemaebene
  Kopieren eines Attributs (Operation S3.1):
  `COPY ATTRIBUTE Telefon FROM CLASS Adresse`
  `VALUE (Person.Name=Adresse.Name)` (in der Umgebung Person)
- auf Objektebene
  Suchen des zugehörigen Objektes (Operation O2.2):
  `_tmp_Adresse = sql_query(„...\)`
  Überschreiben des neuen Attributs durch den Wert aus dem gefundenen Objekt (Operation O1.2):
  `new.Person.Telefon = old.Adresse.Telefon`

Syntax:
`COPY ATTRIBUTE` Attribut `FROM CLASS` Adresse `VALUE` $(a_1 = b_1, \ldots, a_n = b_n)$, wobei a_1

bis a_n Attribute aus Quellklasse 1 und b_1 bis b_n Attribute aus Quellklasse 2 sind, die paarweise verglichen werden sollen.

Syntax für dieses Beispiel:
```
COPY ATTRIBUTE Telefon FROM CLASS Adresse
VALUE (Person.Name=Adresse.Name)
```

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Person
  ```
  _tmp_Adresse = sql_query(„...\)
  new.Person.Name = old.Person.Name
  new.Person.Telefon = _tmp_Adresse.Telefon
  ```
- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Adresse
  ```
  _tmp_Adresse = sql_query(„...\)
  new.Adresse.Name = _tmp_Adresse.Name
  new.Adresse.Telefon = _tmp_Adresse.Telefon
  new.Adresse.Ort = _tmp_Adresse.Ort
  ```
- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Person
  ```
  _tmp_Adresse = sql_query(„...\)
  new.Person.Name = old.Person.Name
  ```
- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Adresse
  ```
  _tmp_Adresse = sql_query(„...\)
  new.Adresse.Name = _tmp_Adresse.Name
  new.Adresse.Telefon = _tmp_Adresse.Telefon
  new.Adresse.Ort = _tmp_Adresse.Ort
  ```

Man könnte die Suche des passenden Adressenobjekts auch nur in der neuen Objektversion durchführen, und den Inhalt des Feldes `Telefon` von dort kopieren – also ohne Zugriff auf die alte Objektversion. Allerdings hat man diesen Zugriff nicht eingespart, denn alle trivialen Konvertierungen sind ja auf jeden Fall als Defaultkonvertierungen vorgesehen. So wird `Telefon` ohnehin vom alten in das neue Adressenobjekt kopiert. Ob es dann von dort oder aus der alten Objektversion gelesen wird, um das Telefonfeld im Personobjekt zu füllen, ist frei wählbar.
In dieser Diplomarbeit wurde die Entscheidung zugunsten der ersten genannten Alternative getroffen, da sich die nötigen Operationen einfach weiterverwenden lassen und z.B. für den Fall „Verschieben von Attributen in eine Unterklasse“ (s. Abschnitt 4.3.4.3) ohnehin benötigt werden.

Sollten mehrere passende Adressenobjekte gefunden werden, gibt es wiederum mehrere Möglichkeiten, darauf zu reagieren:

1. Man kann zufällig eines der passenden Adressenobjekte auswählen und dann vorgehen, als hätte die Suche nur dieses eine Objekt erbracht. Der Nachteil ist offensichtlich: Es kann nicht eindeutig sichergestellt werden, welcher Wert in `Telefon` geschrieben wird, weil nicht sicher ist, welches Adressenobjekt verwendet wird.
2. Man kann in einem solchen Fall die Auswahl eines passenden Objektes umgehen, indem man nur einen konstanten Wert (z.B. einen Leerstring) oder auch den Eintrag „mehrdeutig“ einfügt.
3. Man kann den Typ des Feldes `Telefon` von `string` in `set(string)` ändern und alle gefundenen Telefonnummern übertragen. In diesem Falle wird allerdings eine

Typänderung implizit durchgeführt, die womöglich vom Benutzer nicht gewünscht ist.

4. Es kann gefordert werden, dass der zu vergleichende Wert ein Schlüssel ist, er also innerhalb der Datenbank eindeutig ist. In diesem Falle ist gesichert, dass in obigem Beispiel zu jedem Personobjekt nur ein korrespondierendes Adressenobjekt existiert.

Gerade die ersten drei Lösungsmöglichkeiten könnten je nach Situation gewünscht sein, weshalb in diesem Punkt der Eingriff des Schemaentwicklers notwendig wird. Das System wird also eine feste Auswahl vornehmen und in die Konvertierungsfunktion einen Kommentar mit der Aufforderung zur Überprüfung, ob die gewählte Lösungsmöglichkeit die gewünschte ist, aufnehmen. Die vierte Variante schränkt den Schemaentwickler in der Wahl der zu vergleichenden Attribute sehr ein.
Hier wählt das System immer die Lösungsmöglichkeit 1, da diese keine Typen eigenmächtig ändert, andererseits aber wenigstens eine Information aus der alten Objektversion übernimmt. Dazu wird einfach nach dem ersten gefundenen Objekt die Suche abgebrochen.

4.3.4.2 Verschieben von Attributen in eine Oberklasse

Bei dieser Operation werden ein oder mehrere Attribute entlang der Vererbungshierarchie aus der abgeleiteten in die Basisklasse verschoben. Die abgeleitete Klasse, die das Attribut ursprünglich enthielt, erbt dieses nun von der Basisklasse, so dass sich nach außen keine Änderungen ergeben.
Da das Attribut aber auch an weitere von der Basisklasse abgeleitete Klassen vererbt wird, kann es in vielen weiteren Unterklassen auftauchen. Es können dadurch Konflikte auftreten.

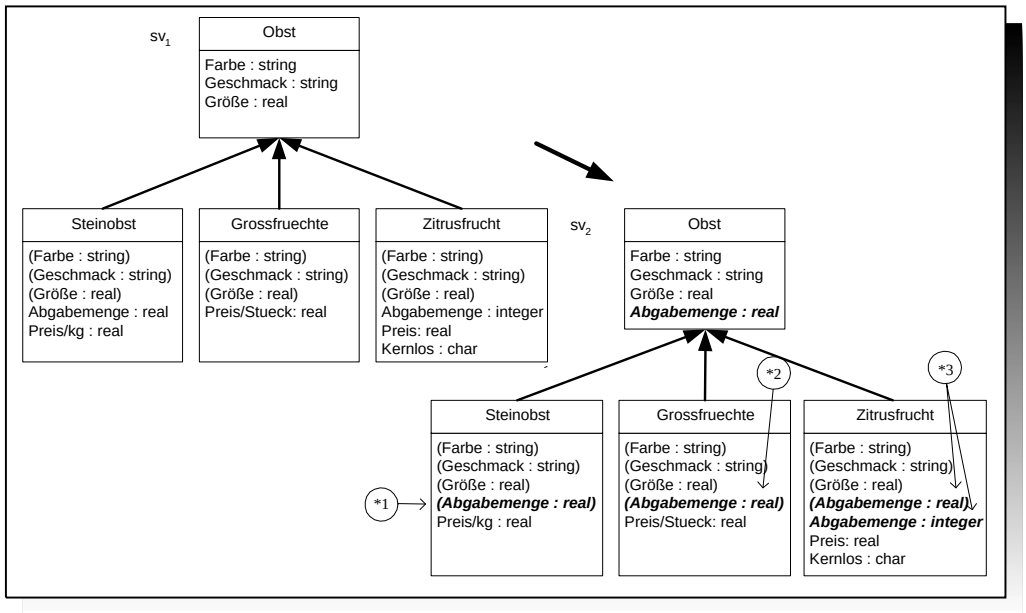

Abbildung 4.21: Konfliktmöglichkeiten beim Verschieben von Attributen in eine Oberklasse

Im Beispiel (s. Abbildung 4.21) wird aus der Unterklasse „Steinobst“ das Attribut „Abgabemenge“ in die Oberklasse „Obst“ verschoben. Es treten dadurch mehrere Effekte auf:

- (markiert mit *1) In der Klasse „Steinobst“ taucht wie gewünscht das verschobene Attribut nun als geerbtes Attribut auf.

- (markiert mit *2) In der Klasse „Grossfruechte“ taucht das Attribut ebenfalls auf, obwohl es vielleicht gar nicht sinnvoll wäre: Bei Großfrüchten wie Melonen und Kürbissen ist die Abgabemenge vielleicht ohnehin immer 1.

- (markiert mit *3) In der Klasse „Zitrusfrucht“ kommt es sogar zu einem Konflikt. Es existiert bereits ein lokales Attribut „Abgabemenge“, das aber einen anderen Typ hat.

  Man kann nun die ursprüngliche lokale Definition als lokale *Re*definition sehen, die das geerbte Attribut überschreibt. Eine Voraussetzung für diese Lösungsmöglichkeit ist, dass der Typ des lokal redefinierten Attributs eine Verfeinerung des geerbten Typs ist.
  Eine weitere Möglichkeit, diesen Konflikt zu vermeiden, besteht darin, erst das kritische Attribut in der betreffenden Unterklasse zu löschen und anschließend die Verschiebung vorzunehmen. In dem Falle hätte man die Problematik auf den Attributinhalt verlagert: Das Attribut in der Klasse „Zitrusfrucht“ hätte zwar einen legalen Typ, allerdings könnte es sein, dass die Konvertierung des Attributinhaltes Probleme bereitet, wenn die Typen nicht verlustfrei ineinander konvertierbar sind. Allerdings kann jederzeit ein NULL-Wert eingesetzt werden. Die Vererbungshierarchie ist damit weiter fehlerfrei.

Falls der Benutzer also vor der Verschiebung des Attributs nicht hinreichend die Konsequenzen für die anderen Klassen in der Vererbungshierarchie bedenkt, kann es zu Problemen kommen.

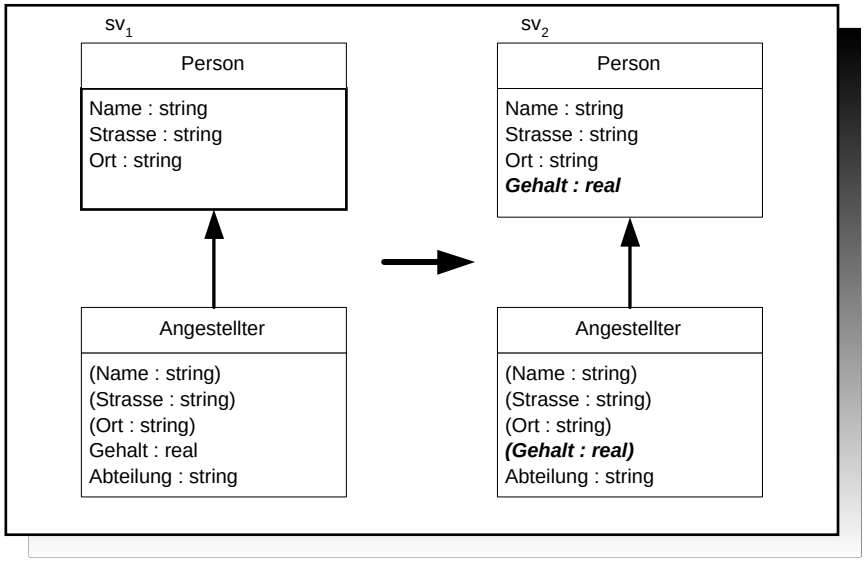

Abbildung 4.22: Beispiel: Verschieben von Attributen in eine Oberklasse

Beispielsituation: siehe Abb. 4.22 „Angestellter“ erbt von „Person“. Das Attribut „Gehalt“ wird von der erbenden Klasse „Angestellter“ in die Vaterklasse „Person“ verschoben. Durch die Vererbung steht es aber weiterhin in der Klasse „Angestellter“ zur Verfügung.

Vorgehensweise:

- auf Schemaebene
  Kopieren des Attributs in die Oberklasse (Operation S3.1):
  `COPY ATTRIBUTE Gehalt FROM CLASS Angestellter` (in der Umgebung Person)
  Dadurch existiert in der Unterklasse für kurze Zeit das Attribut Gehalt doppelt:

Zum einen als geerbtes Attribut und zum anderen als lokal definiertes Attribut. Eine zeitweise Inkonsistenz des Schemas kann dadurch vermieden werden, dass die beiden Operationen COPY und DELETE direkt nacheinander ausgeführt werden müssen. Dazu wird – wie bei jeder Schemaänderung – das Schema eingefroren, die nötigen Änderungen durchgeführt und das Schema wieder aufgetaut. Im eingefrorenen Zustand ist die Konsistenz des Schemas nicht gewährleistet, sie muss nur nach dem Auftauen wieder gesichert sein (s. [Dol99]). Es muss also hier nur verhindert werden, dass das Schema schon nach dem ersten Schritt aufgetaut wird.
Löschen des überflüssigen Attributs in der Unterklasse (Operation S1.2):
`DELETE ATTRIBUTE Gehalt` (in der Umgebung Angestellter)

- auf Objektebene
  Füllen des Attributs in der Unterklasse mit einem Wert aus einer anderen Objektversion (Operation O1.2):
  `new.Angestellter.Gehalt = old.Angestellter.Gehalt`
  Füllen des neuen Attributs in der Oberklasse mit einem NULL-Wert (Operation O1.1):
  `new.Person.Gehalt = 0`

Syntax:
`MOVEUP ATTRIBUTE` Attribut `IN` Unterklasse `TO` Oberklasse

Syntax für dieses Beispiel:
`MOVEUP ATTRIBUTE Gehalt IN Angestellter TO Person`

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Person
  ```
  new.Person.Name = old.Person.Name
  new.Person.Strasse = old.Person.Strasse
  new.Person.Ort = old.Person.Ort
  new.Person.Gehalt = 0
  ```

- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Angestellter
  ```
  new.Angestellter.Name = old.Angestellter.Name
  new.Angestellter.Strasse = old.Angestellter.Strasse
  new.Angestellter.Ort = old.Angestellter.Ort
  new.Angestellter.Gehalt = old.Angestellter.Gehalt
  new.Angestellter.Abteilung = old.Angestellter.Abteilung
  ```

- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Person
  ```
  new.Person.Name = old.Person.Name
  new.Person.Strasse = old.Person.Strasse
  new.Person.Ort = old.Person.Ort
  ```

- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Angestellter
  ```
  new.Angestellter.Name = old.Angestellter.Name
  new.Angestellter.Strasse = old.Angestellter.Strasse
  new.Angestellter.Ort = old.Angestellter.Ort
  new.Angestellter.Gehalt = old.Angestellter.Gehalt
  new.Angestellter.Abteilung = old.Angestellter.Abteilung
  ```

4.3.4.3 Verschieben von Attributen in eine Unterklasse

Diese Operation ist die Umkehrung der vorherigen Operation. Ein Attribut wird von einer Basisklasse zu einer Unterklasse verschoben. Die erbende Klasse, in die das Attribut verschoben wird, enthält dann das Attribut, in der Vaterklasse taucht es nicht mehr auf, ebenso wie in allen anderen erbenden Klassen der Vaterklasse.

Schwierigkeiten können auftreten, wenn das betreffende Attribut in einer der Klassen benötigt wird, in denen es bisher durch Vererbung vorhanden war. Eine mögliche Abhilfe könnte sein, das Attribut dort dann lokal zu definieren. Allerdings ist so nicht mehr die Konsistenz gewährleistet, dass das Attribut in allen Klassen denselben Typ hat.

Lokale Redefinitionen des ursprünglich geerbten Attributs bereiten keine Probleme, sie bleiben bestehen. So wird das Attribut dann in diesen Klassen nicht mehr *re*definiert, sondern im eigentlichen Sinne neu definiert.

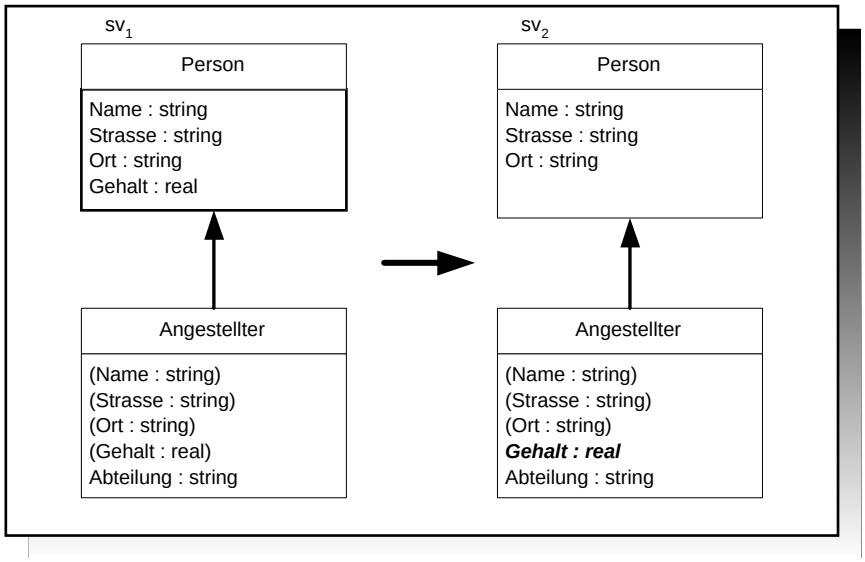

Abbildung 4.23: Beispiel: Verschieben von Attributen in eine Unterklasse

Beispielsituation: siehe Abb. 4.23 „Angestellter“ erbt von „Person“

Vorgehensweise:

- auf Schemaebene
  Kopieren des Attributs in die Unterklasse (Operation S3.1):
  `CREATE ATTRIBUTE Gehalt FROM CLASS Person` (in der Umgebung Angestellter)
  Dadurch existiert in der Unterklasse für kurze Zeit das Attribut Gehalt doppelt: Zum einen als geerbtes Attribut und zum anderen als lokal definiertes Attribut. Hier wird das neue Attribut automatisch umbenannt, um den Namenskonflikt zu vermeiden. Nach dem Löschen kann das Attribut dann den gewünschten Namen erhalten.
  Löschen des Attributs in der Oberklasse (Operation S1.2):
  `DELETE ATTRIBUTE Gehalt` (in der Umgebung Person)

- auf Objektebene
  Füllen des neuen Attributs in der Unterklasse mit einem referenzierten Wert (Operation O1.2):
  `new.Angesteller.Gehalt = old.Person.Gehalt`

Syntax:
`MOVEDOWN` Attribut `IN` Oberklasse `TO` Unterklasse

Syntax für dieses Beispiel:
`MOVEDOWN Gehalt IN Person TO Angestellter`

Default-Konvertierungsfunktion:

- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Person
```
new.Person.Name = old.Person.Name
new.Person.Strasse = old.Person.Strasse
new.Person.Ort = old.Person.Ort
```
- Vorwärtskonvertierungsfunktion (fcf) für die Klasse Angestellter
```
new.Angestellter.Name = old.Angestellter.Name
new.Angestellter.Strasse = old.Angestellter.Strasse
new.Angestellter.Ort = old.Angestellter.Ort
new.Angestellter.Gehalt = old.Angestellter.Gehalt
new.Angestellter.Abteilung = old.Angestellter.Abteilung
```
- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Person
```
new.Person.Name = old.Person.Name
new.Person.Strasse = old.Person.Strasse
new.Person.Ort = old.Person.Ort
new.Person.Gehalt = 0
```
- Rückwärtskonvertierungsfunktion (bcf) für die Klasse Angestellter
```
new.Angestellter.Name = old.Angestellter.Name
new.Angestellter.Strasse = old.Angestellter.Strasse
new.Angestellter.Ort = old.Angestellter.Ort
new.Angestellter.Gehalt = old.Angestellter.Gehalt
new.Angestellter.Abteilung = old.Angestellter.Abteilung
```

4.4 Migration

Eine typische Einsatzmöglichkeit für Schemaänderungen könnte auch sein, zu einer Klasse mehrere Unterklassen anzulegen und die Objekte der ursprünglichen Klasse in die Unterklassen zu verschieben. Dies lässt sich mit den bisherigen Schemaänderungsoperationen nur mit komplizierten Konvertierungsfunktionen erledigen.

Beispiel 4.4.1 *Gegeben seien zwei Schemaversionen sv_1 und sv_2 (s. Abb. 4.24). In Schemaversion sv_1 existiert eine Klasse „KFZ“ mit den Attributen* `Marke`, `Preis` *und* `kW`. *In Schemaversion sv_2 werden zur Klasse „KFZ“ die abgeleiteten Klassen „PKW“, „Sportwagen“ und „LKW“ hinzugefügt, in die sämtliche Objekte aus „KFZ“ wandern sollen und die jeweils noch ein weiteres Attribut besitzen. Die abgeleiteten Klassen haben eine disjunkte Extension, d.h. jedes Objekt kann nur in einer der Unterklassen vorkommen. Dies kann für eine feinere Gliederung gewünscht sein, beispielsweise für verschiedene Steuerklassen o.ä.*
Jedes bisherige Objekt der Klasse „KFZ“ soll nun nach bestimmten Kriterien migriert werden.

Der erste Schritt, also das Anlegen der Unterklassen, ist mit den vorhandenen Schemaänderungsoperationen möglich. Die Migration der Objekte in einzelne Unterklassen kann al-

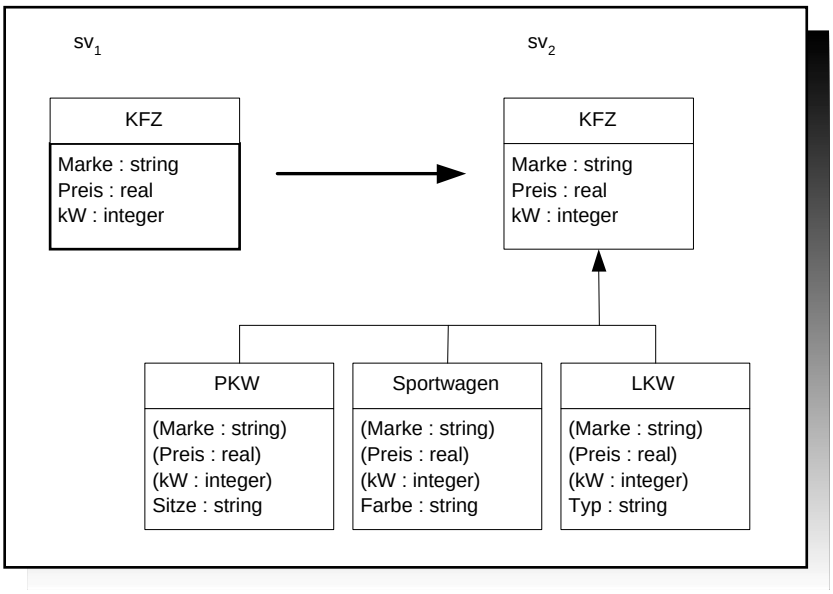

Abbildung 4.24: Die Klasse 'KFZ' mit den Unterklassen 'PKW', 'Sportwagen' und 'LKW'

lerdings noch nicht durchgeführt werden. Ein Trick wäre, bei der Integration der neuen Schemaversion mittels den in dieser Diplomarbeit neu entwickelten `if`-Konstrukten (s. Abschnitt 6.3.8) die neue Objektversion der einen oder anderen Klassenversion zuzuordnen. Man könnte so für jedes Objekt aus „KFZ“ überprüfen, in welche neue Klasse es migriert werden soll.

Eleganter ist diese Aufgabenstellung mit Hilfe der in [FMZ+95] vorgestellten *Migration* zu lösen.
Das Konzept der Migration hat genau genommen nichts mit Schemaänderungen zu tun, aber Ferrandina stellte fest, dass es gut in die Schemaevolution passt. Meine Beobachtung dazu ist, dass diese Aussage auch für die Schemaversionierung zutrifft. Daher wurde im Zuge dieser Diplomarbeit COAST um Migrationskonzepte erweitert.

Eine mögliche Beschreibung für die Verwendung von `if`-Konstrukten ist exemplarisch für die Klasse „Sportwagen“ in Abb. 4.25 skizziert, Abb. 4.26 zeigt den Vorgang schematisch.

```
class Sportwagen inherit KFZ
 type tuple (Farbe: string)
 end;

if(kW > 100, create object KFZ,
  create object Sportwagen)

fcf
{
...
}
```

Abbildung 4.25: Migration mit Hilfe von `if`-Konstrukten

Der Migrationsansatz geht so vor, dass jedes Objekt aus „KFZ“ in Schemaversion sv_1 unabhängig von seinem Wert, in die Klasse „KFZ“ in Schemaversion sv_2 konvertiert wird. Danach werden die Objekte, die den Anforderungen laut einer vorher festgelegten *Migrations-*

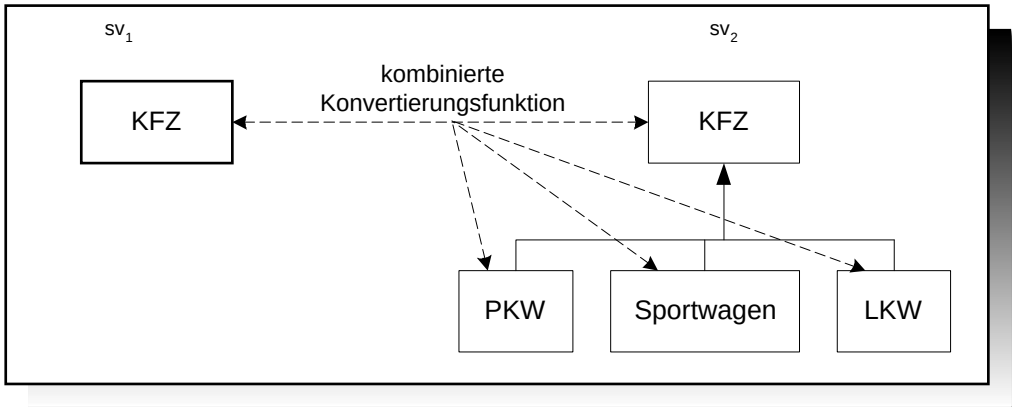

Abbildung 4.26: Konvertierungsansatz mit `if`-Konstrukten

funktion (mf) entsprechen, in eine der Unterklassen migriert. Grafik 4.27 verdeutlicht diese Vorgehensweise.

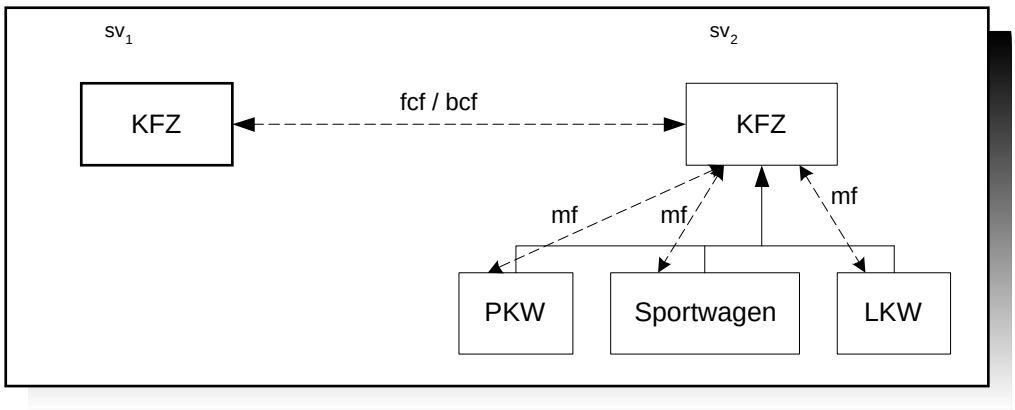

Abbildung 4.27: Konvertierungsansatz mit Migrationsfunktionen („mf“)

Der Vorteil gegenüber der ausschließlichen Verwendung von Konvertierungsfunktionen wie oben beschrieben ist auch, dass nur eine Konvertierungsfunktion zwischen den beiden „KFZ“-Klassen existieren muss, die Unterklassen von „KFZ“ aber keine direkte Konvertierungsfunktionen zu „KFZ“ benötigen.

Auch die Rückrichtung ist ebenso einfach denkbar: Man würde die zu konvertierenden Objekte erst temporär von den Unterklassen nach „KFZ“ migrieren, und sie von dort über die normale Konvertierungsfunktion in die Klasse „KFZ“ der Zielschemaversion konvertieren. Man kann sogar die Objekte aus den Unterklassen als Objekte der Klasse „KFZ“ betrachten, denn sie gehören als Unterklassenobjekte zur Klassenextension von „KFZ“. Nur dann, wenn wirklich eine Migration gefordert ist, beispielsweise, wenn die Objekte wieder in die Oberklasse verschoben werden sollen, um danach die entsprechende Unterklasse zu löschen, muss wirklich eine Migration von der Unterklasse in die Oberklasse stattfinden, wobei allerdings Attributwerte verlorengehen können.

Fabrizio Ferrandina stellt in [FMZ+95] ein Konzept für die Migration vor. Die Grundidee ist dabei, dass ein Objekt seine Klasse wechseln kann. Ferrandina betrachtet allerdings keine versionierten Schemata, die Migration erfolgt innerhalb *einer* Schemaversion bzw. im Zusammenhang mit zwei Schemaausprägungen.
Besonders interessant erscheint die Migration in zwei Fällen:

- Eine neue Klasse wird zur Spezialisierung eingefügt und Objekte aus der Oberklasse müssen in die neue Klasse „nach unten“ verschoben werden
- Eine Klasse wird gelöscht und Objekte aus dieser Klasse sollen gerettet werden, indem sie in Oberklassen „nach oben“ verschoben werden.

Durch eine Migrationsfunktion wird gewährleistet, dass ein Objekt `q` von seiner ursprünglichen Klasse in eine neue Zielklasse wandert.

Die Funktion `migrate` erledigt nach Ferrandina für jedes zu migrierende Objekt `q` die folgenden Schritte:

1. Der Wert des Objektes `q` wird in eine Variable `old` kopiert
2. Die Zielklasse wird gesucht.
3. Das Objekt `q` wird restrukturiert, damit es der Zielklasse entspricht.
4. Die Originaldaten werden aus `old` extrahiert und in `q` übertragen.
5. Die Klassen-ID im Header des Objektes `q` wird auf die neue Klasse aktualisiert.

Eine mögliche Beschreibung der Migrationsfunktion in Beispiel 4.4.1 in der ODL könnte für die Klasse „Sportwagen“ also lauten wie in Abb. 4.28 angegeben.

```
class Sportwagen inherit KFZ
type tuple (Farbe: string)
end;

migration function migrate_auto in class KFZ
{
  if (self->kW > 100,
      self->migrate("Sportwagen"), )
  fcf
  {
    ...
  }
}
```

Abbildung 4.28: Beispiellisting mit Migration

Zusätzlich wird hier nur die Vorwärtskonvertierungsfunktion von „KFZ“ in Schemaversion sv_1 nach „KFZ“ in Schemaversion sv_2 benötigt.

Was in [FMZ+95] nicht geklärt wird, ist die Einbindung neuer Attribute in den Unterklassen. Sinnvoll wäre sicher, in der Migrationsfunktion für die Richtung zu einer Unterklasse die neu hinzukommenden Attribute mit Defaultwerten zu belegen.

4.5 Zusammenfassung und Bewertung

Viele erforderliche Schemaänderungen lassen sich nicht mit den vor dieser Diplomarbeit verfügbaren einfachen Schmemaänderungsoperationen bewerkstelligen. Daher wurden hier *komplexe Schemaänderungsoperationen* entwickelt und eingeführt. Dazu wurde eine Reihe von typischen Schemaänderungsoperationen in einzelne Schritte zerlegt und geprüft, ob sie mit einfachen Schemaänderungsoperationen durchführbar sind oder nicht. Dabei

wurde die Beobachtung gemacht, dass für die Unterstützung von komplexen Schemaänderungen auf Schemaebene vier neue Operationen ausreichen (s. Abschnitt 4.1.2). Zusätzlich wurde für jede Operation neben einem Beispiel eine Befehlssyntax, die mit der in [Her99] vorgestellten ODL (Object Definition Language) in Einklang steht, vorgestellt. Die Auswirkungen der Operation auf Schema- und Objektebene wurden dargestellt und eine Default-Konvertierungsfunktion (in der in dieser Diplomarbeit entwickelten Syntax der Konvertierungssprache, s. Abschnitt 6.3), die vom System erzeugt wird, angegeben.

Die bisherige Taxonomie war bezogen auf das Schema vollständig. Allerdings traten bei Berücksichtigung von Auswirkungen auf die Objektebene Mängel zutage, daher musste sie um wichtige und häufig erforderliche komplexe Operationen ergänzt werden. Diese ließen sich sehr gut, d.h. unter Beibehaltung der bisherigen Struktur, in die existierende ODL einbetten.

Abschließend wurde das Konzept der Migration vorgestellt, mit dem Objekte einer Klasse in eine Ober- oder Unterklasse verschoben werden können. Obwohl die Migration keine Schemaänderung im eigentlichen Sinne ist, wird sie bei Restrukturierungen eines Schemas häufig benötigt und ihre technische Realisierung lässt sich überraschend homogen in die Mechanismen von COAST integrieren.

Kapitel 5

Propagation

Um den Applikationen der verschiedenen Versionen eines Schemas den Zugriff auf die Datenbank zu ermöglichen, müssen die Objekte in den passenden Typen zugreifbar sein. Die Klasse eines Objekts kann sich allerdings von einer Schemaversion zur nächsten ändern und demzufolge liegt auch sie in verschiedenen Versionen, den *Klassenversionen*, vor. Eine physikalische Ausprägung eines Objekts kann jedoch i.A. nicht den Typen mehrerer Versionen seiner Klasse entsprechen. Daher werden versionierte Objekte verwendet. Jedes Objekt, das in einer Schemaversion zugreifbar ist, enthält genau die für diese Schemaversion passende Objektversion, die das Datenbanksystem beim Zugriff auf das Objekt zurückgibt. Änderungen einer Objektversion müssen an andere Objektversionen weitergegeben werden, das ist die *Propagation*.

Durch die Propagation wird das Versionierungskonzept von der Schemaebene auf die Objektebene übertragen, d.h. es findet ein Austausch von Informationen zwischen den Zugriffsbereichen mehrerer Versionen eines Schemas statt.

Dieser Mechanismus arbeitet im wesentlichen so, dass die von der Applikation gewünschte Objektversion, falls nötig, über Konvertierungsfunktionen mit Daten aus einer anderen Objektversion gefüllt wird. Weiter soll ermöglicht werden, den Zugriff auf bestimmte Objekte über bestimmte Schemaversionen steuerbar zu machen, d.h. der Schemaentwickler soll angeben können, welches Objekt in welcher Schemaversion sichtbar ist.

Die im vorigen Kapitel entwickelte Erweiterung auf komplexe Schemaänderungen hat auch Auswirkungen auf die Funktionsweise der Propagation, die bisher in COAST implementiert war. In diesem Kapitel wird die Beobachtung gemacht, dass die Propagationskanten, die zwischen je zwei Versionen derselben Klasse liegen, ebenfalls erweitert werden müssen. Dazu werden verschiedene Ansätze vorgestellt. Die Erweiterung der in COAST verwendeten Propagationskanten basiert auf einem dieser Ansätze, dessen Auswahl begründet wird. Abschließend wird noch auf das Problem der zurücklaufenden Propagation hingewiesen, die vermieden werden muss, und beschrieben, wie eine transitive Propagation zwischen Schemaversionen, die im Ableitungsbaum nicht direkt benachbart sind, durchgeführt wird.

5.1 Ziel

In diesem Kapitel werden die der Propagation zugrundeliegenden Konzepte erläutert und nach einer kurzen Bestandsaufnahme, wie die Propagation in COAST vor der Entstehung dieser Diplomarbeit aussah, der Ansatz der verzögerten Propagation vorgestellt und mit dem Ansatz der sofortigen Propagation verglichen. Dann wird auf die neu erstellte Erweiterung von einfachen Propagationskanten auf kombinierte Propagationskanten, die für

die in dieser Diplomarbeit entwickelten komplexen Konvertierungsfunktionen nötig sind, eingegangen. Anschließend wird der konkrete Ablauf der Propagation im implementierten Prototyp COAST erklärt und einige der zugrundeliegenden Algorithmen beschrieben.

5.2 Konzepte

Die Grundidee bei der Propagationssteuerung besteht darin, in einem Objekt vorliegende Daten in mehreren Versionen anzubieten. Jede Objektversion gehört zu genau einem Objekt. Für jede Schemaversion, in der das Objekt sichtbar ist, gibt es eine Objektversion dieses Objekts. Es muss allerdings nicht immer eine Objektversion existieren, denn durch die verzögerte Propagation (s. Abschnitt 5.4) kann es sein, dass eine Objektversion erst dann erzeugt wird, wenn darauf zugegriffen wird.

5.3 Ausgangssituation

Diese Diplomarbeit basiert auf [Eig97], in der die grundlegende Propagation eingeführt wird. Patricia Eigner bespricht Propagationsflags, die die Propagation zwischen zwei Objektversionen steuern. Des Weiteren bespricht Eigner Vorwärts-, Rückwärts- und Extrakonvertierungsfunktionen, eine Erweiterung der Arbeit von [Wöh96], in der nur Vorwärtskonvertierungsfunktionen beschrieben wurden.

5.4 Die Strategie der verzögerten Propagation

Es gibt zwei Möglichkeiten, veränderte Objekte zu propagieren, zum einen die *sofortige Propagation*, zum anderen die *verzögerte Propagation* ([FMZ+95]). Bei der sofortigen Propagation wird jede Änderung sofort an alle anderen Objektversionen (unter Berücksichtigung der Propagationsflags, die im nächsten Abschnitt vorgestellt werden) weitergegeben. Bei der verzögerten Propagation wird erst dann, wenn tatsächlich auf eine Objektversion zugegriffen wird, die Änderung an der Objektversion durchgeführt.

Die verzögerte Propagation hat damit zwar den Nachteil, dass sie im Moment des Zugriffs die Konvertierungen durchführen muss – die eventuell auch über mehrere Objektversionen, die noch in der Ableitungshierarchie zwischen der gewünschten und der aktuellsten Objektversion liegen, verläuft. Andererseits hat sie den Vorteil, dass nicht jede Änderung auch wirklich propagiert werden muss, wie im folgenden Beispiel verdeutlicht wird.

Beispiel 5.4.1 *Ein Bankkonto sei in Schemaversion sv_1 mit den Attributen Name und Kontostand definiert, in Schemaversion sv_2 mit den Attributen Name, Kontostand und Kontonummer. Wenn nun eine oder mehrere Applikationen, die auf Schemaversion sv_1 zugreifen, zehn Kontobuchungen umsetzen, d.h. zehn mal den Wert des Attributes Kontostand ändern, und dann eine Kontostandsabfrage über eine Applikation auf Schemaversion sv_2 erfolgt, wird die Änderung des Kontostands propagiert. In diesem Falle hat das System aber statt zehn Propagationsvorgängen nur einen durchzuführen. Die vorhergehenden neun Durchführungen der Propagation, also die Auswertung und Ausführung der entsprechenden Konvertierungsfunktionen, werden somit eingespart.*

Es ist natürlich denkbar, die Änderungen, die noch nicht propagiert wurden, in Leerlaufzeiten wie z.B. nachts oder zu Zeiten mit niedriger Auslastung der Datenbank weiterzugeben. Auf diese Weise hätte man bewirkt, dass eine Änderung an einer Objektversion weder sofort nach der Eingabe, noch direkt vor dem Zugriff über eine andere Objektversion durchgeführt werden muss – dies sind die Zeitpunkte, zu denen der Anwender direkt mit der Datenbank kommuniziert und sonst warten müsste.
Technisch kann man sich diese Variante als einen Mechanismus der verzögerten Propagation vorstellen, der zu bestimmten Zeiten eine sofortige Propagation aller noch nicht weitergegebenen Änderungen durchführt.

Allerdings treten Probleme in Verbindung mit komplexen Schemaänderungen auf. Angenommen, es existiert in Schemaversion sv_1 eine Klasse „Person“, die eine andere Klasse „Adresse“ referenziert. In Schemaversion sv_2 sind die Attribute der Klasse „Adresse“ durch die in Abschnitt 4.3.3.4 beschriebene komplexe Schemaänderungsoperation „Zusammenführen von Klassen über Referenzen“ in die Klasse „Person“ mit eingegliedert worden. Dabei ist allerdings in der Propagationskante das Modify-Flag nicht gesetzt (s. Abb. 5.1, im Bild wird bereits die in Abschnitt 5.6.2.2 eingeführte kombinierte Propagationskante verwendet). Wird nun eine Änderung an der Klasse „Adresse“ in sv_1 vorgenommen, muss

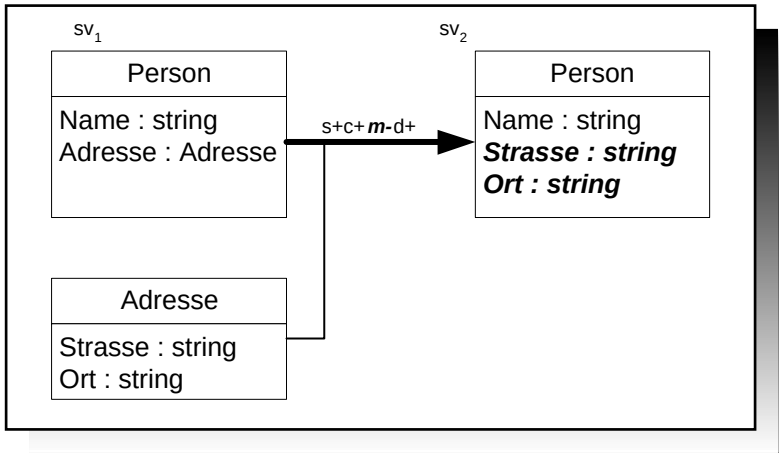

Abbildung 5.1: Propagation bei nicht gesetztem Modify-Flag (m-) aber gesetztem Create-Flag (c+)

diese zwar nicht an sv_2 propagiert werden. Aber es ist möglich, dass der aktuelle Wert der Klasse „Adresse“ beispielsweise durch eine Neuerstellung eingetragen wurde und durch das gesetzte Create-Flag muss diese Information vor dem Überschreiben propagiert werden. In dieser Situation ist also eine *sofortige* Propagation dieses Wertes notwendig.

Es gibt nun mehrere Lösungsansätze:

1. Falls eine beliebige komplexe Schemaänderung durchgeführt wird, müssen alle Objekte in der gesamten Datenbank sofort propagiert werden.

2. Bei einer Schemaänderung, die die Klasse „Person“ betrifft, müssen die zugehörigen Objekte sofort propagiert werden.

3. Es muss nur dann eine sofortige Propagation der Objekte der Klasse „Person“ stattfinden, wenn eine entsprechende komplexe Schemaänderung durchgeführt wurde.

4. Es muss nur dann eine sofortige Propagation der Objekte der Klasse „Person“ stattfinden, wenn die Propagationsflags entsprechend gesetzt sind, also das Modify-Flag nicht gesetzt und das Create- oder das Snapshot-Flag gesetzt ist.

Die feinste Untergliederung zeigt die vierte Lösungsmöglichkeit. Bei allen anderen Lösungsansätzen würde auch dann, wenn eigentlich keine sofortige Propagation erforderlich ist, sofort propagiert. Bei diesem Problem besteht noch weiterer Diskussionsbedarf.

In dieser Hinsicht muss durch die Einführung von komplexen Konvertierungsfunktionen also eine Einschränkung am bisherigen Konzept der verzögerten Propagation in Kauf genommen werden.

5.5 Propagationsflags

Es kann vorkommen, dass in bestimmten Situationen nicht jede Änderung propagiert werden soll. Ein Beispiel wäre die Erstellung einer neuen Schemaversion, die noch im Teststadium ist. Jede Änderung in den bereits existierenden Objektversionen soll zwar in die neue Objektversion übertragen werden, allerdings sollen in dieser neuen Objektversion geänderte Daten nicht nach außen dringen, bis die neue Schemaversion alle Tests bestanden hat und freigegeben wird.

Dazu bietet COAST sogenannte *Propagationsflags* an, die für beide Richtungen – also in Vorwärts- (von alt zu neu) und Rückwärtsrichtung (von neu zu alt) – der Propagation angeben, welche Änderungen weitergegeben werden sollen und welche nicht. Es kann bei der jeweiligen Propagation eine der Optionen s, c, m oder d mit einem + (Flag gesetzt) oder einem – (Flag nicht gesetzt) angegeben werden. Die Propagationsflags sind klassenspezifisch und der jeweiligen Ableitungskante zweier Klassenversionen zugeordnet.

Im Einzelnen gibt es die folgenden Flags:

- Das Snapshot-Flag (s)
  Ist das Snapshot-Flag gesetzt, wird bei Erstellung der neuen Schemaversion der bestehende Datenbestand aus den anderen Objektversionen übernommen. Da durch die Ableitung einer neuen Schemaversion die Erstellung immer in Richtung dieser neuen Schemaversion geht, wird das Snapshot-Flag nur in der Vorwärtsrichtung verwendet.
- Das Create-Flag (c)
  Ist das Create-Flag gesetzt, werden neu erzeugte Objekte auch in der anderen Schemaversion sichtbar.
- Das Modify-Flag (m)
  Ist das Modify-Flag gesetzt, werden Änderungen an Objekten auch in der anderen Schemaversion sichtbar.
- Das Delete-Flag (d)
  Ist das Delete-Flag gesetzt, werden gelöschte Objekte auch in der anderen Schemaversion gelöscht.

5.6 Der Propagationsgraph

Auf Schemaebene werden die Ableitungsbeziehungen zwischen den Schemaversionen in einem Ableitungsgraphen dargestellt. Dieses Konzept soll nun auf die Objektebene übertragen werden: Es ist ein Konvertierungsgraph zu erstellen, entlang dessen Kanten die Propagation stattfinden kann.

In diesem Abschnitt wird besonders oft auf die Unterschiede zwischen Schema, Schemaversionen, Klassen, Klassenversionen, Objekten und Objektversionen eingegangen. Daher

soll hier noch einmal ein kurzer Überblick über die verwendeten Begriffe gegeben werden: Ein *Schema* enthält mehrere *Schemaversionen* (Schreibweise: sv_i). Eine *Klasse* (Schreibweise: c), die in mehreren Schemaversionen auftaucht, heißt dort jeweils *Klassenversion* (Schreibweise: $sv_i.c$). Die Versionen einer Klasse können sich beliebig voneinander unterscheiden.
Ein (Daten-)*Objekt* (Schreibweise: o) liegt in mehreren *Objektversionen* (Schreibweise: ov_i) vor. Diese Objektversionen enthalten die eigentlichen Daten der Datenbank und gehören zu jeweils genau einer Klassenversion, deren Struktur (Typ) sie entsprechen. Es muss nicht zu jeder Schemaversion auch eine Objektversion jedes Objekts geben, etwa weil ein Objekt noch nicht angelegt wurde (beispielsweise aufgrund verzögerter Propagation, s. Abschnitt 5.4) oder die betreffende Klasse in dieser Schemaversion nicht enthalten ist, also keine entsprechende Klassenversion existiert.

5.6.1 Propagationsgraph für einfache Konvertierungsfunktionen

Zwischen je zwei Klassenversionen einer Klasse besteht eine Propagationskante, falls die Schemaversion, in der die eine Klassenversion liegt, von der Schemaversion, in der die andere Klassenversion liegt, abgeleitet wurde[1]. Diese Kante enthält die Konvertierungsfunktion für die Konvertierung von der Quellklassenversion zur Zielklassenversion. In der Konvertierungsfunktion kann für das Befüllen einer Objektversion aus der Zielschemaversion nur auf Attribute aus der Quellklassenversion zurückgegriffen werden. Abb. 5.2 zeigt eine solche Kante.

Beteiligt sei also eine Klasse c_1 und die beiden Klassenversionen $sv_1.c_1$ und $sv_2.c_1$.

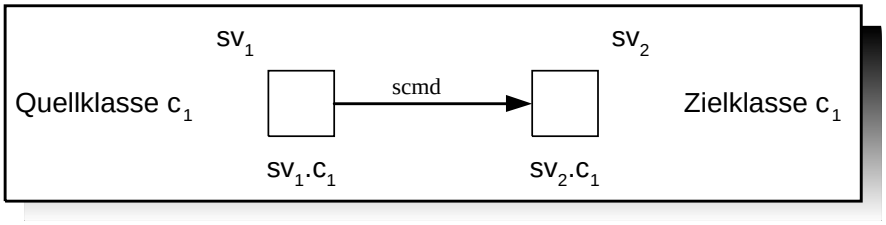

Abbildung 5.2: Propagationskante zwischen zwei Schemaversionen

Diese Propagationskante kann in Vorwärtsrichtung jedes der vier Propagationsflags s, c, m oder d bzw. in Rückwärtsrichtung jedes der drei Propagationsflags c, m oder d enthalten.

Bei einfachen Konvertierungsfunktionen geht eine solche Kante immer von einer Quellklassenversion zu einer Zielklassenversion.

5.6.2 Propagationsgraph für komplexe Konvertierungsfunktionen

Die besondere Problematik für den Propagationsgraphen besteht darin, dass die Konvertierungsfunktionen nicht mehr isoliert zu betrachten sind, sondern durch die Hinzunahme von komplexen Schemaänderungsoperationen Beziehungen zueinander haben. Es stellt sich also die Frage, wo die Propagationsflags angegeben werden müssen und wie sie zu interpretieren sind.

[1] Sollten die beiden Schemaversionen im Ableitungsgraph weiter auseinanderliegen, kann die Propagation in mehrere Teilschritte zerlegt werden. Diese *transitive Propagation* wird in Abschnitt 5.7 beschrieben. Für die Beschreibung der Propagationskanten wird vereinfachend angenommen, dass es sich um zwei direkt voneinander abgeleitete Schemaversionen handelt.

Bei den in dieser Diplomarbeit eingeführten komplexen Schemaänderungsoperationen wird außer auf zwei Objektversionen desselben Objekts noch auf eine oder mehrere zusätzliche Objektversionen eines anderen Objekts zugegriffen. Die zugehörige Konvertierungsfunktion muss also mehr als nur zwei Objektversionen miteinander verbinden und daher auf mehrere Kanten verteilt werden: Der Teil der Konvertierungsfunktion für die Attribute, die aus einer zusätzlichen Objektversion stammen, wird in eine Kante zu eben dieser Objektversion verschoben.

Im Folgenden werden drei Lösungsmöglichkeiten beschrieben, wie die entsprechenden Propagationskanten aussehen können. Dabei wird auf den vereinfachten Fall eingegangen, dass Daten aus genau einer weiteren Quellobjektversion – also Daten aus zwei verschiedenen Objekten – benötigt werden, um die Zielobjektversion mit korrekten Werten füllen zu können. Prinzipiell funktionieren die vorgestellten Möglichkeiten aber auch bei zwei oder mehr zusätzlichen Quellobjektversionen.

5.6.2.1 Von zwei Quellklassenversionen zu einer Zielklassenversion

Zunächst wird die Richtung von zwei Quellklassenversionen zu einer Zielklassenversion gezeigt. Diese Situation kann beispielsweise auftreten, wenn die komplexe Schemaänderungsoperation „Zusammenführen von Klassen über Referenzen“ (s. Abschnitt 4.3.3.4) Verwendung gefunden hat und die Vorwärtsrichtung betrachtet wird.

Beteiligt seien hier also die beiden Klassen c_1 und c_2 und die drei Klassenversionen $sv_1.c_1$, $sv_1.c_2$ und $sv_2.c_1$.

1. **Lösungsmöglichkeit: Mehrere unabhängige Propagationskanten**

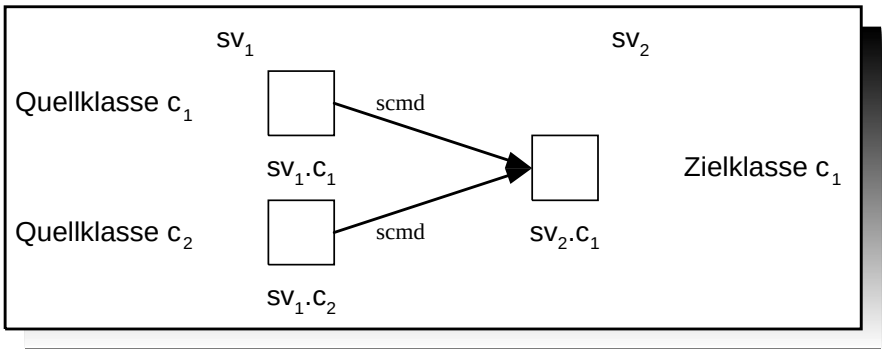

Abbildung 5.3: Zwei unabhängige Propagationskanten zwischen zwei Quellklassen in Schemaversion sv_1 und einer Zielklasse in Schemaversion sv_2

Eine einfache Möglichkeit besteht darin, dass von jeder Quellklassenversion eine Propagationskante zur Zielklassenversion existiert (s. Abb. 5.3). Wenn die Zielobjektversion geschrieben werden soll, werden beide Propagationskanten verwendet, um die entsprechenden Attribute aus den Quellobjektversionen zu erhalten. Ein Beispiel könnte sein, dass in Quellklassenversion $sv_1.c_1$ Adressdaten stehen, in Quellklassenversion $sv_1.c_2$ berufliche Daten und in der Zielklassenversion $sv_2.c_1$ eine Kombination von Attributen wie `Name`, `Strasse`, `Ort`, `Abteilung` und `Gehalt` gewünscht ist.

Da zwei Propagationskanten verwendet werden, kann zu jeder der beiden Kanten unabhängig voneinander bestimmt werden, wie die zugehörigen Propagationsflags aussehen sollen. Die Schwierigkeit bei der Verwendung dieser Lösungsmöglichkeit taucht dann auf, wenn bei den beiden Propagationskanten die Propagationsflags

nicht identisch gesetzt sind, also beispielsweise bei der Kante von Quellklassenversion $sv_1.c_1$ zur Zielklassenversion $sv_2.c_1$ keine Modifikationen weitergegeben werden sollen $(s + c + m - d+)$, bei der anderen aber alle Propagationsflags gesetzt sind $(s + c + m + d+)$. Wenn nun Änderungen in einer der beiden Quellobjektversionen stattgefunden haben und die Zielobjektversion aktualisiert werden soll, würde das bedeuten, dass nur ein Teil der Änderungen (nämlich die aus Quellobjektversion ov_2, da die entsprechende Propagationskante das Modify-Flag gesetzt hat) übertragen werden soll, der andere Teil aber nicht. Das kann zu unvorhersehbaren Ergebnissen führen.

2. **Lösungsmöglichkeit: Eine kombinierte Propagationskante**

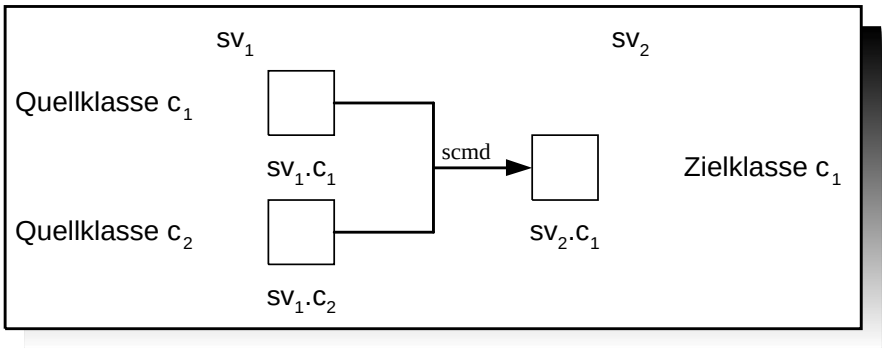

Abbildung 5.4: Kombinierte Propagationskante zwischen zwei Quellklassen in Schemaversion sv_1 und einer Zielklasse in Schemaversion 2

Um die Gefahr widersprüchlicher Propagationsflags zu vermeiden, sollten bei einer komplexen Konvertierung nicht für beide Kanten Propagationsflags angegeben werden können. Dazu liegt der Gedanke nahe, die zwei unabhängigen Kanten zu einer speziellen Kante zu vereinigen, so dass nur ein Satz von Propagationsflags über die Propagation entscheidet.

Wenn nur eine kombinierte Propagationskante zwischen den beiden Quellklassenversionen und der Zielklassenversion existiert, dann enthält diese Kante auch nur einen Satz von Propagationsflags und die in Lösungsmöglichkeit 1 beschriebene Problematik tritt nicht auf: Es wird nur dann eine Propagation durchgeführt, wenn das entsprechende Flag gesetzt ist, und dann aus beiden Quellobjektversionen Daten gelesen.

Allerdings hat auch diese Lösungsmöglichkeit einen Nachteil. Wenn eine der beiden Quellobjektversionen gelöscht wird und das Delete-Flag bei der Propagationskante gesetzt ist, wird die Zielobjektversion ebenfalls gelöscht.

Beispiel 5.6.1 *Als Beispiel diene die Modellierung eines Autos: In Quellklassenversion $sv_1.Auto$ könnte das Auto mit Fabrikat und Werten stehen, in Quellklassenversion $sv_1.Reifen$ der Typ der Reifen stehen. Durch die komplexe Schemaänderungsoperation „Zusammenführen von Klassen über Referenzen“ (s. Abschnitt 4.3.3.4) sei die Zielklassenversion $sv_2.Auto$ erstellt worden, in der alle Attribute der beiden Quellklassenversionen vorkommen (s. Abb. 5.5).*

Es macht wahrscheinlich keinen Sinn, ein komplettes Auto zu löschen, nur weil der Typ seiner Bereifung nicht mehr in der Datenbank steht. Vielmehr sollte man in diesem Falle beim Löschen der Quellklassenversion $sv_1.Reifen$ in der Zielklassenversion $sv_2.Auto$ für das entsprechende Attribut einen NULL-Wert einfügen.

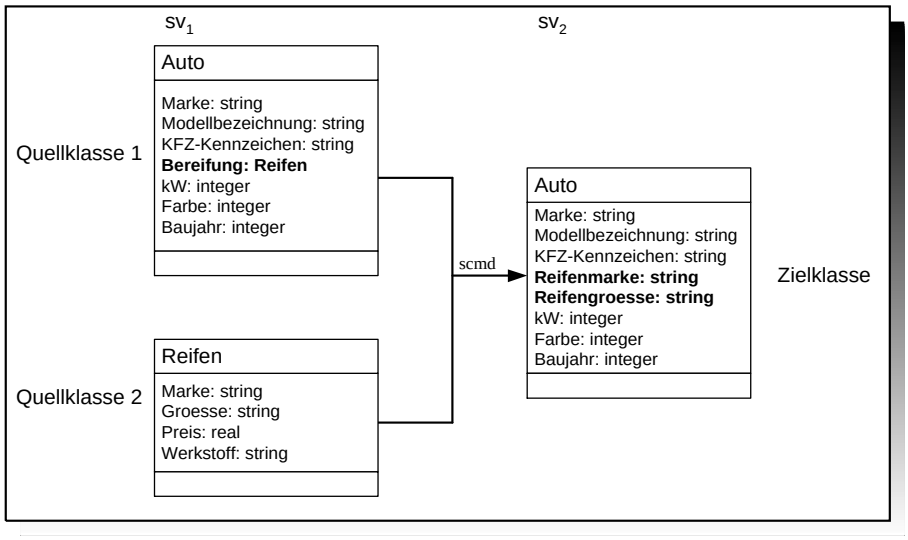

Abbildung 5.5: Quellklassenversionen „Auto“ und „Reifen“ mit Zielklassenversion „Auto“

3. **Lösungsmöglichkeit: Eine Haupt- und eine Neben-Propagationskante**

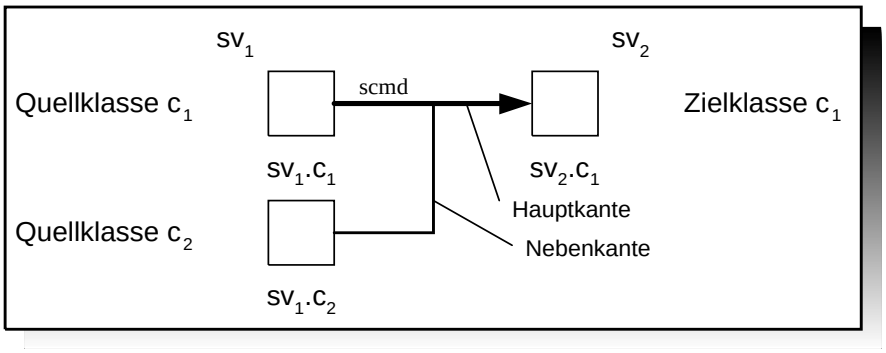

Abbildung 5.6: Haupt- und Neben-Propagationskante zwischen zwei Quellklassen in Schemaversion sv_1 und einer Zielklasse in Schemaversion 2

Verändert man Lösungsmöglichkeit 2 so, dass der Propagation zwischen $sv_1.c_1$ und $sv_2.c_1$ ein größeres Gewicht zugeordnet wird als der durch komplexe Schemaänderungsoperationen zusätzlich auftretenden Klassenversion $sv_1.c_2$, so kann man eine Haupt- und eine Nebenpropagationskanten unterscheiden. Das Resultat ist ein Mechanismus, der es ermöglicht, „wichtige“ und „unwichtige“ Teile zu unterscheiden. Die Propagationsflags sind nur mit der Hauptkante verbunden, d.h. es gibt sie für diese Konvertierung nur an einer Stelle. Auf diese Weise kann am „wichtigsten“ Punkt entschieden werden, welche Löschungen oder Modifikationen propagiert werden sollen.

Die Hauptkante liegt dann immer zwischen den beiden Klassenversionen der Klasse, die auf beiden Seiten der Konvertierung auftaucht. Sie bildet also gewissermaßen die ursprüngliche Propagationskante zwischen zwei Klassenversionen derselben Klasse nach, wie sie bei der Propagation einfacher Schemaänderungen aussieht. Jede zusätzliche Klassenversion einer anderen Klasse, die in die betrachtete Propagation involviert ist, wird durch eine Nebenkante mit der Hauptkante verbunden.

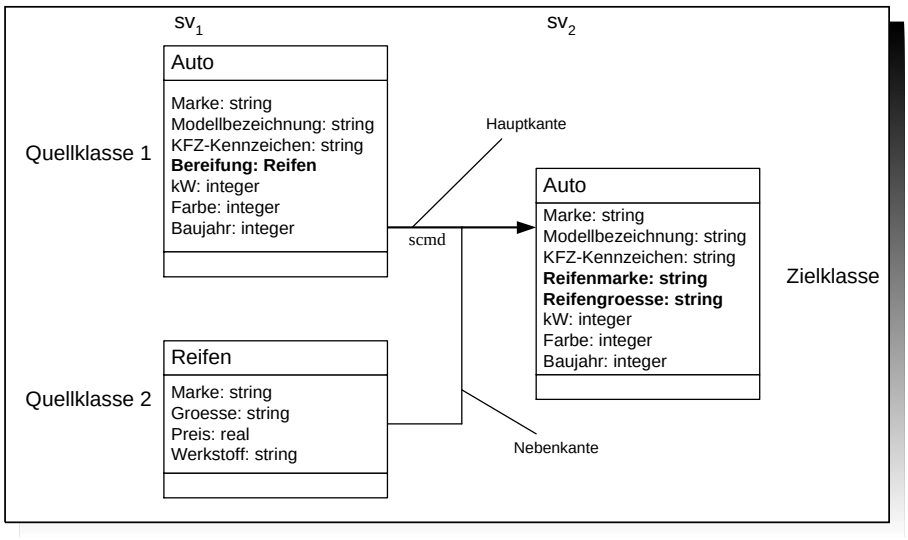

Abbildung 5.7: Hauptpropagationskante zwischen der Quellklassenversion „Auto“ und der Zielklassenversion „Auto“ unter Einbeziehung der Nebenpropagationskante von „Reifen“

Im Beispiel 5.6.1 wurde statt der kombinierten Propagationskante eine Haupt- und eine Nebenkante eingefügt. Auf diese Weise ist klar ersichtlich, welches die „wichtigere“ Information ist (s. Abb. 5.7).

Die Gewichtung der Kanten bewirkt damit, dass beim Löschen der Quellobjektversion $sv_1.Auto$ die Zielobjektversion $sv_2.Auto$ ebenfalls gelöscht wird. Wird die Quellobjektversion $sv_1.Reifen$ gelöscht, wird diese Änderung in der Form propagiert, dass die entsprechenden Felder in der Zielklassenversion mit NULL-Werten belegt werden.

5.6.2.2 Von einer Quellklassenversion zu zwei Zielklassenversionen

Nun wird die Richtung von einer Quellklassenversion zu zwei Zielklassenversionen gezeigt. Diese kann beispielsweise auftreten, wenn die komplexe Schemaänderungsoperation „Aufbrechen von Klassen“ (s. Abschnitt 4.3.3.3) verwendet und die Vorwärtsrichtung betrachtet wird.

1. **Lösungsmöglichkeit: Mehrere unabhängige Propagationskanten**

   Bei der einfachsten Variante existieren von der Quellklassenversion zu jeder der Zielklassenversionen eine Propagationskante (s. Abb. 5.8). Bei verzögerter Propagation kann so bei einem Zugriff auf eine der beiden Zielklassenversionen die Propagation erfolgen. Bei sofortiger Propagation wird die Zustandsänderung in der Quellobjektversion nacheinander in beide Zielobjektversionen propagiert. Die Propagationsflags der beiden Kanten entscheiden darüber, ob die Änderung an die entsprechende Zielobjektversion weitergegeben wird.

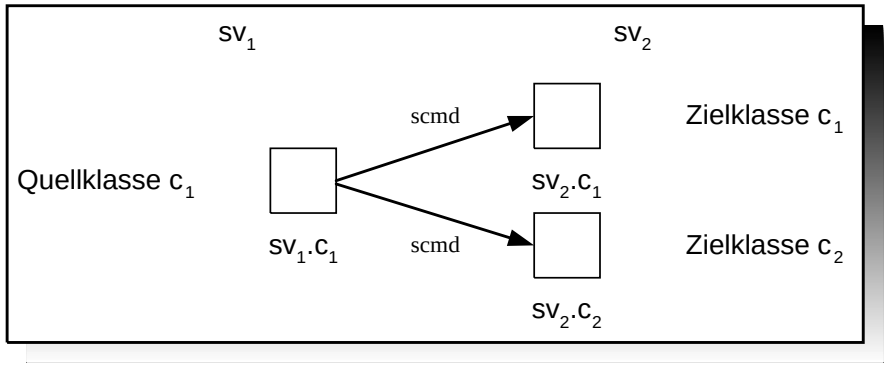

Abbildung 5.8: Zwei unabhängige Propagationskanten zwischen einer Quellklasse in Schemaversion sv_1 und zwei Zielklassen in Schemaversion sv_2

2. **Lösungsmöglichkeit: Eine kombinierte Propagationskante**

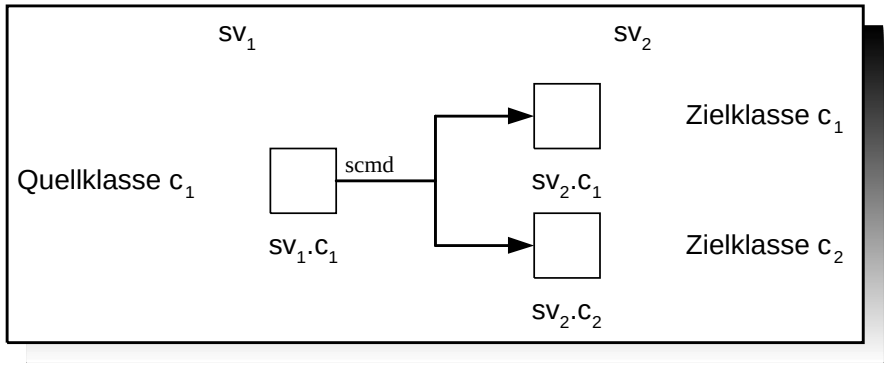

Abbildung 5.9: Kombinierte Propagationskante zwischen einer Quellklasse in Schemaversion sv_1 und zwei Zielklassen in Schemaversion sv_2

Die Verwendung einer kombinierten Propagationskante zwischen einer Quellklasse und zwei Zielklassen (s. Abb. 5.9) führt zu Problemen:
Bei verzögerter Propagation wird gemäß der Kante die Zustandsänderung aus der Quellobjektversion in die entsprechende Zielobjektversion propagiert. Die Schwierigkeit ist, dass die Konvertierungsfunktion für beide Teile in der kombinierten Kante gespeichert ist. Damit bei beliebigen Zugriffen auf eine der beiden Zielobjektversionen immer die nötige Propagation erfolgen kann, muss die Konvertierungsfunktion doppelt gespeichert werden, und zwar für jede der beiden Zielklassenversionen einzeln. Damit läge wieder Lösungsmöglichkeit 1 vor.
Wird mit sofortiger Propagation gearbeitet, taucht das Problem nicht auf, da beide Zielklassenversionen sofort aktualisiert werden.

Durch die Verwendung einer kombinierten Kante kann die Zuordnung von Propagationsflags auch nur für beide Konvertierungen gemeinsam erfolgen. Sollte beispielsweise gewünscht sein, eine Löschung nur zu einer der Zielobjektversionen weiterzugeben, müssen einzelne Propagationskanten erstellt werden.

3. **Lösungsmöglichkeit: Eine Haupt-Propagationskante mit einer Neben-Propagationskante**

Bei der Verwendung von Haupt- und Nebenkanten für eine Propagation von einer Quellklasse zu zwei Zielklassen (s. Abb. 5.10) ist im Vergleich zur umgekehrten Rich-

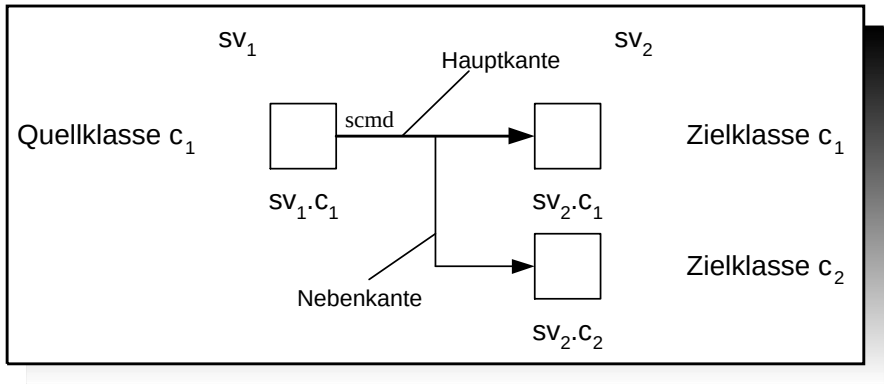

Abbildung 5.10: Haupt- und Neben-Propagationskante zwischen einer Quellklassen in Schemaversion sv_1 und zwei Zielklassen in Schemaversion sv_2

tung nicht mehr gewährleistet, dass die Zielobjektversion an der Hauptkante hängt. Dadurch taucht das Problem auf, dass bei verzögerter Propagation bei einem Zugriff auf die Zielobjektversion $sv_2.c_2$ die Propagationskante, in der die auszuführende Konvertierungsfunktion steht, zwischen zwei anderen Objektversionen (eben denen, die durch die Hauptkante verbunden sind), steht und daher nicht gefunden wird. Die nötige Zustandsänderung wird nicht erkannt und erst beim Zugriff auf die Zielobjektversion $sv_2.c_1$ oder nach einer sofortigen Propagation der korrekte Wert in Zielobjektversion $sv_2.c_2$ übertragen.

Die folgenden Lösungsmöglichkeiten sind denkbar:

- Beim Ändern einer Objektversion, von der eine solche Haupt- und Nebenkante ausgeht, könnte die Änderung sofort (also per sofortiger Propagation) weitergegeben werden.
- Es wäre auch möglich, die nur durch Nebenkanten erreichten Klassen so zu bewerten, dass sie nicht immer den aktuellsten Stand haben müssen. In diesem Fall wäre zu bedenken, dass bei transitiver Propagation (s. Abschnitt 5.7) jede auf dem Propagationsweg dahinter liegende Schemaversion dann auch nur den veralteten Wert erhielte. Insofern ist dieser Lösungsansatz nicht geeignet.
- Eine weitere Möglichkeit ist, bei solchen Konstellationen noch eine zusätzliche einzelne Kante zwischen der Quellklassenversion und der durch die Nebenkante verbundene Zielklassenversion einzufügen, in der eine Kopie der Konvertierungsfunktion oder für bessere Wartbarkeit sogar eine Referenz auf die Konvertierungsfunktion in der Hauptkante liegt.
- Es ist auch vorstellbar, in jeder Klasse eine Liste der mit ihr verbundenen Propagationskanten zu speichern. Auf diese Weise ist bei sofortiger Propagation zudem noch ein schneller Zugriff auf die Liste der durchzuführenden Propagationen möglich.

Die Verwendung von Propagationsflags ist wie bei Lösungsmöglichkeit 2 nur eingeschränkt möglich: Ist beispielsweise das Delete-Flag gesetzt, werden nach einer Löschung der Quellobjektversion automatisch beide Zielobjektversionen entfernt. Hier könnte noch die systemweite Einschränkung eingebaut werden, dass eine Löschung entlang einer Nebenkante nicht durchgeführt wird. In diesem Falle müsste also notfalls die Zielobjektversion $sv_2.c_2$ explizit gelöscht werden.

Bei der Konzeption und bei der Implementierung der Propagationskanten wurde sich im Rahmen dieser Diplomarbeit für Lösungsmöglichkeit 3 entschieden. Die deutlichen Vorteile dieser Variante in der Richtung von mehreren Quellklassenversionen zu einer Zielklassenversion bieten eine solche Wahl an.

5.6.3 Zurücklaufende Propagation

Ein weiterer interessanter Punkt bei der Verwendung von Propagationskanten soll hier noch beleuchtet werden: Die mögliche *zurücklaufende Propagation*. Durch Änderungen oder Löschung von Attributen kann implizit ein anderes Objekt mitverändert oder mitgelöscht werden, obwohl dies vielleicht nicht gewünscht ist (s. Abb. 5.11).

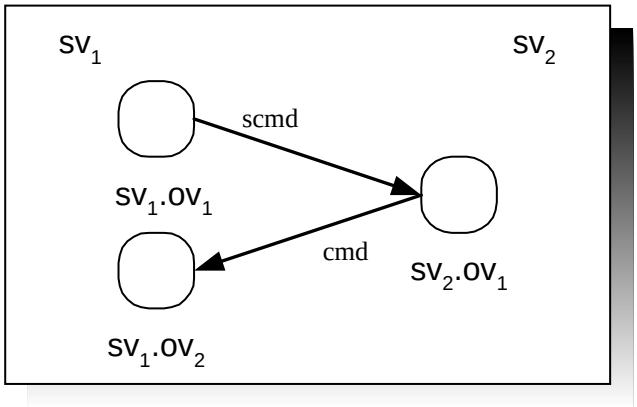

Abbildung 5.11: Eine Objektversion $sv_1.ov_1$ wird gelöscht. Wird dann $sv_1.ov_2$ automatisch mitgelöscht?

Gemäß der Propagationsflags wird beim Löschen der Objektversion $sv_1.ov_1$ in sv_1 ebenfalls die Objektversion in sv_2 gelöscht, da beide durch eine Propagationskante verbunden sind. Ist bei der Propagationskante von $sv_2.ov_1$ nach $sv_1.ov_2$ ebenfalls das Delete-Flag gesetzt, müsste nach der Löschung der Objektversion $sv_2.ov_1$ die Objektversion $sv_1.ov_2$ ebenfalls gelöscht werden.

Der Effekt wäre, dass man durch Löschen der Objektversion $sv_1.ov_1$ implizit auch die Objektversion $sv_1.ov_2$ löscht, was evtl. nicht beabsichtigt und auf den ersten Blick nicht ersichtlich wäre.
In der Konzeption der Propagationskanten wird dieses Problem dadurch umgangen, dass die rückläufige Propagation nicht durchgeführt wird. Wenn eine Änderung in einer Schemaversion auftritt, dann breitet sie sich durch die Propagation nur in andere Schemaversionen aus, aber jede Schemaversion, die bereits aktualisiert ist, wird im Zusammenhang mit dieser Änderung nicht noch einmal erneut verändert. Für die beschriebene zurücklaufende Löschung bedeutet das für den Schemaentwickler, dass er explizit für beide Quellobjektversionen ein `delete`-Statement angeben muss, falls er wirklich beide Quellobjektversionen löschen will.

Bewertung:
Dadurch zeigt sich die Schemaversion sv_2 wieder transparent, d.h. genau so, als wenn diese Schemaversion die einzige wäre – was beabsichtigt war.

5.7 Transitive Propagation

Es kann vorkommen, dass zwei betrachtete Objektversionen o_j und o_k nicht in direkt voneinander abgeleiteten Schemaversionen liegen und damit nicht durch eine direkte Konvertierungsfunktion verbunden sind (s. Abb. 5.12).

Sie müssen nicht von der gleichen Schemaversion abgeleitet sein, d.h. im Ableitungsbaum in gleicher Höhe stehen, sondern der im folgenden beschriebene Fall gilt auch, wenn sv_i von sv_j abgeleitet wäre.
Eine Propagation kann über mehrere Schritte gehen, zur Vereinfachung wurde hier der Fall angenommen, dass nur zwei Propagationsschritte auszuführen sind.

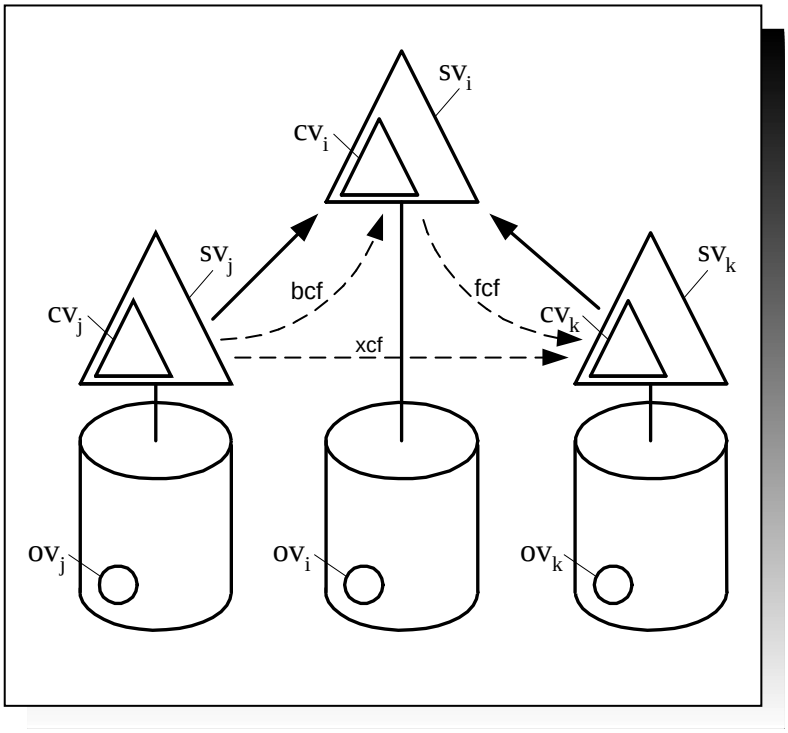

Abbildung 5.12: Transitive Propagation: Eine Objektversion ov_j in Schemaversion sv_j soll in Schemaversion sv_k propagiert werden. Dazu muss sie zunächst in ov_i in der Schemaversion sv_i über die Rückwärtskonvertierungsfunktion ($bcf_{i \leftarrow j}$) und dann von ov_i nach ov_k über die Vorwärtskonvertierungsfunktion ($fcf_{k \leftarrow i}$) konvertiert werden. Alternativ kann die Einführung von direkten Verbindungen über Extrakonvertierungsfunktionen ($xcf_{k \leftarrow j}$) hilfreich sein.

In diesem Fall ist es notwendig, die Propagation in mehreren Schritten vorzunehmen, d.h. die aktuellen Daten erst von sv_k nach sv_i und dann von sv_i nach sv_j zu konvertieren. Dieser Vorgang kann in gleicher Weise auf mehr als zwei Schritte erweitert werden. [Lau00] nennt dies eine *transitive Propagation*.

Bei der transitiven Propagation kann es vorkommen, dass ein Attribut, das in den Klassenversionen $sv_k.c_m$ und $sv_j.c_m$ vorhanden ist, also konvertiert werden muss, in der auf dem Propagationsweg dazwischenliegenden Klassenversion $sv_i.c_m$ nicht existiert. Dadurch wird es auf dem Konvertierungsweg gelöscht und in $sv_j.c_m$ mit einem Nullwert belegt – es wird ja gewissermaßen bei der Konvertierung von sv_i nach sv_j neu angelegt.

```
class cvi
 type tuple (Name: string)
 end;

class cvj inherit cvi
 type tuple (Telefon: string)

fcf
{
  new.Name = old.Name
  new.Telefon = "0"
}
bcf
{
  new.Name = old.Name
}

class cvk inherit cvi
 type tuple (Telefon: string)

fcf
{
  new.Name = old.Name
  new.Telefon = "0"
}
bcf
{
  new.Name = old.Name
}
```

Abbildung 5.13: Drei Klassen in ODL-Syntax, die Klasse cvi besitzt kein Attribut „Telefon“.

Eine Möglichkeit, diese Problematik zu umgehen, ist die Einführung von *Extrakonvertierungsfunktionen* (xcf, s. Abb. 5.12), die zusätzlich zwischen verschiedenen, in der Ableitungshierarchie weiter auseinanderliegenden Schemaversionen eingefügt werden können. Auf diese Weise kann man zum einen den oben beschriebenen Datenverlust-Effekt verhindern, zum anderen aber auch „Abkürzungen“ für die Propagation einfügen. Ein Nebeneffekt ist auch, dass die Propagation dadurch beschleunigt wird. Extrakonvertierungsfunktionen werden in [Eig97] eingeführt und genauer analysiert.

5.8 Zusammenfassung und Bewertung

In diesem Kapitel wurde die Propagation als Mechanismus zur Weitergabe von Informationen über Zustandsänderungen in Objektversionen beschrieben. Dabei wurde die von [Eig97] vorgestellte Propagation von einfachen Schemaänderungen auf die in dieser Diplomarbeit entwickelten komplexen Schemaänderungen erweitert. Für die notwendige Erweiterung der Propagationskanten wurden drei Lösungsmöglichkeiten vorgestellt und diejenige für die Implementierung ausgewählt, die eine Haupt- und eine Nebenkante einführt. Diese stellt die beste der vorgestellten Lösungen dar, da Probleme bei der Verwendung von verschiedenen Belegungen der Propagationsflags vermieden werden. Danach wurde die Erweiterung der Propagation in einem Schritt zwischen zwei benachbarten Schemaversionen auf die transitive Propagation zwischen nicht direkt benachbarten Schemaversionen beschrieben.

Die Propagationssteuerung konnte auf komplexe Schemaänderungen erweitert werden. Es war allerdings notwendig, Einschränkungen in der Semantik der Propagationsflags vorzunehmen, damit der Mechanismus verständlich und handhabbar bleibt. Daher wurde das Konzept der Haupt- und Neben-Propagationskanten eingeführt. Somit werden wichtige Schemaänderungen auf Objektebene unterstützt. Bei der für die praktische Einsetzbarkeit wichtigen verzögerten Ausführung wurde festgestellt, dass diese bei der Verwendung von komplexen Schemaänderungsoperationen zum Teil eingeschränkt werden muss: Teile der Datenbank müssen evtl. sofort propagiert werden. Hier könnte jedoch durch genauere Untersuchung der in Abschnitt 5.4 erwähnten Ansatzpunkte ein erhebliches Verbesserungspotenzial ausgeschöpft werden.

Kapitel 6

Eine Sprache für Konvertierungsfunktionen

Um die gewünschte Propagation von Zustandsänderungen der Datenobjektversionen zu spezifizieren, benötigt man eine Sprache, in der die Konvertierungsfunktion angegeben werden können. In diesem Kapitel wird erläutert, welche Funktionalitäten und Eigenschaften eine solche Sprache haben sollte und schließlich eine Konvertierungssprache vorgestellt, die die spezifizierten Anforderungen erfüllt. Die Entwicklung dieser Sprache war ein weiterer zentraler Bestandteil dieser Diplomarbeit.

6.1 Funktionale Ziele

Die entworfene Sprache sollte sowohl einen bestimmten Funktionsumfang haben als auch nach einer Reihe von Konzeptionsgrundsätzen (s. Abschnitt 6.2) erstellt werden.

Aus den für die in Abschnitt 4.3 vorgestellten typischen Schemaänderungsoperationen nötigen Funktionen und einigen weiterführenden Überlegungen wurde die folgende Liste von Forderungen an den Funktionsumfang der Konvertierungssprache erstellt:

F1 **Es muss ein schreibender Zugriff auf die neue Objektversion möglich sein.**
Eine Konvertierungsfunktion soll Daten, die sie aus einer anderen Objektversion gelesen, berechnet oder als Konstante vorgegeben hat, in eine Objektversion schreiben. Daher muss als Minimalanforderung diese Funktionalität gegeben sein.

F2 **Es muss ein lesender Zugriff auf eine von der Zielobjektversion verschiedene Version des zu konvertierenden Objekts möglich sein.**
Um Werte in eine Zielobjektversion propagieren zu können, muss zumindest eine Quellobjektversion gelesen werden können. Sogar schon für die Identitätsfunktion, die einen Wert einfach in eine neue Objektversion übernimmt, ist diese Funktionalität notwendig.

F3 **Es muss ein lesender Zugriff auf Attribute einer anderen Objektversion eines anderen Objekts möglich sein.**
Da selbst komplexe Konvertierungsfunktionen nur von einer Schemaversion ausgehen, kann die folgende Einschränkung für Konvertierungsfunktionen gemacht werden, ohne die Allgemeinheit der Konvertierungssprache einzuschränken: Wenn eine Konvertierung von Daten aus den beiden Objektversionen $sv_i.c_1$ und $sv_j.c_2$ in die Zielobjektversion $sv_k.c_1$ erfolgen soll (also die Klasse c_1 die konvertierte Klasse ist),

muss die zusätzliche Klassenversion $sv_j.c_2$ aus derselben Schemaversion wie $sv_i.c_1$ stammen. Demnach gilt hier die Forderung:
Wenn $sv_i.c_1$ und $sv_j.c_2$ Quellobjektversionen für die Zielobjektversion $sv_k.c_1$, sind, folgt daraus $j = i$.

F4 **Es muss ein Zugriff auf mehrere Werte eines Attributs mit komplexem Typ möglich sein.**
Hat ein Attribut einen komplexen Typ (`set`, `list`, `tuple`, `bag`, `graph` etc.), muss ein Konstrukt den Zugriff auf jeden einzelnen Wert ermöglichen, der in diesem Attribut gespeichert ist. Deren Anzahl ist erst zur Laufzeit bekannt, dann aber im Moment der Ausführung konstant.

F5 **Es muss eine Operation geben, um Werte in Attribute mit komplexem Typ zu schreiben.**
Soll in Attribute mit komplexem Typ geschrieben werden, ist sicherzustellen, dass eine entsprechende Operation existiert. Beispielsweise ist es für das Überschreiben des vierten Werts eines Attributs vom komplexen Typ `list(integer)` nötig, die Liste einzulesen, abzuändern und die veränderte Liste in das Attribut zurückzuschreiben.

F6 **Es muss lokale Hilfsvariablen geben.**
Für Zwischenberechnungen oder Zwischenspeicherung von Suchergebnissen müssen lokale Variablen benutzt werden können. Deren Gültigkeitsbereich (engl. scope) soll genau der ausgeführten Konvertierungsfunktion entsprechen.

F7 **Es muss eine Operation geben, um globale Konstanten lesen zu können.**
Wenn globale Konstanten wie beispielsweise π oder der Umrechnungskurs von DM zu Euro von außen gesetzt worden sind, sollen sie innerhalb von Konvertierungsfunktionen gelesen werden können. Auf diese Weise können systemweite Konstanten in Konvertierungsfunktionen verwendet werden.

F8 **Die bedingte Ausführung von Operationen muss möglich sein.**
Für Berechnungen innerhalb eines Objekts oder für fallabhängige benutzerdefinierte Konvertierungsfunktionen muss es ein `if`-Konstrukt geben.

F9 **Es muss Konstruktoren und Destruktoren für Objekte geben.**
Durch die Einführung der komplexen Schemaänderungen kann es vorkommen, dass auf Objektebene neue Objekte erzeugt werden müssen – dies kann beispielsweise durch die Ausführung der komplexen Schemaänderungsoperation „Aufbrechen von Klassen“ (s. Abschnitt 4.3.3.3) geschehen. Entsprechend wird auch die Existenz eines Destruktors zum Löschen von nicht mehr benötigten Objekten in der Zielschemaversion gefordert.

F10 **Es muss eine Suchfunktion geben.**
Falls auf eine Quellobjektversion zugegriffen werden soll, deren Referenz nicht vorliegt, muss sie gesucht werden können.

F11 **Es muss eine Existenzabfragefunktion geben.**
Falls ein bereits existierendes anderes Objekt referenziert werden soll, muss es zunächst gesucht werden. Falls die Suche erfolglos ist, ist ein neues Objekt anzulegen, auf das dann die Referenz zeigt. Für die Konvertierungsfunktion gibt diese Operation eine Referenz auf das gewünschte Objekt zurück, unabhängig davon, ob es schon existiert hat oder neu erzeugt wurde.

Außer den notwendigen Anforderungen an die Konvertierungssprache gibt es auch noch Möglichkeiten, die Funktionalität zu erweitern, um dem Programmierer ein mächtigeres Hilfsmittel zur Verfügung zu stellen.

F12 **Es soll ein Konstrukt geben, um Schleifen mit konstanter Anzahl von Wiederholungen ausführen zu können.**
Um die Mächtigkeit der entworfenen Sprache zu erhöhen, soll die Ausführung von Schleifen möglich sein. Die Anzahl der Wiederholungen soll zur Kompilierzeit schon bekannt sein.

6.2 Konzeptionelle Ziele

Im vorangegangenen Abschnitt wurden funktionelle Ziele aufgeführt, die die Sprache erfüllen muss. Hier sollen nun die konzeptionellen Ziele genannt werden, nach denen die Sprache entworfen werden soll.

Eine Sprache für Konvertierungsfunktionen sollte bestimmte Eigenschaften aufweisen. Im Folgenden werden einige Eigenschaften aufgeführt, die beim Entwurf der Konvertierungssprache betrachtet wurden. In [Nic75] werden Vorschläge gemacht, wie eine neu zu entwerfende Programmiersprache aufgebaut sein sollte. Diese sind im Einzelnen:

- Ease-Of-Use (Einfache Verwendbarkeit)
  Die Sprache sollte dem Anwender die Möglichkeit geben, den gewünschten Sachverhalt möglichst einfach auszudrücken. Die Sprache sollte auf möglichst viele der potenziellen Benutzer natürlich wirken.
- High-Level-Design
  Die Sprache sollte mächtig sein, d.h. die Beschreibung eines Problems sollte auf der Abstraktionsebene der Anwendung erfolgen und möglichst wenig Schreibarbeit erfordern.
- Ease-Of-Debugging (Einfache Fehlersuche)
  Die Sprache sollte so aufgebaut sein, dass Fehler möglichst früh auffallen.
- Ease-Of-Documentation (Einfache Dokumentierbarkeit)
  Die Sprache sollte Kommentare, beliebige Einrückungen und „sprechende“ Namen für Variablen und Objekte unterstützen. Dies steigert die Lesbarkeit auch für andere (menschliche Leser) des geschriebenen Codes.
- Transferability (Übertragbarkeit)
  Die in der entwickelten Sprache geschriebenen Anweisungen sollten problemlos auf andere Computersysteme und -architekturen übertragbar sein.

Für die hier zu entwickelnde Konvertierungssprache sind nicht alle Punkte gleichermaßen wichtig – die Übertragbarkeit entfällt, da die Konvertierungssprache nur mit COAST benutzt werden wird.

Daher wurde die obige Liste entsprechend abgeändert und um eigene Punkte erweitert und verfeinert. Die Konzeption der Konvertierungssprache erfolgt nach den folgenden Grundsätzen:

K1 **Die Sprache sollte einfach sein.**
Für den Schemaentwickler, der die Konvertierungssprache nutzt, sollte die Sprache

leicht zu verstehen sein. Idealerweise ähnelt die Sprache daher einer Programmiersprache oder einer gesprochenen Sprache.

K2 **Die Sprache sollte auch Operationen hohen Abstraktionsniveaus anbieten.**
Die Aufgaben der Konvertierungssprache sind meist so feingranular, dass eine Einführung von zusätzlichen Befehlen, die mehrere Befehle ersetzen („syntaktischer Zucker“) kaum notwendig sein wird. Trotzdem wird untersucht, ob an einzelnen Stellen die Vorgabe der Mächtigkeit erfüllt werden kann.

K3 **Die Sprache sollte leicht zu debuggen sein.**
Beim Aufbau der Sprache sollte auf einheitliche Syntax, durchgehend verständliche Bezeichnungen und auch eine optische Wiedererkennbarkeit von typischem Code geachtet werden. Das erhöht die Erkennbarkeit von Tipp- und Denkfehlern bei der Programmierung.

K4 **Die Sprache sollte Eigenschaften aufweisen, die die Lesbarkeit erhöhen.**
Um in der Konvertierungssprache geschriebene Abschnitte optisch so aufbereiten zu können, dass auch ein anderer Benutzer den semantischen Zweck des Geschriebenen leicht nachvollziehen kann, sind einfache Mittel wie mögliche Einrückungen der Zeilen zur besseren Strukturierung anzubieten.

K5 **Es sollten Leerzeilen im Code eingefügt werden können.**
Dies trägt ebenfalls zur optischen Strukturierung des Codes bei.

K6 **Der Programmierer sollte Kommentare verwenden dürfen.**
Kommentare sind sinnvoll, um komplexe Sachverhalte zusätzlich textuell beschreiben zu können.

K7 **Variablen sollten „sprechende“ Namen haben.**
Der Programmierer sollte die Möglichkeit haben, die Bedeutung bestimmter Variablen schon durch ihren Namen ausdrücken zu können.

K8 **Die Sprache sollte unabhängig von der Umgebung des Programmierers sein.**
Die Konvertierungsfunktionen sollten mit jedem beliebigen Editor in reiner ASCII-Darstellung erstellbar sein. (Für den Betrieb ist dann natürlich eine Rechnerumgebung notwendig, auf der COAST läuft.)

K9 **Die Sprache muss so aufgebaut sein, dass keine Benutzereingabe nötig ist und keine sonstigen externen Ereignisse abgefragt werden können.**
Da die Konvertierungssprache für die Spezifikation der Propagation gedacht ist, die immer verzögert ablaufen können soll, darf die Sprache keine Konstrukte anbieten, die eine Benutzereingabe nötig machen. Beispiele wären Eingabedialoge oder Fenster mit Meldungen, die nur durch Klicken auf „OK“ zu schließen sind.

K10 **Die Sprache muss terminiert sein.**
In [Bab90] wird dieser Begriff folgendermaßen definiert:

„*Eine Anweisung (oder auch ein Programmteil oder Programm)* terminiert*, wenn sie in endlicher Zeit ohne Laufzeitfehler und ohne Compilierungsfehler zu Ende – d.h. mit einem definierten Ergebnis – ausgeführt wird.*“

Es sollte bei der Konzeption der Konvertierungssprache darauf geachtet werden, dass keine Konstrukte möglich sind, die nicht terminieren. Typische Beispiele wären `while`-Schleifen, die eine Bedingung haben könnten, die nie falsch wird.

Die letzten beiden Punkte K9 und K10 sind Besonderheiten, die durch die verzögerte Propagation bedingt sind. Die verzögerte Propagation ist nur dann anwendbar, wenn gewährleistet ist, dass sie dasselbe Ergebnis bringt wie eine sofortige Propagation. Würde das System bei der Ausführung von Konvertierungsfunktionen Fragen an den Benutzer stellen, wäre das im sofortigen Fall möglich, im verzögerten Fall wäre der Benutzer aber nicht mehr verfügbar. Ebenso muss der Zugriff auf externe Ereignisse des Systems wie der aktuelle freie Speicherplatz oder die Systemzeit verboten werden, da sonst verschiedene Ergebnisse bei sofortiger oder verzögerter Ausführung errechnet werden könnten.

Im Folgenden werden die Befehle für einfache und komplexe Konvertierungsfunktionen vorgestellt. Dabei werden sowohl die funktionalen Forderungen F1 bis F12 als auch die konzeptionellen Forderungen K1 bis K10 berücksichtigt.

6.3 Einfache Konvertierungsfunktionen

Die Operationen, die nötig sind, um einfache Schemaänderungen auf die Objektebene zu übertragen, die also nur eine Quellobjektversion und eine Zielobjektversion desselben Objekts verwenden, lauten wie folgt:

6.3.1 Schreibzugriff auf Zielattribute

Die errechneten Werte müssen in die Zielattribute geschrieben werden können. Dazu ist es notwendig, die gewünschten Zielattribute spezifizieren zu können. Da aufgrund der Integritätsbedingungen innerhalb einer Klasse jeder Attributsname nur einmal vorkommen kann (s. Invarianten in [Dol99]), braucht nur die mögliche Zweideutigkeit zwischen Quell- und Zielobjektversion betrachtet zu werden.
Die eindeutige Bezeichnung eines Attributs aus der Zielobjektversion lautet:

`new.`Attribut.

Da Konvertierungsoperationen immer nur in eine oder mehrere Objektversionen derselben Schemaversion schreiben, können (abgesehen von lokalen Variablen, s. Abschnitt 6.3.5) nur Zielattribute auf der linken Seite einer Zuweisung stehen. Die Zuweisung und damit der Schreibzugriff wird über das Gleichheitszeichen gesteuert.
Eine Zuweisung eines Wertes in ein Attribut der Zielobjektversion lautet:

`new.`Attribut `=` Wert.

Beispiel 6.3.1 *Soll in ein Attribut* `Bevoelkerung` *vom Typ* `integer` *der Wert 100.000 geschrieben werden, ist dies durch den folgenden Aufruf möglich:*

```
new.Bevoelkerung = 100000
```

6.3.2 Lesezugriff auf Quellattribute

In den meisten Fällen stammt der in ein Zielattribut zu schreibende Wert aus einer anderen Objektversion und ist ggf. noch zu verändern. Um auf ein Attribut der Quellobjektversion desselben Objekts zuzugreifen, zu dem auch die Zielobjektversion gehört, ist zur Unterscheidung der Präfix `old.` vorzusetzen. Ein Lesezugriff eines Attributs aus der Quellobjektversion lautet:

`old.`Attribut

Beispiel 6.3.2 *Soll in ein Attribut* `Hauptstadt` *vom Typ* `string` *der Wert desselben Attributs aus der Quellschemaversion übernommen werden (Identitätsfunktion), geschieht dies durch den folgenden Aufruf:*

```
new.Hauptstadt = old.Hauptstadt
```

6.3.3 Zugriff auf Variablen mit komplexen Typen

Hat ein Attribut einen komplexen Typ (`tuple`, `set`, `list`, ...), muss auch auf die einzelnen Werte des Attributs zugegriffen werden können.
Der Zugriff auf einen Wert eines Attributs mit komplexem Typ erfolgt über den Aufruf:

Attribut.Unterattribut

Um einem Attribut von komplexem Typ einen weiteren Wert zu übergeben, kann einfach eine Zuweisung (s. Abschnitt 6.3.2) erfolgen. Der Wert wird dann in die Menge der Werte des Attributs übernommen bzw. an die Liste der Werte angehängt, in den Graphen der Werte eingefügt etc.

Sollen alle Werte aus einem Attribut vom Typ `set` gelesen und verarbeitet werden, ist dies mit einem Schleifenkonstrukt namens `foreach` möglich. Durch diesen Befehl werden alle Werte des Attributs nacheinander ausgegeben und können so beispielsweise in ein anderes Attribut mit komplexem Typ umkopiert werden. Die Operation, die mit dem jeweils gefundenen Wert des Attributs durchgeführt werden soll, ist im Argument des Befehls, durch Komma getrennt, anzugeben. Diese Operation kann auch leer sein, wenn beispielsweise nur ein Umkopieren der Werte gewünscht ist. Die Syntax lautet damit:

`foreach(`Attribut`,`Operationen`)`

Beispiel 6.3.3 *Sollen die Werte eines Attributs* `Alle_Teilnehmer` *vom komplexen Typ* `set(string)` *aus der Quellobjektversion in ein Attribut* `Teilnehmerliste` *vom komplexen Typ* `list(string)` *in der Zielobjektversion übernommen werden, ist der folgende Befehl zu verwenden:*

```
new.Teilnehmerliste = foreach(old.Alle_Teilnehmer, )
```

6.3.4 Berechnung von Werten

Innerhalb von Konvertierungsfunktionen können Werte aus Quellattributen verändert und dann in das jeweilige Zielattribut geschrieben werden. Beim Entwurf der Konvertierungssprache wurden die folgenden Operationen eingeschlossen, weitere Operationen sind natürlich auch denkbar.

Real- und Integertypen:
Die Grundrechenarten Addition, Subtraktion, Multiplikation und Division sollten dem Programmierer zur Verfügung stehen. Diese sind einfach durch ihr mathematisches Symbol darzustellen: `+ - * /`. Konstanten können einfach als Ziffernfolge eingegeben werden.

Stringtypen:
Die Bearbeitung von Strings ist auch ein wichtiger Punkt beim Entwurf von Konvertierungsfunktionen. Hier wurden die folgenden Operationen entworfen:

- `left(`String`,`n`)` gibt den linken Teil des Strings bis einschließlich der n-ten Stelle aus
- `right(`String`,`n`)` gibt den rechten Teil des Strings ab der n-ten Stelle aus

- `concat(String,String)` fügt zwei Strings aneinander
- `length(String)` gibt die Länge eines Strings aus

Beispiel 6.3.4 *Soll ein Attribut* `Preis_Euro` *vom Typ* `real` *mit dem Wert des Quellattributs* `Preis_DM` *gefüllt werden, dabei aber die Euro-Konstante aufmultipliziert werden, erfolgt das mittels dieser Zeile:*

```
new.Preis_Euro = old.Preis_DM * 1.95583
```

Soll in ein Attribut `PLZ` *vom Typ* `string` *der linken Teil eines Quellattributs* `Ort` *bis zur fünften Stelle geschrieben werden, ist die folgende Operation möglich:*

```
new.PLZ = left(old.Ort, 5)
```

6.3.5 Lokale Variablen

Falls ein Objekt gesucht werden muss, ist es sinnvoll, das Suchergebnis in einer lokalen Variable speichern zu können, falls es in der Konvertierungsfunktion mehrfach benötigt wird. Auch ist eine Verwendung von lokalen Variablen zur übersichtlicheren Darstellung eines Rechenwegs für komplexere Berechnungen sinnvoll.
Lokale Variablen können jeden beliebigen Namen haben, der nicht mit einer Ziffer beginnt (um eine Verwechslungsgefahr mit Konstanten auszuschliessen) und die keinen Punkt enthalten (um eine Verwechslungsgefahr mit einem Attribut, dem der Präfix `new.` oder `old.` vorangestellt wurde oder das von komplexem Typ ist, zu vermeiden). Sie werden mit dem ersten Auftreten in einer Konvertierungsfunktion definiert. In den Beispielen haben lokale Variablen der Übersichtlichkeit halber immer den Präfix `_tmp_`.

Beispiel 6.3.5 *Soll der Wert eines Attributs* `Preis` *vom Typ* `real` *in das Zielattribut* `Angebotspreis_Euro` *vom selben Typ konvertiert werden, wobei zum einen die Umrechnung des Preises von DM nach Euro und zum anderen eine Preisreduzierung um 25 Prozent eingerechnet werden soll, ist der Aufruf*

```
_tmp_Preis_Euro = old.Preis / 1.95583
_tmp_Preis_Euro_Angebot = _tmp_Preis_Euro * 0.75
new.Angebotspreis_Euro = _tmp_Preis_Euro_Angebot
```

sicher aussagekräftiger als die Zeile

```
new.Angebotspreis_Euro = old.Preis * 0.38347
```

6.3.6 Globale Konstanten

Es ist hilfreich, von außen (beispielsweise in Umgebungsvariablen der Unix-Shell, im Hauptmenü von COAST, etc.) Konstanten vorgeben zu können, die in jeder Konvertierungsfunktion gelesen werden können. So wird vermieden, diese Konstanten in jeder Konvertierungsfunktion beispielsweise in lokalen Variablen setzen zu müssen. Auch ist dadurch gesichert, dass jede Konvertierungsfunktion denselben Wert verwendet. Um eine Konstante leichter zu erkennen, sollte ihr Name in Großbuchstaben geschrieben sein.

Beispiel 6.3.6 *Soll der Umrechnungsfaktor von DM nach Euro, der von außen in eine globale Konstante* `EUROKONSTANTE` *geschrieben wurde, verwendet werden, kann der Zugriff etwa so erfolgen:*

```
new.Preis = old.Preis / EUROKONSTANTE
```

6.3.7 Kommentare

Die Verwendung von Kommentaren erhöht das Verständnis der geschriebenen Konvertierungsfunktion. Kommentare beginnen mit `/*` und enden mit `*/`.

Beispiel 6.3.7 *Soll das Beispiel aus dem Abschnitt 6.3.5 kommentiert werden, kann das etwa so erfolgen:*

```
/* Umrechnen des alten Preises in Euro */
_tmp_Preis_Euro = old.Preis / 1.95583
/* Vergeben von 25 Prozent Rabbatt */
_tmp_Preis_Euro_Angebot = _tmp_Preis_Euro * 0.75
/*
Schreiben des neuen Attributswertes
in das Zielattribut
*/
new.Angebotspreis_Euro = _tmp_Preis_Euro_Angebot
```

6.3.8 Bedingungen

Soll die Abarbeitung von Ausdrücken abhängig von den Werten verschiedener Variablen oder Attribute durchgeführt werden, sind entsprechende Bedingungen zu formulieren und diese mit dem `if`-Operator zu überprüfen. Zur Formulierung der Bedingungen können die Vergleichsoperatoren `=`,`<` und `>` benutzt werden. Eine Bedingung wird in der Konvertierungssprache mit dem folgenden Konstrukt überprüft:

`if`(Bedingung,Konsequenz,Alternative)

Die Alternative kann auch leer bleiben, falls bei Nichterfüllung der Bedingung keine weitere Operation erfolgen soll. Die Konsequenz gibt dann den entsprechenden Rückgabewert an.

Beispiel 6.3.8 *Sollen alle die Werte eines Attributs* `Geburtsdaten` *vom Typ* `date`*, die ein Datum nach dem 01.01.1988 aufweisen, in ein Zielattribut* `Kinder` *desselben Typs übernommen werden, kann dies über den folgenden Aufruf erfolgen:*

```
new.Kinder = foreach(old.Geburtsdaten,
     if(old.Geburtsdaten > 01.01.1988, old.Geburtsdaten, ))
```

6.3.9 Schleifen

In vielen Fällen kann der Programmierer viel Zeit sparen, wenn er bestimmte Vorgänge, die mehrfach hintereinander erfolgen sollen, in Schleifen organisiert. Sollten Schleifen angeboten werden, birgt dies aber auch eine Gefahr, denn es kann dann vorkommen, dass der Programmierer einen Fehler macht und beispielsweise eine Schleife erzeugt, die nie beendet wird.
Um aber Situationen zu umgehen, bei denen die Schleife nicht terminiert (Konzeptionsgrundsatz K10), ist die Verwendung von Schleifen auf eine konstante Zahl von Durchläufen festgelegt. Diese Schleifendurchläufe können dann ausgerollt und sequenziell durch das System ausgeführt werden.
Es wurde also eine Einschränkung vorgenommen: Schleifen müssen eine konstante Länge haben, d.h. die Unter- und Obergrenze müssen als Konstante im Programm stehen und dürfen nicht erst zur Laufzeit errechnet werden.

Es würde sich anbieten, mehrere Formen von Schleifenkonstrukten anzubieten, beispielsweise `for`, `while` und `repeat ... until`. Da bei den beiden letzteren Befehlen aber Bedingungen angegeben werden könnten, die das Terminieren der Konvertierungsfunktion in Gefahr bringen, wird daher nur die `for`-Schleife zugelassen. Die Syntax der Schleife lautet wie folgt:

`for(`Zählervariable=Startwert`,`Zielwert`,`Operation`)`

Die Werte von Startwert und Zielwert müssen ganzzahlig und konstant sein. Der Zielwert muss dabei größer als der Startwert sein, da immer hochgezählt wird.

Beispiel 6.3.9 *Sollen in ein Zielattribut* `Namenssuffixe` *vom komplexen Typ* `set(string)` *fünf Präfixe der Längen 1 bis 5 des Attributs* `Vorname` *vom Typ* `string` *geschrieben werden, kann das über den folgenden Aufruf erfolgen:*

```
for(_tmp_zaehler = 1, 5,
     new.Namenssuffixe = left(old.Vorname, _tmp_zaehler))
```

6.4 Komplexe Konvertierungsfunktionen

Die Liste der folgenden Operationen ist bei der Umsetzung von komplexen Schemaänderungsoperationen auf die Objektebene wichtig. Es handelt sich um Operationen, die den Zugriff auf mehrere Klassen regeln.

6.4.1 Zugriff auf andere Klassen

Um komplexe Schemaänderungsoperationen ausführen zu können, ist oft ein Zugriff auf eine weitere Quellobjektversion $sv_i.c_2$ abgesehen von der schon vorgegebenen Quellobjektversion $sv_i.c_1$ nötig, um diese Daten in die Zielobjektversion $sv_j.c_1$ zu schreiben. Der Zugriff erfolgt analog zu den in den Abschnitten 6.3.1 und 6.3.2 beschriebenen Fällen, allerdings wird nach dem Präfix `new.` bzw. `old.` noch der Klassenname eingefügt. Zur besseren Lesbarkeit ist es auch erlaubt, selbst bei Eindeutigkeit den Klassennamen einzufügen.

Eine Mehrdeutigkeit kann dabei entstehen, wenn in einer Klasse c_1 ein Attribut vom Typ `tuple` existiert, das denselben Namen hat wie eine andere Klasse c_2 und dessen anzusprechendes Unterattribut denselben Namen hat wie ein Attribut in Klasse c_2. Dieser Fall muss ausgeschlossen werden, daher gilt die folgende Einschränkung: Beim Zugriff auf ein Attribut mit komplexem Typ, das denselben Namen hat wie eine Klasse, ist der Teil, der sich auf den Zugriff auf das komplexe Attribut bezieht, in runde Klammern zu setzen.

Beispiel 6.4.1 *Soll ein Attribut* `Gehalt` *vom Typ* `real` *in der Klasse* `Person` *mit dem Wert des Attributs* `Gehalt` *desselben Typs überschrieben werden, wobei das Quellattribut der Klasse* `Angestellter` *liegt, so geschieht dies durch den folgenden Befehl:*

```
new.Person.Gehalt = old.Angestellter.Gehalt
```

oder kurz

```
new.Gehalt = old.Angestellter.Gehalt
```

6.4.2 Erstellen und Dereferenzieren von Referenzen

Oftmals ist die Verwendung von Referenzen zum schnelleren Finden der zugehörigen Komponentenobjekte oder auch für andere Assoziationen hilfreich. Um eine Referenz auf ein

anderes Objekt in ein Attribut zu schreiben, wird der folgende Befehl verwendet:

`new.`Zielattribut`= ref(`Referenz`)`

Falls in einer Objektversion eine Referenz auf ein anderes Objekt – beispielsweise ein Komponentenobjekt – existiert und dieses Objekt benötigt wird, muss die Referenz ausgelesen werden. Dies erfolgt mit dem folgenden Befehl:

`deref(`Attribut vom Typ Referenz`)`

Beispiel 6.4.2 *Soll der Wert eines Attributs* `Telefon` *vom Typ* `string` *von einem Objekt* `Adresse` *in das Zielattribut* `Telefon` *desselben Typs übernommen werden und existiert im Attribut* `Adressenreferenz` *vom Typ* `ref` *eine Referenz darauf, können folgende Anweisungen verwendet werden:*

```
_tmp_Adresse = deref(old.Adresse)
new.Telefon = _tmp_Adresse.Telefon
```

6.4.3 Suchen von Objekten

Falls Werte aus einer anderen Objektversion einer anderen Klasse übernommen werden sollen, auf die keine Referenz vorliegt, muss das entsprechende Objekt gesucht werden. Dazu wird die Suchvorgabe in SQL-Syntax übergeben, der Rückgabewert ist eine Referenz auf das erste gefundene Objekt, das diese Suchvorgabe erfüllt. Die Syntax für die Suche lautet:

```
_tmp_Suchergebnis = sql_query(''Suchvorgabe'')
```

Beispiel 6.4.3 *In der Klasse* `Person` *soll das Attribut* `Telefon` *vom Typ* `string` *mit dem Wert des Attributs* `Telefon` *vom gleichen Typ aus der Klasse* `Adresse` *gefüllt werden. Allerdings liegt keine Referenz auf das entsprechende Objekt vor, es muss gesucht werden. Die Konvertierungsfunktion könnte wie folgt aussehen:*

```
_tmp_Adresse = sql_query(''...'')
new.Person.Telefon = _tmp_Adresse.Telefon
```

Dabei ist in der SQL-Abfrage statt ... die entsprechende Bedingung einzufügen, also beispielsweise die Gleichheit des Werts des Attributs `Name` *in der zu findenden Objektversion.*

6.4.4 Erzeugen und Löschen von Objekten

Für die Durchführung einiger komplexer Schemaänderungsoperationen (beispielsweise die Operation „Aufbrechen von Klassen“, s. Abschnitt 4.3.3.3) ist die Erzeugung oder Löschung von Objekten nötig. Dies erfolgt über die folgenden Aufrufe:

`create object` Klassenname

bzw.

`delete object` Referenz

Beispiel 6.4.4 *Es soll ein neues Objekt der Klasse* `Adresse` *angelegt und eine Referenz auf dieses neue Objekt in das Attribut* `Adressenreferenz` *in der Klasse* `Person` *eingefügt werden.*

```
new.Person.Adressenreferenz = ref(create object Adresse)
```

6.4.5 Existenzprüfung von Objekten

Die Konvertierungssprache bietet die Möglichkeit, neue Objekte anzulegen. Soll allerdings ein bereits vorhandenes Objekt verwendet werden, beispielsweise als Komponentenobjekt wie bei der Operation „Aufbrechen von Klassen“ im Abschnitt 4.3.3.3, muss auf die Existenz des gesuchten Objekts geprüft werden können. Der Infix `not` dreht dabei die Aussage um. Dies ermöglicht eine bedingte Erzeugung von Objekten in der folgenden Syntax:

`if(not exists` Objekt `, create object` Klassenname `)`

6.4.6 Zusammenfassung

Die funktionalen Forderungen wurden erfüllt:

- F1 Die Operation `new.`Attribut=Wert lässt einen Schreibzugriff zu (s. Abschnitt 6.3.1).
- F2 Durch die Operation `old.`Attribut wird ein Lesezugriff auf ein Attribut einer existierenden Objektversion realisiert (s. Abschnitt 6.3.2).
- F3 Ein Zugriff auf Attribute anderer Klassen ist durch Einfügen des Klassennamens hinter `old.` möglich (s. Abschnitt 6.4.1).
- F4 Die Operation `foreach(`Attribut`,`Operationen`)` bietet einen Lesezugriff auf einzelne Werte eines Attributs von komplexem Typ (s. Abschnitt 6.3.3).
- F5 Ein weiterer Wert kann in ein Attribut von komplexem Typ geschrieben werden, indem er dem Attribut zugewiesen wird (s. Abschnitt 6.3.3).
- F6 Es können lokale Hilfsvariablen benutzt werden (s. Abschnitt 6.3.5).
- F7 Globale Konstanten können ausgelesen werden (s. Abschnitt 6.3.6).
- F8 Die Operation `if(`Bedingung`,`Konsequenz`,`Alternative`)` lässt eine bedingte Ausführung von Operationen zu (s. Abschnitt 6.3.8).
- F9 Die Operationen `create object` Klassenname bzw. `delete object` Referenz dienen dazu, neue Objekte anzulegen bzw. zu löschen (s. Abschnitt 6.4.4).
- F10 Es existiert eine Operation, um Objekte anhand von zu vergleichenden Werten zu suchen (s. Abschnitt 6.4.3).
- F11 Die Operation `exists` gibt zurück, ob ein Objekt existiert (s. Abschnitt 6.4.5).
- F12 Schleifen werden durch die Operation `for(`Zählervariable=Startwert`,`Zielwert`,`Operation`)` realisiert (s. Abschnitt 6.3.9).

Die einzelnen Sprachelemente entsprechen den geforderten Konzeptionsgrundsätzen:

- K1 Die Sprache hat Ähnlichkeiten zu existierenden Sprachen und ist damit „intuitiv“ lesbar.
- K2 Die Sprache enthält mit lokalen Variablen und Schleifen Konstrukte, die die Mächtigkeit erhöhen.
- K3, K4 Die durchgehende Verwendung von `new.` und `old.` als Präfix für Attribute erhöht die Lesbarkeit des Codes und erhöht damit die Chance, Fehler frühzeitig zu erkennen.
- K5, K6 Die Sprache bietet dem Programmierer die Möglichkeit, Kommentare zu verwenden, außerdem sind beliebige Einrückungen innerhalb der Zeilen und Leerzeilen zum Abgrenzen von Abschnitten möglich.

K7 Bei der Vergabe von Namen für lokale Variablen gibt es nur minimale Einschränkungen.

K8 Die Sprache basiert auf reiner ASCII-Codierung, d.h. Konvertierungsfunktionen können mit jedem verfügbaren Texteditor erstellt werden.

K9 Die Sprache enthält keine Befehle wie `input` oder `get`, die eine Benutzereingabe ermöglichen würden.

K10 Befehle, die die Gefahr bringen könnten, dass die Ausführung der Konvertierungsfunktion nicht terminiert, wurden entsprechend eingeschränkt oder nicht zugelassen.

6.5 Datenerhaltende Konvertierungsfunktionen

Es kann bei der Durchführung einer normalen Propagation der Fall auftreten, dass bei der Konvertierung ein Wert ungewollt durch einen NULL-Wert überschrieben würde.

Beispiel 6.5.1 *Es existiert bereits die Objektversion ov_2 eines Objekts. Nun wird die Objektversion ov_1 geändert (eine neue Telefonnummer wird eingefügt) und ein Lesezugriff auf Objektversion ov_2 folgt (s. Abb. 6.1). Normalerweise müsste nun eine Propagation von sv_1 nach sv_2 erfolgen, in deren Verlauf die Werte in ov_2 durch die Werte aus ov_1 überschrieben und das in ov_1 nicht existierende Attribut* `Alter` *mit einem NULL-Wert belegt würden.*

Ein wünschenswertes Ergebnis wäre in diesem Falle, dass die Objektversion ov_2 sowohl das definierte Alter von 25 als auch die neue Telefonnummer enthält. Die dafür nötige Funktionalität wurde implementiert.

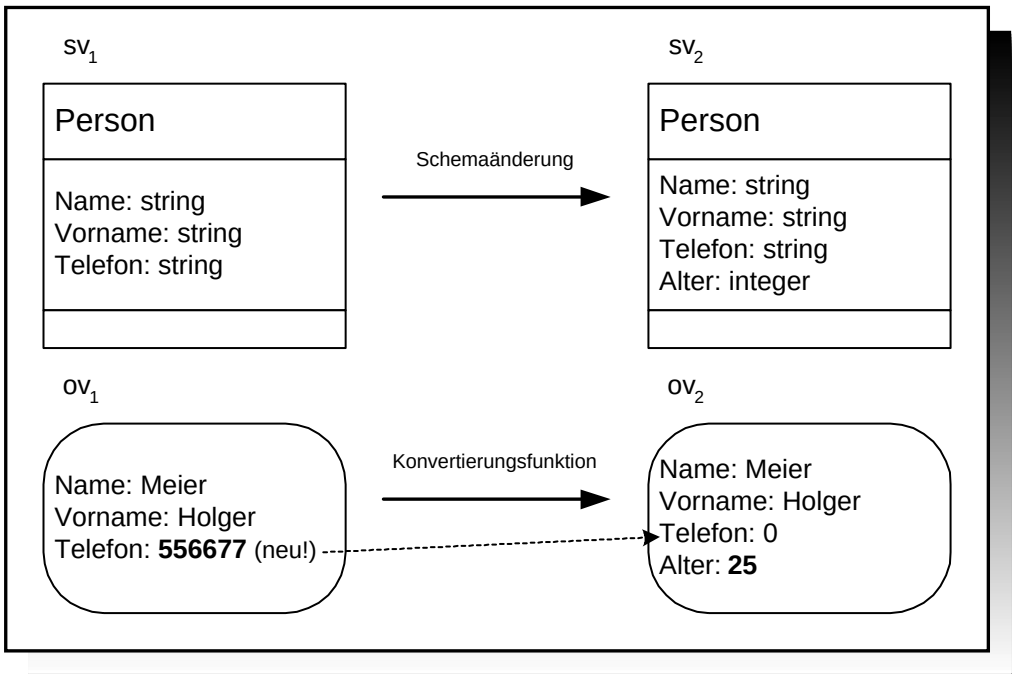

Abbildung 6.1: Beispiel für datenerhaltende Konvertierungsfunktionen

Es wird also festgelegt, dass standardmäßig bereits existierende Attributsinhalte bei Konvertierungen nicht durch Defaultwerte überschrieben werden sollen. Idealerweise ist diese Funktionalität nach Wunsch aktivierbar oder deaktivierbar.

Im implementierten Prototyp COAST wurde im Rahmen dieser Diplomarbeit die Steuerung der beschriebenen Funktionalität über eine globale Konstante namens
`DEFAULT_KILLS_EXISTING`
eingeführt, die standardmäßig den Wert 0 hat. Der Benutzer kann die Deaktivierung der oben beschriebenen Funktionsweise mit dem Befehl

```
SET DEFAULT_KILLS_EXISTING 1
```

bewirken.

6.6 Syntax der Konvertierungssprache in BNF

Der Befehlssatz der entworfenen Konvertierungssprache soll nun abschließend in BNF (Backus-Naur-Form, s. beispielsweise [Sch92]) dargestellt werden. Auf die Angabe der Formatierungszeichen (Einrückungen und Zeilenvorschübe) wurde aus Gründen der Lesbarkeit verzichtet. Das Startsymbol dieser Grammatik lautet `cf`.

```
cf → <cfline> | <cfline><cf>

cfline → <cftarget>' = '<cfrule> | <comment> | 'delete object '<class>

cftarget → <var> | 'new.'<attr> | 'new.'<class>'.'<attr>

aattr → <attr> | 'old.'<attr> | 'old.'<class>'.'<attr>
     | 'new.'<attr> | 'new.'<class>'.'<attr>

cfrule → <aattr> | <cfrule>' + '<cfrule> | <cfrule>' * '<cfrule>
     | <cfrule>' - '<cfrule> | <cfrule>' / '<cfrule>
     | 'ref('<class>')' | 'deref('<class>')' | 'conv('<aattr>')'
     | 'left('<cfrule>,<integer>) | 'right('<cfrule>','<integer>')'
     | 'sql_query('<sql>')' | 'foreach('<aattr>','<cfrule>')'
     | 'for('<var>'='<const>','<const>','<cfrule>')'
     | <if> | <set>

if → 'if('<ifclause>','<ifconsequence>','<ifconsequence>')'
     | 'if('<ifclause>','<ifconsequence>')'

ifclause → <ifclausepos> | 'not '<ifclausepos>

ifclausepos → <attr>' = '<const> | <attr>' = '<attr>
     | <attr>' < '<const> | <attr>' < '<attr>
     | <attr>' > '<const> | <attr>' > '<attr>
     | <var>' = '<const> | <var>' = '<attr>
     | <var>' < '<const> | <var>' < '<attr>
     | <var>' > '<const> | <var>' > '<attr>
     | 'exists '<class>

ifconsequence → <cfrule> | <attr>

comment → '/*'<text>'*/'

set → 'set '<var>' '<const>

attr → <bezeichner>
var → <bezeichner>
sql → <bezeichner>
const → <bezeichner>
class → <bezeichner> | 'create object '<bezeichner>
```

```
text → <bezeichner2> | <text>’ ’<text> | ’ ’<text> | <text>’ ’

bezeichner → <buchstabe><bezeichner2> | <buchstabe>
bezeichner2 → <buchstabe> | <ziffer>
     | <buchstabe><bezeichner2> | <ziffer><bezeichner2>

buchstabe → ’a’ | ... | ’z’ | ’A’ | ... | ’Z’
ziffer → ’0’ | ... | ’9’
```

6.7 Zusammenfassung und Bewertung

In diesem Kapitel wurde die zur Spezifikation der Propagation notwendige Sprache für Konvertierungsfunktionen entwickelt. Dabei wurden sowohl Forderungen an den Funktionsumfang wie auch an die Konzeption aufgestellt, konsequent eingearbeitet und anschließend überprüft.
Zur Abrundung wurde die Konvertierungssprache noch in BNF-Notierung dargestellt.

Damit wurde die Zielsetzung erfüllt: Es steht jetzt eine Sprache zur Spezifikation der Propagation zur Verfügung. Die Sprache umfasst die nötige Funktionaltät sowohl für einfache als auch für komplexe Schemaänderungsoperationen.

Kapitel 7

Implementierung

In diesem Kapitel wird bewusst darauf verzichtet, seitenweise Quellcodes abzudrucken und zu kommentieren. Vielmehr wird auf einige interessante Implementierungsdetails eingegangen und einzelne Fragestellungen, die sich im Zusammenhang mit der Implementierung ergeben haben, beschrieben und Lösungen dazu entwickelt.

Nach einer kurzen Vorstellung des COAST-Prototyps und einzelnen Programmierdetails wird auf die Frage eingegangen, wie und wo die Konvertierungsfunktionen, zu denen im vorangegangenen Kapitel eine Sprache entwickelt wurde, abgespeichert werden sollen. Danach wird auf das Problem eingegangen, wie nach Einführung der komplexen Schemaänderungen weiterhin die Herkunft eines Attributs aus einer anderen Schemaversion notiert werden kann. Zum Schluss werden einige Überlegungen zur Optimierung des Programmablaufs und auch der Schemaänderungsoperationen vorgestellt.

7.1 Funktionsweise von COAST

Die Schemaversionierung ist ein Konzept, das nicht ohne grundlegende Umsetzungen in bestehende Datenbanksysteme integriert werden kann. Aus diesem Grunde wurde der Prototyp COAST (Complex Object and Schema Transformation) entwickelt, der in diesem Abschnitt vorgestellt wird.

Die Grundidee beim Entwurf von COAST ist, bei notwendigen Änderungen an einem bestehenden Datenbankschema nicht dieses bestehende Schema zu verändern, sondern eine Kopie anzulegen und dort die Änderungen einzubringen. Das alte Schema ist damit weiterhin verwendbar. Man spricht in diesem Zusammenhang von *Schemaversionen*. Applikationen greifen immer auf eine Schemaversion zu. In COAST enthält ein Schema alle Schemaversionen dieses Schemas.
Allerdings besteht – anders als beim naiven Kopieren eines Schemas – in COAST eine Beziehung zwischen den Schemaversionen. Sie sind voneinander abgeleitet und werden in einem Ableitungsbaum im Schema gespeichert. Die Wurzel des Ableitungsbaums ist immer die abstrakte Ur-Schemaversion sv_0, die alle Eigenschaften einer Schemaversion hat und damit an die von ihr abgeleiteten Schemaversionen vererbt. Keine Applikation greift auf sv_0 zu, sie dient nur als Ausgangspunkt für die Ableitungshierarchie.

In COAST ist es allerdings möglich, nicht nur eine ganze Schemaversion aus einer anderen Schemaversion abzuleiten, sondern die Kopie in die neue Schemaversion kann auch mit feinerer Granularität geschehen: Eine Klasse kann als *Klassenversion* von einer in eine andere Schemaversion umgesetzt werden. Diesen Vorgang bezeichnet man als *Integration*.

Eine Klasse ist in einer Klassenhierarchie angesiedelt, die analog zu sv_0 bei den Schemaversionen die Klasse `object` als Wurzel hat. `object` ist in sv_0 enthalten.

Die Erstellung einer neuen Schemaversion läuft damit in COAST folgendermaßen ab: Zunächst wird eine neue Schemaversion von sv_0 abgeleitet, dann werden die benötigten Klassen aus der gewünschten Quellschemaversion integriert und ggf. neue Klassenversionen erstellt. Schließlich werden die integrierten Klassenversionen bei Bedarf modifiziert. Soll die neue Schemaversion alle Klassenversionen der Quellschemaversion enthalten, kann auch statt von sv_0 direkt von der Quellschemaversion abgeleitet und dann die nötigen Modifikationen eingebracht werden.

Da in COAST ein Schema in mehreren Schemaversionen vorliegen kann, existiert ein (Daten-)Objekt dementsprechend in mehreren Objektversionen. Zu jeder Schemaversion, in der das Objekt sichtbar ist, existiert genau eine. Es ist allerdings nicht in allen Fällen sinnvoll, jede Änderung einer Objektversion in eine andere Objektversion zu übernehmen. Daher wurden von [Eig97] Propagationsflags beschrieben, die die Propagation von Objekten zwischen zwei Schemaversionen steuern. Sie werden in der vorliegenden Arbeit in Kapitel 5 (Propagation) beschrieben.

7.2 Architektur des COAST-Projekts

COAST besteht aus mehreren Komponenten. Eine schematische Abbildung ist in Abb. 7.1 zu sehen. Auf die Bedeutung der einzelnen Komponenten wird in diesem Abschnitt eingegangen.

7.2.1 Objektmanager

Der Objektmanager ist für Speicherung der Datenbankobjekte zuständig. Des Weiteren generiert und verwaltet er die Objekt-IDs (*oid*s). Für die persistente Speicherung greift er auf ein kommerzielles Produkt, den von AT&T entwickelten Speichermanager EOS [BP94] zurück. Der Objektmanager ist die Schnittstelle von COAST zu EOS, die anderen Komponenten haben keine weiteren direkten Verbindungspunkte zu EOS.

Der Objektmanager von COAST wurde im Rahmen einer Diplomarbeit von [Wöl98] entwickelt.

7.2.2 Schemamanager

Der Schemamanager verwaltet Schemaversionen und Ableitungsbeziehungen zwischen Schemaversionen.

Ein Schema wird persistent in einem Data Dictionary abgelegt. Auf den Schemamanager greifen sowohl der Objektmanager (beim Interpretieren einer Bytekette), der Propagationsmanager (beim Ermitteln von Propagationsflags) und der ODL[1]-Parser (beim Speichern einer Schemabeschreibung in der Metaschemaklasse) zu.

Der Schemamanager von COAST wurde im Rahmen einer Diplomarbeit von [Pri98] entwickelt und von [Dol99] verfeinert und weiterentwickelt.

[1] Object Definition Language, s. Abschnitt 7.2.4

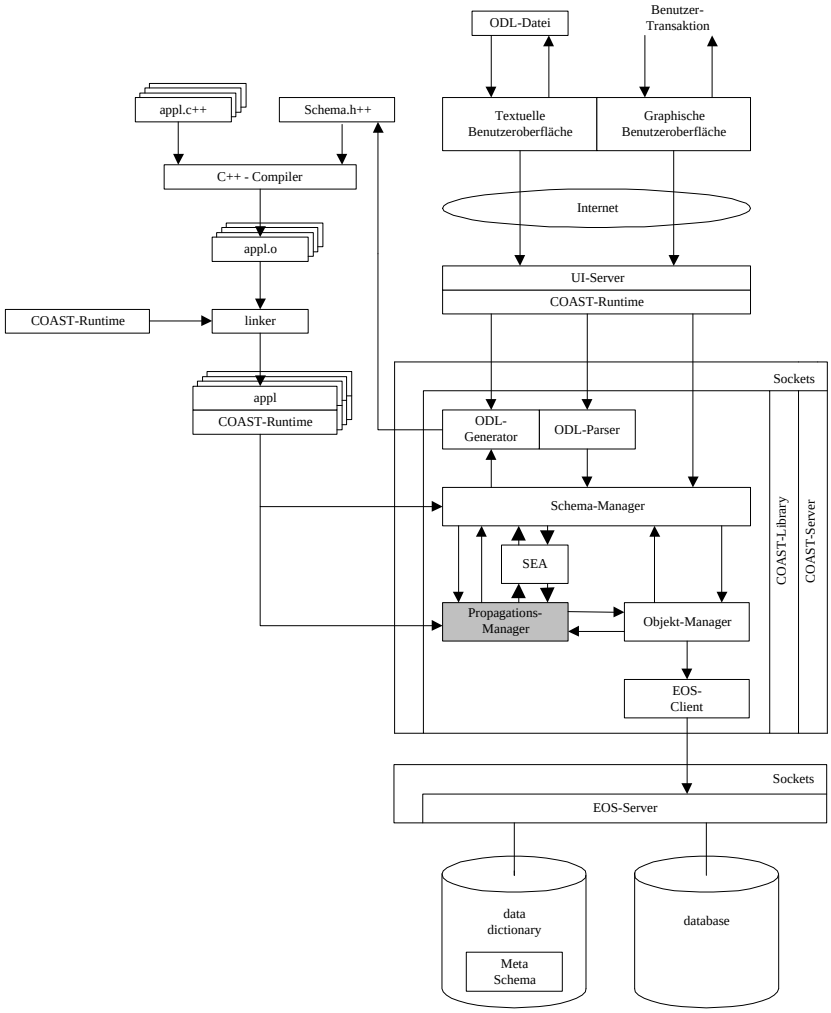

Abbildung 7.1: Die Architektur von COAST, hervorgehoben ist der Propagationsmanager, der die Propagation regelt und der auch Kern dieser Diplomarbeit ist.

7.2.3 Propagationsmanager

Der Propagationsmanager propagiert Objekte zwischen verschiedenen Schemaversionen und vollzieht damit die zentrale Funktion von COAST, die automatische Konvertierung von Datenobjekten im Hintergrund. Dabei beachtet er die Propagationsflags, die im Schema abgelegt sind. Er greift auf den Schemamanager und auf den Objektmanager zu, um seine Aufgabe zu erfüllen.

Die grundlegenden Funktionen des Propagationsmanagers von COAST wurden im Rahmen einer Diplomarbeit von [Eig97] entwickelt und in der vorliegenden Diplomarbeit verfeinert und weiterentwickelt.

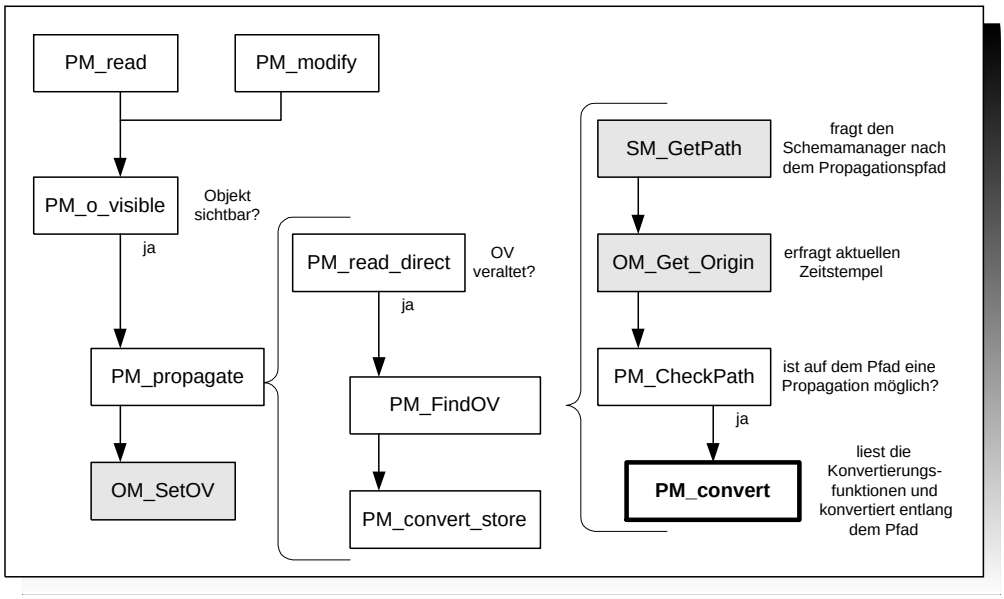

Abbildung 7.2: Die Architektur des Propagationsmanagers, hervorgehoben sind die Zugriffe auf andere Module von COAST. Die besonders betonte Methode PM_convert liest die Konvertierungsfunktionen aus und erledigt die eigentliche Konvertierung.

Die Funktionsweise des Propagationsmanagers wird in Abb. 7.2 vereinfacht dargestellt. Es handelt sich um den folgenden Programmablauf:

Die Applikation, die den folgenden Zugriff tätigt, arbeite mit der Schemaversion sv_{hier}. Bei einem Lesezugriff über `PM_read` oder einen Schreibzugriff über `PM_modify` auf die Objektversion ov_{hier} wird zunächst durch `PM_o_visible` überprüft, ob das Objekt in der gegebenen Schemaversion überhaupt sichtbar ist. Ist das der Fall, wird die Methode `PM_propagate` aufgerufen.
`PM_propagate` ruft zuerst `PM_read_direct` auf, das über Zeitstempelvergleich ermittelt, ob ov_{hier} den aktuellsten Zeitstempel hat. Ist dies nicht der Fall, also in sv_{hier} nicht die aktuelle Version gespeichert, muss die aktuelle Objektversion $ov_{aktuell}$ gesucht werden. Dies wird von `PM_FindOV` erledigt.
`PM_FindOV` kommuniziert mit dem Schemamanager über dessen Methode `SM_GetPath`, um den Pfad zwischen der aktuellsten Objektversion $ov_{aktuell}$ und der zu aktualisierenden Objektversion ov_{hier} zu ermitteln. Dann wird vom Objektmanager der aktuelle Zeitstempel von ov_{hier} erfragt. Die Methode `PM_CheckPath` überprüft mittels der Propagationsflags auf dem vom Schemamanager zurückgegebenen Pfad, ob eine Propagation erfolgen kann. Sind die Propagationsflags in diesem Sinne gesetzt, kann die eigentliche Konvertierung in der Methode `PM_convert` erfolgen.

`PM_convert` geht den Propagationspfad zwischen $sv_{aktuell}$ und sv_{hier} schrittweise durch, liest in jedem Schritt aus der Propagationskante die zu verwendende Vorwärts- oder Rückwärts-Konvertierungsfunktion und ruft den Parser für die Konvertierungsfunktion auf. Dessen Ausgabe und damit die einzelnen Operationen der Konvertierungsfunktion werden ausgeführt. So wird der gesamte Propagationspfad abgearbeitet. Dann wird der in ov_{hier} zu schreibende Inhalt zurückgegeben.
Bevor aber ov_{hier} geschrieben werden kann, muss überprüft werden, ob der momentane Inhalt durch sofortige Propagation an eine andere Objektversion weitergegeben werden muss, bevor er überschrieben wird. Dies wird von der Methode `PM_convert_store` erledigt, die rekursiv aufgerufen wird, falls ein so sofort propagierter Wert vielleicht wiederum einen Wert überschreiben würde, der erst propagiert werden muss.
Schließlich wird der jetzt aktualisierte Wert ov_{hier} persistent in die Datenbank geschrieben, indem die Methode `OM_SetOV` des Objektmanagers aufgerufen wird. Falls zu Beginn `PM_read` aufgerufen wurde, wird noch der aktuelle Wert ausgegeben.

Beim Löschen von Objektversionen (ohne Abbildung) ist der Vorgang ähnlich: Ein Aufruf von `PM_delete` überprüft durch Aufruf von `PM_propagate`, ob der Wert in zu löschenden Objektversion erst noch in eine andere Objektversion propagiert werden muss und führt diese Propagation ggf. aus. Dann wird die Methode `OM_delete` des Objektmanagers aufgerufen, um die Objektversion zu löschen.

Der Parser für die Konvertierungsfunktionen ist in den bereits in [Her99] entwickelten Parser eingebettet. Da dieser ohnehin die Beschreibung der Schemaänderungen in ODL-Syntax liest, wurden für die Konvertierungsfunktionen nur die zwei weiteren Schlüsselworte `fcf` und `bcf` eingefügt und ansonsten die Grammatik des existierenden Parsers erweitert.

Der Parser wird durch die Werkzeuge *flex++* und *bison++* erzeugt. Beides sind C++-Erweiterungen von *lex* und *yacc* [Her95], freien Programmen, die häufig im Compilerbau Verwendung finden. *flex++* generiert aus einer Konfigurationsdatei (mit der Endung `.l`), in der regulären Ausdrücken C++-Code zugeordnet ist, die Methode `yylex`. *bison++* erzeugt aus einer Konfigurationsdatei (mit der Endung `.y`), in der eine kontextfreie Grammatik übergeben wird, die Methode `yyparse`. Beide Teile geben kompilierbaren Quellcode aus, der dann nur noch im Hauptprogramm mit eingebunden werden muss.

Soll eine ODL-Datei mit eingebetteter Konvertierungsfunktion durch den Parser verarbeitet werden, wird sie zunächst durch `yylex` in Tokens zerlegt. Diese werden dann von `yyparse` mit der vorgegebenen Grammatik verglichen. Falls keine Fehler auftreten, kann das Ergebnis im Schema abgelegt werden.

Zusätzlich ist seit der Erstellung des Schemaentwicklungstools SEA, das in [Apo00] beschrieben wird, eine Kommunikation zwischen diesem Modul und dem Propagationsmanager geplant. SEA soll in die Kommunikation zwischen dem Propagationsmanager und dem Schemamanager eingreifen. Es erkennt beispielsweise komplexe Schemaänderungsoperationen und kann die entsprechende Default-Konvertierungsfunktion zurückgeben. Die Funktionalität zur Kommunikation mit SEA ist beiderseitig noch nicht implementiert.

7.2.4 ODL-Parser und ODL-Generator

In COAST wurde eine Sprache ODL (Object Definition Language) für die Beschreibung von Schemaänderungen entwickelt, die vom ODL-Parser gelesen werden kann. In ODL beschriebene Schemaänderungen werden durchgeführt, indem der ODL-Parser mit dem Schemamanager und dem Objektmanager kommuniziert und die einzelnen Änderungsschritte veranlasst. Der ODL-Generator erzeugt aus einem vorhandenen Schema eine Schemabeschreibung in ODL-Syntax, die vom Benutzer verändert und mit dem ODL-Parser neu

eingelesen und interpretiert werden kann. Dabei werden Schemaänderungsoperationen als Befehle eingetragen und so die Historie bei der Entwicklung der jeweiligen Schmemaversion weitgehend beibehalten.

Die ODL-Syntax, der ODL-Parser sowie der ODL-Generator von COAST wurde im Rahmen einer Diplomarbeit von [Her99] entwickelt.

7.2.5 Schemaeditor

Für die Änderung bestehender Schemata kann nicht nur die textuelle ODL-Beschreibung verwendet werden, sondern auch der grafische Editor, der in Java programmiert und Webbrowser-tauglich ist. Über ihn lassen sich sowohl Schemaänderungsoperationen als auch das Setzen von Propagationsflags benutzerfreundlich durchführen.

Der Schemaeditor von COAST wurde im Rahmen einer Diplomarbeit von [Gro00] entwickelt.

7.2.6 Schematool SEA

Es kann für den Benutzer interessant sein, ein vorliegendes Schema in ODL-Syntax ausgeben zu lassen und mit anderen Programmen zu verändern, bis es den Wünschen entspricht. In diesem Falle wird also keine Schemaänderungsoperation in ODL-Syntax hinzugefügt, sondern die eigentliche Struktur des Schemas geändert. Um nun aus Sicht des Datenbanksystems zu erkennen, welche Änderungen erfolgt sind, muss ein analysierender Vergleich zwischen der alten und der neuen Schemabeschreibung erfolgen, bestimmte Änderungen erkannt und durch Schemaänderungsoperationen ausgedrückt und schließlich die neue Schemaversion in Abhängigkeit von der alten Schemaversion in ODL-Syntax ausgegeben werden. Dieselbe Situation tritt auf, wenn beispielsweise zwei völlig unterschiedlich entwickelte Schemabeschreibungen miteinander verglichen werden sollen.

Das Tool SEA (Schema Evolution Assistant), das diese Aufgabe löst, befindet sich zwischen dem Propagations- und dem Schemamanager. Es wurde im Rahmen einer Diplomarbeit von [Apo00] entwickelt.

7.2.7 COAST Runtime

Die Komponente COAST Runtime stellt die Schnittstelle für alle Applikationen dar, die auf COAST zugreifen wollen. Es ist zurzeit noch nicht realisiert.

7.3 Realisierungskonzepte

Nach der obigen Vorstellung der COAST-Architektur wird nun auf einige interessante Aspekte bei der prototypischen Realisierung des Propagationsmanagers auf der Basis der in dieser Diplomarbeit erstellten Konzepte eingegangen.

7.3.1 Entwicklungsumgebung

Der Propagationsmanager wurde in der Programmiersprache C++ implementiert, unter Verwendung der in der Professur vorhandenen Entwicklungsumgebung SNiFF+ [Sys00]. Dieses Produkt bietet die Möglichkeit, im Team an größeren Programmierprojekten zu

arbeiten. Dazu bedient es sich sog. *locks*, um den Schreibzugriff auf gerade in Bearbeitung befindliche Dateien zu sperren, außerdem verwendet es das Konzept der Versionierung, womit man außer der aktuellen auch ältere Versionen von Dokumenten (Quellcode, Dokumentation, etc.) zur Verfügung hat.

7.3.2 Vorgefundene Situation

Eine erste Fassung des Propagationsmanagers hat bereits Patricia Eigner in ihrer Diplomarbeit [Eig97] implementiert. Diese unterstützte allerdings nur das Konzept der einfachen Propagation unter Beachtung von Propagationsflags.

Die tatsächliche Propagation von Daten funktionierte noch nicht zufriedenstellend, es existierten in dieser Hinsicht nur einige Funktionen, die eine festverdrahtete Konvertierung von Objekten zuließen.

7.3.3 Die Konfigurationsdatei .coastrc

Im Rahmen der Implementierung zu dieser Arbeit wurde die Konfigurationsmöglichkeit von COAST bzgl. Netzwerk- und Mehrbenutzerbetrieb erweitert. Dazu gehört eine Konfigurationsdatei `.coastrc`, in der die Umgebung zur Ausführung von COAST individuell definiert wird. Vorher mussten die entsprechenden Daten im Hauptmenü von COAST nach jedem Start eingegeben werden.

Die Datei ist bewusst in reiner textueller Form gehalten, um eine leichte Editierbarkeit von außen zu ermöglichen. Es gibt nur vier Parameter, die ausgelesen werden können:

- Die Variable `COAST-Area` gibt den Pfad an, wo die Datenbank zu COAST abgelegt ist.
- Die Variable `eos-Server` gibt an, auf welchem Rechner der EOS-Server, der die persistente Speicherung der Daten übernimmt, läuft. Hier kann entweder eine IP-Adresse oder ein durch das Unix-System mittels Namensdienst (DNS) auflösbarer Bezeichner verwendet werden.
- Die Variable `.odl-Files` gibt den Pfad an, wo die Textdateien liegen, die die einzulesenden Schemabeschreibungen und -änderungsoperationen in ODL-Syntax enthalten.
- Das Zeichen `#` leitet eine Kommentarzeile ein.

Wenn die Datei `.coastrc` im Heimatverzeichnis des Benutzers beim Start des COAST-Clients nicht existiert, wird sie automatisch angelegt und so ausgefüllt, dass sie sinnvolle Default-Einstellungen vorgibt. Der Benutzer kann dann bei Bedarf die Datei ändern.

Der Inhalt der Datei `.coastrc`, wie er zurzeit automatisch generiert wird, ist in Abb. 7.3 zu sehen.

7.3.4 Wann muss eine Propagation erfolgen?

Beim Zugriff auf eine Objektversion durch eine Applikation, die auf der entsprechenden Schemaversion aufsetzt, muss eine Überprüfung stattfinden, ob die in ihr enthaltenen Daten noch aktuell sind oder ob in einer anderen Objektversion desselben Objekts zwischenzeitlich Änderungen vollzogen wurden, die in dieser Objektversion sichtbar sein müssten.

```
# Diese Datei wurde automatisch durch COAST erzeugt.
#
COAST-Area /home/tokio/specials/coast/.eos/coast_area/
eos-Server woerth
.odl-Files /home/tokio/specials/coast/COAST_Documentation/
```

Abbildung 7.3: Default-Inhalt der Konfigurationsdatei `.coastrc`

Diese Überprüfung arbeitet so, dass in jeder Objektversion ein Zeitstempel der letzten Aktualisierung gespeichert wird. Erfolgt eine Propagation, erhält die so aktualisierte Objektversion denselben Zeitstempel wie die Quell-Objektversion, aus der die Daten propagiert wurden.

Ist nun beim Zugriff auf eine Objektversion der Zeitstempel älter als in einer anderen Objektversion, muss propagiert werden. In dem Falle wird die entsprechende Propagationskante gesucht und die entsprechende Konvertierungsfunktion aufgerufen.

7.4 Speicherung von Konvertierungsfunktionen

Konvertierungsfunktionen liegen nach der Eingabe in textueller Darstellung vor und müssen so gespeichert werden, dass sie schnell gefunden werden, sobald sie zum Einsatz kommen. Die Entscheidung, in welcher Struktur und wo die Konvertierungsfunktionen abgelegt werden, soll hier beschrieben und begründet werden. Es werden zunächst die verschiedenen Möglichkeiten aufgezählt und im Anschluss erklärt und bewertet:

- Speicherung in textueller Form
- Speicherung in einer Baumstruktur

Dazu kommen die verschiedenen Orte, an denen sie abgelegt werden können:

- Speicherung an zentraler Stelle, d.h. in einer Extradatenbank
- Speicherung in der jeweiligen Quellklasse
- Speicherung in der jeweiligen Zielklasse
- Speicherung in der Verbindungskante zwischen Quell- und Zielklasse

Im Folgenden sind mit dem Oberbegriff Konvertierungsfunktionen sowohl Vorwärtskonvertierungsfunktionen (fcf) als auch Rückwärtskonvertierungsfunktionen (bcf) gemeint.

Zunächst soll die **Art der Speicherung** betrachtet werden:

Die Konvertierungsfunktionen können in textueller Form gespeichert werden. In diesem Falle werden die Befehlszeilen, ohne dass sie zuerst von einem Parser übersetzt würden, abgelegt.

Es wäre auch denkbar, die Konvertierungsfunktionen in einer Baumstruktur zu speichern. Dazu müsste der Parser die Konvertierungsfunktionen einlesen und als Syntaxbaum ablegen. Beim Auslesen und Neugenerieren der Zeilen der Konvertierungsfunktion würde dann eventuell die Reihenfolge der Zeilen verändert, allerdings ohne semantische Einbußen.

Die textuelle Darstellung hat Vor- und Nachteile gegenüber einer Baumstruktur.
Zu den Vorteilen zählt die einfache Möglichkeit, Änderungen ohne nötige Nachbehandlung zu speichern. Eine Schwierigkeit bei der Verwendung von Baumstrukturen ist, dass für den Fall, dass der Schemaentwickler die Konvertierungsfunktionen ändern möchte, diese neu aus der Baumstruktur rekonstruiert, d.h. generiert werden müssen. Vorher existierende Kommentare des Benutzers oder bestimmte Schreibweisen (z.B. führende Leerzeichen oder Tabs) sind dabei nicht automatisch wieder in der frisch erzeugten Version wieder zu finden. Ebenso ist nicht gewährleistet, dass die Reihenfolge der rekonstruierten Konvertierungsfunktionen wieder dieselbe ist.
Der Generator, der in [Her99] vorgestellt wurde, gibt eine kanonische Form der ODL vor, die von der Top-Down-Traversierung des Vererbungsgraphen stammt. Diese Vorgehensweise hat sich für die ODL als durchaus praktikabel erwiesen und könnte auch für die Speicherung der Konvertierungsfunktionen Vorteile haben.
Die Reihenfolge für Konvertierungsfunktionen spielt keine Rolle, allerdings muss jeweils innerhalb einer Konvertierungsfunktion die Reihenfolge der Anweisungen beibehalten werden, es sei denn, sie ist ohne semantische Verluste veränderbar.

Bei der Verwendung von Kommentaren ist besondere Vorsicht geboten: Diese würden durch den Parser verworfen, da sie keinen semantischen Anteil an der Konvertierungsfunktion haben. Es ist natürlich auch denkbar, Kommentare in einer Baumstruktur mit zu speichern, allerdings wäre dann zu prüfen, zu welchem Befehl der jeweilige Kommentar (der auch mehrzeilig sein kann) gehört, damit bei einer eventuellen Neuordnung der Zeilen durch das Auslesen des Baums die Kommentare wieder bei den Operationen zu finden sind, wo sie in der ursprünglichen Fassung standen. Dabei ist zu bedenken, dass manche Schemaentwickler Kommentarzeilen jeweils vor die entsprechende Befehlszeile schreiben mögen, andere vielleicht jeweils hinter die entsprechende Befehlszeile. Die eindeutige Zuordnung von Kommentaren zu Befehlszeilen ist also nicht trivial und müsste beispielsweise durch zusätzliche Leerzeilen zwischen den Kommentar-Befehl-Paketen hervorgehoben werden.

Im Gegensatz zur textuellen Darstellung bietet die Baumstruktur die Möglichkeit, die Konvertierungsfunktionen in bereits ausgewerteter Form abzulegen, d.h. sie müssen nicht interpretiert werden, wenn eine Propagation erfolgt. Außerdem kann man vielleicht Änderungen einfacher einbinden, beispielsweise eine Löschung eines Attributs: In diesem Falle könnte bei entsprechender Implementierung ein gesamter Teilbaum gelöscht und so schnell und einfach die Auswirkung der Attributlöschung berücksichtigt werden. In textueller Form geht das nicht, denn es müsste erst wieder durch den Parser zu einem Baum gemacht werden. So müsste bei einer Attributlöschung jede Textzeile der Konvertierungsfunktion daraufhin untersucht werden, ob sie sich durch diese Operation ändern muss oder nicht.

Es entsteht bei der Wahl einer der beiden Alternativen kein Unterschied im Implementierungsaufwand, da ohnehin in beiden Fällen ein Parser bereitgestellt werden muss. Dieser wird für das Einlesen und das Generieren der Konvertierungsfunktionen benötigt.

In dieser Diplomarbeit werden Konvertierungsfunktionen jeweils in der textuellen Darstellung abgelegt, um Änderungen an den Konvertierungsfunktionen für den Schemaentwickler zu erleichtern.

Was den **Speicherungsort** der Konvertierungsfunktionen betrifft, gilt es folgende Vor- und Nachteile abzuwägen:

Eine zentrale Speicherung beispielsweise in eine Datenbank für Konvertierungsfunktionen, die dann abgefragt werden kann, ist möglich. Allerdings ist die Konvertierungsfunktion nicht bei der jeweiligen Schemaversion zu finden, sondern eben an einer ganz anderen Stelle. Das läuft dem Grundprinzip der Kapselung der objektorientierten Datenbank zuwider. Die Speicherung in einer der beiden beteiligten Klassen (bei einfachen Konvertierungsfunk-

tionen) bietet sich an – jedoch ist bei komplexen Konvertierungsfunktionen, bei denen mehr als zwei Klassen involviert sind, die Entscheidung, wo die Speicherung erfolgen soll, unklar. Man könnte die Konvertierungsfunktionen in die jeweils relevanten Teile aufsplitten und diese immer dort unterbringen, wo auch die entsprechenden Klassen liegen. Der Aufwand ist allerdings beträchtlich, wenn zudem noch sichergestellt werden soll, dass ein Teil einer Konvertierungsfunktion nur dann ausgeführt wird, wenn er auch tatsächlich ausgeführt werden soll, und nicht etwa, wenn nur ein Objekt der ihn enthaltenden Klasse propagiert wird. Eine Variante wäre noch die Aufteilung der Vorwärts- und Rückwärtskonvertierungsfunktionen in die Quell- bzw. Zielklasse. Hier käme aber das gleiche Gegenargument mit der verteilten Speicherung in Betracht.

Da es im Propagationsmanager für Verbindungskanten eine eigene Klasse gibt, bietet es sich an, die Konvertierungsfunktion in der Kante zu speichern. In diesem Falle ist die Konvertierungsfunktion immer dann verfügbar, wenn die Kante ohnehin betrachtet wird, nämlich dann, wenn eine Propagation entlang der Kante erfolgen soll. Zudem ist im Falle, dass mehr als zwei Klassen beteiligt sind, die Speicherung ebenfalls einfach, weil weiterhin nur an einer Stelle gesucht werden muss.
Zusätzlich bietet sich dieser Ort an, da bei der Propagation sowieso der Klassenableitungsgraph traversiert wird und man im Falle, dass eine Propagation nötig wird, die entsprechende Kante schon vorliegen hat.

In der im Rahmen dieser Diplomarbeit entstandenen Implementierung werden Konvertierungsfunktionen in der entsprechenden Propagationskante zwischen zwei oder mehr Klassen gespeichert.

7.5 Speicherung der Historie von Schemaänderungen

Die Kenntnis darüber, welche Schemaänderung ausgeführt wurde, um eine neue Schemaversion zu erzeugen, darf nicht verloren gehen. Jedoch will man nicht genau dieselben Schemaänderungsoperationen, sondern eine kanonische Form reproduzieren können, die möglichst kurz und einfach ist. Dabei können gewisse Vereinfachungen vorgenommen werden, eine Optimierung der gesamten Schemaänderungsanweisungen ist möglich.

Es ist hilfreich, eine typische Schemaänderung wie das Kopieren von Attributen oder das Aufbrechen von Klassen so im Schema abzulegen, dass die Zusammenhänge zu anderen Schemaversionen wiederzufinden sind, wenn die Schemaversion erneut in ODL-Syntax generiert werden soll. In diesem Zusammenhang wird im Schema zu jedem Attribut jeder Schemaversion außer dem Attributsnamen und dem Typ noch die Quelle abgespeichert, bestehend aus der Quellklassen-ID und der Quellattributs-ID.
Bei neu angelegten Attributen sind diese Felder leer, bei Attributen, die durch komplexe Schemaänderungen erzeugt wurden, steht dann aber die Herkunft fest und es kann in der generierten ODL-Syntax wieder der entsprechende Befehl (`copy`, `move`, ...) rekonstruiert werden.

In COAST existierte das Konzept der Property ID (*pid*), einer Attributs-ID, die jedem Attribut in jeder Schemaversion zugeteilt wurde. Innerhalb einer Klasse war die *pid* eindeutig. Wenn ein Attribut dieselbe *pid* hat wie ein anderes Attribut in einer anderen Schemaversion, dann bedeutete dies, dass es sich um dasselbe Attribut innerhalb einer Klasse handelte und eine direkte Konvertierung des Attributsinhalts möglich war.
Eine einfache Speicherung dieser *pid* reicht nach der Einführung von komplexen Schemaänderungsoperationen nicht mehr aus, da beispielsweise bei der Operation „Kopieren von Attributen“ zwei Zielattribute dieselbe *pid* innerhalb derselben Schemaversion erhalten

würden, was die Eindeutigkeit einer solchen Kennung zerstört und eine Weiterentwicklung der *pid*s unumgänglich machte.

Der Einfachheit halber wurde in der Implementierung zu dieser Diplomarbeit darauf verzichtet, die Schemaänderungsoperationen, die sich aus anderen Operationen zusammensetzen lassen, speziell abzulegen, es würde in diesem Falle also die ausführliche Beschreibung in ODL-Syntax erzeugt statt der abkürzenden Schreibweise (s. Kapitel 4). Erzeugt werden also nur die Befehle S1.1 bis S1.4, S2.1 bis S2.3 und S3.1 bis S3.4, wie sie in Abschnitt 4.1 vorgestellt wurden. Die Schemaänderungsoperationen, die eine abkürzende Schreibweise für mehrere andere Operationen darstellen, könnten eventuell in einer Überarbeitung der genannten Liste von Schemaänderungsoperationen rekonstruiert werden.

7.6 Effizienzbetrachtungen

Will man das Datenbanksystem für den Anwender schnell machen, ist es sinnvoll, sich Gedanken über gute Strategien für die zu erledigenden Aufgaben zu machen.
Neben dem Konzept der verzögerten Propagation, das bereits in COAST implementiert ist, gibt es noch weitere Punkte, wo eingegriffen werden kann, um die Ausführung von Konvertierungsfunktionen und damit die Ausführung der Propagation performanter zu machen. Ein Optimierungskonzept könnte an zwei Punkten in das System eingreifen:

- Optimierung der intern zu erledigenden Programmschritte zur Durchführung der Konvertierung
- Optimierung der durch den Schemaentwickler vorgegebenen Schemaänderungsoperationen und Konvertierungsfunktionen

7.6.1 Optimierung des Programmablaufs

Die Überlegungen zur Verbesserung der internen Performanz gliedern sich in zwei Teile: Zeit- und Platzeinsparung.

Im Folgenden wird auf Verbesserungen der Propagationsmethodik eingegangen.

- Wie findet man effizient die aktuellste Version eines Objekts?

  Hier bieten sich zwei Alternativen an.

  – Durchsuchen der Objektversionen:
  Das System kann alle vorhandenen Objektversionen durchsuchen, entweder über Tiefensuche (engl. depth first search, dfs) oder Breitensuche (engl. breadth first search, bfs) nach dem Ableitungsgraph der Schemaversionen oder über eine Liste, in der alle aktuell existierenden Objektversionen eingetragen sind. Der Nachteil dabei ist, dass beim Zugriff jede Objektversion auf ihren Zeitstempel untersucht werden muss. Dafür ist ein Vorteil, dass keine zusätzlichen Aktivitäten bei anderen Funktionen wie beispielsweise das Anpassen eines Zeigers auf die aktuellste Objektversion nötig sind.

  – Verwendung eines Extrafelds:
  Für jedes Objekt existiert noch ein Extrafeld, in dem die Objektreferenz der zuletzt veränderten Objektversion steht. Der Nachteil hier ist, dass das Feld immer angepasst werden muss, auch bei transitiver Propagation. Dafür ist ein sofortiger Zugriff möglich.

In COAST ist eine nach Zeitstempel sortierte Liste von Objektversionen und ein Flag, welches die aktuellste Version ist, realisiert. Bei großen Datenbanken, die nur wenige Schemaversionen haben, ist die erste Alternative wahrscheinlich sinnvoller.

- Welche Objektversionen können gelöscht werden, um Speicherplatz zu sparen?

  Im worst case liegt von jedem Objekt eine Objektversion für jede Schemaversion vor, was einen hohen Platzverbrauch darstellt. Um dies zu verbessern, kann untersucht werden, welche Objektversion in dem Sinne redundant ist, als dass man sie verlustfrei löschen und bei Bedarf durch wiederholte Ausführung der Konvertierungsfunktionen rekonstruieren kann. Daher bietet sich an, sich Gedanken über Methoden zu machen, die einzelne Objektversionen verlustfrei löschen. Eine Objektversion ist dann verlustfrei löschbar, wenn sie jederzeit aus einer anderen Objektversion wieder hergestellt werden kann. Ideal ist der Fall, dass jedes Objekt in genau einer, der aktuellsten, Objektversion vorliegt. Das Löschen unnötiger Objekte könnte z.B. in Leerlaufzeiten oder zu Reorganisationszeiten erfolgen. Dies könnte nach dem Prinzip der Speicherbereinigung (engl. garbage collection, s. [ASS91]) geschehen.

  Beispiel 7.6.1 *Objektversion ov_2 werde aufgrund eines Lesezugriffs in sv_2 aus Objektversion ov_1 erzeugt. Danach kann Objektversion ov_2 wieder gelöscht werden – der Schritt ist jederzeit wiederholbar. In diesem Falle sollte ov_1 als die aktuellste Version des Objektes markiert bleiben, obwohl ov_2 später erzeugt wurde. In COAST erhalten in dieser Situation ov_1 und ov_2 denselben Zeitstempel.*

  Beispiel 7.6.2 *Objektversion ov_2 werde aus ov_1 erzeugt, ov_3 danach aus ov_2. Es wird also eine transitive Propagations durchgeführt und ov_2 ist nur ein Zwischenergebnis. Gleichgültig, ob ov_1 oder ov_3 nach einem eventuellen Schreibzugriff die aktuellste Objektversion ist, kann ov_2 gelöscht werden, da die Konvertierung jederzeit wiederholt werden kann.*

7.6.2 Optimierung von Schemaänderungsoperationen

Das Aufschreiben von gewünschten Schemaänderungen in ODL-Syntax kann ähnlich wie die interne Durchführung der Propagation Potenzial zur Optimierung bieten. Dabei wird davon ausgegangen, dass eine Schemaänderung schneller erfolgt, wenn die nötigen Schritte direkt hintereinander stehen. Wenn mehrere Klassen geändert werden sollen und die nötigen Schemaänderungsoperationen immer abwechselnd die eine oder andere Klasse betreffen, ist die Durchführung langsamer, weil immer erst die Umgebung der betreffenden Klasse neu geöffnet werden muss.

So könnte sich anbieten, dem Schemaentwickler ein Werkzeug zu bieten, das beispielsweise die folgenden Vorkommnisse in der Liste der Schemaänderungsoperationen optimiert:

- Streichen von unnötigen Operationsabfolgen wie beispielsweise

```
create attribute unnoetig
delete attribute unnoetig
```

- Verkürzen von Operationssequenzen wie beispielsweise

```
create attribute falscher_name
rename falscher_name to richtiger_name
```

zu

```
create attribute richtiger_name
```

- Umsortierung der Schemaänderungsoperationen. Es könnte sein, dass die Attribute einer Klassenversion in wilder Reihenfolge und über die gesamte ODL-Datei verteilt einzeln in eine neue Klassenversion integriert werden. Der Fall kann beispielsweise bei Verwendung des graphischen Schemaeditors (s. [Gro00]) entstehen, wenn in beliebiger Reihenfolge Schemaänderungen zusammengeklickt wurden.
  In dieser Situation könnte man die Aufrufe der Schemaänderungsoperationen so umsortieren, dass alle Aufrufe, die eine Klasse betreffen, direkt hintereinander und im selben „modify class"-Block der ODL stehen.

- Umschreiben von Abfolgen von Schemaänderungsoperationen, die sich durch eine einzige Schemaänderungsoperation darstellen lassen. Es kann beispielsweise sein, dass der Schemaentwickler ein Attribut von einer Klassenversion in eine andere Klassenversion kopiert und es dann in der ursprünglichen Klassenversion löscht. Dies lässt sich direkt mit der Schemaänderungsoperation `move` ausdrücken (s. Abschnitt 4.3), was das System anstelle der beiden vorgefundenen Zeilen einsetzen könnte.
  Es kann sogar sein, dass durch eine vorhergehende Optimierung die Zeilen in eine andere Reihenfolge gebracht wurden, wodurch der Zusammenhang einer solchen zusammenfassbaren Operation erst offenbar wird.

Es sind natürlich noch weitere Punkte denkbar, an der ein Optimierungsmechanismus ansetzen könnte.

Eine solche Komponente zur Unterstützung des Schemaentwicklungsprozesses ist nicht implementiert. Zur Abrundung der Funktionalität von COAST ist eine entsprechende Entwicklung aber durchaus denkbar.

7.7 Zusammenfassung und Bewertung

Dieses Kapitel beschäftigte sich mit speziellen Problemen, die im Zusammenhang mit der Implementierung auftreten.

Zunächst wurde auf die Architektur von COAST eingegangen und dabei besonderes Augenmerk auf die Funktionsweise des Propagationsmanagers, der Bestandteil dieser Diplomarbeit ist, gelegt. Die einzelnen Komponenten von COAST wurden erläutert und die Kommunikation zwischen ihnen beschrieben.

Es stellte sich dann die Frage, wie und wo Konvertierungsfunktionen abgespeichert werden sollen. Nach einer Diskussion mehrerer Lösungsmöglichkeiten wurde sich dafür entschieden, Konvertierungsfunktionen in textueller Darstellung innerhalb der Propagationskanten abzuspeichern.

Die Einführung von komplexen Schemaänderungsoperationen machte eine Änderung der Notierung der Attributszugehörigkeit zwischen mehreren Schemaversionen nötig: Es reicht nicht mehr aus, eine innerhalb einer Schemaversion eindeutige Attributs-ID (Property ID, *pid*) zu vergeben, die die Information enthält, mit welchem Attribut mit derselben *pid* in einer anderen Schemaversion das Attribut verbunden ist. Die komplexe Schemaänderungsoperation „Kopieren von Attributen" u.ä. führen dazu, dass diese *pid* nicht mehr eindeutig ist, sondern bei zwei Zielattributen dieselbe *pid* notiert ist. Daher musste ein Konzept eingeführt werden, das diese ID erweitert.

Es wurde die Änderung eingeführt, dass in jedem Attribut neben dem eigentlichen Inhalt noch zwei Werte abgespeichert werden, und zwar die Quell-Klassen-ID (*oid*) und die Quell-Attributs-ID (*pid*). Dadurch ist für jedes Attribut eindeutig erkennbar, von wo es stammt, d.h. mit welchem Attribut es in Beziehung steht.

Im Zusammenhang mit Effizienzbetrachtungen wurden mehrere Möglichkeiten vorgestellt, wie der Programmablauf beschleunigt werden kann. Dazu wurde sich bei der Frage, wie ein schneller Zugriff auf die aktuellste Objektversion möglich ist, entschieden, in jedem Objekt noch eine Referenz auf die Objektversion, die als letzte aktualisiert wurde, zu speichern. Außerdem wurden mehrere Bedingungen genannt, wann eine Objektversion ohne Verluste gelöscht werden kann, um Platz zu sparen, da sie sich bei einem Zugriff problemlos aus anderen Objektversionen wieder herstellen lässt.

Schließlich wurden Vorschläge gemacht, welche Optimierungen an eingegebenen Konvertierungsfunktionen durch ein Werkzeug vorgenommen werden könnten, das dem Schemaentwickler als zusätzliche Hilfe an die Hand gegeben werden könnte.

Durch die Einführung und Implementierung von komplexen Schemaänderungen ist COAST leistungsfähiger geworden. Im Gegensatz zu den hartverdrahteten Konvertierungsfunktionen, die vorher existierten, können nun auch beliebige Konvertierungsfunktionen dynamisch vom Schemaentwickler angegeben werden. Durch die angesprochenen noch offenen Punkte sind nur noch Verbesserungen im Detail möglich, es wurde sich auf die wichtigsten Punkte konzentriert.

Kapitel 8

Zusammenfassung und Ausblick

Das Ziel dieser Diplomarbeit war, die automatische Propagation von Objektänderungen zwischen verschiedenen Objektversionen im Prototyp COAST weiterzuentwickeln. Als Erweiterung zu den bereits in [Eig97] vorgestellten *einfachen Schemaänderungen* sollten als neues Konzept *komplexe Schemaänderungen* definiert und beschrieben werden. Diese komplexen Schemaänderungen haben mehr als nur eine Quell-Objektversion oder mehr als nur eine Ziel-Objektversion. Es sollten neben der Spezifikation auch die Auswirkungen dieser Erweiterung auf das vorhandene Konzept sowie die Generierung von Default-Konvertierungsfunktionen untersucht werden.
Schließlich sollte eine Sprache entworfen werden, die die Spezifikation der auf Objektebene nötigen Konvertierung der Daten ermöglicht.

8.1 Zusammenfassung

Zunächst wurde die Notwendigkeit von Schemaänderungen erläutert und verschiedene Ansätze aus der Literatur beschrieben, Schemaänderungen in laufenden Systeme so durchzuführen, dass eine möglichst einfache und automatisch stattfindende Konvertierung der betroffenen Datenobjekte erfolgen kann. Beim Vergleich erweist sich das Konzept der Schemaversionierung als die leistungsfähigste Lösung.
Der Grundgedanke der Schemaversionierung ist, durch jede Schemaänderung eine neue Schema*version* zu erstellen, wobei die alte Schemaversion weiterhin benutzt werden kann. Die Datenobjekte liegen ebenfalls in mehreren Versionen vor und die Schemaänderung wird auf Objektebene nachempfunden, indem Datenänderungen *propagiert*, d.h. die Daten automatisch konvertiert werden. Für die Propagation werden die Beziehungen zwischen den Schemaversionen ausgenutzt.

Mit dem Konzept der Schemaversionierung ist es möglich, mehrere Versionen eines Schemas parallel zu benutzen und nur die auf die Datenbank zugreifenden Applikationen anzupassen, die auch wirklich von der Schemaänderung betroffen sind.
Diese Diplomarbeit ist Teil des COAST-Projekts, das die Schemaversionierung als Prototyp umsetzt.

In COAST existierte vor dieser Diplomarbeit nur die Möglichkeit, einfache Schemaänderungen durchzuführen. Neu wurden *komplexe Schemaänderungsoperationen* eingeführt und das Konzept der Propagation entsprechend erweitert. Komplexe Schemaänderungen unterscheiden sich von einfachen Schemaänderungen dadurch, dass sie Attribute aus mehreren Quellklassen in einer Zielklasse (oder andersherum) vereinen können. Die bereits in

[Wöh96] kurz angeschnittenen Default-Konvertierungsfunktionen wurden genauer untersucht und konkret eingeführt.

Es wurden mehrere typische Schemaänderungsoperationen vorgestellt und darauf untersucht, ob sie mit den bisherigen einfachen Schemaänderungsoperationen durchführbar waren oder ob dazu komplexe Schemaänderungsoperationen nötig sind. Außerdem wurde analysiert, ob das System für die entsprechenden Operationen automatisch sinnvolle Defaultkonvertierungsoperationen generieren kann oder ob ein Eingriff des Schemaentwicklers notwendig ist. Dazu wurden sie in eine von vier Kategorien eingeteilt, die aussagen, ob einfache oder komplexe Schemaänderungsoperationen nötig sind und ob sinnvolle Default-Konvertierungsfunktionen ohne Eingriff des Schemaentwicklers erzeugt werden können oder nicht.
Zu jeder der aufgezählten Schemaänderungsoperationen wurde die entsprechende vom System erzeugte Default-Konvertierungsfunktion aufgeführt und im Falle, dass sie der Schemaentwickler überprüfen muss, angegeben, wo noch potenzieller Bedarf für manuelle Änderungen vorliegt.

Die Auswirkungen der Einführung von komplexen Schemaänderungsoperationen auf die Propagation wurde im nächsten Kapitel analysiert und dabei festgestellt, dass das bisherige Konzept der Propagationskanten zwischen je zwei Objektversionen desselben Objekts nicht mehr ausreicht. Entsprechend wurde das neue Konzept von kombinierten Propagationskanten entwickelt, das Kanten zwischen mehr als nur zwei Objektversionen zulässt. Dazu wurden verschiedene Lösungsmöglichkeiten miteinander verglichen.
Weiter wurden verschiedene Ansätze für die Darstellung und Speicherung von Konvertierungsfunktionen vorgestellt und entschieden, die Konvertierungsfunktionen konkret in textueller Darstellung in den Propagationskanten zu speichern.

Für die Spezifizierung der gewünschten Konvertierungen bei der Propagation wurde eine Konvertierungssprache entwickelt und nach verschiedenen Gesichtspunkten konzipiert. Diese Gesichtspunkte umfassen sowohl den nötigen Funktionsumfang der Sprache wie auch entwurfstechnische Aspekte. Sämtliche Befehle der entwickelten Sprache wurden detailliert vorgestellt und abschließend die Sprache in BNF (Backus-Naur-Form) präsentiert.

COAST ist inzwischen als Prototyp implementiert und wurde u.a. auf der CeBIT '99 vorgestellt (s. [Lau99b] und [LDHA99]). Nach einer Beschreibung der Funktionsweise und des Aufbaus von COAST und insbesondere des Propagationsmanagers wurden einige Implementierungsdetails vorgestellt und verschiedene Betrachtungen zur möglichen Optimierung beschrieben.

Die Ziele der Diplomarbeit wurden damit erreicht: Die Schemaevolution kann mit den Vorzügen der Versionierung durchgeführt werden. Komplexe Schemaänderungen sind nun möglich und wurden ins Modell integriert. Die Propagation wurde entsprechend erweitert und eine Sprache zur Spezifikation der Propagation entwickelt.

8.2 Ausblick

Das Konzept von COAST bietet noch einige Erweiterungsmöglichkeiten.

Zunächst wäre denkbar, den Befehlsumfang der Konvertierungssprache zu erweitern, vielleicht mit dem Ziel, einen möglichst großen Teil der Programmiersprache C anbieten zu können. In diesem Zusammenhang kann auch die Vorschaltung des C-Präprozessors implementiert werden: Man erhielte dadurch mit wenig Aufwand die Möglichkeit, Makros und Includes zu verwenden.
Es sollten die bisherigen Anstrengungen, die Implementierung effizient zu machen, indem

beispielsweise die verzögerte Propagation entwickelt wurde, weitergetrieben werden.
So wäre eine nähere Untersuchung von Optimierungsmöglichkeiten ein lohnendes Ziel, da COAST im momentanen Zustand viele Daten redundant speichert. Der Entwurf eines Optimierungsmanagers, der in die Propagation eingreift oder in Leerlaufzeiten den Bestand der Datenbasis auf unnötige Redundanzen überprüft, könnte den Platzbedarf der Datenbank deutlich senken.
Die Erweiterung des in COAST implementierten graphischen Schemaeditors um einen Editor für Konvertierungsfunktionen, der vielleicht auch gleich bei der Eingabe Konsistenzprüfungen und Optimierungen vornehmen könnte, würde dem Schemaentwickler die Arbeit entschieden erleichtern.
Bisher basiert COAST auf dem Produkt EOS der Firma AT&T (s. [BP94]), das die persistente Speicherung der Daten organisiert. Eine Umstellung von COAST auf ein selbstentwickeltes Fundament würde insofern eine Verbesserung darstellen, dass COAST dann ohne eventuelle rechtliche Probleme zum Verkauf angeboten oder auch als Freeware im Internet zur Verfügung gestellt werden könnte.

Letztlich ist das Ziel der Modellierung einer Datenbank, reale Zusammenhänge möglichst genau abzubilden. Eine Unterstützung der Verwendung und Propagation von Methoden als Teil von Objekten wäre zu begrüßen, da in dem Falle eine bessere Modellierung der Diskurswelt in der Datenbank möglich wäre.

Literaturverzeichnis

[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, J.-M. Nicolas und S. 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.

[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, Spain, September 1991. Morgan Kaufmann Publishers.

[Apo00] Sabbas Apostolidis. Der Schema-Evolutions-Assistent (SEA): Ein Werkzeug zur Unterstützung evolutionärer Schemaänderungen in einem objektorientierten Datenbankmanagementsystem mit Schemaversionierung. Diplomarbeit, University of Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Germany, Juni 2000.

[ASS91] Harold Abelson, Gerald Jay Sussman und Julie Sussman. *Struktur und Interpretation von Computerprogrammen.* Springer-Verlag, Berlin; Heidelberg; New York, 1991.

[Bab90] Robert L. Baber. *Fehlerfreie Programmierung für den Zauberlehrling.* Oldenbourg Verlag, München, 1990.

[Bal96] Helmut Balzert. *Lehrbuch der Software-Technik: Software-Entwicklung.* Lehrbücher der Informatik. Spektrum Akademischer Verlag, Heidelberg; Berlin; Oxford, 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.

[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, United Kingdom, September 1995. Springer-Verlag. Lecture Notes in Computer Science No. 978.

[Boo94] Grady Booch. *Objektorientierte Analyse und Design: Mit praktischen Anwendungsbeispielen.* Addison Wesley, Reading, MA, 1994.

[BP94] Alexandros Biliris und Euthimios Panagos. *EOS User's Guide.* AT&T Bell Laboratories, Murray Hill, NJ 07974, Oktober 1994. Release 2.2, 51 pages.

[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, Australia, August 1990. Morgan Kaufmann Publishers.

[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.

[Cox91] Brad J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison Wesley, Reading, MA, 2. Auflage, 1991.

[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, University of Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Germany, Dezember 1999.

[Dud93] Duden. *Duden Informatik.* Bibliographisches Institut & F.A. Brockhaus, Mannheim, 2. Auflage, 1993.

[Eig97] Patricia Eigner. Objektpropagation in einem Schemaversionierungsansatz: Entwicklung und prototypische Implementierung in einem objektorientierten Datenbanksystem. Diplomarbeit, University of Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Germany, Juli 1997.

[FMZ+95] 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, Zurich, Switzerland, September 1995. Morgan Kaufmann Publishers.

[GLG95] Heino Gärtner, Sven-Eric Lautemann und Martin Gogolla. *(unveröffentlichter Draft).* -, 1995.

[Gro00] Michael Großmann. Der COAST-Schemaeditor: Eine Internet-basierte, plattformunabhängige graphische Benutzungsoberfläche für die Erstellung und Modifikation versionierter Schemata einer objektorientierten Datenbank. Diplomarbeit, University of Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Germany, Februar 2000.

[Her95] Helmut Herold. *lex und yacc: Lexikalische und syntaktische Analyse.* Addison Wesley, Bonn, 2. Auflage, 1995.

[Her99] Detlef Herchen. Schnittstellen zur textuellen Beschreibung versionierter Schemata in objektorientierten Datenbankmanagementsystemen. Diplomarbeit, University of Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Germany, März 1999.

[Heu97] Andreas Heuer. *Objektorientierte Datenbanken: Konzepte, Modelle, Standards und Systeme.* Addison Wesley, 2. Auflage, 1997.

[Kat90] Randy H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, Dezember 1990.

[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.

[Kop95] Helmut Kopka. *LaTeX: Ergänzungen - mit einer Einführung in METAFONT.* Addison Wesley, Bonn, 1995.

[Kop96] Helmut Kopka. *LaTeX: Einführung.* Addison Wesley, Bonn, 2. Auflage, 1996.

[KS96] Gerti Kappel und Michael Schrefl. *Objektorientierte Informationssysteme: Konzepte, Darstellungsmittel, Methoden.* Springer-Verlag, Wien, 1996.

[Lau96] 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, Zurich, Switzerland, September 1996. IEEE Computer Society Press. Workshop Proceedings.

[Lau97] 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, U.K., April 1997. IEEE, IEEE Computer Society Press.

[Lau99a] Sven-Eric Lautemann. The COAST Project Homepage. http://www.coast.uni-frankfurt.de/, 1995-1999.

[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. *Schemaevolution in objektorientierten Datenbanksystemen auf der Basis von Versionierungskonzepten.* Dissertation, University of Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Germany, 2000.

[LDHA99] Sven-Eric Lautemann, Alexander Doll, Jan Haase und Sabbas Apostolidis. 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 No. 1743. Poster Demonstration.

[Ler94] Barbara Staudt Lerner. Type Evolution Support for Complex Type Changes. Technical Report 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. Technical Report 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.

[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, Switzerland, Dezember 1997. Springer-Verlag. Lecture Notes in Computer Science No. 1341.

[LS94] Gamal Labib und Dave Saunders. A Class Versioning Approach for OODBS. In Michael F. Worboys und A. F. Grundy, Hrsg., *Proc. of the 12th British National Conf. on Databases (BNCOD)*, Seiten 9–12, Guildford, United Kingdom, Juli 1994. Poster Summaries.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Nic75] John E. Nicholls. *The Structure and Design of Programming Languages*. Addison Wesley, Reading, Mass., 1975.

[O2 96] O2 Technology, Versailles Cedex, France. *O_2C Reference Manual, Version 4.6*, April 1996.

[Pri98] Manfred Prien. Entwurf und Implementierung eines Schema-Managers für ein objektorientiertes Datenbanksystem mit Schemaversionierung. Diplomarbeit, University of Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Germany, April 1998.

[RR95] Young-Gook Ra und Elke A. Rundensteiner. A Transparent Object-Oriented Schema Change Approach Using View Evolution. In P. S. Yu und A. L. P. Chen, Hrsg., *Proc. of the 11th Int'l Conf. on Data Engineering (ICDE)*, Seiten 165–172, Taipei, Taiwan, März 1995. IEEE, IEEE Computer Society Press.

[Sch92] Uwe Schöning. *Theoretische Informatik kurz gefasst*. BI-Wissenschafts-Verlag, Mannheim, 1992.

[Sci91] Edward Sciore. Multidimensional Versioning for Object-Oriented Databases. In C. Delobel, M. Kifer und Y. Masunaga, Hrsg., *Proc. of the 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Seiten 167–188, Munich, Germany, Dezember 1991. Springer-Verlag. Lecture Notes in Computer Science No. 566.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 3. Auflage, 1997.

[STS97] Gunter Saake, Can Türker und Ingo Schmitt. *Objektdatenbanken*. International Thomson Publishing, Bonn, 1997.

[Sys00] Wind River Systems. The SNiFF+ Homepage. http://www.takefive.com/products/sniff+.html, 1999-2000.

[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, Ireland, August 1993. Morgan Kaufmann Publishers.

[Wöh96] Christian Wöhrle. Schemaversionierung in objektorientierten Datenbanksystemen. Diplomarbeit, University of Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Germany, April 1996.

[Wöl98] Kay Wölfle. Verwaltung und Propagation persistenter Objekte in einem Schemaversionierung unterstützenden objektorientierten Datenbanksystem. Diplomarbeit, University of Frankfurt, Robert–Mayer–Str. 11–15, D-60325 Frankfurt/Main, Germany, 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.

[Zic97] Roberto Zicari. Schema and Database Evolution in Object Database Systems. In *[ZCF+97]*, Kapitel 6, Seiten 411–495. Morgan Kaufmann Publishers, 1997.