Schemaintegration am Beispiel der Produktdatentechnologie

Diplomarbeit am
Lehrgebiet Praktische Informatik I
der Fernuniversität Hagen

Betreuer:
Prof. Dr. Gunter Schlageter

Autor:
Volker Wendrich, Mühlfeldstr. 33, 82216 Maisach
Volker.Wendrich@altavista.net, Mat.-Nr. 4362292

1. Juli 2000

Bestätigung

Hiermit bestätige ich, die Arbeit selbständig verfaßt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und Zitate kenntlich gemacht zu haben.

Abstract

Diese Arbeit vergleicht bekannte Methoden der Schemaintegration bezüglich ihrer Eignung zur Zusammenführung von Produktdatenmodellen. Dabei wird angenommen, daß alle zu integrierenden Schemata einheitlich in der ISO genormten Sprache EXPRESS vorliegen. Der beste Ansatz wird ausgewählt und implementiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation zur Produktdatenintegration

Im internationalen Step (Standard for the Exchange of Product Model Data) Projekt besteht das Ziel, Werkzeuge wie CAD-Tools, Simulatoren und Flächenoptimierer interoperabel zu machen, um Produktdaten unter den Werkzeugen austauschen zu können. Hierzu werden standardisierte Beschreibungsarten für die Produkte entwickelt. In der Produktdatentechnologie hat man es mit komplexen Arbeitsprozessen zu tun, die nicht nur sequentiell ablaufen. Deshalb geht die Zusammenarbeit über binäre Schnittstellen zwischen jeweils zwei Werkzeugen hinaus. Als Folge wird ein integriertes Schema der Produktdaten erforderlich, auf das die einzelnen Werkzeuge überlappende Sichten besitzen. Der Datenaustausch erfordert, daß jedes Werkzeug Updates gegen das Teilschema, auf dem es arbeitet, ausführen kann. Dabei handelt es sich um die wesentliche Hürde, da es für nur lesende Systeme bereits Lösungen gibt (z.B. sogenannte "Data Warehouse Systeme").

In der vorliegenden Arbeit wird die Schemaintegration am Beispiel der Sprache EXPRESS erforscht. Dies hat folgende Gründe: Schemaintegration ist schon lange von der Integration heterogener Datenbanken her bekannt. Dort muß zunächst eine Transformation in ein einheitliches Datenmodell vorgenommen werden. Semantisch reiche Modellierungssprachen haben hier einen geringeren Informationsverlust als ärmere. Deshalb werden sie trotz der schwereren Systematisierbarkeit des Integrationsprozeß oft verwendet. Die Modellierungssprache EXPRESS [EXPR94] ist aufgrund ihres Ursprungs in der Entity Relationship Modellierung und ihres semantischen Reichtums gut geeignet [Tah98]. Beispiele sind Integritätsbedingungen und objektorientierte Konzepte wie Mehrfachvererbung. Für EXPRESS sprechen auch die Standardisierung (ISO10303-11), das mögliche Konfigurationsmanagement aufgrund der ASCII-Darstellung und die Verbreitung in der Industrie. In der Praxis ist eine Transformation in EXPRESS unnötig, wenn alle beteiligten Firmen bereits EXPRESS verwenden. Trotzdem kann das Problem auftreten, daß die verwendeten Teilmodelle nicht alle zu den der ISO10303 Norm untergeordneten Applikationsmodellen, wie AP203 für die 3D-Geometrie, konform sind. Der hierfür noch nötige Integrationsschritt gab den Anstoß für diese Arbeit.

1.2 Motivation zur Automatisierung

Viele Schwierigkeiten bei der Integration, beispielsweise Namensgebung oder Datenkonflikte, sind seit längerem bekannt und finden sich unter anderem in

[Tah98], [Bat86], [Kurs1666], oder [Con97]. In der Produktdatentechnologie kommt hinzu, daß die Modelle sehr groß sein können und starke Unterschiede im Detaillierungsgrad (der Granularität) möglich sind. Die Modelle unterscheiden sich also wesentlich im Grad der Abstraktion und der Struktur. Die Auflösung dieser Unterschiede sollte algorithmisch unterstützt werden, da die Gefahr besteht, bei manueller Vorgehensweise den Überblick zu verlieren.

1.3 Aufgabenstellung

Am Anfang der Arbeit ist eine Einführung und Begriffserklärung nötig. Dies betrifft sowohl EXPRESS als auch die Schemaintegration. Erstes Teilziel ist danach, bestehende Ansätze hinsichtlich ihrer Anwendbarkeit für die Integration von Produktdatenmodellen, die in EXPRESS formuliert sind, zu vergleichen. Kriterien sind Vollständigkeit, Minimalität, Korrektheit, Verständlichkeit und Redundanzfreiheit des erzeugten integrierten Schemas. Dazu sollen bekannte Ansätze zur Integration vorgestellt und bewertet werden. Der beste, ggf. modifizierte, Ansatz soll Vorgabe für die zweite Phase werden.

Ziel der zweiten Phase ist, die Schemaintegration durch ein Programm zu unterstützen. Eingabe sind (zwei) Express-Schemata und die Information über den Zusammenhang dieser Modelle (Inter-Schema-Korrespondenzen). Ausgabe ist ein integriertes Schema und ein Log der einzelnen Schritte. Das Log soll als Eingabe für weitere Vorhaben, wie der schrittweisen Transformation von EXPRESS Modellen (ESPRIT Projekt 25110), dienen.

2 Übersicht

Kapitel 3 erklärt die wesentlichen Sprachkonstrukte von EXPRESS: einfache und komplexe Datentypen, Vererbung, Schlüssel und weitere Integritätsbedingungen.

In Kapitel 4 erfolgt eine Einführung in die Schemaintegration. Es werden Schwierigkeiten bei der Automatisierung, die Eingabe in Form von Inter-Schema-Korrespondenzen und Kriterien für ein gutes Ergebnisschema genannt. Am Kapitelende wird ein sehr simples Verfahren zur Schemaintegration vorgestellt.

Die folgenden vier Kapitel beschreiben je eine komplexere Methodik. Die Auswahl wurde anhand der Abdeckung der Konstrukte von EXPRESS und der Eignung für die Produktdatenintegration getroffen. So behandeln die Kapitel 5, 7 und 8 Aspekte komplexer Datentypen.

Kapitel 5 beschreibt ein Verfahren, das hierarchisch konstruierte Datentypen schrittweise auf eine Normalform transformiert, um die Strukturen besser vergleichbar zu machen. Die Teilschritte können in einer großen Transformation zusammengefaßt werden, um die komplexen Objekte zu transformieren.

In Kapitel 6 werden Vererbungshierarchien vereinigt. Hierbei entstehen neue, als "implizit" bezeichnete Klassen. Der Schwerpunkt liegt auf syntaktisch korrekter Integration.

Kapitel 7 behandelt die Problematik redundanter Pfade durch das Schema. Typisch für komplexe Objekte ist, daß sich die Klassen der Teilschemata nicht eins zu eins entsprechen und somit korrespondierende Attribute nicht immer in korrespondierenden Klassen liegen. Dieses und das nächste Kapitel bieten hierzu einander ergänzende Lösungen. Falls Daten migriert werden sollen, legt SIM die Erzeugung neuer, "virtuell" genannter, Objekte nahe.

Kapitel 8 zeigt, wie Schlüssel Hinweise auf fehlende Beziehungen von einem Teilschema zum anderen geben können (Inter-Schema-Beziehungen). Die zwei vorgestellten Arbeiten entstammen dem Gebiet der View-Integration. Sie versuchen, Integritätsbedingungen als Hilfsmittel zur Schemaintegration zu verwenden. Grundkriterium für das globale Schema ist, daß die einzelnen Views daraus ableitbar sein müssen.

In Kapitel 9 werden die Ansätze kombiniert, wodurch sich Vorschläge für weitere Forschungsprojekte ergeben. Danach wird der beste Ansatz für die folgende Implementierung ausgewählt.

In Kapitel 10 wird die Implementierung der gewählten Methode beschrieben. Es werden die hierfür benötigten Datenstrukturen und Algorithmen vorge-

stellt und die Korrektheit der Vorgehensweise begründet. Anschließend wird die Ausgabe eines Testbeispiels gezeigt.

Kapitel 11 ist eine Gesamtzusammenfassung der Ergebnisse.

3 Einführung in EXPRESS

3.1 Kapitelüberblick

Express ist eine Datenspezifikationssprache, deren festgelegte Syntax es ermöglicht, Datenmodelle in einem lesbaren Format auszutauschen. Das Kapitel stellt die wesentlichen Sprachelemente von EXPRESS vor. Es unterteilt sich in Datentypen und Integritätsbedingungen. Aus Basisdatentypen lassen sich mit Hilfe von Typkonstruktoren beliebig komplexe Typen zusammenbauen. Integritätsbedingungen lassen sich in Vererbungshierarchien, referentielle Integrität, Schlüssel und weitere, nicht in diesen drei Arten enthaltene, untergliedern.

3.2 Einfache und konstruierte Datentypen

Einfache Datentypen sind unter anderem INTEGER, REAL und STRING. Um unterschiedliche Bedeutungen besser zu trennen, können die Basistypen benannt werden.

Beispiel für einen benannten Basistyp:

```
TYPE lautstärke = REAL;
END_TYPE;
```

Die ENUMERATION ermöglicht die Festlegung einer endlichen Anzahl von Alternativen.

Beispiel für eine ENUMERATION:

```
TYPE richtung = ENUMERATION OF (oben, unten, rechts, links);
END_TYPE;
```

Um aus den einfachen Datentypen komplexe Typen konstruieren zu können, gibt es die Schlüsselworte ENTITY, SET, BAG und SELECT.

Mit Hilfe des ENTITY Konstrukts, auch ENTITY-Typ genannt, können Tupel von Werten zusammengefaßt werden. Diese Werte können sowohl Basisdatentypen als auch konstruierte Typen sein. Ein ENTITY wird z.B. wie folgt definiert:

Beispiel für ein ENTITY:

```
ENTITY Person;
  Name, Anschrift: STRING;
END_ENTITY;
```

Dem ENTITY-Typ kommt in EXPRESS eine besondere Bedeutung zu, denn die im nächsten Abschnitt beschriebenen Vererbungsbeziehungen werden nur mit den ENTITY-Typen verknüpft. Dies legt den Gedanken nahe, real existierende Objekte müßten zu ENTITY-Typen gehören. Für die Integration umfangreicher Schemata ist diese Auffassung jedoch sehr hinderlich, da aufgrund von Unterschieden im Detaillierungsgrad beispielsweise ein ENTITY-Typ in Schema A einem aus mehreren ENTITY-Typen zusammengesetzten Konstrukt in Schema B entsprechen kann.

EXPRESS bietet auch Mengen- und Variantenkonstrukte, die mit Hilfe von TYPE benannt werden können. Aggregationen mit Duplikaten werden mit dem Schlüsselwort BAG deklariert, Mengen mit SET.

Beispiel für einen benannten Mengentyp (Bücher):

```
ENTITY Buch
  ISBN: STRING;
END_ENTITY;

TYPE Bücher = SET OF Buch;
END_TYPE;
```

Mit dem Variantenkonstruktor, der durch das Schlüsselwort SELECT eingeleitet wird, können Objekte je nach Kontext unterschiedlich zusammengesetzt werden, wobei immer genau eine der Alternativen zutrifft.

Beispiel für Varianten:

```
ENTITY CD
  Bezeichnung: STRING;
  Spieldauer: INTEGER;
END_ENTITY;

TYPE Verleihgegenstand = SELECT (Buch, CD);
END_TYPE;
```

Varianten sind auch praktisch, um Amtsformulare darzustellen, in denen z.B. nur Verheiratete den Namen des Ehepartners ausfüllen müssen.

Die drei Konstruktorarten können beliebig geschachtelt werden. Dadurch sind sie sehr mächtig. Das Konzept der relationalen Datenbanken ist darin enthalten, wie das folgende Beispiel verdeutlicht.

Beispiel einer “relationalen Datenbank”:

```
ENTITY Datenbank
  Kunden: SET OF Kunde_Tupel;
  Artikel: SET OF Artikel_Tupel;
END_ENTITY;

ENTITY Kunde_Tupel
  Name: STRING;
  Adresse: STRING;
END_ENTITY;

ENTITY Artikel_Tupel
  Nr: INTEGER;
END_ENTITY;
```

Die konstruierten Datentypen müssen nicht rein hierarchisch aufgebaut sein. Sie können auch Querverbindungen haben oder rekursiv sein.

Tupel-, Mengen,- und Variantenkonstruktor stammen geschichtlich von den semantischen Datenmodellen ab, deren erstes das Entity Relationship Modell war. Eine guter Überblick ist durch [Hul87] zu erlangen.

Getrennt hiervon haben sich die objektorientierten Programmiersprachen und Datenbanken entwickelt. Auch von dieser historischen Entwicklung wurden einige Ideen durch EXPRESS übernommen. Beispiele finden sich im nächsten Abschnitt.

3.3 Vererbung und weitere Integritätsbedingungen

Die Semantik der Datentypen kann durch Integritätsbedingungen genauer spezifiziert werden. Als Beispiel wird versucht, ein im bekannten objektorientierten UML-Standard [Fow97] angegebenes Modell in EXPRESS auszudrücken.

Hinweis: Die Begriffe Klasse, Entity, Entity-Typ und Relation werden in dieser Arbeit synonym verwendet. Allerdings können sich komplexe Objekte über mehrere Entity-Typen erstrecken.

Abbildung 1 zeigt die Klassen “Person”, “Kunde”, “Buch” und “Präsenzbestand”, deren Attribute “Name”, “Anschrift”, “Kundennummer” und “ISBN”, die Vererbungsbeziehung “Kunde ist eine spezielle Person” und die Assoziationen “vorgemerkt”, “entliehen” und “enthalten”. Bei letzterer Assoziation

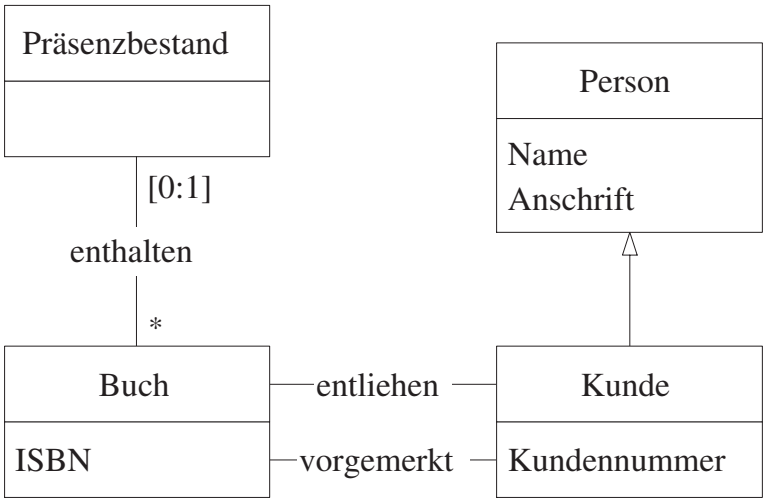

Abbildung 1: Teilschema eines Bibliotheksystems

sind Kardinalitäten angegeben. [0:1] besagt, daß ein Buch in keinem bis einem Präsenzbestand sein muß. Der Stern bedeutet, daß ein Präsenzbestand beliebig viele Bücher haben darf.

Die Grafik wird im weiteren Verlauf schrittweise in EXPRESS umgesetzt. Als erstes werden die Klassen und Attribute als ENTITY-Typen ausgedrückt.

Beispiel zur Umsetzung in Entities:

```
ENTITY Person;
  Name, Anschrift: STRING;
END_ENTITY;

ENTITY Kunde
  Kundennummer: INTEGER;
END_ENTITY;

ENTITY Buch
  ISBN: STRING;
END_ENTITY;

ENTITY Präsenzbestand
END_ENTITY;
```

Vererbung

Die Vererbungsbeziehung impliziert die Integritätsbedingung, daß jedes Objekt einer Subklasse auch zur Superklasse gehören muß. Sie wird mit den Schlüsselwörtern SUBTYPE und SUPERTYPE beschrieben.

Beispiel für eine Vererbungsbeziehung:

```
ENTITY Person;
  SUPERTYPE of Kunde
  Name, Anschrift: STRING;
END_ENTITY;

ENTITY Kunde
  SUBTYPE of Person
  Kundennummer: INTEGER;
END_ENTITY;
```

Geerbte Attribute können überschrieben werden. Dabei muß der neue Typ eine Spezialisierung des bisherigen sein.

EXPRESS bietet die Möglichkeit der Mehrfachvererbung. Werden gleichnamige Attribute von mehreren Superklassen geerbt, die nicht wiederum auf ein Super-Super-Attribut zurückzuführen sind, dann handelt es sich um unterschiedliche Attribute, die beim Zugriff mit dem Namen der entsprechenden Superklasse qualifiziert werden müssen. Wenn andererseits die mehrfach geerbten Attribute von einer gemeinsamen Superklasse stammen, dann gelten sie als ein einzelnes Attribut.

EXPRESS unterstützt auch multiple Klassifikation (ANDOR). Ein Objekt kann zu mehreren Klassen gehören, ohne daß für jede zugelassene Kombination eine Subklasse definiert werden muß. In dieser Arbeit wird hierauf nicht weiter eingegangen, da kein Algorithmus in der Literatur gefunden wurde, der zwei Schemata mit multipler Klassifikation integriert. Die Schemavereinigung ist zwar durch explizite Definition aller Subklassen möglich. Dies birgt aber die Gefahr, daß es exponentiell viele Subklassen werden können.

Assoziationen und referentielle Integrität

Assoziationen werden in EXPRESS durch ein oder zwei (inverses Attribut siehe unten) Referenzattribute beschrieben. Dies bietet die Möglichkeit der Namensgebung für Beziehungen.

Der Wertebereich eines Attributs kann zwar ein komplexer, konstruierter Typ sein. Für Assoziationen zwischen zwei Entities reichen jedoch direkte oder mengenwertige (s. u.) Referenzattribute aus. Direkt soll bedeuten, daß der Wertebereich ein anderer Entity-Typ ist und das Referenzattribut wie ein Zeiger auf ein vorhandenes Objekt dieses Typs verweist. So ermöglicht EXPRESS die Modellierung von *referentieller Integrität*.

Das Schlüsselwort OPTIONAL erlaubt den Verzicht auf das zwangsweise Vorhandensein des Zielobjekts. Im folgenden Beispiel muß die ISBN vorhanden sein, die anderen Attribute dürfen, analog zu Nullwerten, fehlen.

Beispiel für optionale Attribute:

```
ENTITY Buch
  ISBN: STRING;
  bestand: OPTIONAL Präsenzbestand;
  ausleiher: OPTIONAL Kunde;
  vormerker: OPTIONAL Kunde;
END_ENTITY;
```

Abbildung 1 auf Seite 12 sollte anhand der Assoziationen "vorgemerkt" und "entliehen" verdeutlichen, daß die Bedeutung einer Assoziation nicht immer bereits vollständig durch die angrenzenden Klassen festgelegt ist, Assoziationen also eine Bedeutung für sich allein haben können. Ein vollständiges Integrationsverfahren muß also die Semantik der Beziehungen berücksichtigen.

Entlang der Referenzattribute kann durch ein EXPRESS Schema navigiert werden. Ausgehend von einem gegebenen Entity kann eine Kette von Referenzen, "Navigationspfad" genannt, durchlaufen werden, um an den benötigten Wert zu gelangen. Im relationalen Fall würde diese Aufgabe von einem Join übernommen. Wenn aber der Navigationspfad mehr als vier Schritte lang ist, dann wird ein äquivalenter SQL Join unleserlich. Aus diesem Grund ist die Navigation mit Hilfe der Referenzattribute sehr nützlich.

Am Anfang dieses Kapitels (Abschnitt 3.2) kam bereits das Attribut "Kunden" vor. Dabei handelte es sich um ein mengenwertiges Referenzattribut (SET OF Kunde_Tupel). Bei Mengenattributen kann die Anzahl der Elemente durch die Angabe von Kardinalitäten, z.B. [0:5] für 0 bis 5 Bücher oder [0:?] für beliebig viele Bücher, eingeschränkt werden.

Beispiel für Mengen mit Kardinalitätsangabe:

```
ENTITY Kunde
  vorgemerkt: SET [0:5] OF Buch;
  geliehen: SET [0:10] OF Buch;
END_ENTITY;
```

Beispiel für "beliebig viele" Bücher:

```
ENTITY Präsenzbestand
  bücher: SET [0:?] OF Buch;
END_ENTITY;
```

Referenzen sind oft zueinander invers. Wenn also ein Kundenobjekt eine Referenz auf ein ausgeliehenes Buch hat, muß die umgekehrte Referenz vom Buch her gesehen auf denselben Kunden verweisen.

Beispiel für ein inverses Attribut:

```
ENTITY Kunde
...
INVERSE
geliehen:
  SET [0:10] OF Buch FOR ausleiher;
END_ENTITY;
```

Ausleiher ist das Attribut des Entity Buch mit der direkten Referenz zu Kunde. Jeder Kunde kann null bis zehn Bücher leihen. Durch Festlegung der Minimalkardinalität größer als Null kann die *Totalität* der Beziehung erzwungen werden.

In der Produktdatentechnologie werden oft Stücklisten auf der Schemaebene rekursiv beschrieben. Auf Objektebene wird dagegen meist eine hierarchische Struktur verlangt. Diese Baumstruktur ergibt sich aus der Bedingung von Aggregationen, daß jedes Element in genau einen Container eingefügt werden muß. Mit Hilfe der ROLESOF-Funktion kann sichergestellt werden, daß ein Entity nicht in mehrere Mengen eingefügt werden darf. Im folgenden Beispiel dürfen Präsenzbücher aufgrund ROLESOF nur vom Entity-Typ Präsenzbestand, aber nicht von anderer Stelle referenziert werden.

Beispiel für ROLESOF:

```
ENTITY Präsenzbestand
  präsenzbücher: SET OF Präsenzbuch
END_ENTITY;

ENTITY Präsenzbuch
  ISBN: STRING;
INVERSE
  bestand: Präsenzbestand FOR präsenzbücher;
WHERE
  ROLESOF (SELF) = ['Präsenzbestand.präsenzbücher']
END_ENTITY;
```

Schlüssel

Durch Angabe von UNIQUE können ein oder mehrere Attributkombinationen als *Schlüssel* festgelegt werden.

Beispiel für Schlüssel:

```
ENTITY Kunde
  Nr: INTEGER;
  Name: STRING;
  Adresse: STRING;
UNIQUE
  u1: Nr;
  u2: Name, Adresse;
END_ENTITY;
```

Weitere Integritätsbedingungen und abgeleitete Daten

Neben der referentiellen Integrität und den Schlüsseln ist es möglich, weitere, durch Programmfragmente beschriebene Integritätsbedingungen und abgeleitete Daten in das Schema einzubetten. Ein Beispiel für Integritätsbedingungen der komplexeren Art ist das folgende: Ein Buch kann nur entliehen oder vorgemerkt werden, wenn es nicht im Präsenzbestand ist. Ein Kunde darf für ein bestimmtes Buch nur entweder Ausleiher oder Vormerker sein.

Beispiel für eine komplexe Integritätsbedingung:

```
ENTITY Buch
...
WHERE
SIZEOF(
 USEDIN(SELF,
       'Präsenzbestand.Bücher') +
 USEDIN(SELF,
       'Kunde.geliehen' +
 USEDIN(SELF,
       'Kunde.vorgemerkt')) <= 1;
END_ENTITY;
```

Abgleitete Attribute enthalten Daten, die aus den Basisdaten mit Hilfe von Funktionen berechnet werden können. Das folgende einfache Beispiel zeigt die Verwendung eines redundanten Attributs.

Beispiel für ein abgeleitetes Attribut:

```
ENTITY Buch
  ausleiher: Kunde;
DERIVE
  ausleiher_name: STRING := ausleiher.name;
END_ENTITY;
```

3.4 Kapitelzusammenfassung

Die gleiche Realwelt kann mit EXPRESS unterschiedlich modelliert werden. Dies betrifft beispielsweise:

- Varianten, Vererbung und optionale Attribute
- abgeleitete Daten und Integritätsbedingungen
- die Richtung von Referenzen

Da es vermutlich in der Literatur keine Arbeit gibt, die alle oder auch nur mehrere dieser Modellierungskonflikte behandelt, wird hier der Weg eingeschlagen, sich nicht zu sehr auf Details einzulassen, sondern stattdessen vier Schwerpunkte zu bilden. Dabei handelt es sich um die Konstruktion von Datentypen, um Vererbungshierarchien, um referentielle Integrität mit der Möglichkeit einer Navigation durch das Schema, sowie um Schlüssel und

kompliziertere Integritätsbedingungen. Im weiteren Verlauf werden diese vier Gebiete jeweils anhand einer Methode zur Schemaintegration vertieft und anschließend im Zusammenhang betrachtet.

4 Einführung in die Schemaintegration

4.1 Kapitelüberblick

Dieses Kapitel beginnt mit einer Schilderung der Schwierigkeiten automatischer Schemaintegration. Danach werden Inter-Schema-Korrespondenzen, welche die Eingabe der Integrationsverfahren darstellen, beschrieben. Anschließend werden Kriterien für ein gutes Ergebnisschema genannt. Am Ende des Kapitels wird ein sehr simples Verfahren zur Schemaintegration vorgestellt.

4.2 Heutiger Stand

Die Schwierigkeit der Schemaintegration wird in der Literatur sehr unterschiedlich dargestellt. Was spricht für eine hohe Schwierigkeit?

Donald Sanderson, einer der Autoren des EXPRESS-Standards, beschreibt in seiner Arbeit [San94] das spezielle Problem der Erhaltung der Semantik von Objekten, die zwar aus heterogenen Datenbanken stammen, aber im wesentlichen gleichartig aufgebaut sind. Das allgemeine Problem der Integration beliebiger Strukturen wird nicht gelöst. Sanderson führt das Beispiel eines geometrischen, CAD-beschriebenen Körpers an, der per "boundary representation" oder "constructive solid geometry" vollkommen unterschiedlich beschrieben sein kann, um zu erklären, welche Art von Problemen in seiner Arbeit nicht behandelt werden. Es gäbe zwar Transformationen zur Abbildung der Objekte, jedoch existierten keine Verfahren, um solche Transformationen automatisch zu generieren.

Große Schwierigkeiten verursachen auch rekursive Datenstrukturen, wenn zum Beispiel bei zwei Abstammungs-Datenbanken die Verkettung entgegengesetzt ist, in einer von den Eltern zu den Kindern, in der anderen von den Kindern zu den Eltern. Für die Abbildung der Objekte aus rekursiven Strukturen gibt es spezielle Sprachen, wie WOL [WOL94]. Jedoch fehlt wiederum die Computerunterstützung bei der Erstellung der Abbildung.

In der Arbeit zur Methode GIM [GIM97] wird argumentiert, daß meist nur binäre Beziehungen der in der Realwelt vorkommenden Objektmengen betrachtet würden. Dies sei für ein Mengenproblem wie dem Schnitt dreier Mengen nicht ausreichend.

Ein anderes, häufig genanntes pessimistisches Argument ist die Unentscheidbarkeit der Probleme, ob eine Menge referentieller Integritätsbedingungen

widerspruchsfrei ist und ob eine Bedingung aus den anderen abgeleitet werden kann, also redundant ist [Mitc83, Ek95]. Eine Auswirkung ist, daß die Streichung redundanter Integritätsbedingungen, die einer Umgestaltung des Schema gerade im Wege sind, nicht automatisch erfolgen kann. Jeder Algorithmus unterbricht also mitunter öfter als nötig, um den Anwender zu fragen, wie verfahren werden soll. Nötig soll bedeuten, daß Widersprüche aufgrund von Modellierungsfehlern und Inkompatibilitäten der Schemata gemeldet und nicht übersehen werden. Eine halbautomatische Integration, die viele Fehler frühzeitig erkennt, kann den Gesamtprozeß möglicherweise stärker beschleunigen, als ein fiktiver vollautomatischer Algorithmus, der schwere Fehler übersieht.

Für die Produktdatentechnologie ist, wie gesagt, die Integration komplexer Objekte wichtig. Sie ist laut einer Zusammenfassung über den Stand der Datenbankintegration von Parent und Spaccapietra [PS98] noch nicht ausreichend erforscht. Jedoch gibt es Lösungen für Teilaspekte, deren Darstellung einen Schwerpunkt dieser Arbeit bildet.

Warum erscheint die Schemaintegration dennoch in vielen Arbeiten als leicht?

Aufgrund des begrenzten Umfangs der Werke muß, besonders bei semantisch reichen Modellen, eine Teilmenge der Konfliktarten ausgewählt werden. Eine günstige Auswahl erleichtert das Gesamtproblem. Außerdem fehlt eine exakte Problemdefinition, was vereinfachende Annahmen zuläßt. Beides zusammen führt oft zu einem vereinfachten, lösbaren Restproblem.

4.3 Inter-Schema-Korrespondenzen

Alle Integrationsmethodiken versuchen auf unterschiedliche Weise, Konzepte des einen mit solchen des anderen Schemas in Beziehung zu setzen. Um solche Beziehungen auszudrücken, werden *Inter-Schema-Korrespondenzen* definiert.

Zum einen werden die in der Realwelt möglichen Populationen der Klassen über Inklusionsbeziehungen verglichen. Dies ist zur Erkennung korrespondierender Klassen und zur Integration der Vererbungshierarchien nützlich. Ein Vertreter dieser Vorgehensweise ist die bereits angesprochene Methode GIM. GIM verwendet jedoch im Gegensatz zu EXPRESS ein eigenes, semantisch armes, Datenmodell. In [Con97] werden noch weitere, ähnliche Methoden beschrieben, z.B. “Upward Inheritance” und “Formalisierte objektorientierte Integration”. Diesen Arbeiten ist gemeinsam, daß komplexe Objekte mit ihren Beziehungsgeflechten bei den dortigen Schematransformationen nicht unterstützt werden. Aus diesem Grund wird der Ansatz des Populationsvergleichs hier nicht weiter verfolgt.

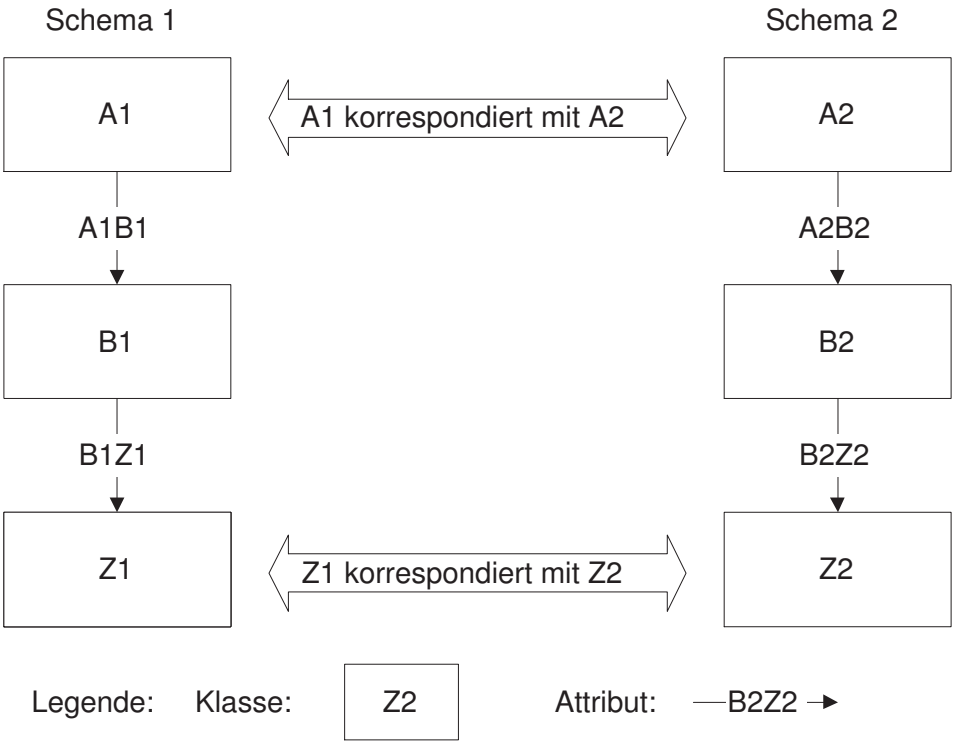

Abbildung 2: Inter-Schema-Korrespondenzen

Zum anderen können informationstragende Modellelemente korrespondieren. Von diesen gibt es vier "atomare" Arten: Attribute, Entities, Assoziationen und Metadaten. Ein Beispiel für Metadaten ist die Zugehörigkeit zu einer bestimmten Klasse. Eine Korrespondenz zwischen Attribut und Metainformation besteht, wenn ein n-wertiges Attribut in Schema A mit der Instanzeigenschaft von n (Sub-)Klassen in Schema B korrespondiert. Die Erkennung von Korrespondenzen zwischen informationstragenden Elementen ist zur Redundanzvermeidung wichtig. Bei den im weiteren Verlauf schwerpunktmäßig vorgestellten Methoden werden Korrespondenzen zwischen Attributen in Kapitel 6, 7 und 8 ausgewertet. Korrespondenzen zwischen Assoziationen werden in Kapitel 6 berücksichtigt. Die Korrespondenz eines Entity mit einer Beziehung kommt in Kapitel 7 vor.

Über den Vergleich der atomaren Elemente hinaus können Korrespondenzen zwischen zusammengesetzten Schemastrukturen aufgestellt werden. Die Objekte der mit Hilfe von ENTITY, SET und SELECT konstruierten Datentypen können als komplex aufgebaute Werte aufgefaßt werden. Dieser Ansatz wird in Kapitel 5 wieder aufgegriffen.

Eine weitere Vergleichsmöglichkeit betrifft *Pfade*. Hier wird neben der eigentlichen Information auch der Weg dorthin in Betracht gezogen. Wird eine Kette von Objektreferenzen entlang der Assoziationen eines Schemas durchlaufen, ergibt sich ein Pfad, der auch Navigationspfad genannt wird. Korrespondierende Pfade beginnen mit einem Objekt einer korrespondierenden Klasse und verlaufen dann auf möglicherweise unterschiedlichen Wegen durch ihr jeweiliges Teilschema, um letztlich zu einem Objekt einer zweiten korrespondierenden Klasse oder zu einem korrespondierenden Attribut zu gelangen. Dies allein genügt jedoch noch nicht für eine *Pfadkorrespondenz*: Werden die Pfade auf Schemaebene ermittelt, muß zusätzlich anhand der Realwelt und/oder anhand des tatsächlichen Datenbankinhalts entschieden werden, ob das Zielobjekt bei gleichem Startobjekt immer ebenfalls gleich sein muß. Beispielsweise könnten im Schema einer Bank die Entities Kunde, Bankberater und Adresse vorkommen. Ein Kunde habe eine Adresse und einen Bankberater, der ebenfalls eine Adresse hat. Bei bloßer Betrachtung des Schemas könnte man meinen, daß redundante Pfade vorliegen. Eine entsprechende Pfadkorrespondenz[1] wäre (Kunde -> Adresse, Kunde -> Bankberater -> Adresse). Jedoch ist die Adresse eines Kunden ein anderes Objekt als die Adresse des Bankberaters desselben Kunden. Also liegt keine Redundanz vor, und die Pfadkorrespondenz ist nicht sinnvoll.

Die Navigation entlang der Pfade funktioniert bei Mengentypen nicht, da dort eine Auswahl des nächsten Objekts getroffen werden muß. Jedoch kann man auch allgemeiner die Mengen der Paare von Objekten, die über eine Kette von Assoziationen miteinander verbunden sind, vergleichen. Dies funktioniert auch mit mengenwertigen Beziehungen. Auf diese Weise bekommen Pfadkorrespondenzen eine symmetrische Darstellung.

In Kapitel 7 wird ein Verfahren vorgestellt, das Pfadkorrespondenzen verarbeitet, um redundante Pfade im Ergebnisschema zu vermeiden.

Inter-Schema-Beziehungen, also Vererbungsbeziehungen und Assoziationen zwischen Elementen unterschiedlicher Schemata können ebenfalls als Korrespondenzen behandelt werden, wenn sie in einem zusätzlichen, neu erstellten Schema modelliert werden, welches anschließend mitvereinigt wird.

[1] Die vollständige Definition einer korrekten Pfadkorrespondenz folgt in Kapitel 7

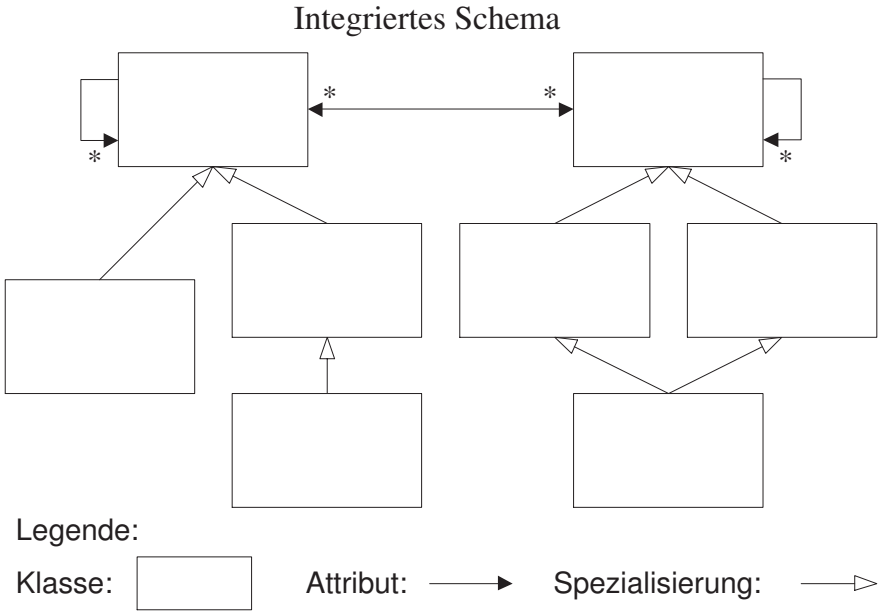

Abbildung 3: Vollständigkeit ohne Minimalität

4.4 Qualitätskriterien für Integrationsverfahren

Folgende Eigenschaften zeichnen eine gute Methode aus:

- Vollständigkeit: Die bisherigen Konzepte müssen alle auch im neuen Schema enthalten sein. Bestehende Daten könnten sonst nicht verlustfrei migriert werden.
- Minimalität: Zusätzliche Konzepte sollen im integrierten Schema nicht vorkommen. Zum Beispiel soll eine "Lösung" wie in Abbildung 3 mit integrierten Vererbungshierachien, aber hierin ganz "nach oben gezogenen" Assoziationen, verhindert werden.
- Korrektheit: Abfragen und Programme sollten so transformierbar sein, daß das gleiche Ergebnis wie vorher geliefert wird. Diese Transformation darf kompliziert sein. Sie muß nicht einmal direkt angegeben werden, sondern es genügt ein Existenzbeweis. Korrektheit ist also nur relativ zu Abfragesprachen definiert [Bri98]. Die direkte Angabe der Transformationen bietet die Chance, Programme und Abfragen zu erhalten.
- Verständlichkeit: Das integrierte Schema soll nicht unnötig komplex sein und die modellierten Konzepte sollten erkennbar bleiben.

- Redundanzfreiheit: Information sollte nicht mehrfach in Attributen, Assoziationen, oder als Metadatum gespeichert sein. Redundante Pfade sind ebenfalls zu vermeiden. Hiermit soll es erschwert werden, die Integrität der Datenbank durch fehlerhafte Programme oder Abfragen zu gefährden.

- Für die einfache Behandlung von Inter-Schema-Beziehungen ist es wünschenswert, daß das Verfahren mehr als zwei Schemata vereinigen kann.

- Eine Undo-Funktionalität bei fehlerhaften Integrationsannahmen wird erleichtert, wenn die Teilschritte des Integrationsverfahrens kommutativ und assoziativ sind.

4.5 Ein einfaches Verfahren

Der Integrationsprozeß läßt sich in die Schritte der Schemaanpassung, der Vereinigung und der Optimierung im Sinn einer nachträglichen Verbesserung unterteilen.

Einfache Verfahren beseitigen als Schemaanpassung Homonyme (gleicher Name für unterschiedliche Dinge) und Synonyme (unterschiedlicher Name für gleiche Dinge) durch Umbenennung. Im Fall abweichender Maßeinheiten korrespondierender Attribute müssen die Werte bei einer Datenmigration später umgerechnet werden. Anschließend werden korrespondierende Klassen und jeweils ihre korrespondierenden Attribute vereinigt. Falls Attribute nur in einem Schema vorkommen, können für sie Subklassen erstellt werden. Integritätsbedingungen werden oder-verknüpft. Als Optimierung werden Redundanzen von Attributen, deren Klassen voneinander abgeleitet sind, durch Entfernung einzelner Attribute beseitigt und die verbliebenen Redundanzen durch Integritätsbedingungen abgesichert.

4.6 Kapitelzusammenfassung

Schemaintegration ist ein komplexes und formal kaum faßbares Gebiet. Daher werden meist nur Teilaspekte behandelt und vereinfachende Annahmen getroffen. Inter-Schema-Korrespondenzen sind die Eingabe der Verfahren. Eine Liste von Qualitätskriterien dient der Beurteilung der Algorithmen. Am Ende wurde eine simple Methodik dargestellt. In den folgenden vier Kapiteln werden ausgewählte Methoden zur Integration von Produktdaten vorgestellt. Dadurch werden die Schwächen der simplen Vorgehensweise aufgedeckt.

5 Komplexe hierarchische Datentypen

5.1 Kapitelüberblick

Dieses Kapitel geht genauer auf den Spezialfall der hierarchisch aufgebauten, konstruierten Datentypen ein. Es wird eine allgemeine Klasse von Transformationsregeln (rewrite rules) vorgestellt, die komplexe Objekte eines Typs auf solche eines anderen Typs abbilden. Diese Regelmenge wird dann auf eine Teilmenge eingeschränkt (simple rewrite rules), die bzgl. Komposition abgeschlossen ist, so daß nach einer weiteren Abbildung die Gesamttransformation wiederum als simple rewrite rule darstellbar ist. Einige spezielle simple rewrite rules genügen, um hierarchische Datentypen auf eine Normalform zu transformieren. Neben der Normalform wird eine "Informationskapazität" definiert. Genau dann, wenn zwei Typen isomorphe Normalformen haben, haben sie die gleiche Informationskapazität. Es existiert dann eine bijektive Abbildung der Mengen unterschiedlicher Objekte zweier Typen.

Die Idee ist also, eine Möglichkeit zu schaffen, Typen gleicher Informationskapazität miteinander zu identifizieren, da deren Objekte Eins zu Eins aufeinander abbildbar sind. Die Namensgebung für die Teilobjekte, aus denen ein komplexes Objekt zusammengesetzt ist, spielt für den Vergleich keine Rolle.

5.2 Definition der Datentypen

In [Abi88] beschreiben Abiteboul und Hull Transformationen für hierarchische Datentypen. Neuerdings steigt das Interesse an hierarchischen Datentypen wegen des sich rasch ausbreitenden XML-Standards [XML98], zu dem dieses Kapitel auch gut passen würde.

Ausgegangen wird von abzählbar unendlichen Basisdomänen[2] wie INTEGER oder STRING und von Konstanten. Daraus können mit Hilfe von Tupel-, Mengen- und Variantenkonstruktor neue Typen gebildet werden. Mit EXPRESS können diese Typen problemlos modelliert werden, wenn das Entity als Tupelkonstruktor und die ENUMERATION mit einem Element für Konstanten verwendet wird.

[2]Basisdomäne hat die Bedeutung eines Wertebereichs. Das englische Wort "domain" bedeutet Definitionsbereich, wird aber manchmal (wie in diesem Kapitel) auch für die Bildmenge verwendet (da sich Basiswertebereich schlecht liest).

Aus Platzgründen wird in diesem Kapitel die in der folgenden Tabelle vorgestellte, an [Abi88] angelehnte, Notation verwendet:

Element nach [Abi88]	Abkürzung nach [Abi88]	EXPRESS-Äquivalent
Konstanten	$CONST_{ja}$	ENUMERATION(ja)
Basisdomänen	$dom_1 ... dom_n$	INTEGER, ...
Tupelkonstruktor	[...]	ENTITY
Mengenkonstruktor	{...}	SET
Variantenkonstruktor	<...>	SELECT

Typen sind rekursiv definiert. Die Teilkonstrukte sollten zum Zweck der besseren Unterscheidung einen Bezeichner bekommen, dies entspricht einem benannten Typ in EXPRESS. Die Definition ist wie folgt:

- Wenn A ein Bezeichner und dom_i eine Basisdomäne ist, so ist A:dom_i ein Typ.
- Ein Typ kann nur aus einer Konstanten bestehen.
- Wenn K ein Bezeichner und T_1 bis T_n Typen mit unterschiedlichen Bezeichnern sind, dann sind $K : \{T_1\}$, $K : [T_1, ..., T_n]$ und $K :< T_1, ..., T_n >$ Typen.

Wenn $P : t$ ein Typ ist, wird t die *Struktur* des Typs genannt. Falls t die Form $\{Q : t_1\}$ hat, sagt man, daß t die Mengenstruktur hat.

Objekte zu Typen werden analog rekursiv definiert. Ihr Aufbau muß der Typdefinition entsprechen: Bei Varianten muß eine Auswahl getroffen werden, bei Mengen müssen die Elemente (wiederum Objekte) festgelegt werden und für Basisdomänen muß ein Wert eingesetzt werden.

In Abbildung 4 wird ein Beispieltyp definiert, der mehrfach in diesem Kapitel verwendet wird. Dieser Typ stellt einen Baum mit der Wurzel Auto dar. Die Blätter sind Basistypen oder Konstanten.

Eine Typdefinition ist noch relativ allgemein, in Abb. 4 kann ein Auto-Objekt eine beliebige Zahl von Zylindern haben. Mit Hilfe von *Termen* kann die Zahl der Zylinder festgelegt werden. Terme passen zu vielen Typen und schränken mögliche Objekte eines jeden dieser Typen ein. Sie werden gebildet, indem im Baum des Typs einige Knoten samt der darunter liegenden Teilbäume durch *Variablen* ersetzt werden. Übrig bleibt ein kleinerer Baum, zu dem umgekehrt viele Typen passen, je nachdem welche Typen durch die Variablen repräsentiert werden. Diese Typen der Variablen können durch einen Vergleich von Typ und Term ermittelt werden. Die Einschränkung der Objektmenge durch

```
Auto:
[   Karosserie:
    [   Farbe:
        <   CONSTrot,
            CONSTblau
        >,
        Seriennummer: INTEGER
    ],
    Scheinwerfer:
    [   Glühbirnenart: STRING
    ],
    Motor:
    <   Dieselmotor:
        {   ZylinderDiesel:
            [   Hubraum: INTEGER
            ]
        },
        Benzinmotor:
        {   ZylinderBenzin:
            [   Hubraum: INTEGER
                Zündkerzenart: STRING
            ]
        }
    >
]
```

Abbildung 4: Ein konstruierter Typ

den Term besteht darin, daß alle Varianten in dem kleineren Baum **ausgewählt** und die vorkommenden Mengen mit **festgelegten** Elementen gefüllt werden müssen. Terme sind rekursiv wie folgt definiert:

- Jedes Objekt vom Typ T ist ein Term von T.
- Wenn $T = P : t$ ein Typ mit der Struktur t und $VARx$ eine Variable ist, dann ist $P : VARx$ ein Term von T.
- Wenn $T = P : [T_1, ..., T_n]$ ein Typ und X_i für alle $i = 1..n$ ein Term von T_i ist, dann ist $T = P : [X_1, ..., X_n]$ ein Term von T.
- Wenn $T = P : R$ ein Typ und X_i für alle $i = 1..n$ ein Term von R ist, dann ist $T = P : \{X_1, ..., X_n\}$ ein Term von T.

- Wenn $T = P :< T_1, ..., T_n >$ ein Typ und X_i für ein bestimmtes i ein Term von T_i ist, dann ist $T = P :< X_i >$ ein Term von T.

Das folgende Beispiel zeigt einen Term, der alle roten Autos umfaßt:

```
Auto:
[   Karosserie:
    [   Farbe:
        <   CONSTrot
        >,
        Seriennummer: VARSeriennummer
    ],
    Scheinwerfer: VARScheinwerfer,
    Motor: VARMotor
]
```

Durch einen Vergleich mit dem ursprünglichen Typ kann abgeleitet werden, daß VARSeriennummer den Typ INTEGER hat. Entsprechend kann der konstruierte Typ der Variable VARMotor ermittelt werden.

Das nächste Beispiel stellt einen Term dar, der alle Autos mit Dieselmotor und 4 Zylindern umfaßt:

```
Auto:
[   Karosserie: VARKarosserie,
    Scheinwerfer: VARScheinwerfer,
    Motor:
    <   Dieselmotor:
        {   ZylinderDiesel: VARZylinder1,
            ZylinderDiesel: VARZylinder2,
            ZylinderDiesel: VARZylinder3,
            ZylinderDiesel: VARZylinder4,
        },
    >
]
```

Wie die Beispiele zeigen, müssen Terme eines Typs T nicht bis zu den Blättern von T reichen. Allerdings müssen sämtliche Entscheidungen bezüglich Varianten und Mengen in demjenigen Teilbaum von T, der durch den Term festgelegt ist, getroffen werden. Demnach ist es im Beispiel mit Termen nicht möglich, die Anzahl der Zylinder festzulegen, aber den Motortyp (Diesel oder Benzin) offenzulassen.

5.3 Abbildung der Objekte

Der Term im letzten Beispiel kann dazu dienen, Informationen über den dritten Zylinder von Diesel-Autos herauszufinden. Die Variable VARZylinder3 kann dann für sich allein zum Aufbau von Objekten eines vollkommen anderen Typs beitragen. Dazu kommt eine *rewrite rule* zum Einsatz. Rewrite rules und *rewrite expressions* sind rekursiv wie folgt definiert:

Gegeben sei ein Term X des Typs S.

- Wenn c ein Wert aus einer Basisdomäne dom_i und P ein Bezeichner ist, dann ist $X \rightarrow P : c$ eine rewrite rule von S nach $P : dom_i$.
- $X \rightarrow CONST_x$ ist eine rewrite rule von S nach $P : CONST_x$.
- Wenn die in X vorkommende Variable $VARy$ den, durch Vergleich von X und S ermittelten, Typ t hat und P ein Bezeichner ist, dann ist $X \rightarrow P : VARy$ eine rewrite rule von S nach $P : t$.
- Einbau einer rewrite expression: Wenn rew eine rewrite expression von $Q : t$ nach R ist, t also die Mengenstruktur hat und die in X vorkommende Variable $VARy$ den gleichen, durch Vergleich von X und S ermittelten, Typ t hat, dann ist $X \rightarrow rew(VARy)$ eine rewrite rule von S nach R.
- Wenn $X \rightarrow Y_1, ..., X \rightarrow Y_n$ jeweils rewrite rules von S nach $T_1, ..., T_n$ respektive sind, dann ist $X \rightarrow P : [Y_1, ..., Y_n]$ eine rewrite rule von S nach $P : [T_1, ..., T_n]$.
- Wenn $T_1, ..., T_n$ Typen und $X \rightarrow Y_i$ eine rewrite rule von S nach T_i für ein bestimmtes $i \in \{1..n\}$ ist, dann ist $X \rightarrow P :< Y_i >$ eine rewrite rule von S nach $P :< T_1, ..., T_n >$.
- Wenn $X \rightarrow Y_1, ..., X \rightarrow Y_n$ rewrite rules von S nach R sind, dann ist $X \rightarrow P : \{Y_1, ..., Y_n\}$ eine rewrite rule von S nach $P : \{R\}$.
- Wenn $X \rightarrow Y_1, ..., X \rightarrow Y_n$ rewrite rules von S nach T sind und T die Mengenstruktur hat, dann ist $X \rightarrow Y_1 \cup, ..., \cup Y_n$ eine rewrite rule von S nach T.
- Erzeugung einer rewrite expression: Wenn m eine nichtleere Menge von rewrite rules von X nach Y ist, dann ist $Q : rew(m)$ eine rewrite expression von $S = P : \{X\}$ nach $T = Q : \{Y\}$

Beispiel: Die Zylinder aus den Dieselmotoren des letzten Beispiels sollen auf den Typ MotorD4, der Objekte mit genau 4 Dieselzylindern beschreibt, abgebildet werden. Zuerst wird MotorD4 definiert.

```
MotorD4:
[   ZylinderDiesel:
    [   Hubraum: INTEGER
    ],
    ZylinderDiesel:
    [   Hubraum: INTEGER
    ],
    ZylinderDiesel:
    [   Hubraum: INTEGER
    ],
    ZylinderDiesel:
    [   Hubraum: INTEGER
    ]
]
```

Anschließend wird eine rewrite rule vom Typ Auto, der wie in Abbildung 4 auf Seite 27 definiert ist, zum Typ MotorD4 erstellt. Die Kontrolle ergibt, daß die Zylindervariablen typrichtig verwendet werden.

```
Auto:
[   VARKarosserie,
    VARScheinwerfer,
    Motor:
    <   Dieselmotor:
        {   ZylinderDiesel: VARZylinder1,
            ZylinderDiesel: VARZylinder2,
            ZylinderDiesel: VARZylinder3,
            ZylinderDiesel: VARZylinder4
        },
    >
]
-->
MotorD4:
[   ZylinderDiesel: VARZylinder1,
    ZylinderDiesel: VARZylinder2,
    ZylinderDiesel: VARZylinder3,
    ZylinderDiesel: VARZylinder4
]
```

Diese rewrite rule paßt nur zu Dieselautos und liefert bei Eingabe eines Benzinautos die leere Menge. Eine weitere, ähnliche rewrite rule kann dazu dienen, die Zylinder aus den Benzin-Autos herauszusuchen und in entsprechende Vierzylinder Benzinmotorenobjekte einzusetzen. Die beiden unterschiedlichen Ergebnistypen von Vierzylindermotoren können auch als Varianten vereint werden. Mengen von ein oder mehreren rewrite rules, deren Ausgangs- und Zieltypen jeweils Varianten von allgemeineren Typen darstellen können, werden rewrite expression genannt. Solche rewrite expressions bilden Objektmengen auf Objektmengen ab. Für jedes Eingabeobjekt wird die passende rewrite rule gesucht. Falls eine solche vorhanden ist, wird die Abbildung durchgeführt und das Ergebnisobjekt in die Ausgabemenge eingefügt. In obigem Beispiel können jetzt die Zylinder aus den Motoren aller Vierzylinder-Autos (nicht nur die der Diesel-Autos) extrahiert werden und auf die entsprechende Variante eines allgemeinen, hier nicht dargestellten, Vierzylindermotor abgebildet werden.

Die Definition legt auch fest, wie rewrite expressions auf der rechten Seite einer rewrite rule geschachtelt verwendet werden dürfen. Als Beispiel für eine solche Schachtelung werden die Zündkerzenarten aus den Benzinzylindern extrahiert. Zuerst wird eine rewrite rule r1 vom Typ Zylinder zum Typ STRING definiert, die auf einzelne Zylinder wirkt.

Wiederholung der Definition des Typs ZylinderBenzin:

```
ZylinderBenzin:
[   Hubraum: INTEGER
    Zündkerzenart: STRING
]
```

Definition der rewrite rule r1:

```
ZylinderBenzin:
[   Hubraum: INTEGER,
    Zündkerzenart: VARZündkerzenart
]
-->
Zündkerzenart: VARZündkerzenart
```

Der Vergleich des Typs ZylinderBenzin mit dem Term ZylinderBenzin auf der linken Seite von r1 ergibt, daß die Variable VARZündkerzenart vom Typ STRING ist. Der Typ der rechten Seite von r1 ist damit bekannt (Zündkerzenart:STRING). Die rewrite rule r1 bildet also tatsächlich Objekte vom Typ ZylinderBenzin zum Typ Zündkerzenart ab.

Eine rewrite expresssion ex({r1}), bestehend aus r1, bildet demnach ZylinderBenzin Mengen in Zündkerzenart Mengen ab.

Als nächstes wird der Zieltyp der Gesamtoperation definiert:

```
häufige Ersatzteile:
[   Glühbirne: STRING,
    Motorersatzteile:
    {   Zündkerzenart: STRING
    }
]
```

Die rewrite expresssion ex kann jetzt in der folgenden rewrite rule r2 auf der rechten Seite geschachtelt verwendet werden, wobei der Typ Auto wie in Abbildung 4 auf Seite 27 definiert ist:

```
Auto:
[   Scheinwerfer:
    [ Glühbirne: VARGlühbirne
    ],
    Motor:
    <   Benzinmotor: VARBenzinmotor
    >
]
-->
häufige Ersatzteile:
[   Glühbirne: VARGlühbirne,
    Motorersatzteile:
        ex(VARBenzinmotor)
]
```

5.4 Abgeschlossenheit bezüglich Komposition

Das letzte Beispiel zeigte, wie sich rewrite rules und -expressions verketten lassen. Das Ergebnis war eine einzige rewrite rule. Es stellt sich die Frage, ob die rewrite rules bezüglich der Komposition abgeschlossen sind. Dies ist nicht der Fall, wie folgende Überlegung veranschaulicht: Im letzten Beispiel wurden im ersten Teilschritt alle Zündkerzen aus Motoren mit einer beliebigen Zylinderzahl extrahiert. Anschließend wurde diese Menge zum Aufbau der häufigen Ersatzteile verwendet. Würde man diese Schritte in Form zweier separater rewrite rules formulieren, dann gäbe es die Möglichkeit, als Vorbedingung auf der linken Seite der zweiten rewrite rule die genaue Anzahl von Zündkerzen vorzugeben. Motoren mit einer nicht passenden Zylinderzahl würden so nichts zum Ergebnis beitragen. Diese Auswahl ist bei kombinierter Formulierung nicht möglich, da die rewrite expression ex eine Menge von

Objekten liefert, deren Anzahl mit den hier definierten Formalismen nicht eingeschränkt werden kann.

Um Abgeschlossenheit bezüglich Komposition zu erreichen, kann man eine Teilmenge der rewrite rules, die *simple rewrite rules*, verwenden. Sie sind wie folgt definiert:

- Auf der linken Seite jeder (äußeren oder geschachtelten) rewrite rule sind Mengenkonstruktoren nicht zugelassen, mit der einzigen Ausnahme von $P : \{CONST_x\}$ (leere Menge oder Menge mit einer einzigen Konstante als Inhalt).
- Feste Objekte dürfen auf den linken Seiten nicht vorkommen.
- Auf jeder linken Seite müssen sämtliche Variablen unterschiedlich bezeichnet sein.
- Wenn eine, möglicherweise geschachtelte, rewrite rule der Form $U \rightarrow CONST_f$ vorkommt, dann muß U eine Konstante der Form $CONST_g$ sein.

Um Mißverständnisse zu vermeiden: Die rewrite rule r2 mit der enthaltenen rewrite expresssion ex aus dem letzten Abschnitt ist simple, obwohl hier Mengen im Spiel sind. Lediglich explizit angegebene Mengen mit einer festen Zahl von Elementen (siehe die Definition von Termen am Anfang des Kapitels) sind verboten.

Der Beweis der Abgeschlossenheit teilt sich im wesentlichen in zwei Hauptschritte. Dafür wird eine andere Teilmenge der rewrite rules, die Menge der *decomposed rewrite rules*, benötigt. Die Definition ist wie folgt:

Ein Term X wird decomposed genannt, wenn die durch einen Vergleich mit dem X zugrundeliegenden Typ ermittelten Typen der in X vorkommenden Variablen entweder

- die Mengenstruktur außer $\{CONST_x\}$ haben oder
- eine Basisdomäne oder Konstante sind.

Eine rewrite rule wird decomposed genannt, wenn die Terme auf den linken Seiten alle decomposed sind.

Anders ausgedrückt, dürfen die Blätter der Terme keine Varianten- oder Tupelkonstruktoren darstellen. Die durch den Term getroffenen Entscheidungen reichen sonst nicht aus.

Im ersten Beweisschritt zeigen Abiteboul und Hull, daß eine simple rewrite rule durch ein oder mehrere decomposed simple rewrite rules ersetzt werden kann. Dies entspricht einer Zerlegung in genauer definierte Varianten.

Im zweiten Beweisschritt wird gezeigt, daß die Komposition zweier decomposed simple rewrite rules durch eine einzige simple rewrite rule ersetzt werden kann. Für den kompletten Beweis dieser beiden Hauptschritte wird wegen der vielen Details auf die Originalarbeit verwiesen.

Insgesamt können die Objekte des Ausgangsschema mit einer einzigen rewrite expression auf Objekte des Zielschema abgebildet werden, auch wenn dieses in einem inkrementellen Prozeß mit mehreren Zwischen-Schemata entstanden sein sollte.

5.5 Normalform und Informationskapazität

Ein Typ S ist in *Normalform*, wenn

- maximal ein Variantenkonstruktor vorkommt, der die Wurzel bilden und mehr als einen Kindknoten haben muß,
- $CONST_f$ kein Kindknoten eines Mengen- oder Tupelkonstruktors ist,
- Kein Kindknoten eines Tupelkonstruktors ein Tupelkonstruktor ist
- und jeder Tupelkonstruktor mehr als einen Kindknoten hat.

In der Originalarbeit wird gezeigt, daß die *Informationskapazität* zweier Typen genau dann gleich ist, wenn deren Normalformen isomorph sind, also bis auf Umbenennungen übereinstimmen. Diese Gleichheit der Informationskapazität wird wie folgt definiert:

Es muß eine feste Schranke k geben, so daß, wenn aus jeder der Basisdomänen eine beliebige Anzahl $> k$ von Werten entnommen wird, die Anzahl der hieraus unterschiedlich zusammensetzbaren Objekte für beide miteinander verglichenen Typen jeweils gleich ist.

Bei Verletzung dieser Äquivalenz ist keine bijektive Abbildung der Objekte möglich.

Typen können mit Hilfe von simple rewrite rules auf Normalform gebracht werden, so daß die damit verbundene Abbildung der Objekte bijektiv ist. Dazu müssen die folgenden Ersetzungen in beliebiger Reihenfolge, so oft wie möglich vorgenommen werden:

1. ersetze
   $P : [T_1, ..., T_{i-1}, Q : [S_1, ..., S_m], T_{i+1}, ..., T_n]$
   durch
   $P : [T_1, ..., T_{i-1}, S_1, ..., S_m, T_{i+1}, ..., T_n]$

2. ersetze
   $P :< T_1, ..., T_{i-1}, Q :< S_1, ..., S_m >, T_{i+1}, ..., T_n >$
   durch
   $P :< T_1, ..., T_{i-1}, S_1, ..., S_m, T_{i+1}, ..., T_n >$

3. ersetze
   $P : [T_1, ..., T_{i-1}, Q :< S_1, ..., S_m >, T_{i+1}, ..., T_n]$
   durch
   $P :< Q_1 : [T_1, ..., T_{i-1}, S_1, T_{i+1}, ..., T_n], ...$
   $Q_m : [T_1, ..., T_{i-1}, S_m, T_{i+1}, ..., T_n] >$

4. ersetze
   $P : \{< T_1, ..., T_n >\}$
   durch
   $P : [Q_1 : \{T_1\}, ..., Q_n : \{T_n\}]$

5. ersetze
   $P : [T_1, ..., T_{i-1}, CONST_f, T_{i+1}, ..., T_n]$
   durch
   $P : [T_1, ..., T_{i-1}, T_{i+1}, ..., T_n]$

6. ersetze
   $P : \{CONST_f\}$
   durch
   $P :< CONST_g, CONST_h >$

Alle diese Transformationen lassen sich laut Abiteboul und Hull durch ein oder mehrere simple rewrite rules realisieren.[3] Die Bijektivität ist anhand der Definition direkt ersichtlich. Allerdings vergrößert die dritte Transformation

[3]Anm. des Autors: Um die rewrite rules tatsächlich anzugeben, müssen die verwendeten Typen als Ganzes bekannt sein.

die Länge des Typausdrucks. So steht am Ende des Verfahrens möglicherweise anstelle von vielen kleinen Variantenkonstruktoren tiefer im Baum ein einzelner, exponentiell vergrößerter, Variantenkonstruktor ganz oben an der Wurzel.

5.6 Bewertung

Die bijektive Abbildung durch eine einzige rewrite rule gewährleistet durch explizite Angabe der Transformation die Kriterien der Korrektheit, Vollständigkeit und Minimalität.

Es sind viele äquivalente Darstellungen derselben Information möglich. Durch die Transformation auf Normalform werden unterschiedlich strukturierte Datentypen besser vergleichbar, so daß bei einem nachfolgenden Vereinigungsschritt die Anzahl der redundanten Definitionen im Ergebnis vermindert wird.

Nachteil ist, daß die Länge der Typdefinition in der Normalform exponentiell größer sein kann, worunter auch die Verständlichkeit des transformierten Schemas leidet. Die Ursache liegt im Variantenkonstruktor (SELECT), der in den meisten oben angegebenen Transformationen vorkommt. Ein Verzicht auf diesen Konstruktor würde die Schemaintegration möglicherweise erleichtern, zumal die Varianten auch, wie bereits erwähnt, Überlappungen mit abgeleiteten Klassen und mit optionalen Attributen aufweisen können. Sicher ist dies jedoch nicht, da abgeleitete Klassen und optionale Attribute nicht exakt die gleiche Semantik wie Varianten haben. Vielleicht wäre es besser, die Baumhöhe der Typen zu beschränken. Gewiß ist weitere Forschung auf dem Gebiet dieses Kapitels nötig.

Da man es in der Praxis nicht nur mit hierarchischen Datentypen zu tun hat, ist der Einsatzbereich der vorgestellten Methode begrenzt.

6 Weak Schema Merge

6.1 Kapitelüberblick

Kosky [Kos94a, Kos94b, Kos94c] definiert gültige Schemata. "Gültig" bezieht sich auf Aspekte der syntaktischen Korrektheit, die in diesem Kapitel vorgestellt werden. Durch Verzicht auf einen dieser Aspekte ergeben sich schwache Schemata (engl. weak Schemas).

Es wird erklärt, wie Kosky eine *Informationsordnung* $\sqsubseteq$ und einen Vereinigungsoperator $\sqcup$ für schwache Schemata definiert und darauf aufbauend das im folgenden beschriebene Problem der impliziten Klassen löst. Die Vereinigung aller schwachen Schemata mit Hilfe des Operators $\sqcup$ ist minimal bezüglich $\sqsubseteq$ und hängt nicht von der Reihenfolge ab. Das Ergebnisschema läßt sich am Ende aus dem vereinigten schwachen Schema gewinnen.

Neben der direkt einsichtigen Form können alle Schemadefinitionen und Operatoren auf eine abgewandelte, aber äquivalente Art dargestellt werden, die den Formalismus des Verfahrens vereinfacht und vervollständigtes Schema genannt wird. Der gesamte Lösungsweg sieht damit wie folgt aus: Die gültigen Teilschemata werden in die gültige vervollständigte Form transformiert und anschließend vereinigt. Das Ergebnis ist ein schwaches, nicht unbedingt gültiges, vervollständigtes Schema. Daraus wird auf algorithmische Weise ein gültiges vervollständigtes Schema zurückgewonnen, welches zum Schluß in die gewöhnliche, nicht-vervollständigte Form gebracht wird.

6.2 Problematik

Mehrfachvererbung

An sich vereinfacht die Möglichkeit der Mehrfachvererbung die Schemaintegration, da es vorkommen kann, daß korrespondierende Klassen von unterschiedlichen Oberklassen abstammen. Hierbei erscheint es für die Interoperabilität günstig, mehrfach geerbte Attribute als eines zu betrachten. Auch EXPRESS verfährt zum Teil auf diese Weise (siehe Seite 13).

Unter diesen Voraussetzungen macht Anthony Kosky auf eine Schwierigkeit aufmerksam, die in Abbildung 5 veranschaulicht wird. Aus Schema 1 stammen die Klassen Arbeitstier, Hund und Polizeihund. Schema 2 enthielt die Klassen Arbeitstier, ArbRegistrierung, Hund und HundRegistrierung. Nach der EXPRESS-Spezifikation muß die zweifach geerbte Referenz Registrierung

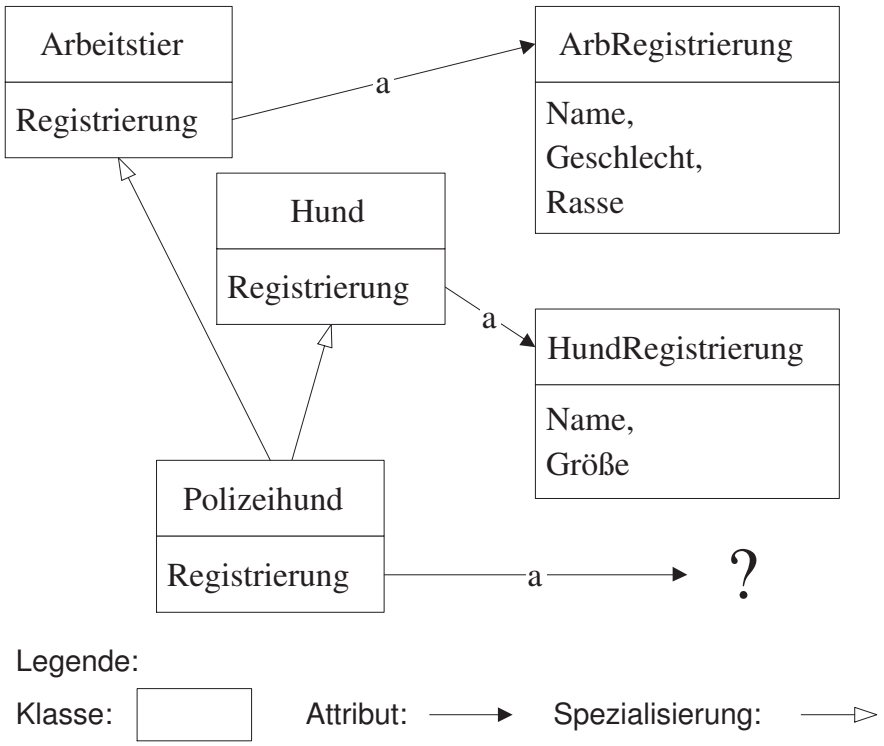

Abbildung 5: Implizite Klassen

der Klasse Polizeihund auf eine Klasse verweisen, die sowohl von ArbRegistrierung, als auch von HundRegistrierung abgeleitet ist. Da eine solche Klasse im vereinigten Schema nicht existiert, muß sie als *implizite Klasse* extra hinzugefügt werden.

Da die impliziten Klassen anfangs noch nicht bekannt sind, müssen Korrespondenzen, an denen implizite Klassen beteiligt sind, während des laufenden Vereinigungsprozeß manuell oder automatisch hinzugefügt werden.

Ein guter Algorithmus sollte vermeiden, daß übermäßig viele implizite Klassen entstehen und die Übersichtlichkeit des Schema verschlechtern.

Unterschiedliche Vereinigungs-Reihenfolgen mehrerer Schemata bedingen eine unterschiedliche Erzeugungs-Reihenfolge der impliziten Klassen. Kosky argumentiert, daß das Ergebnisschema bei einfachen Algorithmen unterschiedlich ausfallen kann.

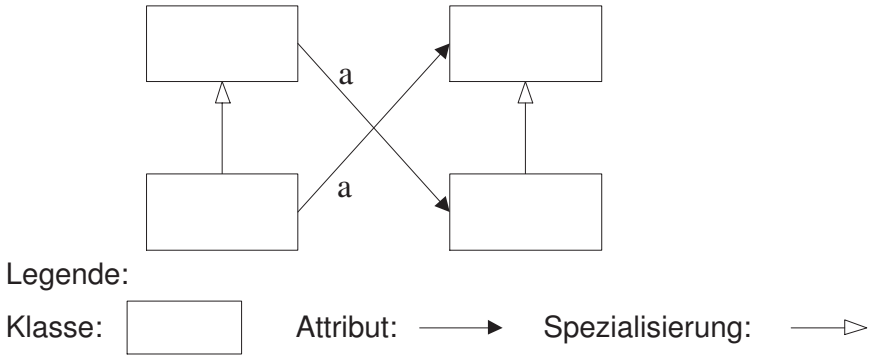

Abbildung 6: Gekreuzte Referenzpfeile

Gekreuzte Referenzen

Ein weiteres Problem ist in Abbildung 6 dargestellt. Diese Situation kann durch Schemavereinigung entstehen, auch wenn jedes Teilschema kreuzungsfrei war. Das überschriebene Attribut ist entgegen dem EXPRESS Standard keine Spezialisierung des geerbten Attributs (Siehe Seite 13).

Nicht-funktionales Schema

Der Begriff "funktionales Schema" wird im nächsten Abschnitt erläutert. Das Problem tritt auf, wenn die Schemavereinigung dazu führt, daß korrespondierende Referenzen aus korrespondierenden Entities unterschiedliche Typen (Zielklassen) haben. Entweder liegt ein echter Typfehler vor, oder es ist eine neue Subklasse gewünscht. Bei der Beschäftigung mit Schemaintegration stößt man meist direkt auf diesen Spezialfall der in Abbildung 5 bereits dargestellten Problematik, die auch den durch Vererbung eingehandelten Typkonflikt einschließt.

6.3 Ausgangslage und Voraussetzungen

Vorab müssen Homonyme und Synonyme für Klassennamen und Assoziationsbezeichner durch Umbenennungen beseitigt sein. Aufgrund des gleichen Namens ergeben sich Inter-Schema-Korrespondenzen zwischen Klassen und zwischen Assoziationen.

6.4 Methode

Jedes Schema $G = (C, L, E, S)$ besteht aus der Menge der Klassen C, der Menge der Assoziationsbezeichner L, der Menge der Assoziationen $E \subseteq C \times L \times C$ und der Menge der Spezialisierungsbeziehungen $S \subseteq C \times C$. Der durch C und E bestimmte Graph ist gerichtet, Assoziationen werden wie in EXPRESS als Objektreferenzen oder Zeiger einer Ausgangsklasse auf eine Zielklasse aufgefaßt. Ein *gültiges Schema* (proper Schema) muß drei Bedingungen erfüllen:

A 1 $\forall p, q, r \in C, a \in L$
$(p, a, q) \in E \wedge (p, a, r) \in E$ impliziert $q = r$

A 2 S muß azyklisch sein.

A 3 $\forall p, q, r, s \in C, a \in L$
$(p, a, q) \in E \wedge (r, a, s) \in E \wedge r \Longrightarrow p$ impliziert $s \Longrightarrow q$

Die erste Bedingung fordert die eindeutige Zuordnung einer Zielklasse (engl. range class, Bedeutung: Bild der Funktion) als Funktion der Startklasse (engl. domain class, Bedeutung: Definitionsbereich) und des Namens der Referenz. Deshalb nennt man ein solches Schema *funktional* (engl. functional). Die zweite Bedingung verbietet Zyklen in der Vererbungshierarchie. Die dritte Bedingung verhindert gekreuzte Referenzen (Abbildung 6). Diese Forderungen passen gut zu EXPRESS (und zu den Spezifikationen der Object Database Management Group [ODMG93]).

Für ein *schwaches Schema* wird die erste Bedingung nur noch für in einer Vererbungsbeziehung stehende Klassen q und r gefordert. In Abbildung 7 darf der Polizeihund vorerst zwei Registrierungen haben, da ArbRegistrierung und HundRegistrierung nicht in einer Vererbungsbeziehung stehen.

Die Informationsordnung $\sqsubseteq$ für schwache Schemata wird wie folgt definiert: Seien $G_1 = (C_1, L_1, E_1, S_1, K_1)$ und $G_2 = (C_2, L_2, E_2, S_2, K_2)$ zwei schwache Schemata.

A 4 $G_1 \sqsubseteq G_2 \iff C_1 \sqsubseteq C_2 \wedge S_1 \sqsubseteq S_2 \wedge$
$\forall (p, a, q) \in E_1 . \exists r \in C_2$ mit $(p, a, r) \in E_2 \wedge r \Longrightarrow_2 q$

Nach dieser Definition muß man gegebenenfalls Objekte beim Transfer vom kleineren in das größere Schema derart ändern, daß sie einen spezielleren

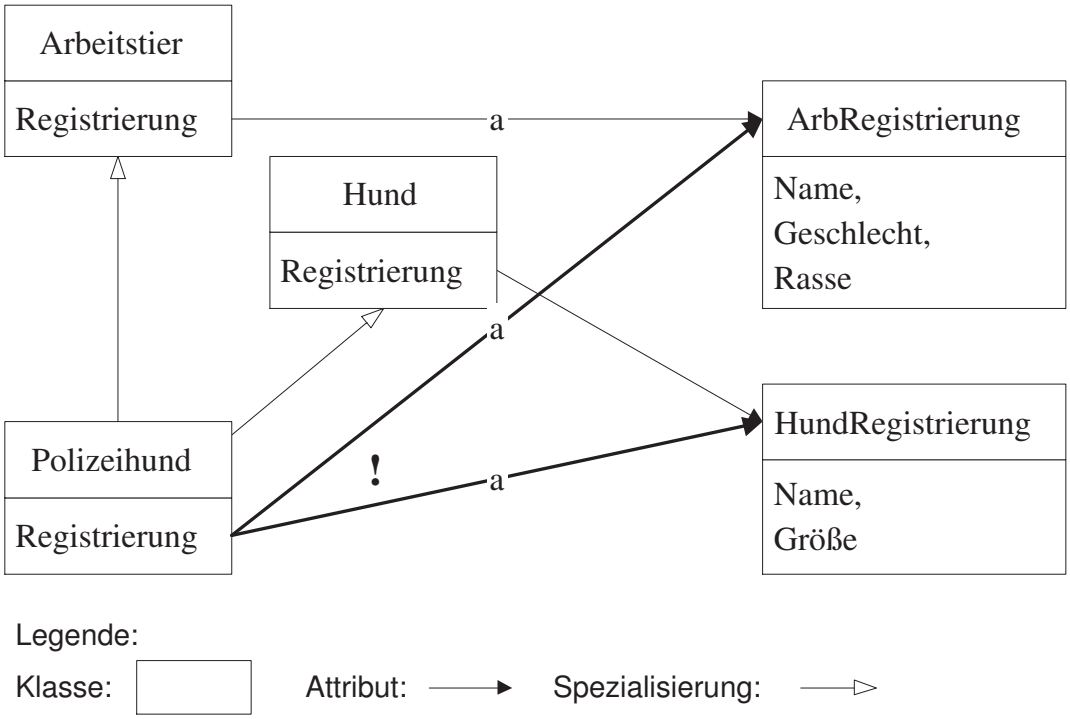

Abbildung 7: Schwaches Schema (Weak Schema)

Typ und neue, mit Nullwerten belegte, Attribute zugewiesen bekommen. Beispielsweise könnte im einen Schema ein Polizist einen Hund haben, während er im anderen Schema einen Polizeihund hat. Die Modellierung ist im zweiten Schema bezüglich des Hundes detaillierter, so daß im vereinigten Schema ein Polizist nur Polizeihunde haben darf. Wenn er privat auch noch einen Hund haben kann, dann muß dieser Referenz ein anderer Name gegeben werden. Als Kardinalität wird im Ergebnisschema die weniger restriktive verwendet.

Umgekehrt werden Objekte des größeren Schemas auf eine Teilmenge ihrer Attribute projiziert und in einen Supertyp umgewandelt, bevor sie in das kleinere Schema einfügbar sind, falls die restriktivere Kardinalität dies erlaubt.

Die obige Definition von $\sqsubseteq$ ist wegen des Vergleichs von E bereits etwas unschön und die weiteren Schritte des Verfahrens würden auf der bisherigen Basis relativ kompliziert werden. Deshalb ist es am einfachsten, noch einmal ganz von vorn zu beginnen, bevor $\sqcup$ definiert wird. **Der Sinn der bisherigen Erläuterungen bleibt erhalten, lediglich die formale Darstellung wird abgewandelt.**

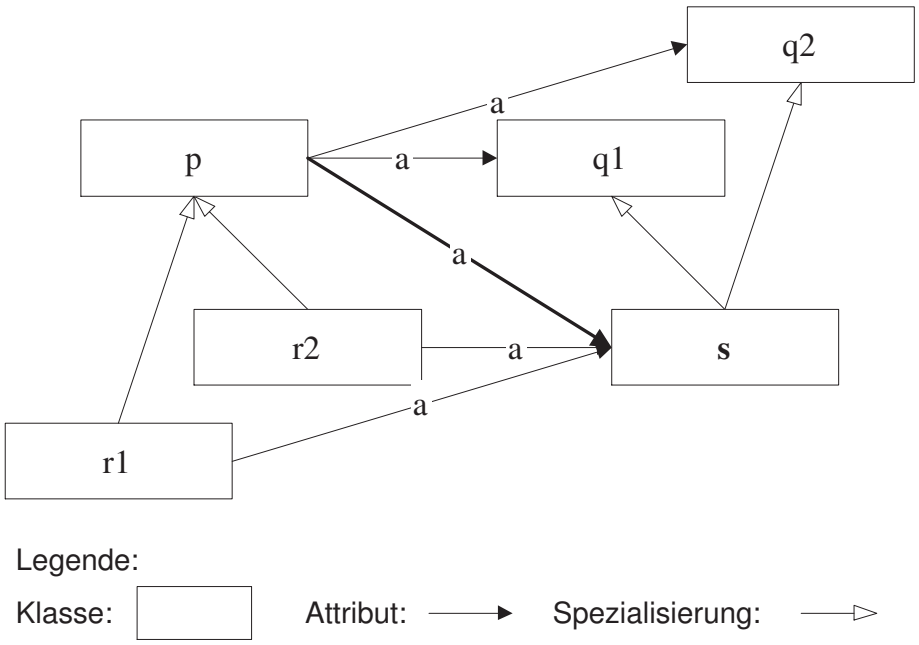

Abbildung 8: Kanonische Klasse

Ein *gültiges vervollständigtes Schema* (engl. wiederum proper Schema, der Begriff "vervollständigtes Schema" stammt zur besseren Unterscheidbarkeit der Definitionen vom Autor) muß drei Bedingungen erfüllen:

B 1 $(p, a, q_1) \in E \wedge (p, a, q_2) \in E$ impliziert
$\exists s \in C.s \Longrightarrow q_1 \wedge s \Longrightarrow q_2 \wedge (p, a, s) \in E$

B 2 $(p, a, s) \in E \wedge r \Longrightarrow p$ impliziert $(r, a, s) \in E$

B 3 $(p, a, s) \in E \wedge s \Longrightarrow q$ impliziert $(p, a, q) \in E$

Abbildung 8 soll die Äquivalenz der Definitionen A1-A3 und B1-B3 veranschaulichen. Um vom ursprünglichen (A1-A3) zum vervollständigten (B1-B3) Schema zu gelangen, wird der fettgedruckte Referenzpfeil vervielfacht, indem seine Startklasse p neue Referenzen zu allen Superklassen seiner Zielklasse s bekommt. (Wenn eine Subklasse r von p nach A3 eine Referenz zu einer Subklasse von s hat, bekommt r nach dem gleichen Prinzip eine direkte Referenz zur Zielklasse s.)

Die Klasse s wird *kanonische Klasse* genannt. Jede Startklasse eines Bündels von a-Referenzen hat wegen B1 eine solche, eindeutig bestimmte, tiefste Klasse in der Vererbungshierarchie. Um vom vervollständigten (B1-B3)

zurück zum ursprünglichen (A1-A3) Schema zu gelangen, werden für jede einzelne Startklasse alle Pfeile entfernt, deren Zielklasse nicht die zugehörige kanonische Klasse ist.

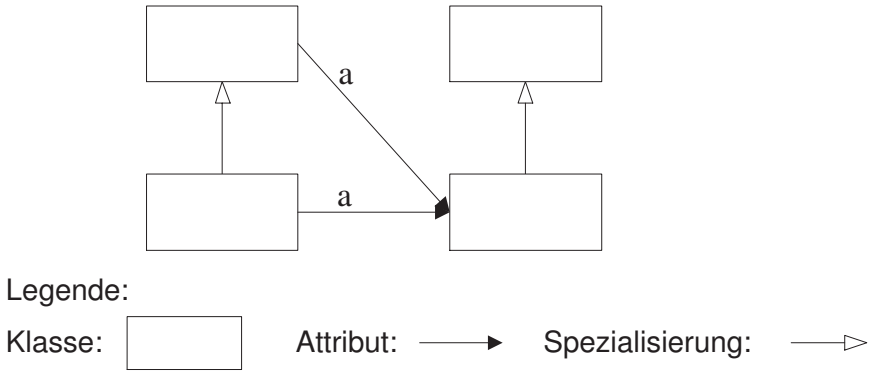

Abbildung 9: Entfernung der gekreuzten Referenzpfeile

Diese Äquivalenz gilt jedoch nur für gültige (korrekte) Schemata. Falls vor der Transformation in die vervollständigte Form gekreuzte Referenzen wie in Abb. 6 bestanden, dann sind diese nach der Rücktransformation verschwunden, siehe Abbildung 9.

Für ein *schwaches vervollständigtes Schema* wird die Forderung B1 fallengelassen. Hier darf die kanonische Klasse fehlen.

In den folgenden zwei Definitionen sind $G_1 = (C_1, L_1, E_1, S_1)$ und $G_2 = (C_2, L_2, E_2, S_2)$ vervollständigte Schemata.

Die Informationsordnung $\sqsubseteq$ wird jetzt einfacher:

B 4 $G_1 \sqsubseteq G_2 \iff (C_1 \sqsubseteq C_2) \wedge (S_1 \sqsubseteq S_2) \wedge (E_1 \sqsubseteq E_2)$

Der zweite Operator $\sqcup$ vereinigt G_1 und G_2. Das vereinigte Schema $G = (C, L, E, S)$ ist wie folgt definiert:

B 5 $C = C_1 \cup C_2$
$L = L_1 \cup L_2$
$S =$ transitiver Abschluß $(S_1 \cup S_2)$
$E = \{(p, a, s) \in (C \times L \times C) |$
$(\exists q, r \in C.(q, a, r) \in E_1 \cup E_2 \wedge p \Longrightarrow q \wedge r \Longrightarrow s)\}$

Die Definition von E [4] sorgt mit Hilfe neuer Referenzpfeile dafür, daß B2 und B3 trotz evtl. neu entstandener Vererbungsbeziehungen erfüllt bleiben. Damit liefert der Vereinigungsoperator ein schwaches vervollständigtes Schema, falls die Vereinigung der Vererbungsbeziehungen zu keinem Zyklus führt.

Die folgenden Behauptungen kommen in der Informatik auch in anderen Zusammenhängen vor und sind leicht zu zeigen. Eine ausführliche Darstellung findet sich z.B. in [Kurs1826] (Theorem 7.5, Any finite lattice is complete): $\sqsubseteq$ ist reflexiv, antisymmetrisch und transitiv und stellt somit eine partielle Ordnung dar. $\sqcup$ ist assoziativ, kommutativ und idempotent. Jedes Schema H, das G_1 und G_2 enthält ($G_1 \sqsubseteq H \wedge G_2 \sqsubseteq H$), muß auch G enhalten ($G \sqsubseteq H$). Auf diese Weise stellt der Vereinigungsoperator die eindeutig festgelegte kleinstmögliche obere Schranke dar. Dies gilt auch für die Vereinigung von n Schemata.

Als nächstes wird gezeigt, wie aus dem vereinigten schwachen vervollständigten Schema wieder ein gültiges vervollständigtes Schema gewonnen werden kann. Dazu werden die fehlenden kanonischen Klassen als gemeinsame Subklasse von solchen Mengen bestehender Klassen, die über die gleiche Abfolge von Referenzen erreichbar sind, neu erzeugt. Daß in einer Menge von Klassen die kanonische Klasse fehlt, ist daran ersichtlich, daß mehrere tiefste Klassen in der Vererbungshierarchie in dieser Menge vorkommen. Die Schritte sind im folgenden formal dargestellt.

Die Funktion R liefert, ausgehend von einer Klasse, die über eine Referenz erreichbaren Klassen. Im folgenden ist $X \subseteq C$, $p \in C$ und $a \in L$.

$R(p, a) = \{q \in C | (p, a, q) \in E\}$

$R(X, a) = \{q \in C | \exists p \in X.(p, a, q) \in E\}$

Die Funktion Min liefert diejenigen Klassen, die in der partiellen Ordnung der Vererbungshierarchie am tiefsten stehen:

$Min_S(X) = \{p \in X | \forall q \in X.q \Longrightarrow p \text{ impliziert } q = p\}$

Mit Hilfe dieser Definitionen wird nun eine Menge von Mengen von Klassen, genannt $Imp \subseteq P(C)$ (P ist die Potenzmenge) konstruiert, die diejenigen Mengen von mindestens zwei minimalen Klassen enthält, die über denselben Pfad erreichbar sind. Die Berechnung der Mengen I^n wird solange fortgesetzt, bis keine neuen Mengen hinzukommen. Da die Potenzmenge von C endlich ist, muß das Verfahren terminieren.

[4] So sähe $\sqcup$ für E ohne vervollständigte Schemata aus (**A5**):
$E = \{(p, a, q) \in (C \times L \times C) | (\exists r \in C.(r, a, q) \in E_1 \cup E_2 \wedge r \Longrightarrow p) \wedge$
$(\forall r, s \in C.p \Longrightarrow r \wedge s \Longrightarrow q \wedge (r, a, s) \in E_1 \cup E_2 \text{ impliziert } q = s)\}$

$I^0 = \{\{p\}|p \in C\}$

$I^{n+1} = \{R(X,a)|X \in I^n, a \in L\}$

$I^\infty = \bigcup_{n=1}^{\infty} I^n$

$Imp = \{Min_S(X)|X \in I^\infty \wedge |Min_S(X)| > 1\}$

Für jede Menge $X \in Imp$ wird eine Klasse $\overline{X}$ neu erzeugt. Nun kann aus dem schwachen Schema (C, L, E, S) das vereinigte gültige Schema $G_p = (C_p, L_p, E_p, S_p)$ gebildet werden:

$C_p = C \cup \{\overline{X}|X \in Imp\}$

$L_p = L$

$\forall X \in Imp.R(\overline{X}, a) := R(X, a)$

$$E_p = \{(x,a,q)|x \in C_p, a \in L, q \in R(x,a)\} \cup \\ \{(x,a,\overline{Y})|x \in C_p, a \in L, Y \in Imp, Y \subseteq R(x,a)\}$$

$$S_p = S \cup \{\overline{X} \Longrightarrow \overline{Y}|X,Y \in Imp, \forall p \in Y.\exists q \in X.q \Longrightarrow p \in S\} \cup \\ \{\overline{X} \Longrightarrow p|X \in Imp, p \in C.\exists q \in X.q \Longrightarrow p \in S\} \cup \\ \{p \Longrightarrow \overline{X}|X \in Imp, p \in C.\forall q \in X.p \Longrightarrow q \in S\}$$

Auf das Schema in Abbildung 7 angewendet, passiert folgendes: Es wird eine neue Klasse X erzeugt, die von ArbRegistrierung und HundRegistrierung abstammt. Polizeihund bekommt eine neue Referenz a zu X. Durch die Umwandlung in ein nichtvervollständigtes Schema würden danach die Referenzen von Polizeihund zu ArbRegistrierung und HundRegistrierung gelöscht werden.

Für den Beweis, daß G_p ein gültiges vervollständigtes Schema ist, wird auf Koskys Originalarbeit [Kos94a] verwiesen.

Abbildung 10 auf Seite 46 soll noch einmal veranschaulichen, was eigentlich gewonnen wurde. Die Vereinigung von Schema 1 mit Schema 2 erzwingt die implizite Klasse x. Wenn anschließend Schema 3 hinzugefügt wird, kommt bei algorithmisch einfach formulierter Vorgehensweise eine weitere Klasse y dazu, die von x abgeleitet ist. Vereinigt man Schema 1 zuerst mit Schema 3, dann führt das zu einem anderen Ergebnis, in welchem x von y abgeleitet wird. Koskys Verfahren erzeugt unabhängig von der Vereinigungsfolge insgesamt nur eine neue Klasse.

Aufgrund der Beschränkung durch die Potenzmenge droht eine exponentielle Laufzeit des Verfahrens. Schemata, bei denen sich das auswirkt, wirken jedoch gekünstelt, d.h. praxisfern.

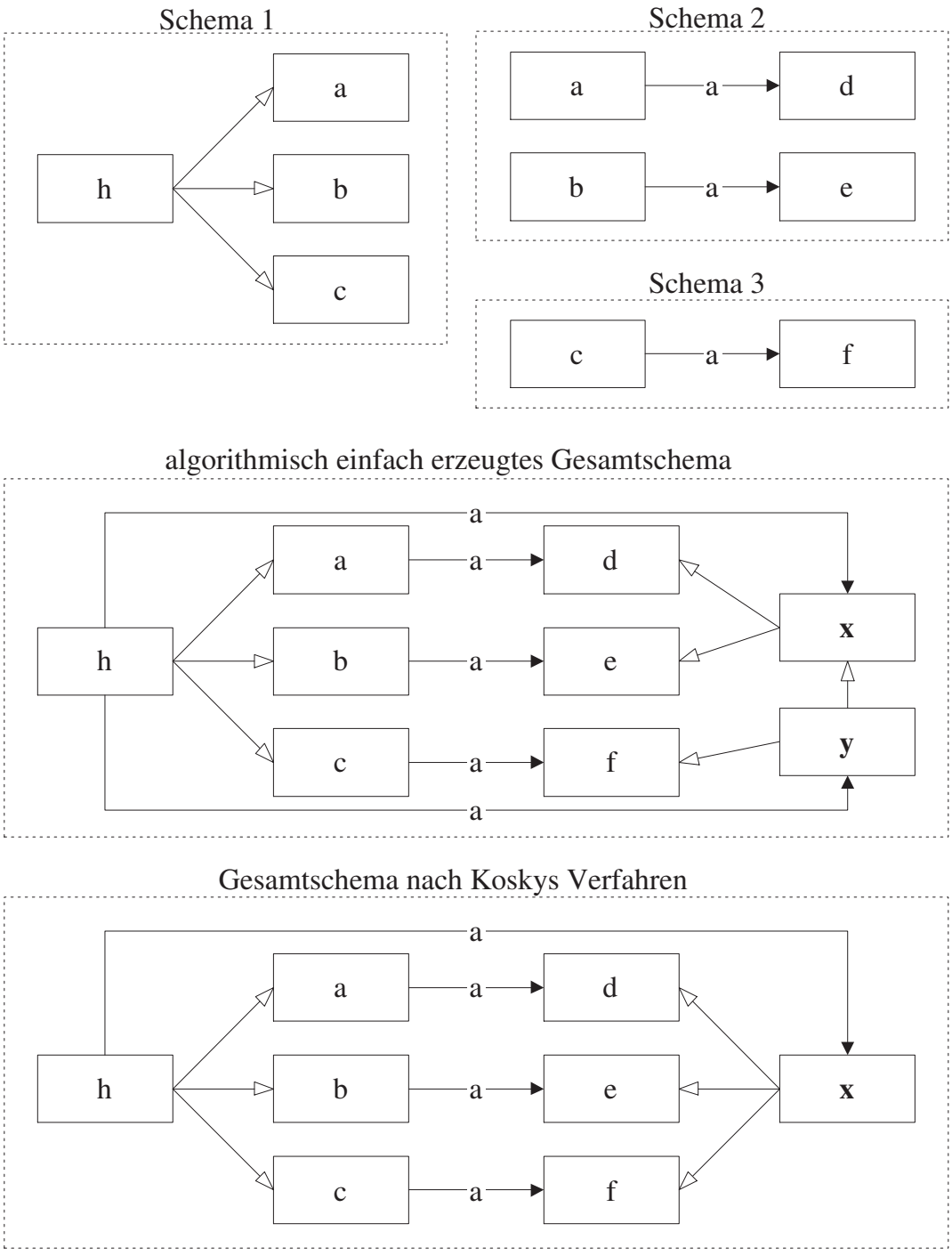

Abbildung 10: Der Vorteil von Koskys Methode

6.5 Bewertung und Zwischenstand

Positiv ist, daß das Verfahren zu einer höheren Automatisierbarkeit des Vereinigungsprozeß führt und daß das Ergebnis der Vereinigung mehrerer Schemata nicht von der Reihenfolge abhängt. Somit können Inter-Schema-Beziehungen durch ein neu erstelltes Zusatz-Schema gut mitbehandelt werden.

Daß nur eine minimale Anzahl zusätzlicher Klassen erzeugt wird, ist ein weiterer Pluspunkt.

Ein Nachteil ist die exponentielle Laufzeit. Es besteht aber die Hoffnung, daß in der Praxis meist "gutartige" Schemata vorkommen.

Trotzdem bleiben zwei wesentliche Probleme:

1. Auf Wertattribute geht das Verfahren nicht ein. Wertattribute nicht zu berücksichtigen ist kein Problem, solange korrespondierende Attribute in korrespondierenden Klassen liegen. Falls dies nicht zutrifft, könnte man auf die (schlechte) Idee kommen, die "falsch positionierten" Attribute in eigene Klassen mit jeweils einem Attribut auszulagern und dann Korrespondenzen zwischen diesen neuen Klassen aufzustellen. Zur Anbindung der neuen Klassen werden neue Beziehungen nötig.

2. Korrespondenzen zwischen einer Klasse Z und einer Assoziation A werden in Koskys Methode ebenfalls nicht berücksichtigt. Dies führt dazu, daß A im Ergebnisschema zwei Klassen direkt verbindet, die auch indirekt über die Klasse Z verbunden sind.

Beide Probleme führen zu einem Schema mit sehr vielen Beziehungen. Dieses Schema ist zwar korrekt, jedoch sind in der Datenbank Referenzen redundant gespeichert. Auch wenn das Schema im Hinblick auf die Anzahl der Klassen minimal ist, ist es dies in Bezug auf die Beziehungen nicht, und die Verständlichkeit nimmt stark ab. Wie solche Redundanzen vermindert werden können, schildern die nächsten beiden Kapitel.

7 SIM

7.1 Kapitelüberblick

Bei SIM [SIM95] werden die Teilschemata durch *Erweiterungen* (engl. augmentations) aneinander angepaßt, bevor sie vereinigt werden. Der Algorithmus verwendet (Navigations-) Pfade, wie sie auf Seite 22 vorgestellt wurden. Die Pfade müssen vom Integrator vorgegeben werden, da dieses Wissen nicht in den Schemata enthalten ist (siehe das Beispiel "Bankberater" auf Seite 22). Der SIM Algorithmus prüft die Pfade und steuert dann die Erweiterungen anhand derselben. Dabei werden Entities, die der zweite Pfad (Partnerpfad) hat, aber der Erste nicht, in den ersten Pfad "eingebaut" und umgekehrt. Die Pfade werden so wie beim Schließen eines Reißverschlusses angeglichen. Hiermit wird referentielle Redundanz beseitigt, da die jeweils letzten Referenzen der Pfade vor der Integration auf das gleiche Objekt zeigten.

Daneben können in vielen Fällen redundante Attribute vermieden werden.

7.2 Problematik

Verletzung der Integrität durch Programme und Abfragen

In einfachen Fällen (keine komplizierte Produktdatenintegration) liegen korrespondierende Attribute in korrespondierenden Klassen. Wenn man in so einem Fall auf die Forderung der Anwendungsmigration verzichtet und redundante Pfade im integrierten Schema zuläßt bzw. mit neuen Integritätsbedingungen absichert, dann wird es möglich, die Integration mit relativ einfachen Verfahren wie der "zusicherungsbasierten Integration" [SPD92] durchzuführen.

Abbildung 11 zeigt einen Ausschnitt eines so entstandenen integrierten Schemas. Die ursprünglichen Schemata hatten die Klassen A1, B1, Z1 bzw. A2, B2, Z2. Die Klassen A1 und A2 sowie Z1 und Z2 korrespondierten und wurden vereinigt als A und Z übernommen. B1 und B2 korrespondierten nicht und wurden einzeln übernommen.

Anwendungen arbeiten oft derart, daß sie eine Klasse als *Einstiegspunkt* verwenden und ein Objekt derselben über eine Abfrage selektieren. In Abbildung 11 ist dies die Klasse A. Von hier ausgehend wird durch das Schema nagiviert, bis die Zielklasse (Z), die das gewünschte Attribut enthält, erreicht wird. Anwendung 1 verwendet den Weg über B1, Anwendung 2 über B2.

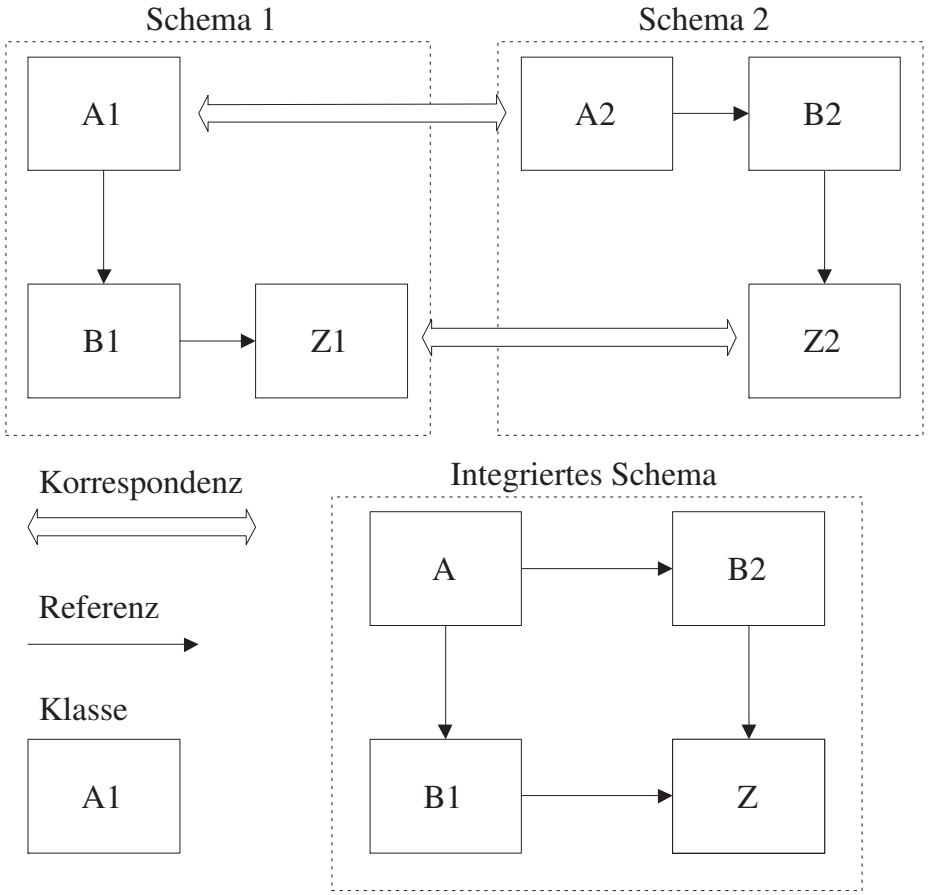

Abbildung 11: Ergebnis ohne Pfadintegration

Solange die Anwendungen alleine arbeiten, funktioniert das, wenn man annimmt, daß die entsprechenden Objekte übernommen wurden oder von den Anwendungen längs der Pfade angelegt wurden.

Ein Problem ergibt sich, sobald beide Anwendungen auf dem gemeinsamen Datenbestand arbeiten, denn jetzt kann es passieren, daß Anwendung 1 mit einem von Anwendung 2 angelegten Start-Objekt beginnt. Diesem Objekt fehlt jedoch die Referenz zu B1, so daß Anwendung 1 hier stolpert.

Auch wenn alle Anwendungen neu erstellt werden, bleiben die genannten Argumente gültig, denn die Erhaltung der Integrität der Datenbank sollte Sache der Datenbank, nicht der Programme, sein.

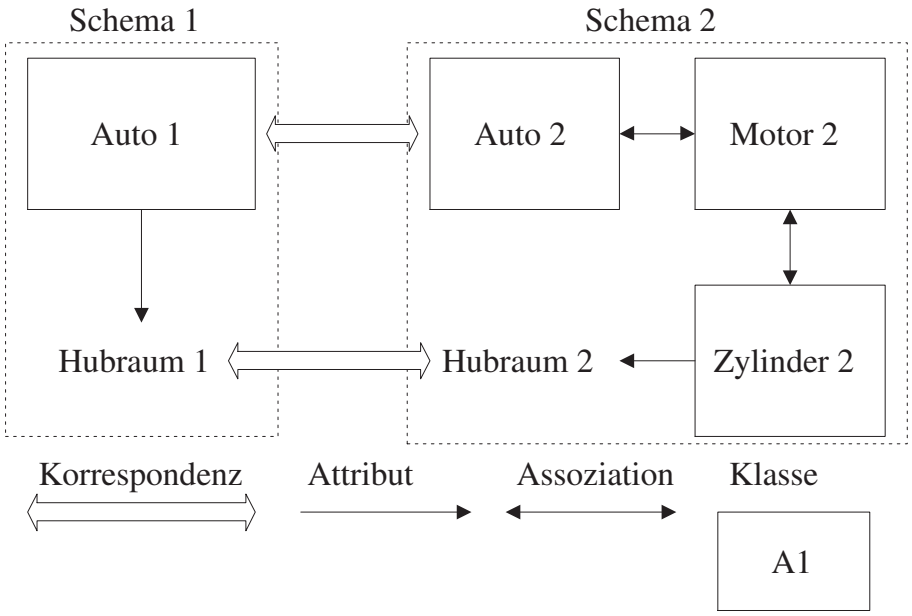

Abbildung 12: Korrespondierende Attribute in nicht korrespondierenden Klassen

Korrespondierende Attribute in nicht korrespondierenden Klassen

Ein verwandtes Problem ist in Abbildung 12 dargestellt. Das korrespondierende Attribut Hubraum liegt in jedem Teilschema in einer unterschiedlichen Klasse. Ohne spezielle Integrationsmethode enthält das Ergebnisschema ein redundantes Attribut.

7.3 Ausgangslage und Voraussetzungen

Wiederum müssen Homonyme und Synonyme durch Umbenennungen beseitigt werden. Danach sind Inter-Schema-Korrespondenzen zwischen Klassen, Attributen und Navigationspfaden aufzustellen.

7.4 Methode

Definitionen

Die Schema-Definition von SIM entspricht der eines gültigen, nichtvervollständigten Schemas bei Kosky (siehe A1-A3 auf Seite 40). Der Schema-

graph besteht aus Klassen-Knoten (Rechteck mit Text), Attribut-Knoten (nur Text), Assoziations-Kanten ($\overset{a}{\longleftrightarrow}$) und Attribut-Kanten ($\rightarrow$). Für Assoziationen soll zwar referentielle Integrität gelten, d.h. eine Objektreferenz muß immer auf ein vorhandenes Objekt des richtigen Typs verweisen, jedoch wird keine Totalität gefordert. Es darf also Objekte geben, die an der Beziehung nicht teilnehmen.

Das Grundprinzip von SIM ist, die Teilschemata aneinander anzupassen, indem die jeweils fehlenden Elemente hinzugefügt werden. Dafür gibt es je einen Erweiterungs-Formalismus (engl. augmentation) für Spezialisierungsbeziehungen und Assoziationen: A_S und A_A.

Für Vereinigung und Schnitt zweier Schemata gibt es die Operatoren $\sqcup_R$ und $\sqcap_R$. Die Operatoren bilden im wesentlichen die Vereinigungs- bzw. Schnittmenge der einzelnen Schemaelemente. Eine solch einfache Definition genügt im Rahmen dieser Arbeit. Hinsichtlich der Vererbungsbeziehungen gelten zwar zusätzliche Regeln - das tiefgestellte R steht für "restriktiv" -, welche durch Anpassung der Vererbungsbeziehungen über A_S erfüllt werden können. Da ein Algorithmus in [SIM95] nur für A_A angegeben ist und für die Vererbungsbeziehungen noch fehlt, wird jedoch für die genaue Beschreibung von $\sqcup_R$, $\sqcap_R$ und A_S auf die Originalliteratur verwiesen und im folgenden nur A_A dargestellt.

Nach allen Erweiterungen werden die Teilschemata vereinigt. Dies könnte z.B. mit Hilfe von Koskys Verfahren (Kapitel 6) anstatt des hier nicht exakt dargestellten Operators $\sqcup_R$ erfolgen.

Für die Erweiterung A_A wird eine eigene Informationsordnung $\sqsubseteq_A$ definiert. Zwei Schemata stehen in dieser Ordnung, falls jedem Element des kleineren Schemas ein oder mehrere Elemente des größeren Schemas derart zugeordnet werden können, daß diese "Abbildung der Schemaelemente" **injektiv** ist. Anders ausgedrückt, dürfen die Bildmengen zweier unterschiedlicher Schemaelemente nicht überlappen, da es sich sonst nicht mehr um eine reine Erweiterung handeln würde. $\sqsubseteq_A$ ist in der Originalarbeit formal exakt definiert. Da diese Definitionen etwas lang sind, werden die möglichen Erweiterungen von Klassen und Assoziationen hier anhand von Beispielen erläutert:

Abbildung 13 zeigt mögliche Erweiterungen. Die Klasse Firma wird in die drei Klassen Firma, Adresse und Zweigstelle abgebildet. Die neu hinzugekommenen Klassen Adresse und Zweigstelle werden über Assoziationen mit der Kardinalität [1,1] an die bestehende Klasse Firma angebunden. Das gleiche geschieht mit der Klasse Autotyp, die um die Klasse Zylinder erweitert wird.

Assoziations- und Attributkanten dürfen von einer bestehenden Klasse los-

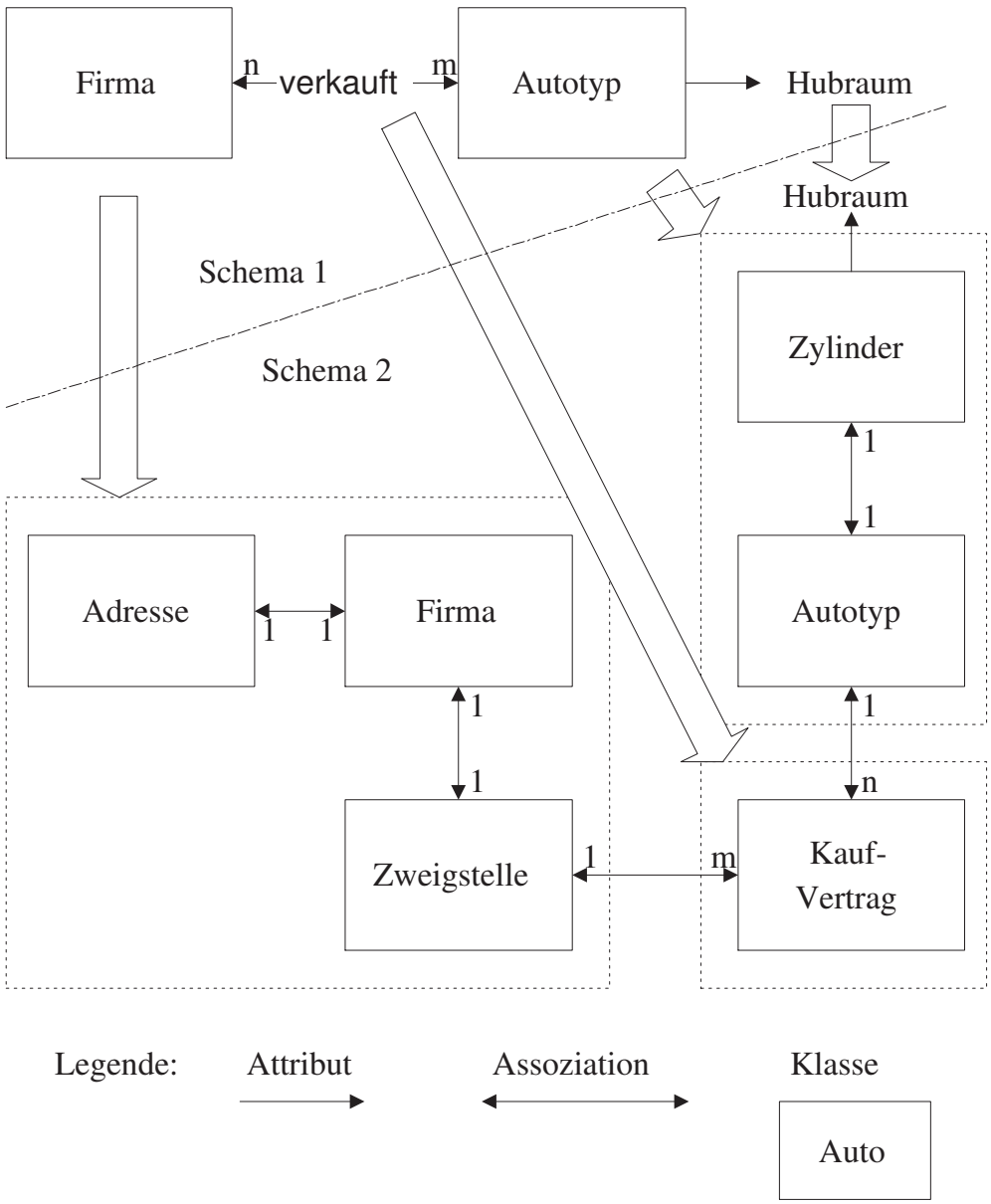

Abbildung 13: Erweiterungen zweier Klassen und einer Assoziation bei SIM

gelöst und stattdessen mit einer anderen Klasse im Bild der ursprünglich angrenzenden Klasse verbunden werden. So wurde im Beispiel das Attribut Hubraum von Autotyp zu Zylinder und die Assoziation "verkauft" von Firma zu Zweigstelle verlagert.

Assoziationen können beibehalten oder in eine neue Klasse (Link-Klasse) mit zwei Assoziationen zu Klassen aus den Bildern der ursprünglich angrenzenden

Klassen abgebildet werden. Im Beispiel wurde die Assoziation “verkauft” in die Link-Klasse Kaufvertrag mit Assoziationen zu Zweigstelle und Autotyp abgebildet. Hierbei müssen die Kardinalitäten m und n wie im Beispiel aus dem Ursprungsschema übernommen werden (m oder n darf auch 1 sein), und die Kardinalitäten der angrenzenden Klassen müssen 1 sein.

Da bestehende Klassen nicht entfernt werden, ist es möglich, alte Datenbestände zu übernehmen. Für die neu hinzugekommenen Klassen gibt es noch keine Objekte. Daher müssen für sie sogenannte *virtuelle Objekte* erzeugt werden. Bei einer Klassen-Erweiterung wird für jedes ursprüngliche Objekt in jeder der neu hinzugekommenen Klassen ein virtuelles Objekt erzeugt und über eine [1:1]-Beziehung mit dem Ursprungsobjekt verknüpft. Eine neue Link-Klasse bekommt virtuelle Objekte für jedes Paar von Objekten der angrenzenden Klassen.

Es ist auch denkbar, bestehende Anwendungen den Schemaerweiterungen entsprechend anzupassen. Wenn beispielsweise die Klasse Autotyp um die Klasse Motor erweitert und das Attribut Hubraum der Klasse Autotyp zur neuen Klasse Motor verlagert würde, so könnte parallel in der Anwendung ein Zugriff Autotyp.Hubraum in Autotyp.Motor.Hubraum umgesetzt werden.

Ein *Pfad* ist ein mit einer Klasse beginnender und mit einem Attribut oder einer Klasse endender Weg durch den Schemagraphen. Beispielsweise sind

$$(\text{Autotyp} \longleftrightarrow \text{Zylinder} \rightarrow \text{Hubraum})$$

oder

$$(\text{Adresse} \longleftrightarrow \text{Firma} \longleftrightarrow \text{Zweigstelle})$$

Pfade. Autotyp ist eine Anfangsklasse, Zylinder eine Zwischenklasse, Zweigstelle eine Endklasse und Hubraum ein End-Attribut.

Bei SIM werden die Pfade in einer Art Reißverschlußverfahren ineinandergeschachtelt. Dadurch werden die Pfade länger. Jedoch werden sie nicht holpriger, da die neuen Objekte immer über totale [1:1]-Beziehungen angebunden werden.

Eine *Pfadkorrespondenz* ist ein Paar von Pfaden. Jedoch sind nicht alle Möglichkeiten zugelassen: Seien G_1 und G_2 zwei Schemata mit den Knotenmengen C_1 bzw. C_2 (Attribute und Klassen). Sei $VC \subseteq C_1 \times C_2$ eine Menge von Korrespondenzen zwischen Klassen und zwischen Attributen. In die Menge PC der *korrekten* Pfadkorrespondenzen dürfen nur solche Paare von Pfaden aufgenommen werden, die den folgenden Bedingungen genügen:

E 1 VC und PC müssen 1:1 Korrespondenzen sein.

E 2 $\forall(p_1 \overset{a_1}{\longleftrightarrow} r_1 \overset{a_2}{\longleftrightarrow} \ldots r_m \overset{a_{m+1}}{\longleftrightarrow} p_2, q_1 \overset{b_1}{\longleftrightarrow} s_1 \overset{b_2}{\longleftrightarrow} \ldots s_n \overset{b_{n+1}}{\longleftrightarrow} q_2) \in PC$
$(p_1, q_1) \in VC$

E 3 $\forall(p_1 \overset{a_1}{\longleftrightarrow} r_1 \overset{a_2}{\longleftrightarrow} \ldots r_m \overset{a_{m+1}}{\longleftrightarrow} p_2, q_1 \overset{b_1}{\longleftrightarrow} s_1 \overset{b_2}{\longleftrightarrow} \ldots s_n \overset{b_{n+1}}{\longleftrightarrow} q_2) \in PC$
$(p_2, q_2) \in VC$

E 4 $\forall(p_1 \overset{a_1}{\longleftrightarrow} r_1 \overset{a_2}{\longleftrightarrow} \ldots r_m \overset{a_{m+1}}{\longleftrightarrow} p_2, q_1 \overset{b_1}{\longleftrightarrow} s_1 \overset{b_2}{\longleftrightarrow} \ldots s_n \overset{b_{n+1}}{\longleftrightarrow} q_2) \in PC$
$(r_i, s_j) \notin VC$

E 5 $\forall(P_1, Q_1), (P_2, Q_2) \in PC$
$P_1 \sqcap_R P_2 = \emptyset \iff Q_1 \sqcap_R Q_2 = \emptyset$

Erläuterung: Wenn ein Modellelement mit mehreren Elementen eines anderen Schema korrespondiert, muß es aufgespalten werden, z.B. durch Verwendung von Subklassen. Der SIM Algorithmus arbeitet nur mit sich Eins zu Eins entsprechenden Korrespondenzen. (E1)

Die Anfangsklassen zweier korrespondierender Pfade müssen korrespondieren (E2), ebenso die Endklassen oder End-Attribute (E3). Eine Zwischenklasse des einen Pfades darf nicht mit einer Zwischenklasse des anderen Pfades korrespondieren (E4). Dies ist jedoch leicht zu beheben, indem die korrespondierenden Pfade jeweils an der Stelle ihrer korrespondierenden Zwischenklasse aufgespalten werden. So ergeben sich zwei Pfadkorrespondenzen.

Falls jedoch eine Zwischenklasse des einen Pfades mit einer Klasse korrespondiert, die nicht Zwischenklasse des anderen Pfades ist, dann ist eine Behebung nicht möglich, denn der zweite Pfad darf nicht um die diese Klasse erweitert werden, da es sie in seinem Schema schon gibt und die Abbildung der Schemaelemente jede Klasse nur einmal abbildet. (Sonst wäre es nur eine Relation. Es dürfen also immer nur neue Elemente dazukommen.) Die Pfade können also nicht vollkommen angeglichen werden.

Deshalb wird (E4) in dieser Arbeit verschärft:

E 4+ Eine Zwischenklasse eines Pfads darf mit keiner Klasse aus dem Schema des Partnerpfads korrespondieren.

Aus der Injektivität folgt, daß es unmöglich ist, zwei Pfade, die keine Elemente gemeinsam haben, überlappend zu machen (E5), da die Mengen der neuen Elemente sich nicht überschneiden dürfen.

Beispiel

Der SIM-Algorithmus leitet Erweiterungen aus einer Menge korrekter Pfadkorrespondenzen ab. Abbildung 14 zeigt hierzu ein Beispiel. Als Eingabe seien die folgenden Pfadkorrespondenzen (Pfad aus Schema 1, Pfad aus Schema 2) gegeben:

Pfadkorrespondenz 1: $(A \longleftrightarrow B \longleftrightarrow X, A \longleftrightarrow V \longleftrightarrow X)$

Pfadkorrespondenz 2: $(A \longleftrightarrow B \longleftrightarrow C \longleftrightarrow Y, A \longleftrightarrow W \longleftrightarrow Y)$

Pfadkorrespondenz 3: $(A \longleftrightarrow Z, A \longleftrightarrow V \longleftrightarrow Z)$

Ohne die dritte Korrespondenz, die zunächst nicht betrachtet werden soll, ist eine Angleichung der Teilschemata möglich: Die Pfade von Korrespondenz 1 und 2 beginnen in Schema 1 mit der gleichen Kante AB, während sie in Schema 2 in unterschiedliche Richtungen starten. Durch die Schema-Abbildung können gemeinsam verlaufende Pfade nicht getrennt werden. Deshalb muß Schema 2 derart erweitert werden, daß die beiden Pfade ebenfalls mit einer gemeinsamen Kante beginnen. Im Beispiel kann die Klasse A um die Klasse B erweitert werden. Dies ist erlaubt, da B noch nicht in Schema 2 vorkommt. B kann nämlich gar nicht in Schema 2 vorkommen, da sonst die Pfadkorrespondenzen nicht korrekt wären (E4+). Die Kanten AW und AV werden anschließend noch von Klasse A zu Klasse B verlagert.

Durch diesen Erweiterungsschritt beginnen nun alle Pfade aus den Korrespondenzen 1 und 2 mit der Kante AB. Da die Klassen B korrespondieren, kann und muß die Kante AB überall weggelassen werden. Übrig bleiben die folgenden beiden verkürzten Pfadkorrespondenzen:

Pfadkorrespondenz 1: $(B \longleftrightarrow X, B \longleftrightarrow V \longleftrightarrow X)$

Pfadkorrespondenz 2: $(B \longleftrightarrow C \longleftrightarrow Y, B \longleftrightarrow W \longleftrightarrow Y)$

Pfadkorrespondenz 1 kann gelöst werden, indem in Schema 1 die Klasse X um die Klasse V erweitert wird, oder indem die Assoziation BX zur Link-Klasse V erweitert wird. Diese Auswahl muß der Anwender treffen. Pfadkorrespondenz 2 wird verkürzt, indem B in Schema 1 um W erweitert oder B in Schema 2

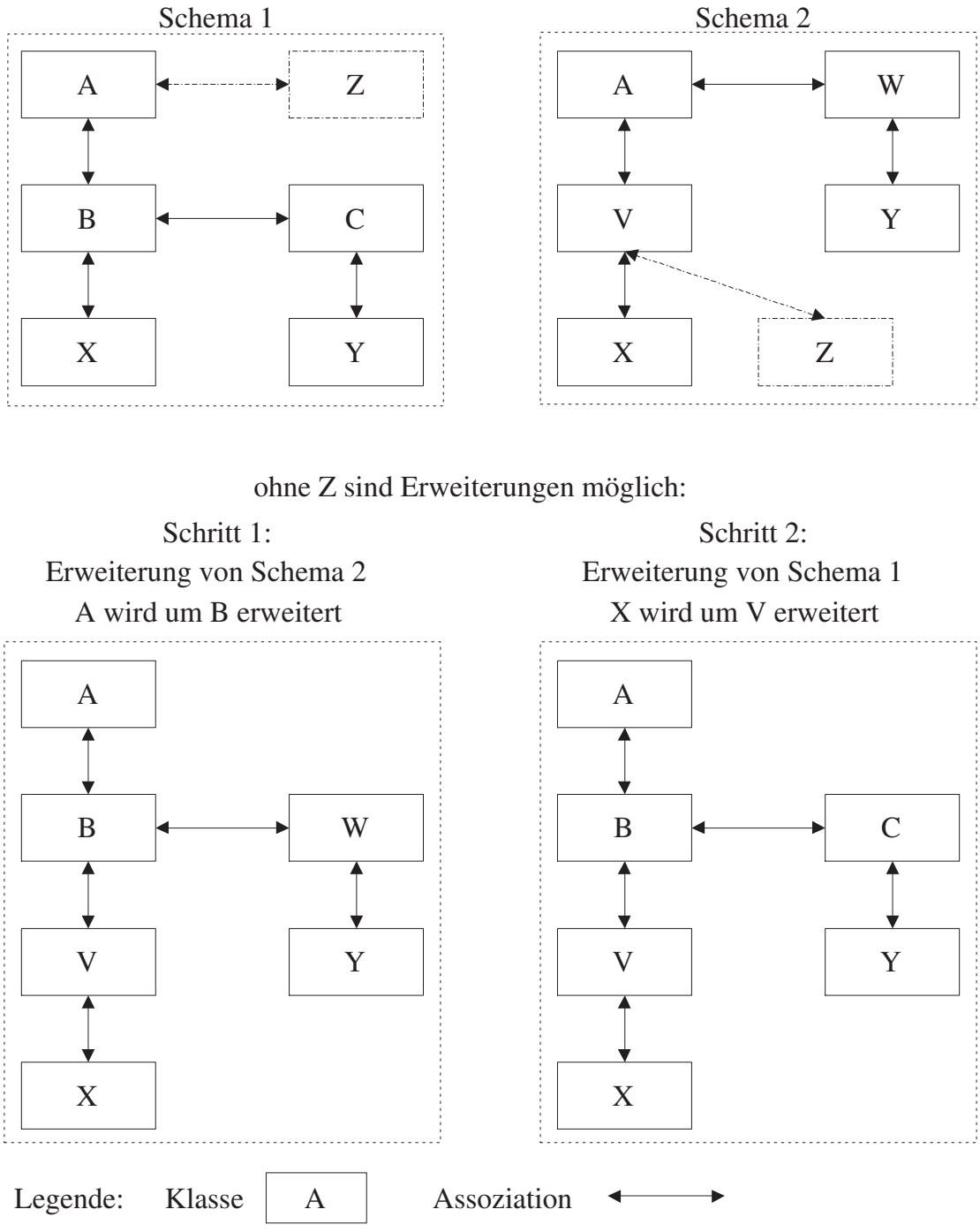

Abbildung 14: Erkennung von Erweiterungen durch Pfadkorrespondenzen

um C erweitert wird. Hier muß wiederum der Benutzer entscheiden, falls er die erste Alternative wählt, bleibt die folgende verkürzte Pfadkorrespondenz:

Pfadkorrespondenz 2: $(W \longleftrightarrow C \longleftrightarrow Y, W \longleftrightarrow Y)$

Diese Korrespondenz kann schließlich gelöst werden, indem eine der Klassen aus Schema 2 um C erweitert wird oder die Assoziation WY aus Schema 2 zur Link-Klasse C erweitert wird.

Was passiert, wenn auch die dritte Korrespondenz mit einbezogen wird? Nachdem die Pfade zu X und Z in Schema 2 anfangs parallel entlang der Kante AV verlaufen, wird es zusätzlich erforderlich, A in Schema 1 um V zu erweitern. Daraus resultiert ein Problem. Je nachdem, mit welchem Schritt begonnen wird, lassen sich zwei Fälle unterscheiden.

Fall 1: Im ersten Schritt wird (wie bisher) in Schema 2 die Klasse A um die Klasse B erweitert. Um die Pfade von A nach X und von A nach Z wie in Schema 2 mit einer gemeinsamen Kante beginnen zu lassen, muß anschließend in Schema 1 A um B erweitert werden. Das geht aber nicht mehr, da es B in Schema 1 schon gibt.

Fall 2: In Schema 1 wird A um V erweitert. Um die Pfade von A nach X und von A nach Y wie in Schema 1 mit einer gemeinsamen Kante beginnen zu lassen, muß danach in Schema 2 A um V erweitert werden. Das geht aber ebenfalls nicht mehr, da es V in Schema 2 schon gibt.

Die Schlußfolgerung ist, daß eine der Korrespondenzen gestrichen werden muß.

Algorithmus

Der Algorithmus sieht folgendermaßen aus:

Solange es noch Pfadkorrespondenzen gibt:

1. Wähle einen Knoten A_1, an dem Pfade beginnen oder enden. Die Auswahl erfolgt automatisch (das Kriterium ist, eine maximal große Partition für den folgenden Schritt zu finden, siehe Kapitel 10), oder das Programm stoppt, um den Integrator zu fragen. Im zweiten Fall steuert der Benutzer die Reihenfolge der Umstrukturierung und hat den größtmöglichen Einfluß auf die sich ergebenden Kardinalitäten.

2. Teile die Pfade in Schema 1 in Partitionen ein. Alle Pfade, die von A_1 aus zum selben Knoten verlaufen, kommen zusammen in jeweils eine Partition. Wähle eine Partition P_1 aus.

3. Sei A_2 das mit A_1 korrespondierende Entity im Partnerschema. Für alle Pfade Pf_1 aus P_1: Berechne die Partition des Partnerpfads Pf_2 von Pf_1 bei A_2.

4. Vergleiche P_1 mit allen P_2.[5] Es gibt vier Möglichkeiten.

   a) $P_1 = P_2$ (Es gibt nur eine Partition P_2):
   Mehrdeutige Erweiterung, der Anwender muß entscheiden, welches Schema erweitert werden soll.

   Die Auswahl entfällt, falls in einer der Partitionen alle Pfade nur noch aus zwei Entities und einer einzigen Beziehung bestehen. In diesem Fall wird automatisch das Schema mit der Partition dieser "kurzen Pfade" erweitert. Es sind entweder alle Pfade einer Partition oder keiner kurz. Andernfalls würde ein Zwischenknoten mit einem Endknoten zusammenfallen. Bei korrekten Korrespondenzen korrespondiert ein Zwischenknoten mit keinem Knoten des Partnerschema (E4+), ein Endknoten dagegen immer (E2, E3), so ergäbe sich ein Widerspruch.

   Anschließend muß der Integrator immer bestimmen, ob die Erweiterung als Link-Klasse ausgeführt werden soll.

   b) $\forall P_2.P_2 \subset P_1$: Schema 2 wird *automatisch* erweitert.

   c) $P_1 \subset P_2$ (Es gibt nur eine Partition P_2): Schema 1 wird *automatisch* erweitert. Die genaue Vorgehensweise, um Überlappungen frühzeitig zu entdecken, ist: Gehe zu 3. zurück. Vertausche die Rollen von Schema 1 und Schema 2 und verwende P_2 statt P_1. Wieder bei 4. angekommen, ist jetzt b) oder d) anstatt c) möglich.

   d) Fehler: Wenn keine der bisherigen drei Alternativen zutrifft, liegt *Überlappung* vor. In diesem Fall muß der Anwender Pfadkorrespondenzen streichen.

5. Passe die Pfade entsprechend der durchgeführten Erweiterung an. Betroffen sind alle Pfade der größeren bzw. ausgewählten Partition im (evtl. nach Vertauschung) ersten Schema und deren Partnerpfade im erweiterten Schema. Die letzteren beginnen bzw. enden jetzt mit der neu hinzugekommenen Klasse. Im ersten Schema verkürzen sich die Pfade. Falls aufgrund der Verkürzung beide Pfade einer Korrespondenz kurz sind, wird diese Korrespondenz gestrichen.

[5] Um den Vergleich durchführen zu können, müssen die Partitionen des zweiten Schema umgerechnet werden: Statt der Pfade des zweiten Schema werden die korrespondierenden Pfade des ersten Schema verwendet. Andernfalls wäre der Schnitt immer leer.

Man kann sich überlegen, daß die Korrektheitseigenschaften (E2) - (E4) bei einer Erweiterung erhalten bleiben. (E1), (E4plus) und (E5) können jedoch verletzt werden, z.B. wenn sich Pfade in einem Schema kreuzen (siehe Kapitel 10.9).

Da bei jedem Durchlauf mindestens ein Pfad gekürzt wird, muß der Algorithmus terminieren. Die Datenstrukturen und weitere Details für diesen Algorithmus werden in Kapitel 10 beschrieben.

7.5 Bewertung und Zwischenstand

Der Hauptvorteil von SIM ist, daß Redundanz bei Wertattributen und Referenzen vermindert wird, so daß die Datenbank nicht so leicht inkonsistent werden kann.

Ein weiterer Vorteil von SIM ist, daß Anwendungen bei der Navigation durch das Schema öfter zu einem gewünschten Ziel-Attributwert gelangen, wenn der Pfad mit einem *fremden* Start-Objekt beginnt, jedenfalls dann, wenn das fremde Ziel-Attribut mit dem "eigenen" korrespondiert. Die Verständlichkeit des Schemas wird erhöht, da es weniger Unklarheiten darüber gibt, welcher Pfad zum benötigten Attribut der Richtige ist.

Bezüglich Vollständigkeit und Minimalität leistet sich das Verfahren keine Schwächen. Zwar werden die Kardinalitätsbereiche erweitert, dies kann aber für die Migration alter Datenbestände auch nötig sein.

Auch mehr als zwei Teilschemata können durch SIM erweitert werden. Hierbei hängt das Ergebnis nicht von der Reihenfolge ab.

Da SIM das Schema relativ rigoros umbaut, kann es sein, daß das Ergebnis gravierende Mängel enthält. Ein Beispiel ist die Korrespondenz

(Tabelle $\longleftrightarrow$ Zeile $\longleftrightarrow$ Zelle, Tabelle $\longleftrightarrow$ Spalte $\longleftrightarrow$ Zelle).

In jedem Schema werden unterschiedliche Anordnungen der Zellen verwendet. Die Integration zu (Tabelle $\longleftrightarrow$ Zeile $\longleftrightarrow$ Spalte $\longleftrightarrow$ Zelle) wäre falsch, da eine Spalte nicht Teil einer Zeile ist. Bei zweifelhaften Vorgaben ist also die Korrektheit gefährdet.

Bei der Bildung der Partitionen wird nur der Name des nächsten Knotens im Pfad berücksichtigt, aber nicht die Semantik der Beziehungen. Abb. 1 aus Kapitel 3 (Seite 12) mit den Assoziationen "vorgemerkt" und "entliehen" zeigte ein Beispiel für die Notwendigkeit unterschiedlicher Semantiken. Würde man die Partitionen feiner untergliedern, ergäbe sich das Problem, daß manchmal zwei parallele Beziehungen unterschiedlicher Semantik zur selben Linkklasse

erweitert werden sollen, dies aber ohne simultane Behandlung nicht geht, da die Schemaabbildung jede Klasse nur einmal "einbaut". (Sonst wäre es keine Abbildung, sondern nur eine Relation.) Bei SIM hat man sich für die simultane Behandlung entschieden. Es ist deshalb nicht möglich, "entliehen" zu einer Linkklasse mit dem Rückgabedatum zu erweitern, aber "vorgemerkt" zu belassen.

Vererbungshierarchien werden bisher ausgeklammert. Zum einen sollte als Pfadkorrespondenz auch die Möglichkeit einbezogen werden, daß eine Anfangs- bzw. Endklasse des ersten Pfades mit einer Klasse korrespondiert, die mit der Anfangs- bzw. Endklasse des zweiten Pfades nicht übereinstimmt, sondern nur in einer Vererbungsbeziehung steht. Zum anderen können korrespondierende Pfade auf unterschiedlichen Höhen in der Vererbungshierarchie verlaufen. Wenn Zwischenknoten in einer Spezialisierungsbeziehung stehen, sollte es die Möglichkeit geben, die Pfade an dieser Stelle zu splitten.

Insgesamt ist SIM eine wesentliche Methode für die Ermöglichung der Updatefähigkeit, deren Ansatz noch genauer erforscht werden muß.

8 Schlüssel und weitere Integritätsbedingungen

8.1 Kapitelüberblick

Inter-Schema-Beziehungen verbinden zwei, aus unterschiedlichen Teilschemata stammende, Klassen. Die Betrachtung der Schlüssel und anderer Integritätsbedingungen wie Totalität kann dabei helfen, solche Assoziationen zu finden. Dazu werden im folgenden zwei Methoden dargestellt. Beide führen eine View-Integration durch. Sie unterscheiden sich in Bezug auf die Möglichkeit, Inserts gegen die ursprünglichen Views durchführen zu können.

Views sind für die Produktdatenintegration generell nützlich. Die Programmierung der Werkzeuge wäre unnötig kompliziert, wenn jedes Werkzeug das komplette Schema kennen müßte. EXPRESS bietet mit Hilfe der SCHEMA-Deklaration die Möglichkeit, Teilschemata mit einem eigenen Namensraum zu definieren. Views können abgeleitete Daten enthalten, die es auch in EXPRESS gibt.

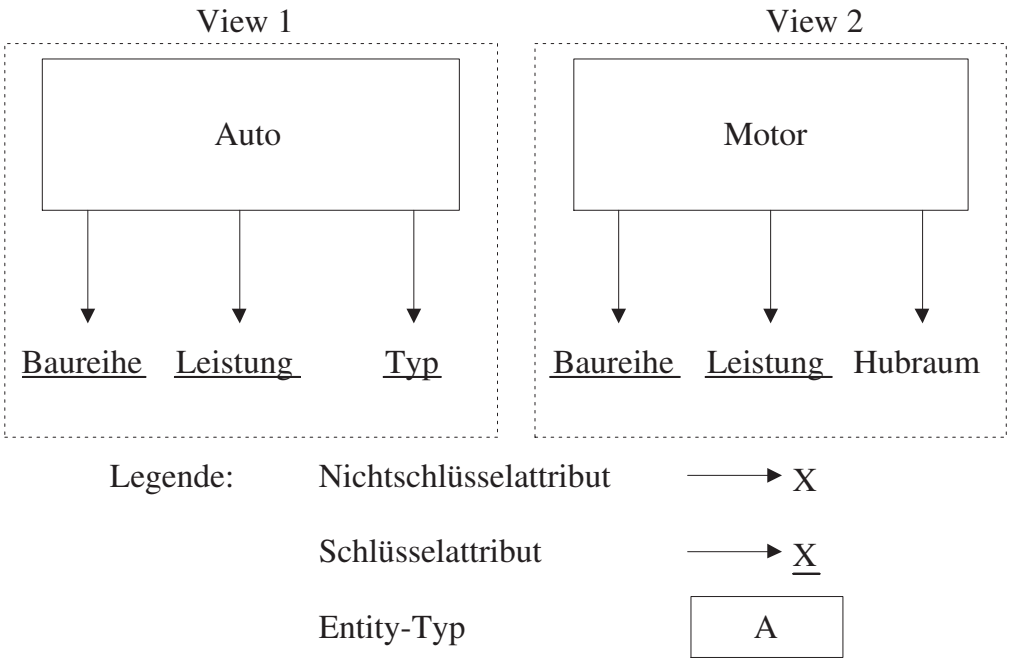

Abbildung 15: Schlüssel mit korrespondierenden Attributen

8.2 Problematik

Abbildung 15 zeigt die Entities Auto und Motor, deren Schlüsselattribute Baureihe und Leistung jeweils korrespondieren. Das Entity Auto hat zusätzlich das nicht korrespondierende Schlüsselattribut Typ. Diese Konstellation kann wie folgt interpretiert werden: Mit dem gleichen, vorhandenen Motor können unterschiedliche Autos gebaut werden. Es darf aber kein Auto aus nicht vorhandenen Motoren gebaut werden.

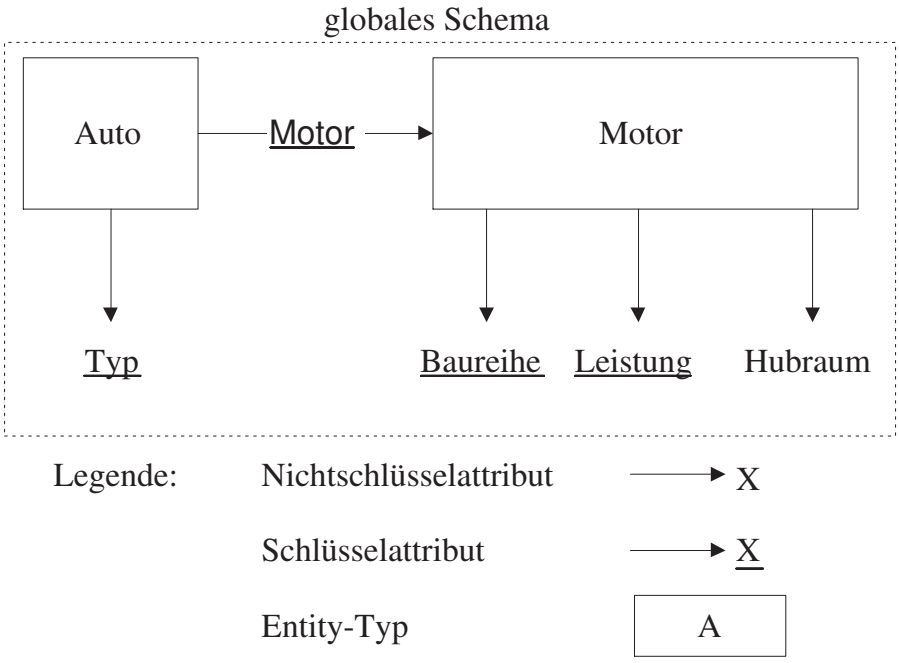

Abbildung 16: Gewünschtes Ergebnis nach Analyse der Schlüssel

Die Methode SIM ist hier nicht direkt anwendbar, da eine Referenz von Auto zu Motor, die zur Definition korrespondierender Pfade genutzt werden könnte, fehlt. Nach Erzeugung dieser Referenz in der Richtung, wie es die Schlüssel erfordern, könnte eine redundanzfreie Lösung wie in Abbildung 16 aussehen.

Dieses Beispiel sollte als Motivation dienen, Integritätsbedingungen genauer zu betrachten.

8.3 Die Methode von Biskup und Convent

Die Arbeit [Bis86] baut auf der relationalen Theorie auf. Der Begriff der Relation ist in diesem Kapitel ein Synonym zu Entity. Die Autoren unterscheiden Integritätsbedingungen nach ihrem Zweck: Zur Modellierung oder als Hilfsmittel zur Schemaintegration (Integrationsbedingungen). Zunächst werden die Views zu einem *kombinierten Schema*, welches zusätzliche Integrationsbedingungen enthält, vereinigt. Durch Transformationen werden die Integrationsbedingungen anschließend soweit möglich vereinfacht oder ganz entfernt. Am Ende dieses Prozeß steht ein *globales Schema*. Das Ergebnis gilt als korrekt, wenn jeder ursprüngliche View mit einem relationalen Ausdruck aus dem globalen Schema abgeleitet werden kann, es also zu jeder beliebigen Belegung eines Views mit Daten auch eine Belegung des globalen Schemas derart gibt, daß der zugehörige relationale Ausdruck die Daten des Views liefert.

Diese Bedingung ist bei Verwendung des Schemas aus Abbildung 16 als globales Schema erfüllt. Relational ausgedrückt, muß für die Relation Motor aus dem zweiten View in Abbildung 16 nichts gemacht werden, während für die Relation Auto aus dem ersten View ein natürlicher Join der Relationen Auto und Motor mit anschließender Projektion auf Baureihe, Leistung und Typ durchgeführt werden muß.

Gegen den Join-View kann im allgemeinen kein Insert durchgeführt werden, da das für einen neuen Motor erforderliche Attribut Hubraum in diesem View gar nicht vorkommt.

Die im Beispiel verwendete Integrationsbedingung kann als Mengeninklusion der sich durch Projektion auf die Attribute Baureihe und Leistung in den beiden Views ergebenden Relationen formuliert werden:

Auto [Baureihe, Leistung] $\subset$ Motor [Baureihe, Leistung]

Anders wäre es bei Gleichheit dieser projizierten Relationen:

Auto [Baureihe, Leistung] = Motor [Baureihe, Leistung]

Solange das Attribut Typ noch zum Schlüssel der Auto-Relation im ersten View gehört, bleibt das bisherige globale Schema sinnvoll, denn die Änderung bedeutet nur, daß alle Motortypen auch in Autos eingebaut werden müssen, also die Totalität der Referenz von Auto zu Motor im globalen Schema. Es können immer noch unterschiedliche Autotypen mit dem gleichen Motor vorkommen.

Anders sieht es aus, wenn das Attribut Typ aus dem Schlüssel entfernt wird, so daß jedem Auto genau ein Motor entspricht. In diesem Fall ergibt sich

ein anderes globales Schema. Es besteht jetzt aus einer einzigen Relation mit allen Attributen aus den beiden Views: Rglobal (Baureihe, Leistung, Typ, Hubraum).

Obwohl lediglich ein Schlüssel anders definiert wurde, hat sich das Integrationsergebnis verändert. Diese Möglichkeit ist in den drei anderen bisher vorgestellten Arbeiten nicht enthalten.

Biskup und Convent stellen noch einige andere Integrationsbedingungen und einen Algorithmus zur Erstellung des globalen Schema vor. Dieser Algorithmus versucht bei jedem Schritt, eine Integrationsbedingung zu entfernen oder zu vereinfachen und die Relationen entsprechend zu kombinieren. Jedoch sind immer noch manuelle Eingriffe erforderlich, wenn sich mehrere Integrationsbedingungen behindern, indem sie sich z.B. auf dieselbe Relation beziehen.

Die vollautomatische Integration ist deshalb nicht möglich, da die Probleme der Widerspruchsfreiheit und der Implikation von Inklusionsbedingungen, wie schon gesagt, nicht entscheidbar sind [Mitc83, Ek95].

8.4 Die Methode von Vidal und Winslett

Die Autorinnen von [Vid94] verwenden ein kombiniertes und ein globales Schema, wie in der vorhergehenden Methode dargestellt. Jedoch sollen auf den ursprünglichen Views auch Änderungen durchführbar sein, die in eine Änderung auf dem globalen Schema transformiert werden.

Beispiel: Einfügeoperationen gegen den Join-View Auto aus dem letzten Abschnitt sollten ermöglicht werden, falls ein neuer Auto-Typ mit einem bereits bestehenden Motor erzeugt werden soll. Dagegen sollten Inserts mit nicht existierenden Motor-Attributen zurückgewiesen werden.

Die Lösung besteht zum einen darin, Integrationsbedingungen mit einer zusätzlichen Update-Semantik auszustatten. Insert- und Delete-Operationen werden entsprechend dieser Semantik entweder abgelehnt oder ausgeführt und nicht propagiert oder ausgeführt und propagiert.

Zum anderen werden die Integrationsschritte feiner in Zwischenschritte untergliedert. Abbildung 17 zeigt ein Beispiel für so einen Zwischenschritt. Der zusätzliche Entity-Typ Motor2 enthält einen Teil der Attribute der tatsächlich in Autos eingebauten Motortypen. Das globale Schema bleibt gegenüber dem Beispiel im letzten Abschnitt unverändert.

Für das Schema im Beispiel sind mehrere Integrationsbedingungen nötig:

- (am2-R1) Referentielle Integrität von am2.

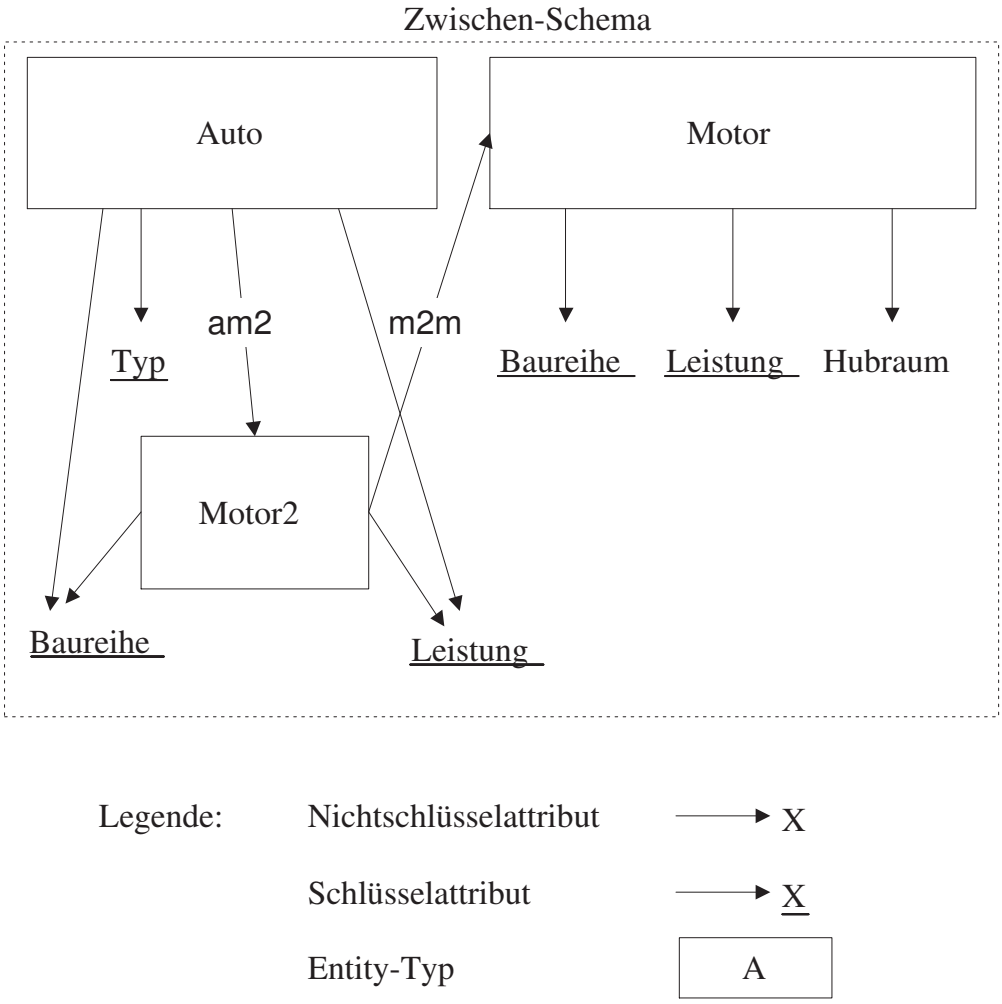

Abbildung 17: Zwischenschritt der Transformation zum globalen Schema

- (am2-T1) Totalität von am2.
- (am2-P1) Ausgehend von einem bestimmten Auto muß der indirekte Zugriff über die Referenz am2 zum Attribut Baureihe denselben Wert liefern, wie der direkte Zugriff.
- (am2-P2) Ausgehend von einem bestimmten Auto muß der indirekte Zugriff über die Referenz am2 zum Attribut Leistung denselben Wert liefern, wie der direkte Zugriff.
- (m2m-I1) Es gilt die Inklusion Motor2 [Baureihe, Leistung] $\subset$ Motor [Baureihe, Leistung]

Die Update-Semantik der Integrationsbedingungen ist in der folgenden Tabelle aufgelistet. Die zweite Spalte beinhaltet das Entity, gegen welches ein Insert oder Delete ausgeführt werden soll. AP bedeutet, daß die Aktion ausgeführt und propagiert wird:

Integrationsbedingung	Entity	Insert	Delete
am2-R1, am2-T1, am2-P1, am2-P2	Auto	AP	ausgeführt
am2-R1, am2-T1, am2-P1, am2-P2	Motor2	ausgeführt	AP
m2m-I1	Motor2	abgelehnt	ausgeführt
m2m-I1	Motor	ausgeführt	AP

Da jede Bedingung ihre eigene Update-Semantik hat, können Widersprüche auftreten. Dies ist hier nicht der Fall. Es kommt zwar das Entity "Motor2" in zwei Zeilen vor, aber für einen Widerspruch muß neben dem Entity auch die Beziehung übereinstimmen (Die Beziehungen am2 und m2m sind unterschiedlich).

Ein Update gegen einen ursprünglichen View würde nun in einen Update gegen das Zwischenschema transformiert und bei Erfolg (keine Ablehnung) in einen Update gegen das globale Schema weitertransformiert werden (es kann auch mehrere Zwischenschemata geben). Im Beispiel würde die Erzeugung eines neuen Auto-Typs den entsprechenden Insert in Auto des Zwischen-Schemas bewirken. Dieser Insert wird nach Motor2 propagiert. Falls der entsprechende Motor schon vorhanden ist, passiert nichts und die erfolgreiche Operation kann gegen das globale Schema weitertransformiert werden. Falls jedoch ein tatsächlicher Insert in Motor2 nötig ist, kommt die Update-Semantik von m2m-I1 zum Tragen und die ganze Aktion wird zurückgewiesen.

8.5 Bewertung

Die in diesem Kapitel vorgestellten Methoden enthalten einige nützliche Erkenntnisse: Die Wahl der Schlüssel kann das Ergebnis der Schemaintegration mitbestimmen. Integritätsbedingungen können, als Integrationsbedingung verwendet, Inter-Schema-Korrespondenzen ersetzen und Redundanzen absichern. Falls es nicht gelingt, störende Bedingungen wegzutransformieren und das integrierte Schema Views enthalten soll, sollten die Integritätsbedingungen mit einer Update-Semantik versehen werden, sofern sie über View-Grenzen verlaufen. In der bisherigen Version von EXPRESS fehlt die Möglichkeit, eine Update-Semantik anzugeben.

Die Korrektheit ist gewährleistet, da die einzelnen Views durch explizit angegebene Regeln aus dem globalen Schema abgeleitet werden. Auch die Krite-

rien Minimalität und Vollständigkeit werden erfüllt, und die Verständlichkeit steigt parallel zur Reduzierung der Redundanz. Es ist möglich, mehr als zwei Schemata zu vereinigen.

Die in den anderen Kapiteln vorgestellten Arbeiten werden durch die hier vorgestellte Methodik ergänzt. Neben SIM stellt dieses Verfahren eine zweite Art dar, korrespondierende Attribute aus nicht korrespondierenden Klassen zu behandeln, die im vereinigten Schema auch in keiner Vererbungsbeziehung stehen würden.

9 Kombination der Verfahren

9.1 Kapitelüberblick

Die vorgestellten Verfahren decken jeweils nur einen Teilaspekt des Gesamtproblems ab. Deshalb ist es wünschenswert, die Methoden zu kombinieren. Eine Möglichkeit hierzu wird im ersten Teilabschnitt beschrieben. Anschließend wird eine geeignete Methode für die Implementierung ausgewählt.

9.2 Kombination der Verfahren

Jedes der Verfahren vermeidet andere eklatante Fehler im Ergebnisschema. Deshalb stellt sich die Frage, ob und wie die vier vorgestellten Verfahren kombiniert werden können. Die im folgenden beschriebene Vorgangsweise ist naheliegend:

1. Durch einen Schlüsselvergleich gemäß Kapitel 8 wird nach fehlenden Beziehungen zwischen den Schemata gesucht. Die gefundenen neuen Assoziationen werden in einem zusätzlichen, neuen Teilschema gesammelt, welches anschließend mitverarbeitet wird. Die Methode eignet sich gut für den ersten Schritt, da die bestehenden Schemata unangetastet bleiben und man sich daher keine Möglichkeiten verbaut.

2. Hierarchisch aufgebaute Datentypen werden mit dem Verfahren von Abiteboul und Hull aus Kapitel 5 umstrukturiert, um eine bessere Vergleichbarkeit zu erzielen. Diese Aktion kann in jedem Teilschema separat durchgeführt werden.

3. Anschließend werden Pfadkorrespondenzen ermittelt, mit denen die Methode SIM aus Kapitel 7 gestartet wird, um Unterschiede in der Granularität auszugleichen. Die Teilschemata werden durch Erweiterungen aneinander angeglichen. Diese Erweiterungen betreffen möglicherweise auch die umgebauten hierarchischen Datentypen und werden deshalb erst nach dem zweiten Schritt durchgeführt.

4. Die Vereinigung der Teilschemata findet mit Hilfe des Verfahrens von Kosky aus Kapitel 6 statt. So ergibt sich ein syntaktisch korrektes Gesamtschema.

5. Das Gesamtschema kann mit (in dieser Arbeit nicht vorgestellten) Methoden, wie [Bri98], noch weiter verbessert werden.

Die Kombination der Verfahren wirft neue Fragestellungen auf, die in weiteren Arbeiten geklärt werden müßten. So sind beispielsweise die im ersten Schritt gefundenen Inter-Schema-Beziehungen nicht an der Bildung von Pfadkorrespondenzen im dritten Schritt beteiligt, da mit den anderen Teilschemata jeweils nur eine der beiden Klassen jeder neu gefundenen Assoziation korrespondiert. Ganz auszuschließen ist aber nicht, daß über die neuen Beziehungen nach der späteren Vereinigung redundante Pfade verlaufen.

Ein andere offene Frage wurde bereits bei der Bewertung von SIM am Ende von Kapitel 7 angesprochen: Wie muß SIM angepasst werden, um in Vererbungshierarchien sinnvoll angewendet werden zu können?

Nach diesem Ausblick auf zusätzlich mögliche Forschungsvorhaben kehrt der nächste Abschnitt zurück zur Aufgabenstellung, um eines der Verfahren für die geforderte Implementierung zu wählen.

9.3 Auswahl einer Methode zur Implementierung

Das wichtigste Ziel, welches die vorgestellten Methoden erfüllen müssen, ist Redundanzfreiheit, wie in der Einleitung beschrieben. So sollen konsistente Updates möglich werden. Zwar kann Redundanz auch durch Integritätsbedingungen abgesichert werden. Dann leidet aber die Updatefähigkeit, sobald die bei komplexen Modellen unvermeidlichen Views verwendet werden (siehe Kapitel 8). Zur Redundanzfreiheit tragen die Methoden der Kapitel 5 (Abiteboul und Hull), 7 (SIM) und 8 (Biskup und Convent bzw. Vidal und Winslett) bei. Die Methode von Abiteboul und Hull eignet sich jedoch in der dargestellten Form nicht zur Implementierung (siehe die Beurteilung am Ende von Kapitel 5). Somit schränkt sich die Auswahl auf SIM und die Verfahren aus Kapitel 8 ein.

Daß SIM gewählt wurde, hat die folgenden Gründe: Zur Zeit ist die Integration hierarchischer XML [XML98] Daten sehr gefragt. Würde man nur die Wurzeln dieser baumförmigen Strukturen vereinigen, bliebe das Schema hierarchisch. Dadurch wird aber Redundanz in den übrigen Knoten erzeugt, und die Anwendungen müssen mehrere Pfade in Betracht ziehen, um zu einem Attribut zu finden. Die Folgerung ist, daß alle korrespondierenden Knoten vereinigt werden müssen. Selbst wenn anfangs hierarchische Schemata vorlagen, ist das vereinigte Schema dann nicht mehr hierarchisch, sondern es enthält redundante Pfade. Bei SIM kommt das Ergebnisschema der hierarchischen Form je näher, desto mehr Pfadkorrespondenzen angegeben werden, so daß diese Methode die Integration von XML Schemata erleichtert. Dagegen entfernt sich das Ergebnisschema durch die Verfahren aus Kapitel 8 durch

neue Querverbindungen von der hierarchischen Form.

9.4 Zusammenfassung

Die vorgestellten Methoden können kombiniert werden, was neue Fragestellungen aufwirft. Einzeln decken sie nur Teilaspekte des Gesamtproblems ab. Es gibt also keinen klaren Sieger, der fast alles löst. SIM trägt viel zur Redundanzverminderung bei, bietet Ansätze für weitere Forschungsvorhaben und wäre auch für die Integration von Schemata nützlich, die hierarchisch bleiben müssen. Deshalb wurde SIM zur Implementierung ausgewählt.

10 Implementierung

10.1 Kapitelüberblick

Dieses Kapitel beschreibt zuerst (Abschnitt 10.2) die bestehende C++ Umgebung und die Integration der neuen Klassen darin (Abschnitt 10.3).

Anschließend werden die benutzten Teile der C++ Standardbibliothek vorgestellt (Abschnitt 10.4). Abschnitt 10.5 erklärt dann, wie die verwendeten Datenstrukturen aufgebaut sind.

Abschnitt 10.6 nennt die groben Schritte des Programmablaufs.

Die Beschreibung der Implementierung der Kernbestandteile des SIM Algorithmus zerfällt in drei Teile. Der erste Abschnitt 10.7 beschreibt diejenigen Prüfungen für einzelne Pfade und Korrespondenzen, die direkt aus der Originalarbeit [SIM95] hervorgehen. Im zweiten Teil 10.8 wird dargestellt, welche Aktionen für einen Schemaerweiterungsschritt nötig sind. Hierbei ändern sich die Pfade. Deshalb muß nachvollziehbar sein, daß die Bedingungen (E1 - E5) aus Kapitel 7 erhalten bleiben. Die Originalarbeit ist an dieser Stelle etwas unklar. Der dritte Teilabschnitt 10.9 modifiziert daher (E1 - E5), um den Algorithmus sicher zu machen. Dabei erweitern sich die Prüfungen des Abschnitts 10.7.

Abschnitt 10.10 enthält ein Beispiel für den Ablauf des Programms: Gezeigt werden die Eingabeschemata, die Korrespondenzdatei, der Dialog mit dem Benutzer, das Protokoll und das Ausgabeschema.

10.2 Die bestehende Umgebung

1996 wurde am Lehrgebiet Praktische Informatik I unter der Leitung von Prof. Schlageter damit begonnen, Software für das STEP Projekt zu entwickeln. Die für diese Arbeit bereitstehenden Komponenten wurden von Mitarbeitern des Lehrgebiets in der Programmiersprache C++ geschrieben.

Abbildung 18 auf der folgenden Seite zeigt die Verwendung der übernommenen Klassen (nicht fettgedruckt). Die Pfeile weisen jeweils vom Aufrufer zum Aufgerufenen. Insbesondere handelt es sich um:

- Interne EXPRESS Datenstrukturen (C++ Objekte), insbesondere für Schemata, Entities und Attribute. Mengen dieser Objekte werden als verkettete Listen (wiederum C++ Objekte) verwaltet, z.B. MemberList für Attribute eines Entity's. (Unten in Abbildung 18.)

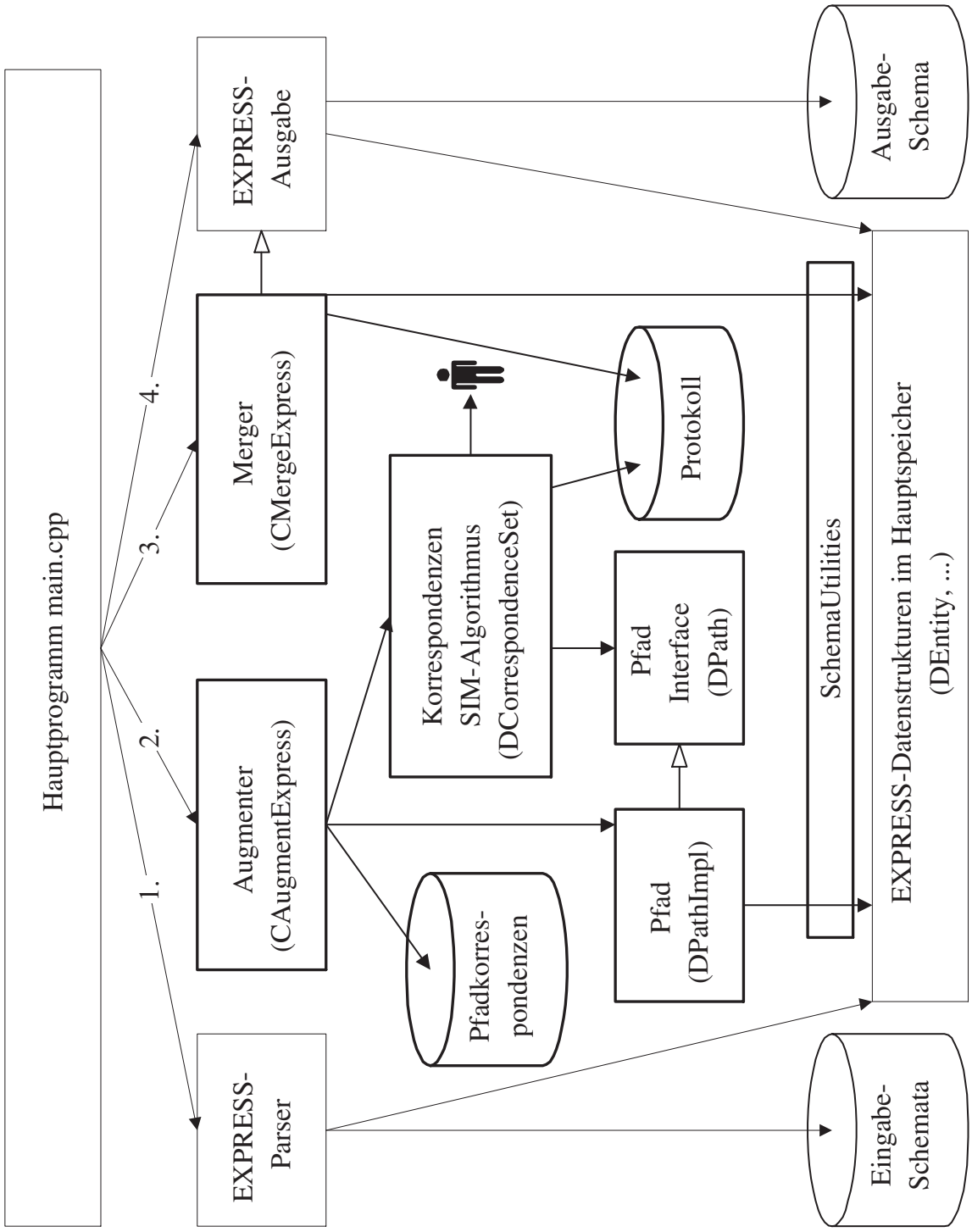

Abbildung 18: Aufruffolge

- Eine Kombination aus Scanner und Parser zum Einlesen der Schemata aus einer Textdatei und zum Aufbau der internen Objekte. (Links in Abbildung 18.)
- Ein Generator zur Erzeugung des EXPRESS Ausgabeschemas als Textdatei aus den internen Datenstrukturen. (Rechts in Abbildung 18.)
- Ein Hauptprogramm zur Auswertung der Programmoptionen von der Kommandozeile und zur Steuerung des Ablaufs. (Oben in Abb. 18.)

10.3 Die neuen Klassen

Hinzugekommen sind zwei unabhängige Verarbeitungsstufen, die nacheinander ausgeführt werden, aber zu Testzwecken auch einzeln ausführbar sind. In Abbildung 18 sind die neuen Klassen fettgedruckt.

- Der Augmenter (Mitte bis Mitte links in Abbildung 18) erweitert Schemata entsprechend der in einer Datei vorgegebenen Pfadkorrespondenzen. Dazu gehören die Klassen CAugmentExpress, DCorrespondenceSet, DPath und DPathImpl. Diese Stufe führt den SIM Algorithmus aus und wird in den nächsten Abschnitten genau erklärt.
- Der Merger (Mitte rechts in Abbildung 18) vereinigt alle Schemata in eines. Hierbei werden Entities und Typen, deren Name nur in einem der Schemata vorkommt, in das Ergebnisschema übernommen.

  Bei gleichnamig benannten Typen schreibt der Merger eine Warnung in die Protokolldatei. Für den SIM Algorithmus ist dies ausreichend. Für das Gesamtproblem wären Erweiterungen, z. B. gemäß des Algorithmus von Abiteboul und Hull aus Kapitel 5 denkbar.

  Gleichnamige Entities werden vereinigt, indem alle Attribute, deren Name nur in einem der Entities vorkommt, übernommen werden. Kommt ein Attributname in beiden Entities vor, so wird bei ungleichen Typnamen eine Fehlermeldung ausgegeben. Hier bestünde beim Algorithmus von Kosky aus Kapitel 6 die Alternative, eine Subklasse zu erzeugen. Bei gleichen Typnamen werden die Kardinalitätsbereiche vereinigt.

  Der Merger ist von der bestehenden Klasse CDictIterator abgeleitet, die für die Ausgabe des EXPRESS Schema zuständig ist, um deren Fähigkeiten beim Schreiben von Typinformation für Attribute in die Protokolldatei wiederzuverwenden.

Die aufgrund der Anbindung der neuen Klassen notwendigen Erweiterungen der EXPRESS Umgebung bestehen in einigen zusätzlichen Methoden. Durch Subklassen könnten die meisten dieser Änderungen an den bestehenden Klassen vermieden werden.

10.4 Die C++ Standardbibliothek

Ebenfalls 1996 wurde ein neuer C++ Standard [ISO96, Bre97] verabschiedet, der mittlerweile von den Compilerherstellern umgesetzt wurde. Wesentliche Neuerungen des ISO96 Standards sind:

- Container, deren Elemente in sortierter Form in balancierten Binärbäumen gespeichert sind (Stichwort: red black tree [Kle95]). Es handelt sich dabei um Templates, die mit den Klassen der enthaltenen Elemente und der ggf. verwendeten Schlüssel instanziert werden müssen. Daraus resultiert Typsicherheit.

  Im einzelnen stehen die folgenden, intern als Baum realisierten, Container zur Verfügung:

  - set: Menge, in der jedes Element maximal einmal vorkommt (Element = Schlüssel)
  - multiset: Menge mit Duplikaten (Element = Schlüssel)
  - map: Assoziativer Container mit eindeutigen Schlüsseln (Element $\neq$ Schlüssel)
  - multimap: Assoziativer Container mit Duplikat-Schlüsseln (Element $\neq$ Schlüssel)

- Algorithmen, die mit Hilfe von Iteratoren (erweiterten Zeigern) auf den Containern arbeiten.

  Zum Einsatz kommen die folgenden Algorithmen für die Mengen S_1 und S_2. Die genannte Komplexität entspricht jeweils einer einfachen Implementierung:

  - set_intersection: Schnittmenge $S_1 \cap S_2$ mit der Komplexität $O(|S_1| \log(|S_2|))$
  - set_difference: Mengendifferenz $S_1 - S_2$ mit der Komplexität $O(|S_1| \log(\max(|S_1|, |S_2|)))$
  - includes: Mengeninklusion $S_1 \subset S_2$ mit der Komplexität $O(|S_1| \log(|S_2|))$

  - find: Suchen der Elemente zu einem gegebenen Schlüssel in der Menge S_1. Das Ergebnis sei S_2.
    Die Komplexität ist mit Duplikaten $O(|S_2| + \log(|S_1|)$, ohne Duplikate $O(\log(|S_1|))$.
  - erase: Wie find, mit S_2 als Menge der zu löschenden Elemente des angegebenen Schlüssels.
  - insert: Einfügen eines Elementes in die Menge S_1.
    Die Komplexität ist $O(\log(|S_1|))$.

  Die Maximumbildung bei der Mengendifferenz ist wegen des Aufbaus der Ergebnismenge nötig. Die Angaben zur Komplexität entstammen [Bre97]. Sie sind laut derselben Quelle durch einen Vergleich der Größen der Mengen noch optimierbar.

- Typprüfung bei der Ein- und Ausgabe (Ersetzung von printf durch Streams).
- Erweiterte Einsatzmöglichkeiten für das Schlüsselwort const (Erleichterung der Fehlersuche).

Die bestehende Umgebung ist im älteren C++ Dialekt geschrieben, der sich mit dem neueren ISO96 Dialekt nicht beliebig mischen läßt (insbesondere stdio.h und fstream.h). Jede der hinzugekommenen Klassen ist daher in einem einheitlichen Dialekt geschrieben: DCorrespondenceSet im ISO96 Dialekt, die anderen Klassen in der früheren Sprachvariante. Dies allein genügt noch nicht, da über die Headerdateien beide Sprachvarianten für den Compiler während desselben Laufs sichtbar sind. Eine zusätzliche Trennung der Dialekte erfolgt zum einen mit Hilfe der abstrakten Basisklasse DPath. Die Headerdatei DPath.h enthält keinerlei Datenstrukturen, sondern nur Methodendeklarationen, deren Parameter ausschließlich Zeichenketten (char *) und andere DPath Objekte sind (siehe Seite 104). Zum anderen wurden die Datenstrukturen in DCorrespondenceSet.h als void Pointer definiert (siehe Seite 103). Bei dieser Alternative entfällt eine zusätzliche abstrakte Basisklasse, dafür muß gecastet werden. Die Bibliotheksfunktionen, die es früher nicht gab oder die nicht funktionierten, verkürzen die Entwicklungszeit sehr. Auch die Qualität erhöht sich durch Verwendung vorgefertigter, erprobter Komponenten.

Die Baumstrukturen ermöglichen Einfügen, Auffinden und Löschen in logarithmischer Zeit. Bei der hier vorliegenden Problematik ändern sich die Schlüssel häufig (siehe Folgeabschnitte). Ein direktes Ändern der Schlüssel

würde die Sortierfolge zerstören und wird deshalb durch eine Lösch- und eine Einfügeoperation ersetzt.

Eine Alternative zu Bäumen ist das Hashing. Wegen der häufigen Löschungen sind jedoch für diese Anwendung die Baumcontainer besser geeignet.
Bem.: Die Spezifikation der Hashtable ist zudem nicht rechtzeitig fertig geworden, um in den ISO96 Standard aufgenommen zu werden, so daß C++ zur Zeit keine Standard Hashtable zur Verfügung stellt.

Insgesamt sind Performanceoptimierungen beim SIM Algorithmus unwichtig, da sehr viele Dialoge mit dem Benutzer vorkommen (wegen des häufigen Falls der mehrdeutigen Schemaerweiterung) und die Antwortzeiten so kurz sind, daß sie nicht wahrgenommen werden. Dagegen verkürzt SIM den Gesamtprozeß der Schemaintegration aufgrund einer kontrollierten, mit Fehlerprüfungen versehenen, Vorgehensweise.

10.5 Datenstrukturen

Pfade bestehen im wesentlichen aus zwei Listen für die Knoten (Entities oder Datentypen) und Attribute, welche der Pfad durchläuft. Die Listen des bestehenden Systems wurden hierfür wiederverwendet. Bis auf den letzten müssen alle Einträge der Knotenliste des Pfads Entities sein. Der letzte Knoten darf auch ein anderer Datentyp (z.B. STRING) sein. Normale Pfadschlüssel dienen dazu, einen Pfad wiederzufinden. Sie ergeben sich aus der Verkettung der Namen der ersten beiden (Entity- bzw. Datentyp-) Knoten des Pfads. Genauso ergibt sich der inverse Pfadschlüssel aus der Verkettung der Namen des letzten und des vorletzten Knoten der Liste. Er wird nur verwendet, wenn der letzte Knoten ein Entity ist. Unterschiedliche Pfade haben den gleichen Schlüssel, wenn sie in der gleichen Richtung starten und den gleichen inversen Schlüssel, wenn sie mit derselben Kante enden.

Für Pfadkorrespondenzen gibt es keine eigene Klasse. Stattdessen verwaltet die Klasse DCorrespondenceSet Mengen von Pfadkorrespondenzen. Hierfür bedient sie sich der oben beschriebenen vordefinierten Containerklassen set, map und multimap. Jedes Schema besitzt eine PathMultimap, in der alle durch dieses Schema verlaufenden Pfade gespeichert sind (siehe Abb. 19). Die PathMultimap ist eine Template-Instanz der multimap, die Pfadschlüssel auf Pfade abbildet. Ein Pfad wird immer unter dem normalen Pfadschlüssel eingetragen. Wenn der inverse Pfadschlüssel existiert, wird derselbe Pfad nochmals unter diesem Namen eingetragen. Um ein einfaches Löschen zu ermöglichen, speichert jeder Pfad zwei Iteratoren (Zeiger) auf die beiden Einträge in der Multimap. (Dies funktioniert mit Baumstrukturen, da sich deren

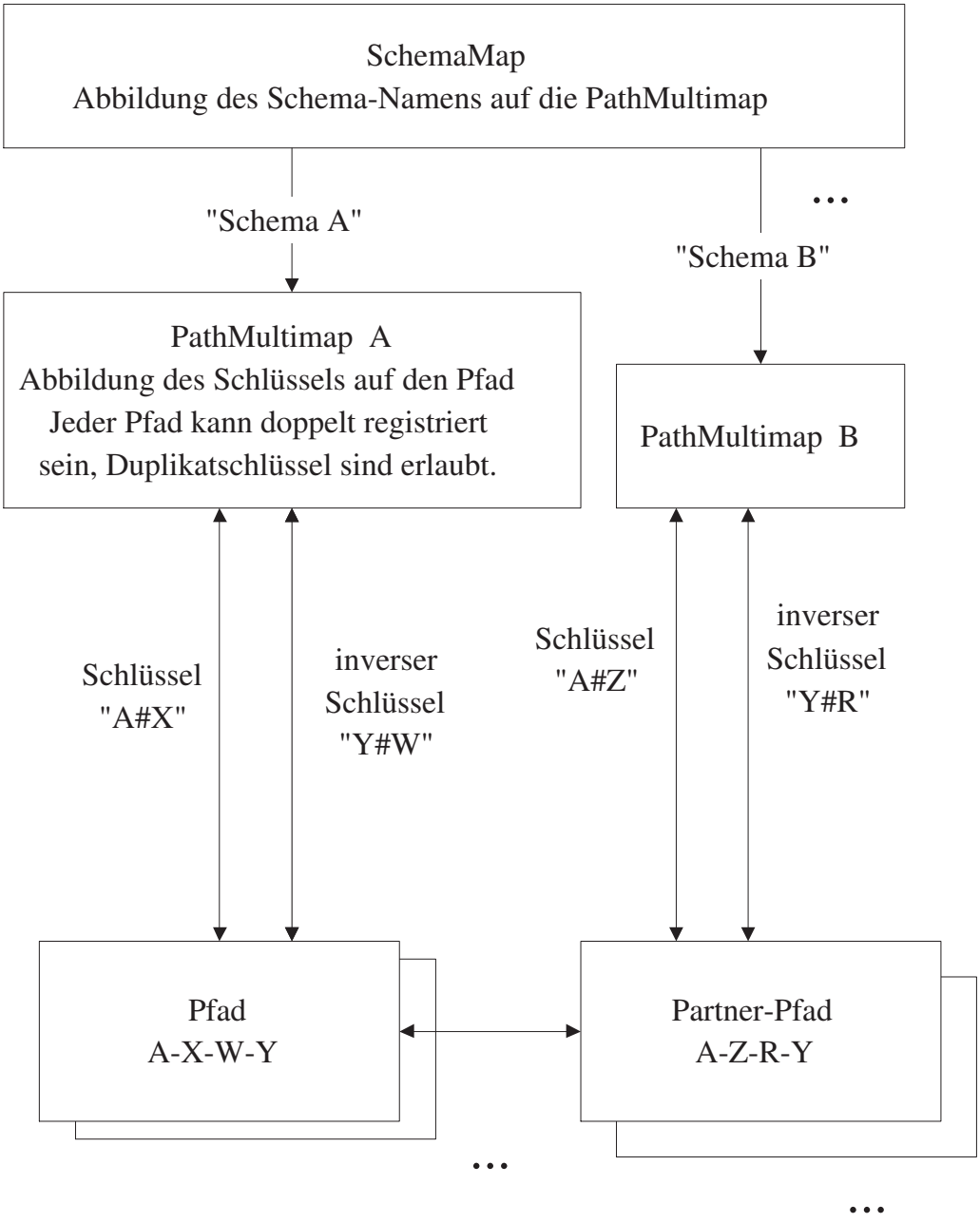

Abbildung 19: Datenstrukturen

Elemente beim Löschen im Gegensatz zu Vektoren nicht verschieben.)

Die beiden Pfade einer Korrespondenz liegen in unterschiedlichen Multimaps und sind gegenseitig direkt verbunden. Man kann so ohne Suchaufwand direkt

vom Pfad zum Partnerpfad und zurück gelangen.

Um zu einem Schema die zugehörige PathMultimap zu finden, besitzt die Klasse DCorrespondenceSet eine SchemaMap, die Template Instanz der C++ map ist und Schemanamen auf PathMultimaps abbildet.

Die dritte verwendete Datenstruktur der Standardbibliothek sind Mengen (set). Mengen werden zum einen bei der Gültigkeitsprüfung für Pfadkorrespondenzen eingesetzt. Die Namen der Knoten eines Pfads und die Namen aller Entities im Schema des Partnerpfads werden hierfür in Template Instanzen von C++ sets gespeichert. Der Standardalgorithmus "set_intersection" prüft dann die Gültigkeit in Form einer leeren Schnittmenge.

Zum anderen kommen Pfadmengen bei der Entscheidung über die nächste Schemaerweiterung zum Einsatz. Hier hilft der Standardalgorithmus "includes".

Wenn die Pfadkorrespondenzen widersprüchlich sind, dient der Algorithmus "set_difference" dazu, Kandidaten für Streichungen zu ermitteln.

10.6 Beschreibung des Programmablaufs

Die Hauptschritte des Verfahrens werden hier grob beschrieben.

- Zuerst wird der bestehende EXPRESS-Parser aufgerufen, um die Eingabeschemata aus einer Datei in das interne Format im Hauptspeicher zu transferieren.

- Im zweiten Schritt wird der Augmenter gestartet. Er liest Pfadkorrespondenzen aus einer Datei ein. Hierbei bedient er sich des bestehenden Scanners, um Einheitlichkeit bei der Namenserkennung zu garantieren, Blanks und Zeilenvorschübe zu entfernen und EXPRESS-Kommentare verwenden zu können.
  Syntax der Pfadkorrespondenz-Datei (regulär):

```
Start              ::= Pfadkorrespondenz*
Pfadkorrespondenz  ::= "(" Pfad "," Pfad ")"
Pfad               ::= Schema-Name ":"
                         Entity-Name Attribut-Name+
Schema-Name        ::= SIMPLE_ID
Entity-Name        ::= SIMPLE_ID
Attribut-Name      ::= SIMPLE_ID
```

  SIMPLE_ID wird als Token vom bestehenden Scanner geliefert. Die angegebenen Attribute dürfen explizit (normal) oder invers sein.

Beispiel für eine Pfadkorrespondenz:
`(schema1: A refb refc refy, schema2: A refw refy)`

Erklärung: Der erste Pfad beginnt in "schema1" mit dem Entity "A". Dieses Entity besitzt das Referenzattribut "refb", welches auf ein weiteres, hier nicht angegebenes Entity verweist. Dieses neu erreichte Entity besitzt ein Referenzattribut "refc" zu einem dritten Entity, welches ein Referenz- oder Wertattribut "refy" hat. (Analog für "schema2")

In CAugmentExpress wird die reguläre Syntax der Korrespondenzdatei mit einem handgeschriebenen Zustandsautomaten überprüft, der die Token wiederum vom EXPRESS-Scanner bezieht. Sobald die erste Beziehung eines Pfades eingelesen wurde, wird ein DPathImpl-Objekt erzeugt und mit Hilfe der Methode DPath.initialize() verifiziert, ob der Pfadanfang tatsächlich wie angegeben durch sein Schema verläuft. Die Methode DPath.append() verlängert den Pfad und kontrolliert die Richtigkeit der Angaben. Wenn Pfad und Partner-Pfad einer Korrespondenz erfolgreich eingelesen wurden, wird das Paar der Klasse DCorrespondenceSet übergeben. Die Methode addCorrespondence() kontrolliert, ob die Anfangs-Entities und End-Typen korrespondieren und daß kein Zwischen-Entity mit einem Entity des Partnerschema (Schema des Partner-Pfad) korrespondiert. Wenn alle Korrespondenzen erfolgreich übergeben wurden, wird die Methode DCorrespondenceSet.generateAugmentations() gestartet, um den SIM-Algorithmus auszuführen. Die sich ergebenden Schemaerweiterungen werden über die Methode DPath.augmentSchema() ausgeführt.

- Im dritten Schritt werden alle EXPRESS-Schemata mit Hilfe des Mergers vereinigt.

- Im vierten Schritt wird das Ergebnis-Schema unter Verwendung der bestehenden Umgebung aus dem Hauptspeicher in eine Ausgabedatei geschrieben.

Die folgenden Abschnitte stellen die Implementierung der Kernbestandteile des SIM Algorithmus mit den beschriebenen Datenstrukturen dar.

10.7 Korrektheitsprüfungen Teil 1

Dieser Abschnitt beschreibt die Prüfungen einzelner Pfade und Korrespondenzen, wie in der Originalarbeit beschrieben. Erweiterte Prüfungen folgen im übernächsten Abschnitt, da die Kenntnis des nächsten Abschnitts dafür nötig

ist. Die beschriebenen Korrektheitstests müssen einmal am Anfang durchgeführt werden. Bei einem Erweiterungsschritt, der im nächsten Abschnitt genauer dargestellt wird, bleiben sie gültig und müssen nicht wiederholt werden. Warum sie gültig bleiben, wird im übernächsten Abschnitt diskutiert. Die Prüfung einer Menge von Pfadkorrespondenzen auf Überlappung muß bei jedem Erweiterungsschritt erneut stattfinden und gehört deshalb nicht zu den hier beschriebenen anfänglichen Prüfungen.

Zuerst wird die Korrektheit der Pfade geprüft. Hierzu wird der in der Korrespondenzdatei angegebene Verlauf jedes Pfads durch sein Schema nachverfolgt. Der Aufwand ist proportional zur Gesamtlänge L_g aller Pfade, multipliziert mit dem Aufwand zum Finden von Entities im Schema und Attributen in Entities. Da letztere in doppelt verketteten Listen gespeichert sind, steigt die Suchdauer linear mit deren Anzahl. E sei die Maximalzahl von Entities in einem Schema. A sei die Maximalzahl von Attributen in einem Entity. Die Komplexität ist dann $O(L_g \times (E + A))$. Die weiteren Prüfungen betreffen Korrespondenzen, die aus zwei korrekt eingelesenen Pfaden gebildet werden sollen. Die folgenden Prüfungen sind auszuführen:

- Bedingung (E1) aus Kapitel 7 wird im Rahmen der Syntaxprüfung für die Korrespondenzdatei verifiziert. Korrespondenzen dürfen nur paarweise, d.h. Eins zu Eins angegeben werden. Der Aufwand ist proportional zur Größe der Korrespondenzdatei.

  Zusätzlich sollte geprüft werden, ob ein neuer Pfad im Schema bereits vorkommt, d.h. doppelt definiert wurde. Für den Vergleich muß für jedes Paar von Pfaden zweimal der includes Algorithmus auf die Entity Mengen der Pfade angewendet werden. L sei die Maximallänge eines Pfads, M die Maximalzahl der Pfade in einem Schema. Die Komplexität ist dann für jeden includes Aufruf $O(L \log(L))$, insgesamt also $O(M^2 L \log(L))$.

- Die Anfangs- und Endknoten der Pfade müssen korrespondieren. Hierzu werden die Namen verglichen (Bedingungen (E2) und (E3) aus Kapitel 7). Der Aufwand ist proportional zur Zahl M der Pfade.

- Anschließend erfolgt die Prüfung der Bedingung (E4+) aus Kapitel 7. Hierzu werden für jede hinzukommende Pfadkorrespondenz die Zwischenknoten des ersten Pfads in einem set der Standardbibliothek, die Entities des Schema des zweiten Pfads in einem zweiten set gespeichert. Der Standardalgorithmus set_intersection muß die leere Menge liefern. L_g sei die Gesamtlänge aller Pfade, L die Maximallänge eines Pfads

und E die Maximalzahl von Entities in einem Schema. Die Komplexität ist dann $O(L_g \log(E) + L_g \log(L) + E \log(E))$. Der erste Summand ist der Gesamtaufwand der Schnittmengenbildungen, der zweite Summand der Aufwand für das Speichern der Pfadknoten in sets und der letzte Summand die Zeit für das Speichern der Entities in sets.

- Um (E5) zu prüfen, muß jeder neu hinzukommende Pfad mit allen bisher gespeicherten Pfaden seines Schemas geschnitten werden, d.h. es ist jeweils die Schnittmenge der Entitymengen der Pfade zu bilden. Die gleichen Operationen sind mit den korrespondierenden Pfaden durchzuführen. Das Ergebnis (leere Menge oder nicht) muß jedesmal dasselbe sein. Die Prüfung kann aufgrund der Erkenntnisse des übernächsten Abschnitts 10.9 entfallen. Die Komplexität wäre wie bei der obigen Prüfung von (E1).

Die gesamte Komplexität aller Tests ist
$O(L_g \times (E + A + \log(L)) + M^2 \times L \log(L) + E \log(E))$

10.8 Ablauf des Erweiterungsschritts

Dieser Abschnitt stellt die Implementierung des in Kapitel 7 auf Seite 58 beschriebenen Erweiterungsschritts des SIM Algorithmus dar. Ausgangspunkt ist ein manuell oder automatisch gewählter Eintrag einer PathMultimap M_1. Die genaue Methode der Auswahl wird im nächsten Abschnitt beschrieben. Der Eintrag enthalte einen Pfad namens Pf_1, der mit einem Entity namens $Ent1_1$ beginne. (Erstes Entity im Pfad, erstes Schema.) Die Größe der Multimap sei $|M_1|$. Das zu $Ent1_1$ korrespondierende Entity sei $Ent1_2$. (Erstes Entity im Pfad, zweites Schema.)

Zunächst wird mit Hilfe der Bibliotheksfunktion find() die Partition P_1 von Pf_1 bestimmt, deren Größe $|P_1|$ sei. Aufgrund der Sortierung ist der Aufwand $O(\log(|M_1|) + |P_1|)$. Das Ergebnis wird als temporäre Menge (C++ set) mit dem Aufwand $O(|P_1| \log(|P_1|))$ gespeichert.

Ausgehend von P_1 folgen Tests auf Inklusion oder Überlappung mit allen Partitionen der Partnerpfade zu den Pfaden aus P_1. Für jeden Pfad aus P_1 wird dazu mit Hilfe der direkten gegenseitigen Verweise korrespondierender Pfade der Partnerpfad Pf_2 bestimmt. Die Partition von Pf_2 sei P_2 mit der Größe $|P_2|$. Sie wird mit dem Aufwand $O(\log(|M_2|) + |P_2|)$ berechnet, wobei $|M_2|$ die Größe der Multimap des Partnerschemas ist. Statt der sich für P_2 ergebenden Pfade werden jeweils (pro P_2) deren korrespondierende Pfade aus dem ersten Schema in einer zweiten temporären Menge gespeichert.

Um die Mengenbeziehung zwischen P_1 und der gerade betrachteten Partition P_2 festzustellen, wird zweimal der Algorithmus "includes" aufgerufen. Als Parameter werden die temporären Mengen übergeben. Der Aufwand hierfür ist $O(|P_1| \times \log(|P_2|) + |P_2| \times \log(|P_1|))$. Insgesamt sind $|P_1|$ solcher Vergleiche durchzuführen. Wenn M und P Maximalgrößen einer Multimap und einer Partition sind, ergibt sich als grobe Obergrenze für diesen Schritt $O(P^2 \times \log(P))$.

Wenn keine der Partnerpartitionen P_2 mit P_1 überlappt, ist der Vergleich mit der ersten Partnerpartition ausschlaggebend.

Es gibt vier Fälle:

1. P_2 ist echt in P_1 enthalten.
2. P_1 ist echt in P_2 enthalten.
3. P_1 und P_2 sind gleich und P_1 enthält keinen kurzen Pfad.
4. P_1 und P_2 sind gleich und P_1 besteht aus kurzen Pfaden.

Im zweiten und vierten Fall werden die Rollen von P_1 und P_2 vertauscht.

Es kann so immer gewährleistet werden, daß P_2 in P_1 enthalten ist und P_1 keinen kurzen Pfad enthält. Die Begründung ist wie folgt:

In Fall 1 und Fall 2 wird eine Partition gewählt, die mehr als einen Pfad enthält. Solche Partitionen enthalten keinen kurzen Pfad. In Kapitel 7 auf Seite 58 wurde bereits erklärt, warum entweder alle Pfade einer Partition kurz sind oder kein Pfad kurz ist. Zudem folgt aus der Eins zu Eins Eigenschaft für Korrespondenzen (E1), daß eine Partition mit kurzen Pfaden nur aus einem einzelnen Pfad bestehen kann, da mehrere kurze Pfade immer der gleiche Pfad wären, was (E1) widersprechen würde.

In Fall 4 kann P_2 keinen kurzen Pfad enthalten. Andernfalls bestünden P_1 und P_2 nach der gleichen Überlegung wie im letzten Absatz aus einem einzelnen kurzen Pfad. Dann wäre jedoch die Korrespondenz bereits gelöst und die Pfade wären aus den Multimaps gelöscht.

Es ist also sicher, daß nach einer eventuellen Vertauschung der Rollen P_1 keinen kurzen Pfad enthält. Der zweite Knoten jedes Pfades von P_1 ist daher ein Zwischenknoten, und jeweils das gleiche Entity, hier $Ent2_1$ genannt (zweites Entity der Pfade im ersten Schema), da die Pfade einer Partition in der gleichen Richtung starten.

Aus der Bedingung (E4+) auf Seite 54 folgt, daß $Ent2_1$ im anderen Schema noch nicht vorkommt und daher dort als neues Entity eingebaut werden kann,

hier $Ent2_2$ genannt (neues zweites Entity im zweiten Schema), so daß es mit $Ent2_1$ korrespondiert.

Wenn alle Partnerpfade der Pfade aus P_1 in der gleichen Partition liegen, wie es bei sich gegenseitig enthaltenden, gleichgroßen Partitionen der Fall ist, ist auch der zweite Knoten der Partnerpfade immer der gleiche. In diesem Fall besteht die Option, $Ent2_2$ als Linkklasse zwischen dem ersten und dem zweiten Entity der Partnerpfade einzubauen. Hierzu wird die bestehende Beziehung zwischen den beiden Entities im Partnerschema gelöst und die Entities werden mit neuen Beziehungen zu $Ent2_2$ entsprechend Abbildung 13 auf Seite 52 versehen.

Wird diese Option abgelehnt, oder verteilen sich die Partnerpfade auf unterschiedliche Partitionen, wird eine Entity Erweiterung von $Ent1_2$ im Partnerschema um $Ent2_2$ gemacht. $Ent2_2$ wird mit einer neuen 1:1 Beziehung mit $Ent1_2$ verbunden. Es kann jetzt mehrere zweite Entities im Partnerschema geben. Deren Beziehungen zu $Ent1_2$ werden gelöst und stattdessen mit $Ent2_2$ verbunden.

Die Anzahl dieser Aktionen (Entity einbauen und Beziehungen anpassen) ist $O(|P_1|)$.

Im ersten Schema verkürzen sich alle Pfade der Partition P_1. Sie beginnen jetzt alle mit $Ent2_1$.

Die Partnerpfade dieser Pfade werden derart angepaßt, daß sie mit $Ent2_2$ beginnen oder enden.

Nachdem alle beteiligten Pfade nun mit einem anderen Entity als bisher, nämlich $Ent2_1$ oder $Ent2_2$ enden, ändern sich die Schlüssel der betroffenen Pfade. Um die Sortierfolge zu erhalten, werden die geänderten Einträge aus den PathMultimaps entfernt und neu eingefügt. Der Aufwand hierfür ist $O(|P_1| \times \log(|P_1|))$.

Die Maximalzahl von Erweiterungen ist L_g, d.h. die Gesamtlänge aller Pfade, wenn jedesmal nur ein Pfad gekürzt wird. Die gesamte Komplexität für die Durchführung aller Erweiterungen ist dann
$O(L_g \times (\log(M) + P^2 \times \log(P))$.

Der Aufwand für die Tests des letzten Abschnitts war
$O(L_g \times (E + A + \log(L)) + M^2 \times L \log(L) + E \log(E))$

Insgesamt ergibt sich
$O(L_g \times (E + A + \log(M) + \log(L) + P^2 \times \log(P)) + M^2 \times L \log(L) + E \log(E))$

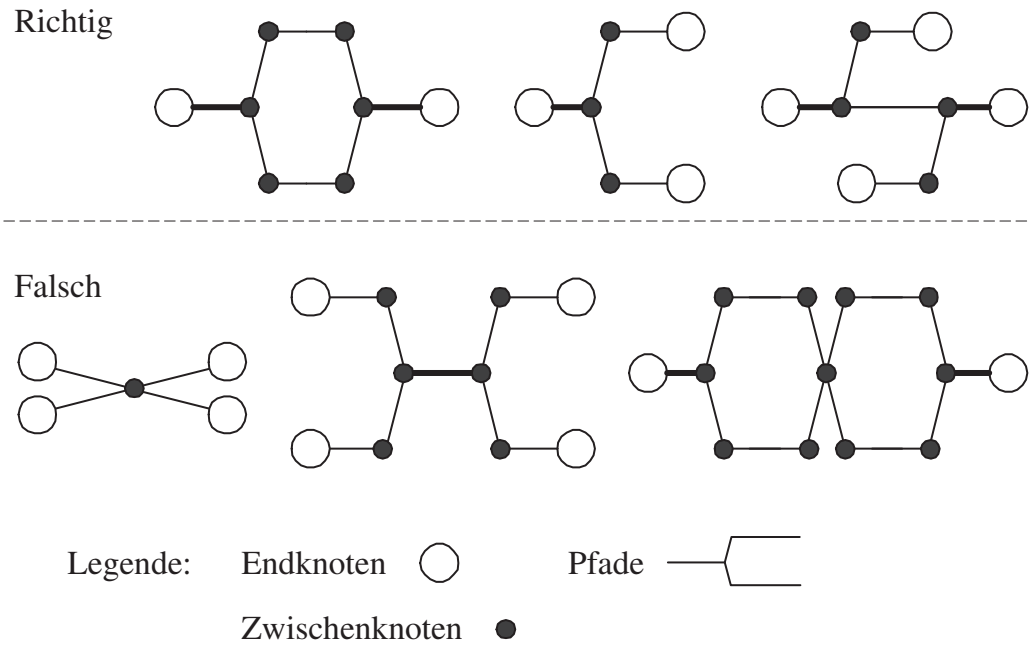

Abbildung 20: Mögliche Pfadverläufe im selben Schema

10.9 Korrektheitsprüfungen Teil 2

Für die Korrektheit des Algorithmus ist die wesentliche Frage, ob die Bedingungen (E1) - (E5) aus Kapitel 7 (Seite 54), die anfangs einmal geprüft werden (siehe Abschnitt 10.7), bei jedem Erweiterungsschritt erhalten bleiben. Es zeigt sich, daß Änderungen an den Bedingungen nötig sind. Jedoch ändert sich nichts Grundlegendes an den Ideen, die dem SIM Algorithmus zugrundeliegen.

Welche Bedingungen bleiben erhalten? Da $Ent2_1$ mit $Ent2_2$ (siehe letzten Abschnitt) korrespondiert, bleiben die Bedingungen (E2) und (E3) erhalten. Auch (E4) bleibt erhalten, da keine neuen Zwischenklassen entstehen.

Warum bleiben die Bedingungen (E1), (E4+) und (E5) nicht immer erfüllt? Die Ursache liegt darin, daß über den Verlauf der Pfade in *einem* Teilschema wenig festgelegt wird. ((E2), (E3) und (E4) sind Bedingungen zwischen den Schemata.) So können sich Pfade bei einem Zwischenknoten überkreuzen. (E5) verlangt lediglich, daß dann auch die Partnerpfade irgendwo überlappen. Das Problem besteht darin, daß der Zwischenknoten nur einmal im Partnerschema eingebaut werden darf. Ebenso dürfen sich Pfade mehrfach kreuzen, teilweise parallel verlaufen, etc., was zu ähnlichen Schwierigkeiten

führt.

Abbildung 20 zeigt erlaubte und verbotene gemeinsame Verläufe von Pfaden durch ein Schema. Als Unterscheidungskriterium kann die folgende zusätzliche Bedingung identifiziert werden:

E 6 Von jedem Zwischenknoten eines Pfads ausgehend gibt es einen Weg durch das Schema, entlang dem alle Pfade, zu denen der Zwischenknoten gehört, gemeinsam verlaufen, ohne diesen Weg wieder zu verlassen.

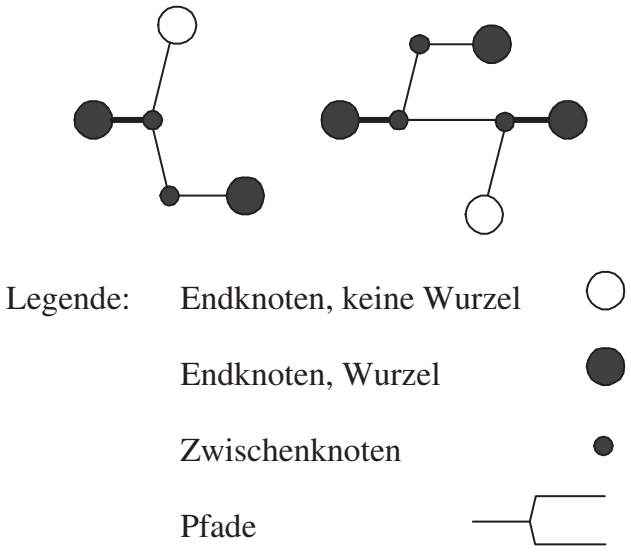

Abbildung 21: Wurzelknoten

Eine Folgerung aus (E2, E3) (Endknoten korrespondieren) und (E4) (Zwischenknoten korrespondieren nicht) ist, daß die gemeinsam verlaufenden Pfade mit dem gleichen Knoten enden müssen. Dieser Knoten wird im folgenden Wurzel genannt.[6] Die von der Wurzel in der gleichen Richtung startenden Pfade werden Wurzelpartition genannt. Abbildung 21 zeigt Beispiele dafür, welche Endknoten auch Wurzel sind. Die Endknoten kurzer Pfade, die keine Zwischenknoten besitzen, werden ebenfalls zu den Wurzeln gezählt.

SIM Erweiterungsschritte dürfen jetzt nur noch Wurzelpartitionen als erste Partition (siehe letzten Abschnitt) verwenden. Die an der Erweiterung teilnehmenden Pfade werden also von dem Ende her bearbeitet, das mit der Wurzel übereinstimmt.

[6]Da es sich um keinen Baum handelt, sollte eine bessere Bezeichnung gefunden werden. “Gemeinsamer Endknoten“ ist etwas lang.

Die insgesamt größte Partition P ist immer eine Wurzelpartition. Andernfalls wäre der zweite Knoten der ausgehenden Pfade ein Zwischenknoten Z. Z muß zu einem weiteren Pfad gehören, der nicht unter den ausgehenden Pfaden ist. Eine zu Z aufgrund (E6) existierende Wurzelpartition wäre dann größer als P (Widerspruch).

Der implementierte Algorithmus wurde daher so geändert, daß für Erweiterungsschritte die größte Partition verwendet wird. Bei mehreren gleichgroßen Partitionen wird eine beliebige Partition automatisch herausgegriffen, hier könnte man auch den Anwender über die Auswahl entscheiden lassen.

Die Komplexität der Suche nach einer maximal großen Partition beträgt $O(M \log(M))$, wenn M die Anzahl der Pfade in der Multimap eines Schemas ist. Dieser "worst case" tritt ein, wenn alle Partitionen die Größe Eins haben. Es ist dann M mal eine Partition in der Zeit $O(\log(M))$ zu ermitteln. Größere Partitionen können in Sortierfolge durchlaufen werden. Dann nimmt die Zahl der Rechenschritte ab. Da beim Aufwand für Korrektheitsprüfungen (siehe Abschnitt 10.7) bereits der Term $O(M^2)$ vorkam, bleibt die Komplexität insgesamt gleich.

Die Prüfung von (E6) wurde aus Aufwandsgründen nicht implementiert. Eine Möglichkeit wäre, alle Partitionen eines Schemas zu durchlaufen. Für jede Partition wird der Weg der Gruppe der ausgehenden Pfade nachverfolgt. Es gibt drei Fälle, die sich in der Art der erreichten Zwischenknoten unterscheiden:

- Neue Pfade stoßen zur Gruppe. In diesem Fall dürfen sich die Pfade der erweiterten Gruppe im restlichen Verlauf nicht mehr trennen.
- Die Gruppe spaltet sich auf. Die Teilstränge müssen getrennt nachverfolgt werden.
- Die Gruppe setzt den Weg ungeändert zum nächsten Knoten fort.

(E6) ist verletzt, wenn die ersten beiden Fälle gleichzeitig bei einem Zwischenknoten zutreffen oder wenn der zweite Fall eintritt, nachdem der erste Fall bereits bei einem Vorgängerknoten zutraf.

Die Nachverfolgung endet, wenn der gemeinsame Endknoten oder ein bereits besuchter Zwischenknoten erreicht wird. Die Komplexität ist dann proportional zur Gesamtlänge L_g aller Pfade.

Es wird nun erklärt, warum (E6), (E1) und (E4+) erhalten bleiben und (E5) aus den anderen Bedingungen folgt.

Warum bleibt (E6) erhalten? In keinem der Teilschemata werden neue Zwischenknoten erzeugt, die neue oder erweiterte Wurzeln erforderlich machen würden. $Ent1_1$ und $Ent1_2$ waren Wurzeln der an der Erweiterung teilnehmenden Pfade. Die neuen Endknoten $Ent2_1$ und $Ent2_2$ können wieder als Wurzeln verwendet werden.

Warum bleibt (E1) erhalten? Da Pfade nur paarweise gelöscht werden (wenn beide kurz sind), bleiben die Pfadkorrespondenzen Eins zu Eins Beziehungen. Zusätzlich muß ausgeschlossen werden können, daß es in derselben Partition zwei gleiche Pfade gibt. (Andernfalls würde drohen, daß eine Partition aus mehreren kurzen Pfaden besteht. Die obige, sichere Auswahl von $Ent2_1$ wäre dann gefährdet, da das zweite Entity bei kurzen Pfaden nicht existiert.) Im zweiten Schema der Erweiterung werden Endknoten nur ausgetauscht, so daß sich Pfade nicht angleichen können. Im ersten Schema verkürzen sich die Pfade. Dennoch können zwei Pfade nicht gleich werden, wie die folgende Überlegung verdeutlicht: Wenn die Pfade im ersten Schema keinen gemeinsamen Zwischenknoten besitzen, werden sie bei Erweiterungen nicht aneinander angeglichen (Injektivität). Wenn sie einen gemeinsamen Zwischenknoten besitzen, werden sie vom Ende der Wurzel her bearbeitet. Wegen (E1) müssen die Pfade vor dem Änderungsschritt teilweise unterschiedlich gewesen sein. Da die Änderungen vom Ende der Wurzel her nur den gemeinsam verlaufenden Teil betreffen, bleiben die getrennt verlaufenden Abschnitte weiterhin verschieden.

Warum bleibt (E4+) erhalten? Die Schwachstelle des ursprünglichen Verfahrens besteht darin, daß der Knoten $Ent2_1$ zum neuen Endknoten wird, aber nicht ausgeschlossen werden kann, daß dies ein Zwischenknoten Z eines anderen Pfades $Pf_{Problem}$ ist. Aufgrund des Erweiterungsschrittes korrespondiert Z jetzt mit $Ent2_2$. (E4+) wird verletzt. Durch die zusätzliche Bedingung (E6) folgt aber nun, daß $Pf_{Problem}$ an der Erweiterung teilnimmt, da er von der Wurzel ausgehend mitbehandelt wird. Im Partnerschema gibt es keine derartige Schwachstelle, da $Ent2_2$ dort neu eingebaut wird. (E4+) bleibt also erhalten.

Warum kann (E5) aus den anderen Bedingungen gefolgert werden? Wenn zwei Pfade in einem Schema einen Zwischenknoten gemeinsam haben, haben sie nach dem Vorhergehenden auch die Wurzel, also einen Endknoten, gemein. Wenn sie einen Endknoten gemeinsam haben, dann haben sie im Partnerschema aufgrund von (E1) und (E2, E3) den korrespondierenden Knoten gemeinsam.

Die Argumentation zu den einzelnen Bedingungen hat gezeigt, daß die Wurzelpartition nur im ersten Schema gefordert wird. Wenn die Partition des

ersten Schemas in der Partnerpartition enthalten ist, müssen nach der bisherigen Beschreibung des SIM Algorithmus die Rollen getauscht werden. Dies kann zu einem Fehler führen, wenn die Partnerpartition keine Wurzelpartition ist. Eine Lösung wäre, die insgesamt größte Partition aller Teilschemata zu verwenden. In der Hauptschleife des Algorithmus gäbe es dann statt der vier nur noch drei Fälle (Partnerpartition enthalten, Gleichmächtigkeit, Überlappung).

Bei der Spezialisierung auf hierarchisch strukturierte Teilschemata wie XML ergibt sich eine Vereinfachung. Die Entsprechung zu (E6) ist hier, daß Pfade bei einem beliebigen Knoten starten und dann nur noch in Richtung der Wurzel des XML Baumes verlaufen dürfen.

10.10 Testbeispiel

Zum Test der Implementierung wurden die aktuellen C++ Compiler der Hersteller Borland (C++ Builder 5) und Microsoft (Visual C++ 6) verwendet.

Das folgende Anwendungsbeispiel zeigt die Integration zweier Schemata, die analog zu Abb. 14 in Kapitel 7 auf Seite 56 aufgebaut sind:

Die beiden Eingabeschemata

```
SCHEMA schema1;

ENTITY A;
 refb: SET [5:8] OF B;
 refz: Z;
END_ENTITY;

ENTITY B;
 refc: C;
 refx: X;
 INVERSE
 refb: SET [3:4] OF A FOR refb;
END_ENTITY;

ENTITY C;
 refy: Y;
 INVERSE
 refc: SET [0:1] OF B FOR refc;
END_ENTITY;

ENTITY X;
 INVERSE
 refx: SET [0:1] OF B FOR refx;
END_ENTITY;

ENTITY Y;
 val: STRING;
 INVERSE
 refy: SET [0:1] OF C FOR refy;
END_ENTITY;
```

```
ENTITY Z;
 INVERSE
 refz: SET [0:1] OF A FOR refz;
END_ENTITY;

END_SCHEMA;


SCHEMA schema2;

ENTITY A;
 ref: V;
 refw: W;
END_ENTITY;

ENTITY V;
 refvx: X;
 refz: Z;
 INVERSE
 ref: SET [0:1] OF A FOR ref;
END_ENTITY;

ENTITY W;
 refy: Y;
 INVERSE
 refw: SET [0:1] OF A FOR refw;
END_ENTITY;

ENTITY X;
 val: STRING;
 INVERSE
 refvx: SET [0:1] OF V FOR refvx;
END_ENTITY;

ENTITY Y;
 val: STRING;
 INVERSE
 refy: SET [0:1] OF W FOR refy;
END_ENTITY;

ENTITY Z;
```

```
 INVERSE
 refz: SET [0:1] OF V FOR refz;
END_ENTITY;

END_SCHEMA;
```

Die Datei mit Pfadkorrespondenzen

Die Korrespondenzen entsprechen dem Beispiel aus Kapitel 7 auf Seite 55. Ein konfliktfreier Durchlauf ist möglich, da die dritte, widersprechende Korrespondenz auskommentiert ist.

```
(*_____________________________________________________________

 Pfadkorrespondenzen
 Syntax:
 Start::=  Pfadkorrespondenz*
 Pfadkorrespondenz ::= "(" Pfad "," Pfad ")"
 Pfad ::=  Schema-Name ":" Entity-Name Attribut-Name+
  ____________________________________________________________*)

(schema1: A refb refx, schema2: A ref refvx)

(schema1: A refb refc refy, schema2: A refw refy)

(*schema1: Z refz, schema2: Z refz ref*)
```

Die neuen Aufrufoptionen

Das Hauptprogramm wurde um drei neue Optionen erweitert. Mit der Option -a wird die Datei der Pfadkorrespondenzen bestimmt und die Erweiterungsstufe in den Ablauf eingeschlossen. Die Option -m bewirkt die Ausführung der Merge Stufe. Die Angabe -v erweitert die Ausgabe um einige, für die Fehlersuche nützliche, Details. Ruft man das Programm ohne Parameter auf, ist die Ausgabe wie folgt.

```
C:\Merge\Borland\bin>import

SDAI+ Generation -- EXPRESS+ Importer/Exporter/Compiler
(C) 1997, 1998 FernUniversitaet Hagen
```

```
Usage: import [flags] [srcfile]
Flags:
 -k: keep temporary files (for debugging) - currently always on!
 -e: extract pure EXPRESS part from input file
 -p: extract EXPRESS+ part from input file (= copy input file)
 -t: extract EXPRESS+ part from input file, with all the
      templates and relationships expanded
 -c: compile to C++
 -i: generate improved default PI
 -d: generate EXPRESS-C
 -a<korr_file>: augment all SCHEMAs using correspondences from <korr_file>
 -m: merge all SCHEMAs from input file into one pure EXPRESS SCHEMA
 -v: more verbose output
press [return]...
```

Der Dialog mit dem Integrator

Der Dialog wird erst komplett wiedergegeben und anschließend erklärt.

```
C:\Merge\Borland\bin>import -aSchema1.pfk -m -e Schema1.exp

SDAI+ Generation -- EXPRESS+ Importer/Exporter/Compiler
(C) 1997, 1998 FernUniversitaet Hagen


PASS 1: Parsing source...
Schema 'schema1'
End Schema 'schema1'
Schema 'schema2'
End Schema 'schema2'

Syntax ok!

PASS 1: Finished!
PASS 2: Augmenting the schemas ...

Automatische Entity-Erweiterung
Was soll man tun ?
1) Entity W aus Schema schema2 bei Entity B in Schema schema1
```

```
 einbauen ?
2) Entity C aus Schema schema1 bei Entity B in Schema schema2
 einbauen ?
1
Welche Erweiterungsart soll, die Kardinalitaeten betreffend,
durchgefuehrt werden ?
1) Entity B aus Schema schema1 um Entity W aus Schema schema2
 erweitern ?
2) Beziehung B<->C aus schema1 mit Entity W aus Schema schema2
 identifizieren ?
2
Welche Erweiterungsart soll, die Kardinalitaeten betreffend,
durchgefuehrt werden ?
1) Entity B aus Schema schema1 um Entity V aus Schema schema2
 erweitern ?
2) Beziehung B<->X aus schema1 mit Entity V aus Schema schema2
 identifizieren ?
2
Welche Erweiterungsart soll, die Kardinalitaeten betreffend,
durchgefuehrt werden ?
1) Entity W aus Schema schema2 um Entity C aus Schema schema1
 erweitern ?
2) Beziehung W<->Y aus schema2 mit Entity C aus Schema schema1
 identifizieren ?
1

PASS 2: Finished!

PASS 3: Merging the Schemas ...
FEHLER beim Merge Vorgang. Bitte lesen Sie die Datei prot_merge.txt
PASS 3: FEHLER!

PASS 4: Generating pure EXPRESS...

generating file Schema1.pure.exp...
        done
press [return]...
```

Erklärung des Ablaufs:
Nach Prüfung der Pfade und Einzelprüfung der Korrespondenzen werden die Pfade in PathMultimaps gespeichert, die hier für das Beispiel wiedergegeben

sind. Die Zeilen sind wie folgt aufgebaut:
`Schlüssel ( Pfad )`, wobei der Pfad so dargestellt wird:
`[Start-Entity]-Attribut-[Zwischen-Entity 1]- ... -[End-Entity]`.
Die Pfade, die jeweils in der gleichen Zeile der Multimap stehen, korrespondieren. (Sie sind intern verkettet, dies wird hier durch die gleiche Zeile nur veranschaulicht.)

```
PathMultimap von Schema1:
A#B ( [A]-refb-[B]-refx-[X] )
A#B ( [A]-refb-[B]-refc-[C]-refy-[Y] )
X#B ( [A]-refb-[B]-refx-[X] )
Y#C ( [A]-refb-[B]-refc-[C]-refy-[Y] )

PathMultimap von Schema2:
A#V ( [A]-ref-[V]-refvx-[X] )
A#W ( [A]-refw-[W]-refy-[Y] )
X#V ( [A]-ref-[V]-refvx-[X] )
Y#W ( [A]-refw-[W]-refy-[Y] )
```

Das Programm beginnt damit, einen Pfad zu wählen, der zu einer Partition maximaler Größe gehört.
Der gewählte Pfad ist (`A#B ( [A]-refb-[B]-refx-[X] )`),
denn `A#B` kommt zweimal vor und ist die größte Partition.
Der Partnerpfad ist (`A#V ( [A]-ref-[V]-refvx-[X] )`).
Die Partnerpartition ist echt enthalten. (Zweimal `A#B` gegen einmal `A#V`.)
Das Partnerschema wird also automatisch erweitert, ohne daß das Programm unterbricht. Das Ergebnis ist wie nach Schritt 1 aus Abb. 14 auf Seite 56. In der ersten Multimap haben sich beide Pfade der Partition `A#B` verkürzt und die Multimaps sehen jetzt wie folgt aus.

```
PathMultimap von Schema1:
B#X ( [B]-refx-[X] ) SHORT
B#C ( [B]-refc-[C]-refy-[Y] )
X#B ( [B]-refx-[X] ) SHORT
Y#C ( [B]-refc-[C]-refy-[Y] )

PathMultimap von Schema2:
B#V ( [B]-ref-[V]-refvx-[X] )
B#W ( [B]-refw-[W]-refy-[Y] )
X#V ( [B]-ref-[V]-refvx-[X] )
Y#W ( [B]-refw-[W]-refy-[Y] )
```

Jetzt haben alle Partitionen nur noch die Größe Eins. Es kann also mit einer beliebigen Partition fortgesetzt werden. Schritt 2 weicht aufgrund dieser willkürlichen Auswahl vom Beispiel in Kapitel 7 ab. Es werden nun, ausgehend vom Entity B, die Partitionen `B#C` und `B#W` verglichen. Diese sind gleich groß und enthalten sich gegenseitig. Da beide Pfade noch über zwei Beziehungen verlaufen, also noch nicht kurz sind, hat der Integrator eine strukturelle Entscheidung zu treffen. Im obigen Ablaufbeispiel wurde entschieden, Schema 1 um das Entity "W" zu erweitern. Außerdem hat er die Möglichkeit, das neue Entity "W" als Linkklasse einzubauen, die angenommen wird. Nach dem zweiten Schritt haben sich die PathMultimaps wiederum geändert:

```
PathMultimap von Schema1:
B#X ( [B]-refx-[X] ) SHORT
W#C ( [W]-refc-[C]-refy-[Y] )
X#B ( [B]-refx-[X] ) SHORT
Y#C ( [W]-refc-[C]-refy-[Y] )

PathMultimap von Schema2:
B#V ( [B]-ref-[V]-refvx-[X] )
W#Y ( [W]-refy-[Y] ) SHORT
X#V ( [B]-ref-[V]-refvx-[X] )
Y#W ( [W]-refy-[Y] ) SHORT
```

Auch die weiteren beiden Schritte sind mehrdeutige Erweiterungen wie in Schritt 2. Jedoch besteht jetzt immer eine der beteiligten Partitionen aus kurzen Pfaden, so daß keine manuellen Strukturentscheidungen (für die Anordnung von Entities und Attributen ausschlaggebende Benutzereingaben) mehr erforderlich sind. Lediglich nach den Kardinalitäten (Link-Klasse) wird jeweils noch gefragt. Der nächste Multimap Zustand ist wie folgt.

```
PathMultimap von Schema1:
W#C ( [W]-refc-[C]-refy-[Y] )
Y#C ( [W]-refc-[C]-refy-[Y] )

PathMultimap von Schema2:
W#Y ( [W]-refy-[Y] ) SHORT
Y#W ( [W]-refy-[Y] ) SHORT
```

Nach dem letzten Schritt sind die Multimaps leer, und der Algorithmus terminiert.

Insgesamt wurden drei Strukturentscheidungen automatisch getroffen, einmal aufgrund Mengeninklusion und zweimal aufgrund kurzer Pfade. Eine Strukturentscheidung und drei Kardinalitätsentscheidungen mußten vom Integrator vorgegeben werden. Die Häufigkeit der Unterbrechungen für Kardinalitätsentscheidungen würde sich noch erhöhen, wenn man den zu erweitenden Knoten bei jedem Hauptschritt mit mehreren gleichgroßen Partitionen vom Integrator auswählen ließe, anstatt wie hier automatisch ein Entity zu wählen. Man hätte dann z.B. im letzten Schritt neben der Erweiterung von W um C und der Verwendung von C als Linkklasse die dritte Alternative, Y um C zu erweitern, wodurch sich nochmals andere Kardinalitäten ergäben.

Der Fehler beim Merge Vorgang kann durch Ansicht des folgenden Protokolls der einzelnen Aktionen ermittelt werden.

Die Protokolldatei

Die Protokolldatei untergliedert jeden Erweiterungsschritt in seine elementaren Aktionen und zeigt auch die Transformationen der Merge Stufe genauer an.

```
Schema 'schema2' wird erweitert.
       Einbau des Entity 'B' aus Schema 'schema1' (ohne seine Beziehungen)
       zwischen Entity 'A' und Entity 'V'.
       Attributentfernung aus Entity 'A':  ref :   V;
       Attributerzeugung für Entity 'B':  ref :   V;
       Erstellung einer neuen Beziehung zwischen Entity 'A' und Entity 'B'.
              Hin-Attribut:  refb :   B FOR refb;
              Rück-Attribut:  refb :   A;
       Anpassung der Domäne eines Attributs von Entity 'V':
              vorher :  ref :   SET [0 : 1] OF A FOR ref;
              nachher:  ref :   SET [0 : 1] OF B FOR ref;
       Attributentfernung aus Entity 'A':  refw :   W;
       Attributerzeugung für Entity 'B':  refw :   W;
       Anpassung der Domäne eines Attributs von Entity 'W':
              vorher :  refw :   SET [0 : 1] OF A FOR refw;
              nachher:  refw :   SET [0 : 1] OF B FOR refw;
Schema 'schema1' wird erweitert.
       Einbau des Entity 'W' aus Schema 'schema2' (ohne seine Beziehungen)
       zwischen Entity 'B' und Entity 'C'.
       Attributentfernung aus Entity 'B':  refc :   C;
       Attributerzeugung für Entity 'W':  refc :   C;
```

```
        Erstellung einer neuen Beziehung zwischen Entity 'B' und Entity 'W'.
                Hin-Attribut:  refw :   W FOR refw;
                Rück-Attribut:  refw :   B;
        Anpassung der Domäne eines Attributs von Entity 'C':
                vorher :  refc :   SET [0 : 1] OF B FOR refc;
                nachher:  refc :   SET [0 : 1] OF W FOR refc;
Schema 'schema1' wird erweitert.
        Einbau des Entity 'V' aus Schema 'schema2' (ohne seine Beziehungen)
        zwischen Entity 'B' und Entity 'X'.
        Attributentfernung aus Entity 'B':  refx :   X;
        Attributerzeugung für Entity 'V':  refx :   X;
        Erstellung einer neuen Beziehung zwischen Entity 'B' und Entity 'V'.
                Hin-Attribut:  ref :   V FOR ref;
                Rück-Attribut:  ref :   B;
        Anpassung der Domäne eines Attributs von Entity 'X':
                vorher :  refx :   SET [0 : 1] OF B FOR refx;
                nachher:  refx :   SET [0 : 1] OF V FOR refx;
        Anpassung des obigen neu erzeugten Attributs
        und ggf. seines Gegenattributs:
                vorher          :  refx :   X;
                vorher  (Gegen) :  refx :   SET [0 : 1] OF V FOR refx;
                nachher (Gegen) :  refvx :   SET [0 : 1] OF V FOR refvx;
                nachher         :  refvx :   X;
Schema 'schema2' wird erweitert.
        Einbau des Entity 'C' aus Schema 'schema1' (ohne seine Beziehungen)
        zwischen Entity 'W' und Entity 'Y'.
        Attributentfernung aus Entity 'W':  refy :   Y;
        Attributerzeugung für Entity 'C':  refy :   Y;
        Erstellung einer neuen Beziehung zwischen Entity 'W' und Entity 'C'.
                Hin-Attribut:  refc :   C FOR refc;
                Rück-Attribut:  refc :   W;
        Anpassung der Domäne eines Attributs von Entity 'Y':
                vorher :  refy :   SET [0 : 1] OF W FOR refy;
                nachher:  refy :   SET [0 : 1] OF C FOR refy;
        Anpassung des obigen neu erzeugten Attributs
        und ggf. seines Gegenattributs:
                vorher          :  refy :   Y;
                vorher  (Gegen) :  refy :   SET [0 : 1] OF C FOR refy;
                nachher (Gegen) :  refy :   SET [0 : 1] OF C FOR refy;
                nachher         :  refy :   Y;
Augmentations OK.
```

```
Schema 'schema2' wird mit dem Ausgabeschema 'schema1' gemerged.
       Entity 'A' wird mit dem gleichnamigen Entity gemerged.
              Die untere Kardinalitätsgrenze des bestehenden Mengenattributs
              'refb' wird im Ausgabeschema auf 1 erniedrigt.
       Entity 'V' wird mit dem gleichnamigen Entity gemerged.
              Attribut 'refz' aus Entity 'V' wird in das gleichnamige
              Entity übernommen.
              Attribut 'ref' aus Entity 'V' ersetzt das gleichnamige
              Attribut des gleichnamigen Entity (grösserer Kard.-Bereich).
       Entity 'W' wird mit dem gleichnamigen Entity gemerged.
              Attribut 'refw' aus Entity 'W' ersetzt das gleichnamige
              Attribut des gleichnamigen Entity (grösserer Kard.-Bereich).
       Entity 'X' wird mit dem gleichnamigen Entity gemerged.
              Attribut 'val' aus Entity 'X' wird in das gleichnamige
              Entity übernommen.
       Entity 'Y' wird mit dem gleichnamigen Entity gemerged.
       Entity 'Z' wird mit dem gleichnamigen Entity gemerged.
              FEHLER: Die gleichnamigen Attribute 'refz'
              der Entities 'Z' sind nicht typverträglich.
       Entity 'B' wird mit dem gleichnamigen Entity gemerged.
              Die untere Kardinalitätsgrenze des bestehenden Mengenattributs
              'refb' wird im Ausgabeschema auf 1 erniedrigt.
       Entity 'C' wird mit dem gleichnamigen Entity gemerged.
FEHLER beim Merge Vorgang.
```

Durch das Protokoll kommt heraus, daß das Attribut "refz" des Entity "Z" in beiden Schemata auf unterschiedliche Entities (A bzw. V) zeigt. Nachdem dieser Fehler in Schema 2 durch Umbenennung von refz in refz2 korrigiert wurde, läuft der Algorithmus fehlerfrei durch. Der nächste Abschnitt zeigt das sich so ergebende, korrekte Ergebnischema.

Ergebnisschema

Abbildung 22 zeigt eine graphische Darstellung des entstandenen Schema. Obwohl die Ausgangsschemata keine strukturellen Zyklen hatten, besitzt das Ergebnisschema den "Kreis" (Z, A, B, V). Ursache ist, daß die dritte Korrespondenz nicht mit angegeben werden konnte. Das folgende Listing zeigt die Textform des Resultatschemas.

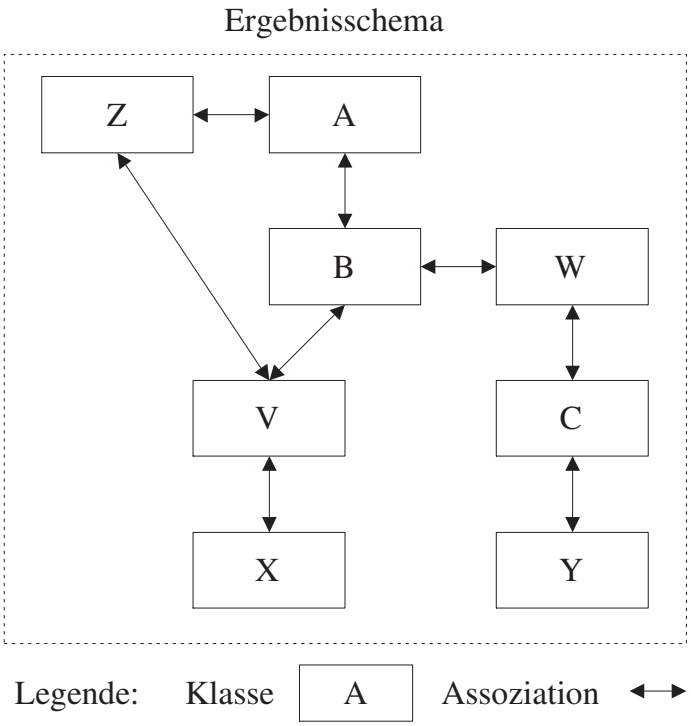

Abbildung 22: Ergebnis nach SIM Algorithmus und Merge der Teilschemata

```
(*------------------------------------------------------------

 S c h e m a 1 . p u r e . e x p

 generated at:  Fri Jun 16 14:02:14 2000

 GENERATED FILE. DO NOT EDIT!
  ------------------------------------------------------------*)

(*////////////////////// SCHEMA: schema1 //////////////////////*)

SCHEMA schema1;

ENTITY A;

 refb :   SET [1 : 8] OF B;
 refz :   Z;
END_ENTITY; (* A *)
```

```
ENTITY B;

  INVERSE
 refb :   SET [1 : 4] OF A FOR refb;
 refw :   W FOR refw;
 ref :   V FOR ref;
END_ENTITY; (* B *)

ENTITY C;

 refy :   Y;

  INVERSE
 refc :   SET [0 : 1] OF W FOR refc;
END_ENTITY; (* C *)

ENTITY X;

 val :   STRING;

  INVERSE
 refvx :   SET [0 : 1] OF V FOR refvx;
END_ENTITY; (* X *)

ENTITY Y;

 val :   STRING;

  INVERSE
 refy :   SET [0 : 1] OF C FOR refy;
END_ENTITY; (* Y *)

ENTITY Z;

  INVERSE
 refz :   SET [0 : 1] OF A FOR refz;
 refz2 :   SET [0 : 1] OF V FOR refz2;
END_ENTITY; (* Z *)

ENTITY W;
```

```
 refc :   C;

  INVERSE
 refw :   SET [0 : 1] OF B FOR refw;
END_ENTITY; (* W *)

ENTITY V;

 refvx :   X;
 refz2 :   Z;

  INVERSE
 ref :   SET [0 : 1] OF B FOR ref;
END_ENTITY; (* V *)

END_SCHEMA; (* schema1 *)
```

10.11 Kapitelzusammenfassung

Auf den vorhandenen Klassen aufbauend, wurde eine mögliche C++ Implementierung des SIM Algorithmus beschrieben. Durch Entkopplung der Abhängigkeiten mit Hilfe von abstrakten Basisklassen wird eine Verwendung mehrerer Sprachdialekte in einem Programm ermöglicht. So konnten die vorgefertigten Containerklassen des neueren C++ Standards die Entwicklungszeit verkürzen.

Bei Schemata mit sich kreuzenden Pfaden kann der SIM Algorithmus ein falsches Ergebnis liefern. Z.B. kann es passieren, daß ein Entity zweimal neu in ein anderes Teilschema eingebaut wird. Als Ausweg wurde die zusätzliche Bedingung (E6) für den Verlauf der Pfade innerhalb eines bestimmten Teilschema gefunden. Es wurde gezeigt, daß dann die Bedingungen (E1) bis (E4+) aus Kapitel 7 bei Schemaerweiterungen erhalten bleiben und auf die Bedingung (E5) verzichtet werden kann.

Das Ablaufbeispiel hat gezeigt, daß die Mehrzahl der manuell zu treffenden Entscheidungen die Kardinalitäten des Ergebnisschema betreffen. Es sind aber noch weitere Tests mit großen Schemata nötig, um die relative Häufigkeit der automatischen und der manuellen Erweiterungsschritte besser einschätzen zu können.

11 Zusammenfassung und Ausblick

Die Integration großer Schemata zu einem unabhängigen Neusystem, ob für die Produktdatentechnologie oder bei einer Firmenfusion, ist auch meist ein umfangreiches Projekt, das sich über Monate oder Jahre hinziehen kann. Ursache ist zum einen, daß es noch kein Verfahren gibt, welches einen Großteil der Konflikte behandeln kann. Zum anderen können wichtige Vorgaben, z.B. Schlüssel und Integritätsbedingungen, noch teilweise fehlen, und die Inter-Schema-Korrespondenzen müssen erst aufgestellt werden.

Unterschiede im Detaillierungsgrad erfordern eine besondere Vorgehensweise, um Redundanz zu vermeiden und konsistente Updates zu ermöglichen. Hierfür gibt es Ansätze, die durch einen Vergleich von Schlüsseln oder Navigationspfaden korrespondierende Attribute aus nicht korrespondierenden Klassen integrieren können.

Diese Ansätze müssen noch geeignet mit den umfangreichen Modellierungsmöglichkeiten von EXPRESS, wie Vererbung und konstruierte Datentypen, kombiniert werden. Zudem ist die hier vorgestellte theoretische Arbeit über die hierarchischen Datentypen für die Praxis nicht geeignet.

Diese Fragestellungen werden sicherlich in zukünftigen Arbeiten noch viel genauer erforscht, da die zunehmende Vernetzung das Volumen und die Möglichkeiten des Datenaustauschs stark erweitert.

Im zweiten Teil der Arbeit wurde der SIM Algorithmus implementiert. Die grundlegenden Ideen wurden beibehalten. Es kam heraus, daß die Bedingungen für korrekte Pfadkorrespondenzen modifiziert werden müssen, um bei Schemaerweiterungen erhalten zu bleiben. Als möglicher Ausweg wurde beschrieben, wie Pfade in einem einzelnen Teilschema verlaufen müssen, um den korrekten Ablauf sicherzustellen.

Danksagung

Der Autor möchte sich sehr herzlich bei Prof. Dr. Gunter Schlageter,
Dr. Wolfgang Wilkes und Dipl. Inform. Martin Pein für die Ausgabe und optimale Betreuung dieser Arbeit am Lehrgebiet praktische Informatik I der Fernuniversität Hagen bedanken.

A Header - Dateien

Dieser Anhang zeigt die Schnittstellen der neu erstellten Programmteile.

A.1 DCorrespondenceSet.h

```
/*_____________________________________________________________

 DCorrespondenceSet.h

 Verarbeitung einer Menge von Pfadkorrespondenzen mit
        Hilfe des SIM Algorithmus.

 Version
        31.04.2000      erstellt        (Volker Wendrich)

  _____________________________________________________________*/

#ifndef __DCorrespondenceSet__
#define __DCorrespondenceSet__
/*____________________ #includes ______________________________*/

#include "DPath.h"

/*____________________ DCorrespondenceSet _____________________*/

class DCorrespondenceSet
{
 public:
 // Konstruktor
 DCorrespondenceSet ();

 // Destruktor
 ~DCorrespondenceSet ();

 // Access
 // Bildet aus den zwei Eingabepfaden eine Korrespondenz.
 // Der Eingabeparameter pLinenum wird später angezeigt, falls
 // mehrere Korrespondenzen widersprüchlich sind.
 // Er sollte die Zeilennummer der Korrespondenzdatei enthalten.
```

```
 //
 // Returnwert: true (1) gdw. Die Korrespondenz ist korrekt,
 //      sonst false (0)
 int addCorrespondence(  DPath *   pDPath1,
       DPath *   pDPath2,
       char * pLinenum);


 // Führt den SIM-Algorithmus aus
 // Der Benutzer wird gefragt, wenn eine automatische
 // Entscheidung nicht möglich ist
 // Returnwert: true (1) gdw. Die Korrespondenzen sind
 //  widerspruchsfrei, sonst false (0)
 int generate_augmentations();

 private:
  void *  c;
  void *  cm;
  void *  pr;
};
#endif
```

A.2 DPath.h

```
/*_____________________________________________________

 DPath.h

 Diese Datei enthaelt die Deklarationen
 eines Pfades durch das Modell.

 Version
        31.04.2000      erstellt        (Volker Wendrich)

  _____________________________________________________*/

#ifndef __DPath__
#define __DPath__

/*_________________________ DPath _____________________*/
```

```
class DPath
{
 public:
 // Konstruktor 1
 // Der Pfad verläuft im Schema pSchemaName und beginnt
 // mit dem Entity pEntityName und dem Attribut
 // pAttributeName dieses Entity.
 DPath::DPath (
  char *  pSchemaName,
  char *  pEntityName,
  char *  pAttributeName
 );

 // Konstruktor 2
 // Der Partner Pfad der Korrespondenz wird
 // zusätzlich mit übergeben. Daraus resultiert eine
 // gegenseitige Verkettung der Pfade.
 DPath::DPath (
  char *  pSchemaName,
  char *  pEntityName,
  char *  pAttributeName,
  DPath *  pPartnerPath
 );

 // Destruktor
 ~DPath ();

    // Access

    // Muss nach dem Konstruktor aufgerufen werden, um zu prüfen,
    // ob die Angaben beim Konstruktoraufruf korrekt waren.
    // Wenn ja, ist der Returnwert true, sonst false.
    virtual bool    initialize()=0;


    // Verlängert den Pfad um ein weiteres Attribut, d.h. vom
    // letzten Entity des Pfads wird versucht, über das gegebene
    // Attribut zu einem neuen Datentyp zu gelangen.
    // Returnwert true gdw. das Weiterhangeln war erfolgreich
    virtual bool    append(char *      pAttributeName)=0;
```

```
// Liefert eine Zeichenkette:
// <Name des ersten Entity im Pfad>#<Name des zweiten Knoten im Pfad>
virtual char *  getKey()=0;


// Wie getKey mit dem letzten Entity und dem vorletzten Knoten.
// Wenn der Pfad nicht mit einem Entity endet,
// ist der Rückgabewert NULL
virtual char *  getInverseKey()=0;


// true gdw. Der Pfad hat die Länge wie nach initialize(),
// d.h. genau ein Attribut
virtual bool    isShort()=0;


// Liefert den Namen des ersten Entity im Pfad,
// wie beim Konstruktor übergeben
virtual char *  getStartEntityName()=0; // =0 bedeutet abstract


// Liefert den Namen des zweiten Entity im Pfad,
// falls vorhanden, sonst NULL
virtual char *  getSecondEntityName()=0;


// Liefert den Namen des vorletzten Entity im Pfad,
// wenn der letzte Knoten ein Entity ist,
// sonst wird der Name des letzten Entity zurückgegeben
virtual char *  getVorletztesEntityName()=0;


// Liefert den Namen des nächsten Zwischen-Entity im Pfad,
// falls ein weiteres vorhanden ist (das letzte Entity
// zählt nicht mit), sonst NULL.
// Die Aufzählung beginnt nach einem Aufruf von
// getStartEntityName() wieder von vorn
// (mit dem zweiten Knoten) und reicht bis zum
// vorletzten Entity-Knoten
```

```
virtual char *  getNextEntityName()=0;


// Liefert den Namen des letzten Knoten des Pfads
virtual char *  getTargetTypeName()=0;


// Liefert den Namen des Schema des Pfads
virtual char *  getSchemaName()=0;


// Liefert den Namen des ersten Entity im Schema des Pfads,
// - hat nichts mit den Entities des Pfads zu tun
virtual char *  getFirstEntityNameOfSchema()=0;


// Liefert den Namen des nächsten Entity im Schema des Pfads,
// - hat nichts mit den Entities des Pfads zu tun -
// Der Returnwert ist NULL, wenn es keines mehr gibt
// Die Aufzählung beginnt nach einem Aufruf von
// getFirstEntityNameOfSchema() wieder von vorn
// (mit dem zweiten Entity im Schema)
virtual char *  getNextEntityNameOfSchema()=0;


// Falls pKey aus den Namen der ersten beiden Entities von
// pDPath2 besteht, wird das zweite Entity von pDPath2 aus
// dessen Schema kopiert und neu in das Schema dieses Pfads
// eingebaut, falls nicht, wird das vorletzte
// Entity von pDPath2 eingebaut (inverse augmentation).
// Zwischen dem ersten und dem neuen Entity wird eine neue
// Beziehung erstellt.
// Die alte Beziehung vom ersten zum zweiten Knoten wird
// zwischen neues und zweites Entity verlagert.
// Für die neue und die verlagerte Beziehung werden die
// Kardinalitäten entsprechend geändert, falls der Parameter
// pIsEdgeAugmentation true/1 ist.
// (Bei EdgeAugmentation ist das neue Entity die Link-Klasse
// einer m:n Beziehung)
// Für die neue und die verlagerte Beziehung wird versucht,
// die Namen aus dem Schema von pDPath2 zu übernehmen.
```

```
// Am Ende wird pDPath2 verkürzt.
// Die Methode kann auch mehrfach aufgerufen werden, wenn
// das neue Entity schon vorhanden ist, wird es nicht nochmals
// eingebaut und keine neue Beziehung erstellt.
// Dies ist nützlich, wenn ein Bündel von Pfaden (pDPath2) in
// einer gemeinsamen Richtung startet.
//
// Vorbedingung: pDPath2 muss mindestens zwei Attribute haben
// Rückgabe: true bei Erfolg
virtual bool    augmentSchema(  char *   pKey,
        DPath *   pDPath2,
        int pIsEdgeAugmentation)=0;


// Pfadverkürzung am Anfang
virtual void    RemHead()=0;


// Pfadverkürzung am Ende
virtual void    RemTail()=0;


// Liefert eine neu erzeugte Zeichenkette, um den Pfad für
// Debug-Zwecke auszudrucken
virtual char *  display()=0;


// Liefert den Partnerpfad (siehe Konstruktor 2)
virtual DPath *  getPartnerPath()=0;


// Liefert einen Zeiger auf diesen Pfad
virtual void *  getIterator1()=0;


// Liefert einen zweiten Zeiger auf diesen Pfad
virtual void *  getIterator2()=0;


// Merkt sich einen Zeiger auf diesen Pfad
virtual void    setIterator1(void * pIterator1)=0;
```

```
    // Merkt sich einen zweiten Zeiger auf diesen Pfad
    virtual void    setIterator2(void * pIterator2)=0;
};
#endif
```

A.3 DPathImpl.h

```
/*_____________________________________________________

 DPathImpl.h

 Diese Datei enthaelt Implementierungs-Details
 eines Pfades durch das Modell und wird nur für
 Aufrufe von einem Pfad zum anderen gebraucht.

 Die Schnittstelle zum Pfad steht in DPath.h

 Version
 31.04.2000      erstellt (Volker Wendrich)

  _____________________________________________________*/

#ifndef __DPathImpl__
#define __DPathImpl__

#include "DPath.h"

class DPathImpl:public DPath
{
 public:
  // Konstruktor 1, siehe DPath.h
  DPathImpl::DPathImpl (
   char *  pSchemaName,
   char *  pEntityName,
   char *  pAttributeName
  );

  // Konstruktor 2, siehe DPath.h
```

```
DPathImpl::DPathImpl (
 char *  pSchemaName,
 char *  pEntityName,
 char *  pAttributeName,
 DPath *  pPartnerPath
);

// Destruktor
~DPathImpl ();


// Access - DPathImpl spezifisch

// Liefert das Schema des Pfads
Schema *                getSchema();

// Liefert das erste Entity
EntityDecl *            getFirstEntity();

// Liefert das zweite Entity, falls existent, sonst null
EntityDecl *            getSecondEntity();

// Liefert das erste Attribut
// (dessen Name im Konstruktor angegeben wurde)
Attribute *          getFirstAttribute();

// Liefert das Attribut ==> des "mittleren" Entity zum
// letzen Entity/Knoten, der Pfad muss wie folgt aussehen:
// Entity1 <--> Entity2 ==> Entity3/Type
// Bei anderen Pfadarten ist der Rückgabewert null
Attribute *             getSecondAndLastAttribute();

// Setzt den Partner-Pfad zu diesem Pfad
void    setPartnerPath(DPath *   pPartnerPath);


// Access - siehe DPath.h
bool append(char *      pAttributeName);
      bool initialize();
      char *   getKey();
      char *   getInverseKey();
```

```
        bool     isShort();

        char *   getStartEntityName();
        char *   getSecondEntityName();
        char *   getVorletztesEntityName();
        char *   getNextEntityName();
        char *   getTargetTypeName();

        char *   getFirstEntityNameOfSchema();
        char *   getNextEntityNameOfSchema();
        char *   getSchemaName();
        bool     augmentSchema( char *   pKey,
              DPath *   pDPath2,
              bool pIsEdgeAugmentation);
        void     RemHead();
        void     RemTail();
        char *   display();
        DPath *  getPartnerPath();

        void *  getIterator1();
        void *  getIterator2();
        void    setIterator1(void * pIterator1);
        void    setIterator2(void * pIterator2);


        private:
        // Name des Schema des Pfads
        char *   cSchemaName;

        // Schema des Pfads
        Schema *   cSchema;

        // Name des ersten Entity des Pfads
        char *   cEntityName;

        // Name des ersten Attributs des Pfads
        char *   cAttributeName;

    // Liste aller Attribute des Pfads
        MemberList *  cAttributeList;
```

```
        // Zwischen-Knoten im Pfad,
        // NICHT enthalten: Start-Entity, letztes Entity/Datentyp
        TmplTextList *   cDPathNodeList;

        // Zeiger zum Iterieren der cDPathNodeList
        void *   cTypeListPointer;

        // Partner Pfad
        DPath *  cPartnerPath;

        // Zeiger zum Iterieren aller Entities des Schema des Pfads
        EntityDecl *   cEntityOfSchemaIterator;

        // Speichert einen Zeiger, der auf diesen Pfad zeigen sollte
  void * cIterator1;

        // Speichert einen weiteren Zeiger, der auf diesen
        // Pfad zeigen sollte
  void * cIterator2;
};
#endif
```

A.4 CAugmentEXPRESS.h

```
/*
 C A u g m e n t E X P R E S S

 Parst eine Datei mit Pfadkorrespondenzen und ruft
    den Augmenter DCorrespondenceSet auf,
    um Teilschemata einander anzupassen.

 Version
 31.04.2000      erstellt (Volker Wendrich)
*/

#ifndef __CAugmentEXPRESS_H__
#define __CAugmentEXPRESS_H__

/*____________________ CAugmentEXPRESS _____________________*/
```

```
class CAugmentEXPRESS {
public:
 // Konstruktor & Destruktor

 CAugmentEXPRESS (){}

 ~CAugmentEXPRESS (){}

    // Methoden

 // Parst die Korrespondenzdatei
 // Legt Pfade (DPath) an und bildet Korrespondenzen daraus
 // (DCorrespondenceSet.addCorrespondence())
 // Nach erfolgreichem Parsen und Anlegen der Pfade und
 // Korrespondenzen wird der SIM-Algorithmus gestartet
 // (DCorrespondenceSet.generate_augmentations())
 bool checkCorrespondencesAndAugmentSchema (
   char * pCorrespondenceFileName);
};
#endif
```

A.5 CMergeEXPRESS.h

```
/*
 C M e r g e E X P R E S S

 Vereinigt mehrere EXPRESS Schemata zu einem Schema

 Version
 31.04.2000      erstellt (Volker Wendrich)
*/

#ifndef __CMergeEXPRESS_H__
#define __CMergeEXPRESS_H__

/*____________________ #includes ____________________*/

#include "CDictIterator.h"

/*____________________ Konstanten ____________________*/
```

```
enum {

 mkATTRIBUTE,
 mkDERIVE,
 mkINVERSE,
 mkUNIQUE,
 mkWHERE,
 mkMETHODS,

 mkALL = -1
};

extern FILE *   fIntegrationsProtokoll;

//////////////////// CMergeEXPRESS ////////////////////

/*_____________________ CMergeEXPRESS ___________________*/

class CMergeEXPRESS : public CDictIterator {
public:
 // Konstruktor & Destruktor

 CMergeEXPRESS () : CDictIterator ()
 {
  m_stream =  fIntegrationsProtokoll;
 }

 ~CMergeEXPRESS (){}


 // Merge aller Schemata:
 bool merge ();


 // Merge zweier Schemata, das Ergebnis steht in pSchema1
 bool mergeSchema (  Schema *   pSchema1,
      Schema *   pSchema2);


 // Merge zweier Entities, das Ergebnis steht in pEntity1
```

```
 bool mergeEntity (  EntityDecl * pEntity1,
      EntityDecl * pEntity2);
};
#endif
```

A.6 SchemaUtilities.h

```
/*-------------------------------------------------------

 SchemaUtilities.h

 Diese Datei enthaelt nützliche Funktionen
 fuer die Zugriffe auf das Modell.

 Version
 31.04.2000      erstellt (Volker Wendrich)

  -------------------------------------------------------*/



// Löscht das explizite oder inverse Attribut pAttribute
// aus dem Entity pEntity, falls vorhanden
void            removeAttributeFromEntity(
       EntityDecl *   pEntity,
       Attribute *   pAttribute);



// Sucht das Entity namens pEntityName im Schema pSchema
EntityDecl *    findEntityInSchema(
       Schema *   pSchema,
       char *   pEntityName);



// Sucht die Konstante namens pConstantName
// im Schema pSchema
DConstant *     findConstantInSchema(
       Schema *   pSchema,
```

```
        char *   pConstantName);


// Sucht den Basistyp namens pTypeName im Schema pSchema
BaseType *      findTypeInSchema(
        Schema *    pSchema,
        char *   pTypeName);


// findet explizite und inverse Attribute
Attribute *     findAttributeInEntity(
        EntityDecl *   pEntityDecl,
        char *   pAttributeName);


// findet nur explizite Attribute
Attribute *     findExplicitAttributeInEntity(
        EntityDecl *   pEntityDecl,
        char *   pAttributeName);


// findet nur inverse Attribute
Attribute *     findInverseAttributeInEntity(
        EntityDecl *   pEntityDecl,
        char *   pAttributeName);


// Sucht die Domänenregel namens pDomainRuleName
// im Entity pEntity
DomainRule *   findDomainRuleInEntity(
        EntityDecl *   pEntity,
        char *   pDomainRuleName);


// Sucht den Schlüssel namens pDomainRuleName
```

```
// im Entity pEntity
UniqueRule *   findUniqueRuleInEntity(
       EntityDecl *   pEntity,
       char *   pUniqueRuleName);



// Hilfsfunktion, um die Attribute des Entity
// pEntityDecl aus dem Schema pSchema anzuzeigen
void           printAttributesOfEntity(
       Schema *   pSchema,
       EntityDecl *   pEntityDecl);



// Hilfsfunktion, um die Attribute der
// MemberList pAttributeList anzuzeigen
void           printAttributeList(MemberList *   pAttributeList);



// Liefert den Namen des Typs, auf den das Attribut zeigt
char *         dereferenziere(Attribute *   pAttribute);



// wird nicht mehr benötigt
char *         typeToString(BaseType *   pBaseType);



// Ausdruck eines Attributs mit Hilfe von CGenerateExpress
void           printAttribute(Attribute *   pAttribute);



// Sucht das Gegen - Attribut zu einem gegebenen Attribut
// (egal ob explizit oder invers)
// Das gegebene Attribut pAttribute muss im Entity pEntity
// des Schema pSchema existieren, sonst ist das Ergebnis NULL
Attribute *     findReverseAttribute(
```

```
        Schema *   pSchema,
        EntityDecl *   pEntity,
        Attribute *   pAttribute);


// liefert die Kardinalitäten für Aggregations-Attribute
// (egal ob explizit oder invers)
// bei anderen Attributen ist der Returnwert NULL
DExpression * getBoundSpec (Attribute *   pAttribute);



// liefert die Mengen-Art für Aggregations-Attribute
// (egal ob explizit oder invers)
// bei anderen Attributen ist der Returnwert NULL
int getSetOrBag (Attribute *   pAttribute);



// hängt pAttribute von pEntityFrom ab und eine Kopie
// an pEntityTo an.
// falls pIsEdgeAugmentation = 1, werden Aggregationstypen
// (pAttribute = SET ...) zu gewöhnlichen Attributen
// Falls das Attribut im Ziel-Entity schon vorkommt,
// wird es nicht angehängt und der Returnwert ist NULL.
// Nach einer erfolgreichen Operation wird das kopierte
// Attribut zurückgegeben.
Attribute *   moveAttribute(Attribute *   pAttribute,
        EntityDecl *   pEntityFrom,
        EntityDecl *   pEntityTo,
        int pIsEdgeAugmentation);



// Zwischen pEntityFrom und pEntityTo wird eine neue 1:1
// oder 1:n Beziehung erstellt. Wenn pIsEdgeAugmentation=0,
// dann wird eine 1:1 Beziehung erstellt. Wenn
// pIsEdgeAugmentation=1, dann wird eine 1:card[pAttribute]
// Beziehung erstellt.
// pName wird der Name des Hin-Attributs, falls pName=NULL
```

```
// wird ein neuer Name automatisch generiert.
// pReverseName wird zum Namen des Gegen-Attributs, falls
// pReverseName=NULL wird der Hin-Name verwendet.
void createAssociation(  Attribute *   pAttribute,
       char *   pName,
       char *   pReverseName,
       EntityDecl *   pEntityFrom,
       EntityDecl *   pEntityTo,
       int pIsEdgeAugmentation);



// wie fflush, direkt fflush funktioniert nicht,
// druckt die letzte Zeile vor einem Fehler
void            flush(FILE *stream);
```

Literatur

[Abi88] Serge Abiteboul, Richard Hull, *Restructuring Hierarchical Database Objects*, Theoretical Computer Science 62, pp. 3-88, 1988

[Bat86] Batini, Lenzerini, Navathe, *A Comparative Analysis of Methodologies for Database Integration*, ACM Computing Services, Vol. 18, pp. 323-364, ACM digital library www.acm.org/dl, 1986

[Bis86] Joachim Biskup, Bernhard Convent, *A Formal View Integration Method*, Proceedings of International Conference of the Management of Data, ACM digital library www.acm.org/dl, 1986

[Bre97] Ulrich Breymann,
Die C++ Standard Template Library,
ISBN 3-8273-1067-9, Addison Wesley 1997

[Bri98] Peter McBrian and Alexandra Poulovassilis,
A Formalisation of Semantic Schema Integration,
Information Systems, Vol. 23, No. 5, pp. 307-334, 1998

[Con97] Stefan Conrad, *Föderierte Datenbanksysteme*, ISBN 3-540-63176-3, Springer 1997

[Ek95] L. Ekenberg, P. Johannesson, *Conflictfreeness as a Basis for Schema Integration*, Proceedings of CISMOD-95, pp. 1-13, http://www.dsv.su.se/∼johanw/articles/CISMOD95.html, 1995

[EXPR94] ISO 10303-11: *Express Language Reference Manual*, http://www.cs.etsu.edu/faculty/sanderso, 1994

[Fow97] Martin Fowler, *UML konzentriert*, Addison Wesley, ISBN 3-8273-1329-5, 1997

[GIM97] Ingo Schmitt, *Schemaintegration für den Entwurf föderierter Datenbanken*, Dissertation, infix Verlag, ISBN 3-89601-443-9, 1998

[Hul87] Richard Hull, Roger King, *Semantic Database Modeling: Survey, Applications, and Research Issues*, ACM Computing Surveys, Vol. 19, No. 3, ACM digital library www.acm.org/dl, September 1987

[ISO96] ISO/ANSI WG21/N0836: *ANSI C++ Standard*, 1996

[Kle95] Bernd Owsnicki-Klewe, *Algorithmen und Datenstrukturen*, Wißner, 1994

[Kos94a] P. Buneman, S. Davidson, A. Kosky,
Theoretical Aspects of Schema Merging,
EDBT' 92, Technical Report, University of Pennsylvania,
http://www.cis.upenn.edu/∼db/schemas.html, 1994

[Kos94b] P. Buneman, S. Davidson, A. Kosky, M. VanInwegen,
A Basis for Interactive Schema Merging, Technical Report, University of Pennsylvania, http://www.cis.upenn.edu/∼db/schemas.html, 1994

[Kos94c] A. Kosky, *A Formal Model for Databases with Applications to Schmema Merging*, Technical Report, University of Pennsylvania, http://www.cis.upenn.edu/∼db/schemas.html, 1994

[Kurs1666] Peter Dadam, *Kurs 1666 Datenbanken in Rechnernetzen* der Fernuniversität Hagen, 1996

[Kurs1667] Gunter Schlageter et. al., *Kurs 1667 Informationsverwaltung für Design-Anwendungen* der Fernuniversität Hagen, 1995

[Kurs1826] S. Ceri, G. Gottlob, L. Tanca, *Kurs 1826 Logic Programming and Databases* der Fernuniversität Hagen, KE3, pp. 123, 1997

[Mitc83] J.C. Mitchell, *The Implication Problem for Functional and Inclusion Dependencies*, Information and Control 56, pp. 154-173, 1983

[ODMG93] Cattell, R. G. G., *Object Databases: The ODMG-93 Standard*, Morgan Kaufmann, 1993

[PS98] C. Parent, S. Spaccapietra, *Issues and Approaches of Database Integration*, Communications of the ACM 41, Article 166, ACM digital library www.acm.org/dl, 1998

[San94] Donald Bruce Sanderson, *Loss of Data Semantics in Syntax Directed Translation*, Rensselaer Polytechnic Institute, Troy, New York, http://www.cs.etsu.edu/faculty/sanderso, 1994

[SDAI94] ISO/CD 10303-22: *Standard Data Access Interface*, 1994

[SIM95] Peter Fankhauser, Regina Motz, Gerald Huck,
SIM Schema Integration Methodology,
http://www.darmstadt.gmd.de/oasys/projects/irodb/more.html, 1995

[SPD92] S. Spaccapietra, C. Parent, Y. Dupont, *Model Independent Assertions for Integration of Heterogeneous Schemas*, VLDB Journal, 1(1):81-126, 1992

[Tah98] Shirin Tahzib, *Suitability of Express for Database Integration*, http://www.mis.coventry.ac.uk/research/isse/datmod/express.html, 1998

[Vid94] Vania Vidal, Marianne Winslett, *Preserving Update Semantics In Schema Integration*, ACM digital library www.acm.org/dl, 1994

[WOL94] S. Davidson, A. Kosky, *WOL: A Language for Database Transformations and Constraints*, University of Pennsylvania, http://www.cis.upenn.edu/∼db/schemas.html, 1994

[XML98] T. Bray, J.Paoli, C. M. Sperberg-McQueen, *Extensible Markup Language (XML) 1.0*, W3C Recommendation REC-xml-19980210, http://www.w3.org/REC-xml, 1998