

Universität-Gesamthochschule Paderborn

Fachbereich 17 - Mathematik/Informatik
Arbeitsgruppe Softwaretechnik
Warburger Straße 100
33098 Paderborn

Transformationsbasierte Interaktive Datenbankschemamigration

Diplomarbeit
für den integrierten Studiengang Informatik
im Rahmen des Hauptstudiums II

von

Christian Rummel
Am Berkeibach 1
59872 Meschede

vorgelegt bei
Prof. Dr. Wilhelm Schäfer

und
Prof. Dr. Gregor Engels

Paderborn, Dezember 1998

Danksagung

Ich danke allen, die mich bei der Erstellung dieser Arbeit unterstützt haben, besonders meinem Betreuer Herrn Dipl.-Inform. Jens Jahnke für die interessante Aufgabenstellung und seine wertvollen Anregungen und Hinweise, sowie Herrn Prof. Dr. W. Schäfer und Herrn Prof. Dr. G. Engels für die Korrektur der Arbeit.

Eidesstattliche Erklärung

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Paderborn, den 3. Dezember 1998

(Christian Rummel)

Inhaltsverzeichnis

1 Einleitung	5
1.1 Motivation	5
1.2 Überblick und Einordnung	6
1.3 Ziele	7
1.4 Kapitelübersicht	8
2 Verwandte Arbeiten	9
2.1 Transformationsbasierte Migration	9
2.2 Migrationsmetamodell	10
2.3 Informationskapazität	12
2.4 Konkurrierende Ansätze	13
3 Datenmodelle	15
3.1 Das relationale Datenmodell	15
3.2 Objektorientierte Datenmodelle	17
4 Das Migrationsmetamodell	19
4.1 Beschreibung des M^3	19
4.2 Formale Definition des Migrationsmetamodells	20
4.3 Interpretation eines Schemas im Migrationsmetamodell	23
4.4 Darstellung einer Datenbank als Graph	25
5 Abbildung konkreter Datenmodelle auf das Migrationsmetamodell	33
6 Restrukturierungstransformationen	35
6.1 Eigenschaften von Transformationen	35
6.2 Aufbau und formale Beschreibung einer Transformation	38
6.3 Reversibilität von Transformation	46

7 Katalog der Transformationen	49
7.1 Transformation von Typen	49
7.2 Transformation von Beziehungen zwischen Typen	54
7.3 Transformation von Vererbungsbeziehungen	60
7.4 Anwendungsbeispiel	67
8 Zusammenfassung und Ausblick	71
Literaturverzeichnis	73

1 Einleitung

1.1 Motivation

Datenbanken spielen in der Informationstechnik eine zentrale Rolle. Sie dienen als Speichersort für Daten verschiedenster Anwendungen.

Um den unterschiedlichen Anforderungen gerecht zu werden und um sich den jeweiligen technischen Gegebenheiten anzupassen, sind im Laufe der Zeit verschiedene Datenmodelle entwickelt worden. Die ältesten sind das hierarchische Modell und das Netzwerkmodell. Derzeit vorherrschend ist das relationale Datenmodell. In letzter Zeit beginnen objektorientierte Modelle, sich immer mehr zu verbreiten. Ein möglicher Standard für objektorientierte Modelle zeichnet sich mit dem ODMG-Datenmodell ([Cattell97]) ab, doch derzeit benutzen die meisten objektorientierten Datenbanksysteme noch proprietäre Modelle. Darüber hinaus gibt es auch Mischformen zwischen den unterschiedlichen Datenmodellen, die zumeist ebenfalls proprietär sind, wie etwa objektrelationale Modelle, die sich am relationalen Datenmodell orientieren (bzw. sogar daraus entstanden sind) und zusätzlich einige objektorientierte Eigenschaften wie etwa Vererbung haben.

Wegen der schnellen Entwicklung der Informationstechnik kommt es vor, daß Anwendungen bestimmte Datenmodelle verwenden, es sich aber im Nachhinein herausstellt, daß ein anderes Datenmodell den Anforderungen der Anwendung besser gerecht wird. So ist etwa das objektorientierte Datenmodell für Anwendungen mit komplex strukturierten Daten besser geeignet als das relationale Datenmodell. Dennoch benutzen solche Anwendungen häufig das relationale Modell, sei es, weil zum Zeitpunkt der Entwicklung der Anwendung noch keine objektorientierten Datenbanksysteme zur Verfügung standen, oder auch, weil relationale Systeme weiter verbreitet sind und daher leichter Entwickler für solche Systeme gefunden werden können.

Aus diesen Gründen ist es häufig wünschenswert, bestehende Datenbankanwendungen auf ein anderes Datenmodell zu migrieren. Dabei sollte es einerseits möglich sein, bestehende Daten übernehmen zu können, andererseits sollten aber auch die Vorteile des Zielmodells möglichst gut ausgenutzt werden, was unter Umständen eine erhebliche Umstrukturierung der Daten erforderlich macht. Außerdem sollte die Semantik des Ursprungsschemas erhalten bleiben. Idealerweise sollte sie nach der Migration mit entsprechenden Konzepten des Zielmodells ausgedrückt werden.

Auch bei der Föderation von Datenbanken, bei der die Daten in den bestehenden Datenbanken bleiben und nicht in eine einheitliche neue Datenbank überführt werden, besteht das Problem, ein Modell für eine einheitliche Sicht auf die Daten in den unterschiedlichen Datenbanken zu finden.

Wie eine solche Migration oder Föderation im einzelnen abläuft, hängt von den beteiligten Datenmodellen ab. Ein Ansatz, der das Ziel verfolgt, möglichst viele Datenmodelle zu berücksichtigen, sollte daher eine Möglichkeit bieten, die modellunabhängigen Migrationsschritte getrennt von den modellspezifischen zu behandeln.

1.2 Überblick und Einordnung

Diese Diplomarbeit ist Teil des Projektes VARLET (Verified Analysis and Reengineering of Legacy database systems using Equivalence Transformations), in dem am Lehrstuhl für Softwaretechnik an der Universität-Gesamthochschule Paderborn eine Datenbank-Migrationsumgebung entwickelt wird.

Mit dieser Datenbank-Migrationsumgebung können Datenbanken zwischen verschiedenen Datenmodellen migriert werden. Eine wichtige Aufgabe bei der Migration einer Datenbank ist die Migration des Datenbankschemas. In VARLET geschieht dies folgendermaßen: Ein Schema des Ursprungsmodells, welches in einer gängigen Repräsentation des Ursprungsmodells vorliegt (etwa SQL für das relationale Modell) wird als Eingabe benutzt. In einem initialen Schritt wird dieses Schema in ein internes, sogenanntes Migrationsmetamodell (M^3) überführt. Dieser Schritt kann automatisch erfolgen. In diesem Migrationsmetamodell wird ein benutzerinteraktives Redesign des Schemas durchgeführt. Schließlich wird das Schema dann wiederum automatisch in eine Repräsentation des Zielmodells (etwa ODMG für das objektorientierte Modell) überführt.

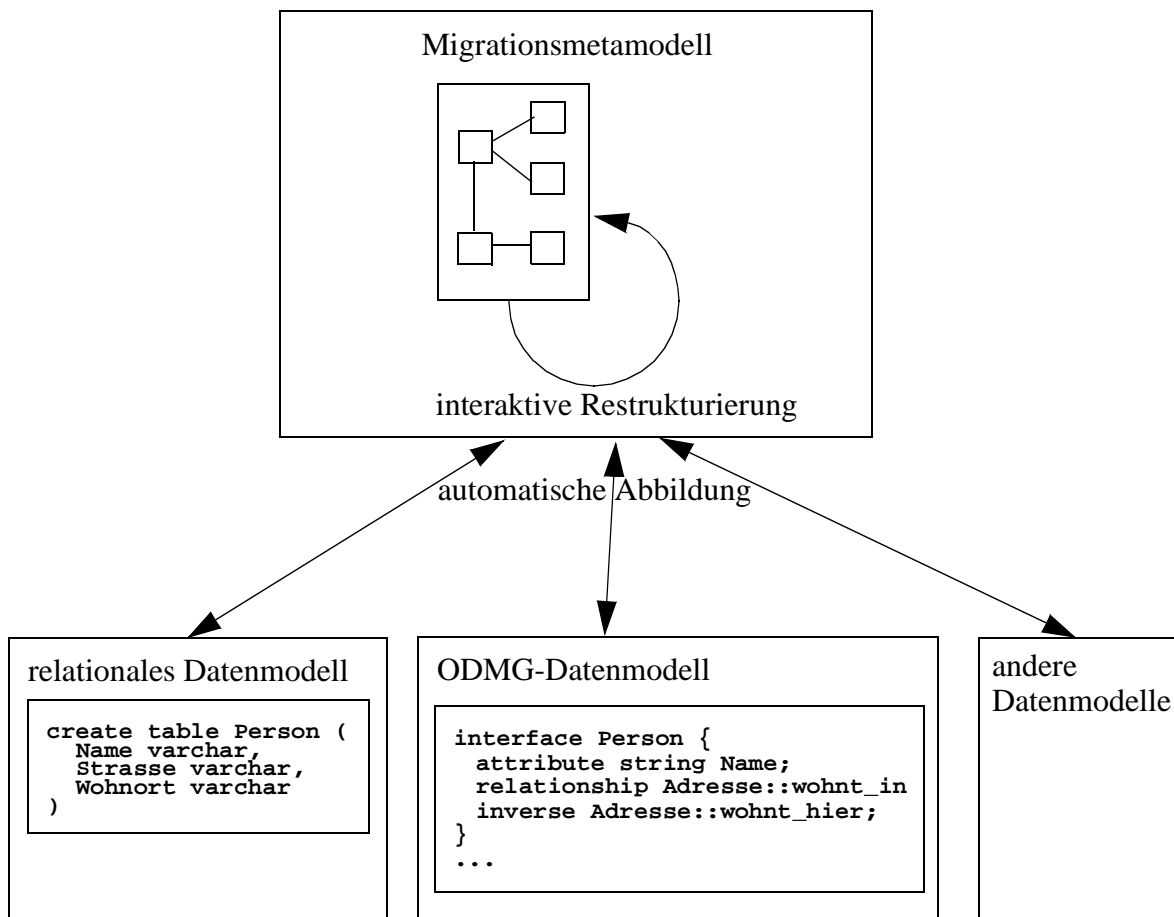


Abbildung 1: Migrationsprozeß in VARLET

Dieser Ansatz ermöglicht eine Trennung der datenbankmodellspezifischen Aspekte der Schemamigration von den Aspekten, die allen Modellen gemeinsam sind. Die modellspezifischen

Aspekte werden durch die Überführung der konkreten Datenmodelle ins Migrationsmetamodell (und umgekehrt) behandelt, während für die modellunabhängigen Schritte das M^3 benutzt wird.

Das Redesign im M^3 geschieht interaktiv, weil eine eindeutige Abbildung der Konzepte verschiedener Datenmodelle in der Regel nicht möglich ist. Daher soll einem Benutzer die Möglichkeit gegeben werden, diesen Prozeß zielgerichtet und der konkreten Anwendung angemessen durchzuführen.

Das Thema dieser Arbeit ist genau dieses Redesign des Schemas im Migrationsmetamodell.

1.3 Ziele

Die wichtigsten Ziele dieser Diplomarbeit sind:

- Grundlage für das Redesign ist das Migrationsmetamodell. Es ist wichtig, daß damit möglichst alle Konzepte der zu behandelnden Datenmodelle ausgedrückt werden können. Außerdem sollten sich diese möglichst einfach darauf abbilden lassen. In dieser Arbeit wird das M^3 formal definiert und seine Eigenschaften diskutiert. Darüber hinaus wird ein Mechanismus vorgestellt, mit dem sich konkreten Datenmodelle darauf abbilden lassen.
- Das Redesign des Schemas geschieht mithilfe von Restrukturierungstransformationen. Eine Restrukturierungstransformation formt die Konstrukte eines abgegrenzten Teils des Schemas in andere Konstrukte um. Das Redesign ist dann eine geeignete, vom Benutzer festgelegte Hintereinanderausführung einzelner Restrukturierungstransformationen. Ziel ist es, eine möglichst vollständige Menge von Transformationen zu finden, so daß alle gewünschten Umwandlungsschritte durchgeführt werden können.
- Eine Restrukturierungstransformation soll so aufgebaut sein, daß sie die Semantik des Ursprungsschemas möglichst genau erhält. Die Bedeutung des umgewandelten Schemas soll gleich sein wie die des Ursprungsschemas, lediglich mit Konzepten des Zieldatenmodells ausgedrückt. Darüber hinaus ist es oft wünschenswert, das Schema während des Redesigns zu erweitern oder zu reduzieren. VARLET ermöglicht daher auch Änderungen, die dem Schema Information hinzufügen oder entfernen. Dem Redesigner sollten solche Transformationen dann aber kenntlich gemacht sein, so daß er sich dieser Änderungen bewußt ist. Daher ist es wichtig, die Transformationen bezüglich der Erhaltung der Semantik bewerten zu können. Zu diesem Zweck werden bestimmte Eigenschaften für Schemata im M^3 formuliert, anhand derer sich Transformationen in unterschiedliche Klassen hinsichtlich der Semantikerhaltung einteilen lassen.
- Um die Eigenschaften der Transformationen nachweisen zu können, müssen sie in einer dafür geeigneten Notation vorliegen. Die formale Spezifikation der Transformationen ist daher ein weiteres wichtiges Ziel dieser Arbeit.

1.4 Kapitelübersicht

Kapitel 2 stellt einige wichtige verwandte Ansätze vor, die diese Diplomarbeit beeinflusst haben und beschreibt andere Systeme mit einer ähnlichen Zielsetzung wie VARLET.

In Kapitel 3 werden Eigenschaften von Datenmodellen diskutiert und das relationale und das ODMG-Datenmodell vorgestellt.

In Kapitel 4 wird das Migrationsmetamodell beschrieben und formal definiert. Darauf aufbauend wird gezeigt, wie sich eine Datenbank graphisch darstellen läßt.

Kapitel 5 beschreibt die Abbildung konkreter Datenbankmodelle auf das M^3 .

Kapitel 6 beschreibt den Aufbau von Restrukturierungstransformation, wie sie beschrieben werden, wie sie angewendet werden und nennt wichtige Eigenschaften, die sie haben können.

In Kapitel 7 werden die einzelnen Restrukturierungstransformationen spezifiziert und ihre Eigenschaften diskutiert.

In Kapitel 8 werden die Ergebnisse zusammengefaßt und ein Ausblick gegeben.

2 Verwandte Arbeiten

In diesem Kapitel werden Arbeiten vorgestellt, aus denen wichtige Konzepte in diese Diplomarbeit übernommen wurden, sowie konkurrierende Ansätze, die ähnliche Ziele verfolgen.

2.1 Transformationsbasierte Migration

[Hainaut91] und [Hainaut92] beschreiben das Werkzeug TRAMIS zur Restrukturierung von Datenbanken. Dessen Ansatz berücksichtigt lediglich relationale Datenbanken. Dennoch enthält er Konzepte, die sich auf die Restrukturierung im M^3 übertragen lassen.

Grundlage für die Restrukturierung sind Schematransformationen. Eine solche Schematransformation besteht aus zwei Teilen, der sogenannten Schema-Abbildung T und der sogenannten Instanz-Abbildung t .

Die Schema-Abbildung T einer Transformation wird durch je eine Vorbedingung P und eine Nachbedingung Q beschrieben. Diese Bedingungen sind Aussagen im relationalen Kalkül. P gibt Eigenschaften an, die das Schema erfüllen muß, damit die Transformation anwendbar ist. Q beschreibt das Schema nach der Transformation.

Die Instanz-Abbildung t führt die Umwandlung der Daten des Schemas vor der Transformation in das Schema nach der Transformation durch.

Zur Klassifikation von Transformationen werden die Begriffe *Reversibilität* und *symmetrische Reversibilität* einer Schematransformation eingeführt. Diese Eigenschaften beschreiben die Transformationen hinsichtlich der Erhaltung der Semantik des Schemas: Wenn eine Transformation wieder rückgängig gemacht werden kann, ist sichergestellt, dass dem Schema keine Information verloren gegangen ist, die Semantik also erhalten geblieben ist.

Eine Transformation (T,t) ist *reversibel*, wenn es eine andere Transformation (T^{-1},t^{-1}) , die Umkehrtransformation, gibt, so daß die durch (T,t) entstandene Datenbank sich durch (T^{-1},t^{-1}) wieder in die Ursprungsdatenbank rücktransformieren läßt. Eine Transformation ist *symmetrisch reversibel*, wenn sie reversibel ist und darüber hinaus auch die Umkehrtransformation reversibel ist.

Reversibilität und symmetrische Reversibilität sind transitiv bezüglich der Hintereinanderausführung von Transformationen.

Zur Verdeutlichung sei hier beispielhaft eine einfache Transformation aus [Hainaut91] angegeben. Sie beschreibt die aus der relationalen Theorie bekannte project/join-Dekomposition [Fagin77].

P :

$$R(U)$$

$$I \cup J \cup K = U ; R : I \rightarrow J | K$$

Q :

$$R1(I, J), R2(J, K)$$

t :

$$R1 = R[I, K]$$

$$R2 = R[J, K]$$

Die Vorbedingung P beschreibt eine relationale Tabelle R mit einer Spaltenmenge U . I , J und K sind Teilmengen von U , die zusammen die gesamte Menge U ergeben und für die mehrwertige Abhängigkeit $I \rightarrow J|K$ erfüllt ist. Die Nachbedingung Q beschreibt das Schema nach der Transformation: Aus R sind zwei neue Relationen $R1$ und $R2$ entstanden. Sie haben die Spaltenmengen $I \cup K$ und $J \cup K$. Die Instanz-Abbildung t gibt an, daß die Instanzen der neu entstandenen Relationen durch die Projektion von R auf $I \cup K$ bzw. $J \cup K$ gewonnen wird.

Die Umkehrtransformation zu dieser Transformation kann angegeben werden durch Vertauschen von Vorbedingung P und Nachbedingung Q . Die Instanz-Abbildung der Umkehrtransformation ist

$$t' : R = R1 \otimes R2$$

also der natürliche Join der beiden entstandenen Relationen $R1$ und $R2$.

Da die Schema-Abbildung der Umkehrtransformation meistens durch Vertauschen von Vor- und Nachbedingung gewonnen werden kann und daher nur die Instanz-Abbildung explizit angegeben werden muß, wird in [Hainaut91] als Kurzschreibweise für Transformation und Umkehrtransformation die Notation (P, Q, t_1, t_2) benutzt. Die Umkehrtransformation hat dann Q als Vorbedingung, P als Nachbedingung und t_2 als Instanz-Abbildung.

Die für diese Diplomarbeit wichtigen Konzepte sind einerseits die Aufteilung einer Transformation in Schema- und Instanz-Abbildung sowie die Reversibilitätsbegriffe. Diese Konzepte werden auf die Transformationen im Migrationsmetamodell anzuwenden sein.

2.2 Migrationsmetamodell

In [JJ94] wird ein Metamodell für Datenmodelle beschrieben. Das Metamodell ist hierarchisch aufgebaut. Die Elemente des Datenmodells sind darin im Sinne einer Vererbungshierarchie klassifiziert. Elemente eines konkreten Datenmodells lassen sich durch Vererbung an den Blattknoten der Vererbungshierarchie in das Metamodell integrieren.

Das Metamodell unterscheidet zunächst sogenannte Units und Links. Units sind Elemente, die Daten speichern können, Links drücken Beziehungen zwischen Daten aus. Units und Links werden dann genauer nach ihren Eigenschaften spezialisiert. Das Metamodell ist in Abbildung 2 zu sehen.

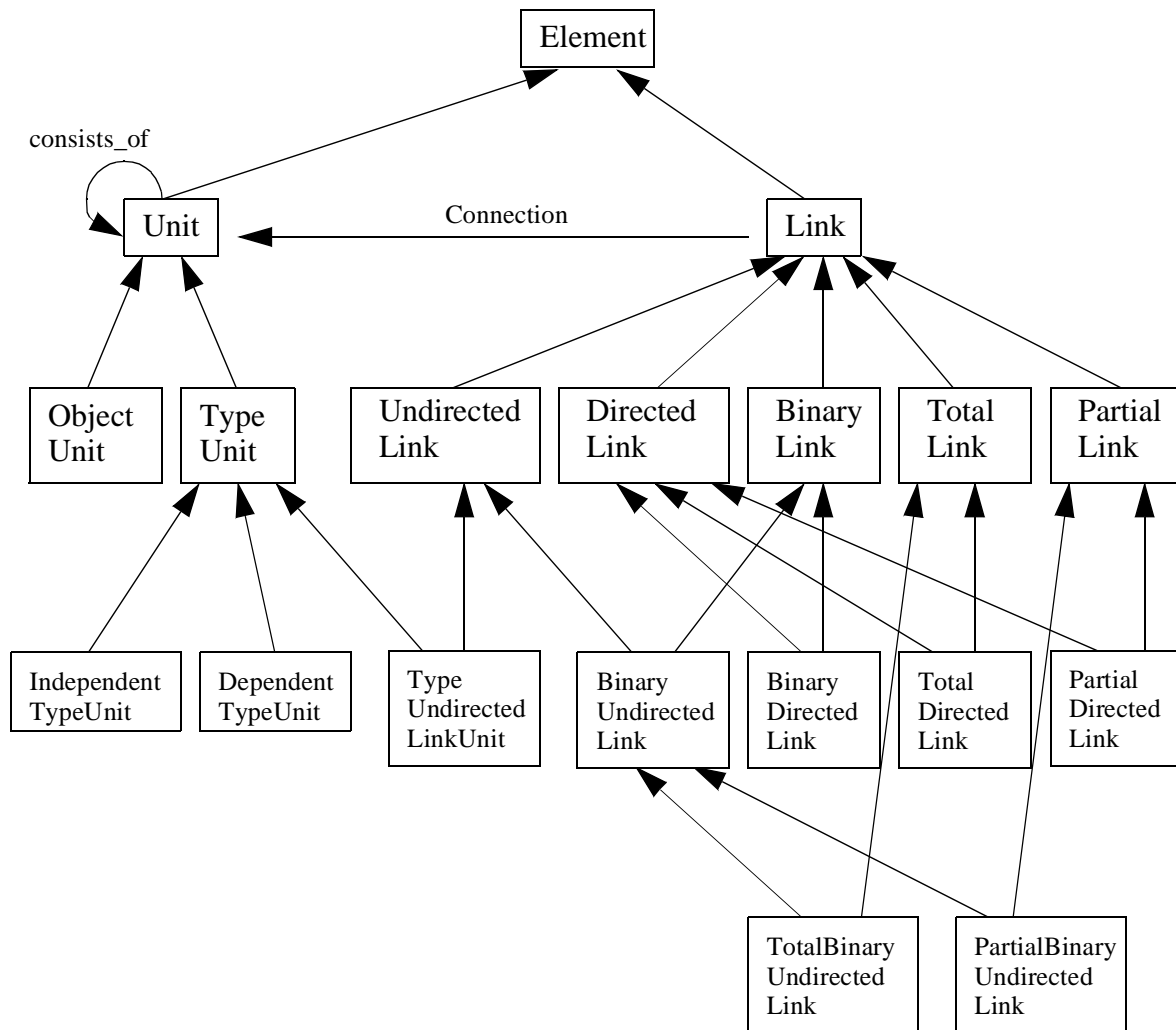


Abbildung 2: Metamodell nach [JJ94]

Die Ausdruckskraft dieses Metamodells reflektiert die von [JJ94] besonders berücksichtigten Datenmodelle, nämlich Entity-Relationship Diagramme und das relationale Datenmodell. Es wird vorgeschlagen, das Metamodell bei Bedarf zu erweitern, um es an andere Datenmodelle anzupassen.

Die Integration konkreter Datenmodelle geschieht, indem zunächst die Konstrukte des Quell- und des Zieldatenmodells in das Metamodell klassifiziert werden. Dann wird das zu migrierende Schema gemäß der Klassifizierung des Quelldatenmodells analysiert, also festgestellt, welche Schemaelemente welche Konstrukte instanzieren. Für jedes Schemaelement wird dann ein entsprechendes Konstrukt des Zieldatenmodells gesucht, welches schließlich mit einem entsprechenden Element des Zielschemas instanziiert wird.

Zur Spezifikation des Metamodells und der Schemata wird die deduktive Wissensrepräsentationssprache Telos verwendet. Das Schema wird durch Aussagen in dieser Sprache repräsentiert. Über weitere Regeln kann dann die Klassifikation der Schemakonstrukte ins Metamodell durchgeführt und schließlich das Zielschema instanziiert werden. Dabei ist eine Interaktion

durch einen Benutzer möglich (und notwendig).

Die für diese Diplomarbeit wichtige Idee aus [JJ94] ist das Metamodell. Nachteil dieses Ansatzes ist, daß er zwar bei nicht eindeutig zu treffenden Entscheidungen einen Benutzereingriff erlaubt, jedoch keine zielgerichtete Restrukturierung ermöglicht. Daher wird in dieser Diplomarbeit für die Integration konkreter Schemata sowie die eigentliche Migration ein anderer Ansatz verfolgt.

2.3 Informationskapazität

In [Tresch95] werden dynamische Veränderungen in Objekt-Datenbanken diskutiert. In diesem Zusammenhang werden Eigenschaften eines Datenbankschemas definiert, die auch für diese Diplomarbeit nützlich sind. Der für die Definition der Eigenschaften wichtige Begriff ist die *Informationskapazität*:

Es sei S ein Datenbankschema. σ bezeichne den Zustand der zu S gehörigen Datenbasis. Dann ist die *Informationskapazität* DB_S eines Schemas S die Menge aller möglichen Zustände der zu diesem Schema gehörenden Datenbasis σ .

Änderungen eines Schemas werden in [Tresch95] so definiert:

Die *Schemaänderung* S in S' ist eine Funktion $\mu : S \rightarrow S' = \mu(S)$.

Zu einer solchen Schemaänderung gehört immer auch eine Beschreibung der Änderung der Datenbasis:

Eine *Reorganisation* der Datenbasis mit Schema S und Zustand σ_S in eine Datenbasis mit neuem Zustand zum Schema S' ist eine Funktion

$$\rho_{S \rightarrow S'} : DB_S \rightarrow DB_{S'}, \rho_{S \rightarrow S'} : \sigma_S \rightarrow \sigma_{S'} = \rho_{S \rightarrow S'}(\sigma_S)$$

Die Gesamtheit von Schemaänderung μ und Datenbasisreorganisation ρ wird in [Tresch95] eine *Schemaevolution* genannt.

Mit Hilfe die Funktion ρ werden Eigenschaften einer Datenbasisreorganisation definiert:

Ist ρ injektiv, so ist die Reorganisation *verlustfrei*.

Ist ρ surjektiv, so ist die Reorganisation *vollständig*.

Daraus werden Eigenschaften von Schemata abgeleitet:

Schema S' ist *strukturell dominant* gegenüber Schema S , genau dann wenn es eine verlustfreie Reorganisation $\rho : DB_S \rightarrow DB_{S'}$ gibt.

Schema S' ist *strukturell äquivalent* mit Schema S , genau dann wenn es eine verlustfreie und vollständige Reorganisation $\rho : DB_S \rightarrow DB_{S'}$ gibt.

Darüber hinaus werden Eigenschaften von Schemaevolutionen definiert:

Eine Schemaevolution (μ, ρ) ist

- kapazitätserhaltend, genau dann wenn ρ bijektiv ist
- kapazitätserweiternd, genau dann wenn ρ injektiv, aber nicht surjektiv ist
- kapazitätsreduzierend, genau dann wenn ρ surjektiv, aber nicht injektiv ist
- kapazitätsändernd, genau dann wenn ρ weder injektiv noch surjektiv ist

Diese Eigenschaften werden auch zur Klassifizierung der Restrukturierungstransformationen in dieser Diplomarbeit verwendet.

2.4 Konkurrierende Ansätze

ONTOS.

Die Firma ONTOS bietet als kommerzielles Produkt die Objektdatenbank *OIS* (Object Integration Server) an ([ONTOS96]). Teil dieses Produktes ist ein sogenannter *Schema Mapper*, der die Abbildung eines existierenden relationalen Schemas auf ein Schema im (objektorientierten) Modell von OIS ermöglichen soll. Dazu werden die Relationen des relationalen Schemas auf Klassen im OIS-Objektmodell abgebildet. Die Spalten der Relationen werden zu Attributen der entsprechenden Klasse. Die Zuordnung der Relationen zu den Klassen und der Spalten zu den Attributen wird dann in einer sogenannten *type map* bzw. *property map* gespeichert.

Das Werkzeug bietet nur wenige Möglichkeiten zur Umstrukturierung des Schemas. Darüber hinaus stellt es recht hohe Anforderungen an das benutzte relationale Schema.

In VARLET sollen gerade auch diese Aspekte besonders berücksichtigt werden: Die zu migrierende Datenbank mit ihren Anwendungen wird auf vielfältige Weise analysiert, so daß ein eventuell nur unzureichend definiertes Schema mit zusätzlichen Informationen angereichert werden kann ([Bewermeyer98], [JSZ97]). Auch die Restrukturierung soll möglichst flexibel sein.

OpenDM.

Das im C-LAB Paderborn entwickelte Werkzeug *OpenDM* (Open Database Middleware) föderiert mehrere Datenbanken, die auch in unterschiedlichen Modellen vorliegen können ([OpenDM96], [Radeke95]). Es können dann einheitliche Benutzersichten auf die Gesamtheit der föderierten Datenbanken angeboten werden. Die Abbildung der Benutzersichten auf die tatsächlichen Datenbanken geschieht über mehrere Schichten. Die erste Schicht, das sogenannte *lokale Schema*, bilden die Schemata der Datenbanken; die letzte, das sogenannte *externe Schema*, ist die gewünschte Benutzersicht. Dazwischen liegen drei weitere Schichten, in denen die Schemata schrittweise ineinander überführt werden können. Die einzelnen Schichten sind:

1. lokales Schema
2. Komponentenschema
3. Exportschema
4. föderiertes Schema
5. externes Schema

Diese mehrstufige Schichtenarchitektur wurde in [SL90] definiert.

Die Übergänge zwischen den einzelnen Schichten werden jeweils durch eine textuelle Beschreibung definiert. Beim Zugriff auf die Daten durch einen Benutzer oder eine Anwendung, der das externe Schema zu Grunde liegt, wird die Abbildung der Daten auf das lokale Schema zur Laufzeit durchgeführt. Es findet also keine Migration des Gesamtdatenbestandes in eine einheitliche Datenbasis statt, sondern die Daten verbleiben in den ursprünglichen Datenbanken, und auch neue Daten werden dort eingefügt.

Eine grundlegende Umstrukturierung des Schemas ist nicht möglich. Da jede der Abbildungen von Hand programmiert werden muß, ist die Transformation der Daten fehleranfällig.

3 Datenmodelle

Dieses Kapitel beschreibt Eigenschaften von Datenmodellen und stellt das relationale und das objektorientierte Datenmodell als die von VARLET besonders berücksichtigten Modelle vor.

Datenmodelle stellen die Möglichkeit bereit, die Struktur von Daten zu beschreiben und die Daten dann zu manipulieren und auf sie zuzugreifen. Diese Strukturbeschreibung einer Datenbank wird *Schema* genannt, die Daten selbst sind dann die *Instanz* des Schemas.

Die Art dieser Strukturierungs- und Zugriffsmöglichkeiten sind die wesentlichen Eigenschaften eines Datenmodells. [Ullman88] nennt einige Kriterien, anhand derer sich Datenmodelle unterscheiden lassen:

- Objekt- oder Wertsemantik

In Datenmodellen mit Wertsemantik sind Daten allein durch ihre Werte unterscheidbar. Die Objektsemantik ordnet Daten eine Identität zu, anhand derer sie unabhängig von ihrem Wert unterschieden werden können.

- Umgang mit Redundanz

Ein grundsätzliches Ziel von Datenbanken ist es, die mehrfache Speicherung von Daten zu vermeiden. Neben dem zusätzlichen Speicherplatz, der dadurch verbraucht wird, können durch Redundanz Inkonsistenzen entstehen, wenn die Daten an einer Stelle verändert werden und an anderen nicht.

- Umgang mit m:n-Beziehungen

In einer Datenbank müssen oft Daten gespeichert werden, bei denen eine Gruppe von Elementen zu vielen Elementen einer anderen Gruppe in Beziehung stehen und umgekehrt. Eine Speicherstruktur für solche Beziehungen, die schnelle Anfragen ermöglicht, ist nicht immer leicht zu realisieren.

3.1 Das relationale Datenmodell

Grundlage des relationalen Datenmodells ist der mengentheoretische Begriff der *Relation*. Eine Relation ist eine Teilmenge des kartesischen Produkts mehrerer Mengen. Die Mengen, die in diesem Zusammenhang dann auch *Domänen* genannt werden, stellen im relationalen Datenmodell die Wertebereiche einfacher, atomarer Datentypen wie Zahlen oder Zeichenketten dar.

Das Schema einer relationalen Datenbank besteht aus der Beschreibung mehrerer solcher Relationen, dem *Relationenschema*. Die Relationen selbst sind dann die *Extension* oder der Datenbank.

Ein Relationenschema besteht aus dem eindeutigen Namen der Relation und einer Liste der an ihr beteiligten Domänen. Jeder der Domänen wird ein in dieser Relation eindeutiger Name zugeordnet. Die einzelnen Domänen mit ihren Namen werden *Attribute* der Relation genannt. Üblicherweise wird ein Relationenschema dann wie folgt notiert:

Relationenname(Attribut1, Attribut2, ...)

Die übliche Darstellungsform einer Relation ist eine Tabelle. Jedes der an der Relation beteiligten Attribute wird durch eine Spalte repräsentiert, in denen die an der Relation beteiligten Werte eingetragen werden. Die Zeilen der Tabelle bilden dann die Tupel der Relation. Die folgende Abbildung verdeutlicht dies:

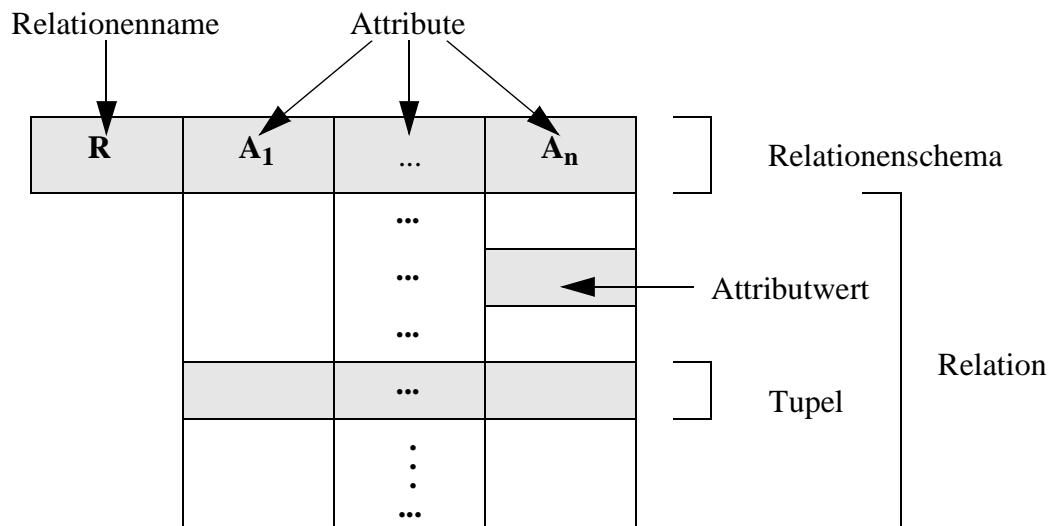


Abbildung 3: Begriffe im Relationenmodell (nach [SST97])

Zwischen den Attributen einer Relation können Abhängigkeiten bestehen. Eine *funktionale Abhängigkeit* besteht, wenn Werte einer Teilmenge B von Attributen die Werte einer anderen Attributmenge C eines jeden Tupels festlegen. Eine solche funktionale Abhängigkeit wird mit $B \rightarrow C$ notiert.

Daraus läßt sich die Definition des Schlüssels einer Relation herleiten: Eine Attributmenge B ist *Schlüssel* einer Relation R , wenn alle Attribute von R funktional abhängig von B sind.

Anhand dieser Begriffe lassen sich Normalformen für relationale Datenbankschemata definieren. In der Literatur (z. B. [EN94] oder [Ullman88]) sind 5 aufeinander aufbauende Normalformen gebräuchlich. Ihr Ziel ist, Redundanz in Datenbanken zu vermeiden.

Eine weitere Abhängigkeit, die in einer relationalen Datenbank bestehen kann, ist die *Inklusionsabhängigkeit*. Zwischen einem Attribut A einer Relation R besteht eine Inklusionsabhängigkeit zu einem Attribut A' einer Relation R' , wenn alle Werte, die in der Extension für A vorkommen, stets auch für das Attribut A' in der Extension vorhanden sind. Eine solche Inklusionsabhängigkeit wird mit $R(A) \subseteq R'(A')$ notiert. Sie drückt eine Beziehung zwischen den Attributen und ihren Relationen aus. Mit den Arten der Beziehungen können die Inklusionsabhängigkeiten klassifiziert werden ([FV93]). So lassen sich Abhängigkeiten, die eine Vererbungsbeziehung ausdrücken (I-IND) von einfachen Beziehungen zwischen Relationen (R-IND) unterscheiden.

Zur Definition von Relationen und zur Manipulation von Daten sowie zur Formulierung von Anfragen gibt es für relationale Datenbanken die standardisierte Sprache SQL (Structured Query Language) ([DD97]). Sie enthält die Befehle

- create zur Definition von Relationen
- insert zum Einfügen von Daten
- update zum Ändern von Daten
- delete zum Löschen von Daten
- select zur Formulierung von Anfragen

Diese Sprache wird von fast allen relationalen Datenbanksystemen verwendet, oft jedoch mit proprietären Erweiterungen.

Das relationale Datenmodell ist ein Modell mit Wertsemantik. Daten sind allein durch ihren Wert identifizierbar. Es ist nicht möglich, zwei Tupel, die in allen Attributen gleiche Werte enthalten, zu unterscheiden.

Relationale Datenbanken erfordern oft eine redundante Speicherung von Daten. So ist es etwa zum Ausdruck von Beziehungen zwischen Relationen nötig, den Schlüssel einer Relation in Form eines sogenannten Fremdschlüssels ein zweites Mal in der anderen Relation zu speichern.

Um eine n:m-Beziehung zwischen zwei Relationen auszudrücken, muß eine zusätzliche, sogenannte Beziehungsrelation definiert werden, die dann die Schlüssel der von zwei in Beziehung stehenden Tupeln jeweils als Fremdschlüssel enthält.

Diesen Nachteilen stehen als Vorteile gegenüber, daß sich relationale Datenbanksysteme aufgrund der einfachen Struktur des relationalen Modells leicht implementieren lassen. Es gibt viele ausgereifte Implementierungen, die auch wichtige andere (modellunabhängige) Eigenschaften wie Transaktionsschutz und Replikationsfähigkeit bieten. Anwendungen, die nur mit einfach strukturierten Daten arbeiten, können mit relationale Datenbanksystemen effizient implementiert werden. Darüber hinaus steht mit SQL eine deklarative Anfragesprache zur Verfügung, die eine einfache Formulierung von Anfragen ermöglicht und dabei von Optimierungsstrukturen abstrahiert.

3.2 Objektorientierte Datenmodelle

Mit dem Aufkommen objektorientierter Datenbanksysteme entstand eine Vielzahl von unterschiedlichen objektorientierten Datenmodellen, da in Ermangelung eines Standards jeder Hersteller sein eigenes Datenmodell definiert hat. Um dem abzuwehren, wurde von der *Object Management Group* mit dem OMDG-Datenmodell für objektorientierte Datenbanken ([Cattell97]) ein Standard geschaffen, der von den Herstellern mehr und mehr unterstützt wird. Es besteht aus folgenden Konstrukten:

- Die grundlegenden Komponenten sind *Objekte* und *Literale*. Objekte haben eine eindeutige Identität, Literale nicht.
- Der *Zustand* eines Objektes ist definiert durch die Werte seiner *Attribute* und seine *Beziehungen* zu anderen Objekten.
- Das *Verhalten* eines Objekts ist definiert durch die *Operationen*, die von ihm oder mit ihm ausgeführt werden können.

- Objekte und Literale können klassifiziert werden nach ihren *Typen*. Alle Elemente eines Typs haben die gleiche Zustandsmenge (d. h. Attribute und Beziehungen) und das gleiche Verhalten (d. h. Operationen). Ein Objekt wird auch als *Instanz* seines Typs bezeichnet. Zwischen den Typen können Vererbungsbeziehungen bestehen.

Die Definition von ODMG-Schemata geschieht mit der Objektdefinitionssprache ODL (*Object Definition Language*). Mit ihr werden die zu einem Schema gehörenden Typen spezifiziert.

Das ODMG-Datenmodell ist ein Modell mit Objektsemantik. Jedem Objekt wird bei seiner Erzeugung eine eindeutige Identität zugeordnet, über die es sich unabhängig von seinem Zustand von allen anderen Objekten unterscheiden läßt.

Diese Objektidentität ermöglicht es, Redundanz zu vermeiden, da sich darüber immer auf ein Objekt Bezug nehmen läßt, ohne etwa - wie im relationalen Modell - ein Hilfskonstrukt wie Fremdschlüssel verwenden zu müssen.

Zum Ausdruck von n:m-Beziehungen stehen in der ODL explizite Sprachmittel zur Verfügung, mit denen sich die Eigenschaften einer Beziehung spezifizieren lassen.

4 Das Migrationsmetamodell

In diesem Kapitel wird das Migrationsmetamodell beschrieben, mithilfe dessen die in den weiteren Kapiteln definierten Redesign-Transformationen durchgeführt werden.

Wie schon in Kapitel 2 erwähnt, stammt die Idee für das M^3 aus [JJ94]. Das dort beschriebene Modell orientiert sich recht deutlich am relationalen Datenmodell. So gibt es dort etwa keine Komponente, die Vererbungsbeziehungen direkt widerspiegelt. Die wichtigsten Komponenten sind *Units* und *Links*. Die *Units* beschreiben die Daten selbst, während die *Links* Beziehungen zwischen ihnen ausdrücken.

Diese Komponenten finden sich auch im dem in dieser Diplomarbeit definierten Metamodell. Darüber hinaus werden hier objektorientierte Konzepte stärker berücksichtigt. Tatsächlich orientiert sich das M^3 deutlich am ODMG-Datenmodell für objektorientierte Datenbanken. Dies entspricht auch der im VARLET-Projekt vorrangig berücksichtigten Migrationsrichtung, nämlich der auf objektorientierte Systeme.

Außerdem erweisen sich bei näherer Betrachtung objektorientierte Modelle als konzeptionell sehr reichhaltig. Sie sind ausdrucksstärker als andere verbreitete Modelle, und somit geeignet, auch die Konstrukte anderer Datenmodelle zu repräsentieren.

Es folgt zunächst eine informale Beschreibung der einzelnen Komponenten des M^3 . Danach wird es in einer algebraischen Notation formal definiert. Diese Definition ist Grundlage für die dann folgende graphische Darstellung der Schemaelemente.

4.1 Beschreibung des M^3

Ein Schema im Migrationsmetamodell besteht aus folgenden Komponenten:

Atomare Typen.

Atomare Typen beschreiben Daten, die im M^3 keine weitere interne Struktur mehr haben, etwa Zahlen oder Zeichenketten. Sie entsprechen dem, was in [JJ94] *TypeUnit* genannt wird. In relationalen Modellen sind es die Attribute einer Relation. Das ODMG-Datenmodell definiert als vergleichbares Konstrukt die *Atomic Objects*.

Komplexe Typen.

Komplexe Typen beschreiben eine Zusammenfassung anderer (atomarer oder komplexer) Typen zu einem neuen Typ. Dieser wird mit Hilfe von Aggregationen aus atomaren und komplexen Typen gebildet. Im relationalen Modell entspricht dies einer einzelnen Relation, wobei sich eine Relation nur aus atomaren, und nicht aus komplexen Typen zusammensetzen kann. Im objektorientierten Sinn kann ein komplexer Typ als eine Klasse aufgefaßt werden. Folglich gibt es auch abstrakte komplexe Typen, von denen es keine Instanzen gibt. Sie dienen lediglich zur Beschreibung gemeinsamer Eigenschaften anderer, nicht abstrakter Typen, die dann über den Vererbungsmechanismus abgeleitet werden können.

Aggregationen.

Aggregationen sind das Mittel, mit dem aus (atomaren und komplexen) Typen weitere (komplexe) Typen zusammengesetzt werden. Eine einzelne Aggregation beschreibt einen (atomaren oder komplexen) Typ innerhalb eines komplexen Typs. Er ist innerhalb des aggregierenden Typs über einen Namen identifizierbar. Außerdem legt die Aggregation eine Kardinalität fest, die angibt, ob der aggregierende Typ mehrere Exemplare des aggregierten Typs enthält und ob diese geordnet zu betrachten sind oder nicht. Die möglichen Kardinalitäten sind folglich *one*, *set* und *list*.

Assoziationen.

Eine Assoziation beschreibt Beziehungen zwischen den Instanzen zweier komplexer Typen. In jedem der beiden Typen ist der jeweils andere durch einen Rollennamen bekannt. Weiterhin wird für jeden Typ festgelegt, wieviele seiner Instanzen an der Beziehung teilnehmen können. Dabei wird einerseits berücksichtigt, ob die Beziehung bezüglich des jeweiligen Typs total ist, also jede Instanz in Beziehung zu einer Instanz des anderen Typs steht oder nicht, und ob sie funktional ist, also ob jede Instanz nur zu höchstens einer Instanz des anderen Typs in Beziehung steht.

Vererbungsbeziehungen.

An einer Vererbungsbeziehung sind zwei komplexe Typen beteiligt, der Ober- und der Untertyp. Instanzen des Untertyps sind dann ebenfalls Instanzen des Obertyps. Sie enthalten dann auch alle seine Aggregationen und Assoziationen. Zu jedem Typ darf es nur einen Obertyp geben.

4.2 Formale Definition des Migrationsmetamodells

Im folgenden wird das M^3 in algebraischer Notation formal definiert. In [Hohenstein93] wird eine ähnliche Darstellung eines erweiterten Entity-Relationship-Modells erarbeitet.

Ein Schema im Migrationsmetamodell ist definiert durch ein Tupel von Mengen $S = (Complex, Atomic, Aggr, Assoc, Inherit)$. Die Mengen bedeuten im Einzelnen:

- *Complex* ist die Menge der komplexen Typen. Jedes $c \in Complex$ hat die Signatur n, a . Dabei ist n der eindeutige Name des komplexen Typs. Der boolsche Wert a gibt an, ob der Typ abstrakt oder konkret ist.
- *Atomic* ist die Menge der atomaren Typen. Die Signatur besteht lediglich aus dem Namen n .
- $Types = Atomic \cup Complex$ bezeichnet die Gesamtheit der atomaren und komplexen Typen.
- *Aggr* ist die Menge der Aggregationen. Jedes $a \in Aggr$ hat die Signatur $c, n, m \rightarrow t$. Dabei gilt:
 - $c \in Complex$ ist der aggregierende komplexe Typ
 - $t \in Types$ ist der aggregierte Typ
 - n ist der Name des aggregierten Typs im aggregierenden Typ

- m die Kardinalität der Aggregation (*set*, *list* oder *one*)

Für $a = c, n, m \rightarrow t \in Aggr$ sind folgende Hilfsfunktionen definiert:

- $ctype: Aggr \rightarrow Complex, ctype(a) = c$
 - $type: Aggr \rightarrow Types, type(a) = t$
 - $name: Aggr \rightarrow String, name(a) = n$
 - $card: Aggr \rightarrow \{set, list, one\}, card(a) = m$
- *Assoc* ist die Menge der Assoziationen. Jede Assoziation $l \in Assoc$ hat die Form $c, n, m, t \leftrightarrow c', n', m', t'$. Dabei sind
 - $c, c' \in Complex$ die beteiligten komplexen Typen.
 - n und n' die Rollennamen der Assoziation in den beteiligten Typen.
 - m und m' sind boolsche Werte und geben an, ob die Assoziation bezüglich der jeweiligen Seite funktional ist oder nicht.
 - t und t' sind boolsche Werte und geben an, ob die Assoziation bezüglich der jeweiligen Seite total ist oder nicht.

Die möglichen Assoziationen können durch Mengenausdrücke klassifiziert werden:

- $Assoc_{1x} = \{l = c, n, m, t \leftrightarrow c', n', m', t' \in Assoc | (m = one)\}$ ist die Menge der 1:x-Assoziationen.
- $Assoc_{x1} = \{l = c, n, m, t \leftrightarrow c', n', m', t' \in Assoc | (m' = one)\}$ ist die Menge der x:1-Assoziationen.
- $Assoc_{nx} = \{l = c, n, m, t \leftrightarrow c', n', m', t' \in Assoc | (m = many)\}$ ist die Menge der n:x-Assoziationen.
- $Assoc_{xn} = \{l = c, n, m, t \leftrightarrow c', n', m', t' \in Assoc | (m' = many)\}$ ist die Menge der x:n-Assoziationen.
- $Assoc_{lt} = \{l = c, n, m, t \leftrightarrow c', n', m', t' \in Assoc | (t = total)\}$ ist die Menge der linksseitig totalen Assoziationen.
- $Assoc_{lp} = \{l = c, n, m, t \leftrightarrow c', n', m', t' \in Assoc | (t = partial)\}$ ist die Menge der linksseitig partiellen Assoziationen.
- $Assoc_{rt} = \{l = c, n, m, t \leftrightarrow c', n', m', t' \in Assoc | (t' = total)\}$ ist die Menge der rechtsseitig totalen Assoziationen.
- $Assoc_{rp} = \{l = c, n, m, t \leftrightarrow c', n', m', t' \in Assoc | (t' = partial)\}$ ist die Menge der rechtsseitig partiellen Assoziationen.

Durch Schnittmengenbeziehungen können die Assoziationen weiter spezialisiert werden. So ist etwa die Menge der n:1-Assoziationen durch folgenden Mengenausdruck beschrieben: $Assoc_{n1} = Assoc_{nx} \cap Assoc_{x1}$. Die linksseitig totalen, rechtsseitig partiellen Assoziationen sind $Assoc_{ltrp} = Assoc_{lt} \cap Assoc_{rp}$. Und schließlich die linksseitig totalen, rechtsseitig partiellen n:1 Assoziationen:

$$Assoc_{n1ltrp} = Assoc_{n1} \cap Assoc_{ltrp} = Assoc_{nx} \cap Assoc_{x1} \cap Assoc_{lt} \cap Assoc_{rp}$$

- $CTypeAttr = Aggr \cup Assoc$ bezeichnet die Vereinigung der Aggregationen und Assoziationen, die in ihrer Gesamtheit dann auch die Attribute eines komplexen Typs genannt werden.
- $Inherit$ ist die Menge der Vererbungsbeziehungen zwischen komplexen Typen. Jedes $v \in Inherit$ hat die Form $c \rightarrow c'$, mit $c, c' \in Complex$ und die Bedeutung: c ist direkter Untertyp von c' . Für indirekte Vererbungsbeziehungen wird folgende Kurzschreibweise eingeführt:

$$c \gg c' \Leftrightarrow \exists c_1, c_2, \dots, c_k \in Complex:$$

$$(c \rightarrow c_1) \in Inherit, (c_1 \rightarrow c_2) \in Inherit, \dots, (c_{k-1} \rightarrow c_k) \in Inherit$$

Abbildung 4 zeigt eine graphische Darstellung des der Elemente des Migrationsmetamodells.

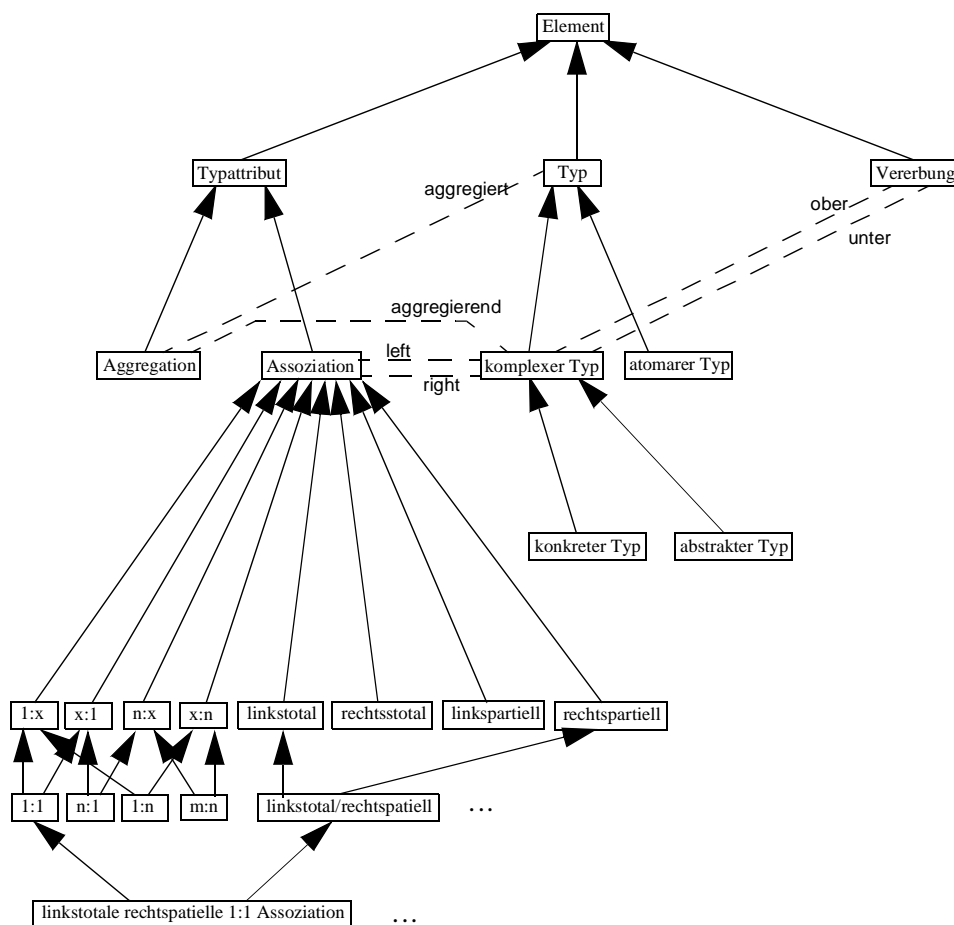


Abbildung 4: Migrationsmetamodell

4.3 Interpretation eines Schemas im Migrationsmetamodell

Das Thema dieser Arbeit ist die Migration von Datenbank-*Schemata*. Um jedoch bestimmte Eigenschaften einzelner Restrukturierungsschritte nachweisen zu können, ist es nötig, auch mögliche *Instanzen* eines Schemas zu betrachten, ist doch die Möglichkeit eines Transfers bestehender Daten in das neue Datenbankschema eines der wichtigsten Kriterien eines guten Migrationsverfahrens. In den folgenden Kapiteln wird - bei der Beurteilung der Eigenschaften einer Restrukturierungstransformation - die Untersuchung der Transformierbarkeit von Instanzen eine entscheidende Rolle spielen.

Das Migrationsmetamodell muß daher neben der Beschreibung des Schemas auch die Beschreibung einer zugehörigen Datenbasis ermöglichen. Eine formale Repräsentation der Datenbasis ist durch die im folgenden angegebene *Interpretation* eines Schemas gegeben.

Die Interpretation eines Schemas im M^3 enthält für alle Komponenten des M^3 eine Funktion, die die Elemente des Schemas auf die Instanzen der Datenbasis dieses Schemas abbildet.

Es werden zunächst einige im folgenden benutzte Bezeichnungen eingeführt:

- $|FUN|$ ist die Menge aller Funktionen.
- $|REL|$ ist die Menge aller Relationen.
- $|PRED|$ ist die Menge aller logischen Ausdrücke.
- M, N seien Mengen. $f: M \rightarrow N$ sei eine Funktion. $f(M)$ ist dann folgende Menge: $\{y \in N | \exists(x \in M, f(x) = y)\}$
- $P(M)$ ist die Potenzmenge von M

Eine Interpretation eines Schemas S im Migrationsmetamodell ist ein Tupel von Funktionen $I_S = (I_{Complex}, I_{Atomic}, I_{Aggr}, I_{Assoc}, I_{Inherit})$.

- Interpretation der komplexen Typen:

Die Interpretation eines komplexen Typ ist die Menge seiner Instanzen. Jede Instanz eines komplexen Typs wird durch eine eindeutige natürliche Zahl interpretiert. Dies entspricht dem Konzept der Objektidentität in objektorientierten Datenmodellen. Die Funktion $I_{Complex}$ bildet folglich jeden komplexen Typen des Schemas auf eine Menge von natürlichen Zahlen ab. Diese Zahlen entsprechen den Instanzen des Typs.

$$I_{Complex}: Complex \rightarrow P(IN)$$

Aufgrund der Semantik der Vererbung liegen in der Instanzenmenge eines Typs auch die Instanzen seiner direkten und indirekten Untertypen. Es gilt daher für alle $c \in Complex$:

$$I_{Complex}(c) = \bigcup_{c' \gg c \in Inherit} I_{Complex}(c')$$

Abstrakte komplexen Typen können nicht direkt instanziiert werden. Die Interpretation eines abstrakten Typs ist daher gleich der Menge der Instanzen aller seiner

direkten und indirekten Untertypen. Für die Interpretation eines abstrakten Typs $c \in Complex$ gilt daher:

$$I_{Complex}(c) = \bigcup_{c' \rightarrow c \in Inherit} I_{Complex}(c')$$

- Interpretation der atomaren Typen:

Die Interpretation eines atomaren Typs ist sein Wertebereich, zuzüglich eines Wertes $NULL$, der einen nicht definierten Wert für alle atomaren Typen repräsentiert. Wie der Wertebereich aussieht, hängt natürlich vom jeweiligen atomaren Typen ab.

$$I_{Atomic}: Atomic \rightarrow \bigcup_{a \in Atomic} P(dom(a)) \cup NULL$$

- Als Kurzschreibweise für die Interpretation eines beliebigen (atomaren oder komplexen) Typs wird folgende Bezeichnung verwendet:

$$I_{Types}(t) = \begin{cases} I_{Complex}(t), & t \in Complex \\ I_{Atomic}(t), & t \in Atomic \end{cases}$$

- Interpretation der Aggregationen:

Die Interpretation einer Aggregation ist eine Funktion, die jede Instanz des aggregierenden komplexen Typs auf die Menge der von ihr aggregierten Instanzen des aggregierten Typs abbildet.

$$I_{Aggr}: Aggr \rightarrow |FUN|$$

$$I_{Aggr}(a = c, m, n \rightarrow t) = f_a$$

f_a bildet jede Instanz von c auf eine Menge von Instanzen des aggregierten Typs t ab:

$$f_a: I_{Complex}(c) \rightarrow P(I_{Types}(t))$$

Die Kardinalität der Aggregation läßt sich durch folgende Aussagen beschreiben:

Falls die Aggregation die Kardinalität *one* hat, gilt für alle $o \in I_{Complex}(c)$:

$|f_a(o)| = 1$, also jeder Instanz von c wird durch f_a nur eine Instanz von t zugeordnet.

Falls die Aggregation die Kardinalität *list* hat, ist für alle $o \in I_{Complex}(c)$ eine Ordnung $<_a$ auf der Menge $|f_a(o)|$ definiert.

Für Aggregationen der Kardinalität *set* ist keine weitere Einschränkung erforderlich.

- Interpretation der Vererbungsbeziehungen:

Die Vererbungsbeziehung wird durch eine Teilmengenbeziehung der Instanzmengen der beteiligten komplexen Typen interpretiert. Die Menge der Instanzen des Untertyps ist eine Teilmenge der Instanzen des Obertyps. Die Interpretationsfunk-

tion bildet folglich jede Vererbungsbeziehung auf eine Aussage über die Instanzmenge der komplexen Typen ab.

$$I_{Inherit} : Inherit \rightarrow |PRED|$$

$$I_{Inherit}(v = c \rightarrow c') = \forall o \in I_{Complex}(c) : o \in I_{Complex}(c')$$

- Interpretation der Assoziationen:

Eine Assoziation ist eine Relation zwischen den Instanzmengen (also den Interpretationen) der beiden an ihr beteiligten komplexen Typen. Die Interpretationsfunktion bildet daher jede Assoziation auf eine Relation über die Instanzmenge der komplexen Typen ab.

$$I_{Assoc} : Assoc \rightarrow |REL|$$

$$I_{Assoc}(l = c, n, m, t \leftrightarrow c', n', m', t') = R_l \text{ mit}$$

$$R_l \subseteq I_{Complex}(c) \times I_{Complex}(c')$$

Die Totalitäten und Funktionalitäten lassen sich durch Aussagen über diese Relationen beschreiben:

Falls $m = true$ kommt jede Instanz von c nur in höchstens einem Tupel von R_l vor. Entsprechendes gilt für $m' = true$ und die Instanzen von c' .

Falls $t = true$ kommt jede Instanz von c in mindestens einem Tupel von R_l vor. Entsprechendes gilt für $t' = true$ und die Instanzen von c' .

Zur Verdeutlichung sei hier ein kleines Beispiel angegeben:

Es seien $c, c' \in Complex$ zwei komplexe Typen und $l = (c, n, m, t \leftrightarrow c', n', m', t') \in Assoc$ eine Assoziation zwischen den beiden. Die Interpretation von c sei $I_{Complex}(c) = \{o_1, o_2, o_3, o_4\}$ und die Interpretation von c' sei $I_{Complex}(c') = \{o'_1, o'_2, o'_3\}$.

Eine mögliche Interpretation von l ist dann:

$$R_l = I_{Assoc}(l) = \{(o_1, o'_1), (o_2, o'_1), (o_3, o'_2), (o_4, o'_3)\}$$

l erfüllt die Anforderungen einer beidseitig totalen n:1-Assoziation ist.

4.4 Darstellung einer Datenbank als Graph

Um ein Schema anschaulich darstellen zu können und um Transformationen leichter definieren zu können, werden im folgenden graphische Darstellungsformen für die Komponenten eines Schemas und ihre Interpretation, also die Datenbasis, eingeführt. Eine Datenbank kann dann als ein Graph, der *Schema-Instanz-Graph*, dargestellt werden.

Die Darstellung orientiert sich an der Sprache Progres [Zündorf95]. Diese Sprache ermöglicht eine hierarchische Typisierung der Elemente eines Graphen. Dies geschieht durch die Definition von Knoten- und Kantentypen. Jeder Knoten und jede Kante eines Graphen besitzt dann einen der definierten Typen. Bei der Definition eines Knotentyps können diesem Attribute gegeben werden, die weitere Eigenschaften der zugehörigen Knoten beschreiben.

Es werden nun die zur Beschreibung von Schema-Instanz-Graphen benötigten Knoten- und Kantentypen vorgestellt.

Knotentypen.

Die Knotentypen lassen sich in zwei wesentliche Gruppen aufteilen: *Schematypen* und *Instanztypen*. Die Schematypen geben Elemente des dargestellten Schemas wieder, während die Instanztypen die Objekte der Datenbasis darstellen. Sie werden über eine besondere Kante mit dem entsprechenden Schemaknoten, zu dem sie gehören, verbunden.

Tabelle 1 zeigt die zur graphischen Darstellung von Schema-Instanz-Graphen benötigten Knotentypen.

Knotentyp	Obertyp	Attribute	Beschreibung
Elem	-	-	gemeinsamer Obertyp für alle Elemente des Schema-Instanz-Graphen
SchemaElem	Elem	-	gemeinsamer Obertyp für alle Schemaelemente
NamedElem	SchemaElem	name:String	gemeinsamer Obertyp für Elemente, die einen Namen haben
Type	NamedElem	-	gemeinsamer Obertyp für atomare und komplexe Typen
Atomic	Type	-	atomare Typen
Complex	Type	is_abstract:bool	komplexe Typen
CTypeAttr	NamedElem	-	gemeinsamer Obertyp für Aggregationen und Assoziationen
Aggr	CTypeAttr	card:enum(one, set, list)	Aggregationen
Assoc	CTypeAttr	total:bool functional:bool	Assoziationen
Inherit	SchemaElem	-	Vererbungsbeziehungen

Tabelle 1: Knotentypen

Knotentyp	Obertyp	Attribute	Beschreibung
InstElem	Elem	-	gemeinsamer Obertyp für alle Instanzelemente
Type_i	InstElem	-	Gemeinsamer Obertyp für Instanzen von atomaren und komplexen Typen
Atomic_i	Type_i	value:void	Instanzen atomarer Typen
Complex_i	Type_i	identity:integer	Instanzen komplexer Typen
CTypeAttr_i	InstElem	-	gemeinsamer Obertyp für Instanzen von Aggregationen und Assoziationen
Aggr_i	CTypeAttr_i	-	Instanzen von Aggregationen
Assoc_i	CTypeAttr_i	-	Instanzen von Assoziationen

Tabelle 1: Knotentypen

Die Hierarchie der Knotentypen ist in Abbildung 4 dargestellt.

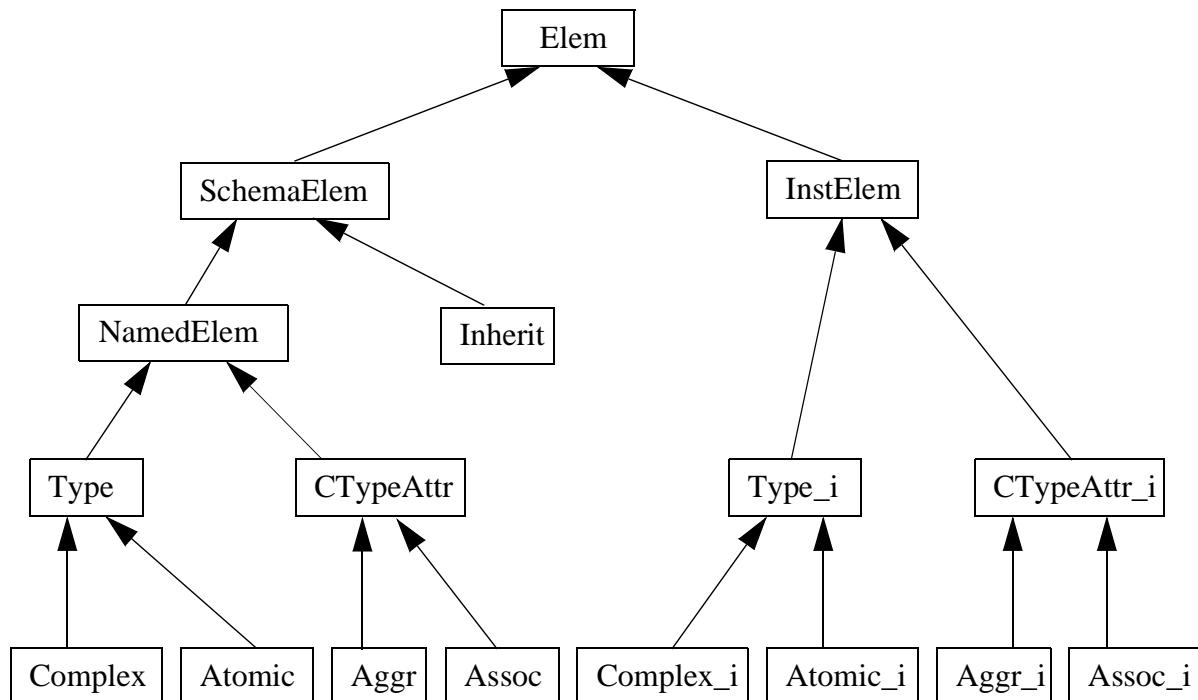


Abbildung 5: Hierarchie der Knotentypen

Kantentypen.

Wie bei den Knotentypen gibt es auch bei den Kantentypen zwei wesentliche Gruppen von Typen, die *Schemakanten* und die *Instanzkanten*. Schemakanten verbinden immer zwei Schemaknoten und Instanzkanten immer zwei Instanzknoten. Eine Besonderheit ist der Kantentyp *inst_of*: Er verbindet einen Instanzknoten mit einem zugehörigen Schemaknoten.

Man beachte, daß es auf Instanzebene keine Kanten gibt, die Vererbungsbeziehungen ausdrücken. Das liegt daran, daß die Vererbungsbeziehung allein Aussagen über Typen macht, und auf die Instanzebene keinen direkten Einfluß hat.

Tabelle 2 zeigt die zur Darstellung von Schema-Instanz-Graphen benötigten Kantentypen.

Kantentyp	Obertyp	Quelle	Ziel	Beschreibung
edge	-	Elem	Elem	Gemeinsamer Obertyp für alle Kantentypen
inst_of	edge	InstElem	SchemaElem	Verbindet Instanzelemente mit einem Schemaelement
has	edge	Complex	CTypeAttr	gemeinsamer Obertyp für hasAggr und hasAssoc
hasAggr	has	Complex	Aggr	verbindet komplexe Typen mit ihren Aggregationen
hasAssoc	has	Complex	Assoc	verbindet komplexe Typen mit ihren Assoziationen
aggr	edge	Aggr	Type	verbindet Typen mit den sie aggregierenden Aggregationen
assoc	edge	Assoc	Assoc	verbindet zusammengehörige Assoc-Knoten
inh	edge	Inherit	Complex	gemeinsamer Obertyp für super und sub-Kanten
super	inh	Inherit	Complex	verbindet Inherit-Knoten mit dem zugehörigen Obertyp
sub	inh	Inherit	Complex	verbindet Inherit-Knoten mit dem zugehörigen Untertyp

Tabelle 2: Kantentypen

Kantentyp	Obertyp	Quelle	Ziel	Beschreibung
has_i	edge	Complex_i	CTypeAttr_i	gemeinsamer Obertyp für hasAggr_i und hasAssoc_i
hasAggr_i	has_i	Complex_i	Aggr_i	verbindet Instanzen komplexer Typen mit den Instanzen ihrer aggregierten Aggregationen
hasAssoc_i	has_i	Complex_i	Assoc_i	verbindet Instanzen komplexe Typen mit ihren Assoziationen
aggr_i	edge	Aggr_i	Type_i	verbindet Instanzen von Typen mit den Instanzen sie aggregierenden Aggregationen
assoc_i	edge	Assoc_i	Assoc_i	verbindet zusammengehörige Assoc_i-Knoten

Tabelle 2: Kantentypen

Darstellung der Elemente des Schema-Instanz-Graphen.

- Darstellung von komplexen Typen

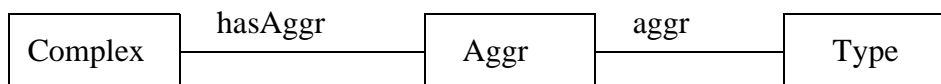
Ein komplexer Typ wird durch einen Knoten vom Typ *Complex* dargestellt.

- Darstellung von atomaren Typen

Ein atomarer Typ wird durch einen Knoten vom Typ *Atomic* dargestellt.

- Darstellung von Aggregationen

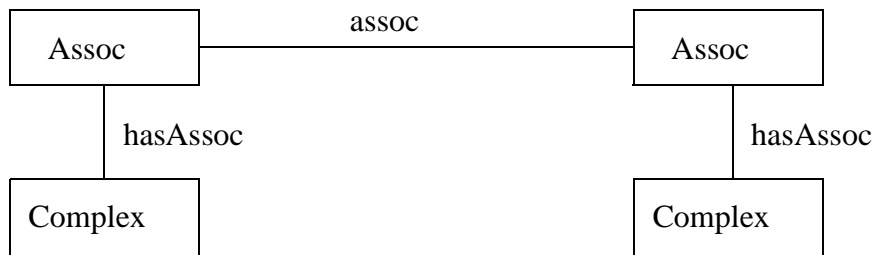
Eine Aggregation wird durch einen Knoten vom Typ *Aggr* dargestellt. Der Knoten ist über eine Kante vom Typ *hasAggr* mit dem aggregierenden Typ (Knotentyp *Complex*) verbunden und über eine Kante vom Typ *aggr* mit dem aggregierten Typ (Knotentyp *Type*).



- Darstellung von Assoziationen

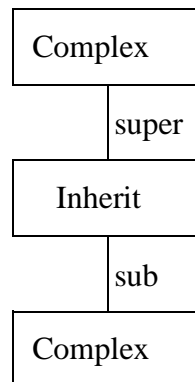
Assoziationen werden durch ein Paar von Knoten des Typs *Assoc* dargestellt, die mit je einem der beteiligten komplexen Typen durch Kanten vom Typ *hasAssoc*

verbunden sind. Untereinander sind die beiden Knoten durch eine Kante vom Typ *assoc* verbunden.



- Darstellung der Vererbungsbeziehungen

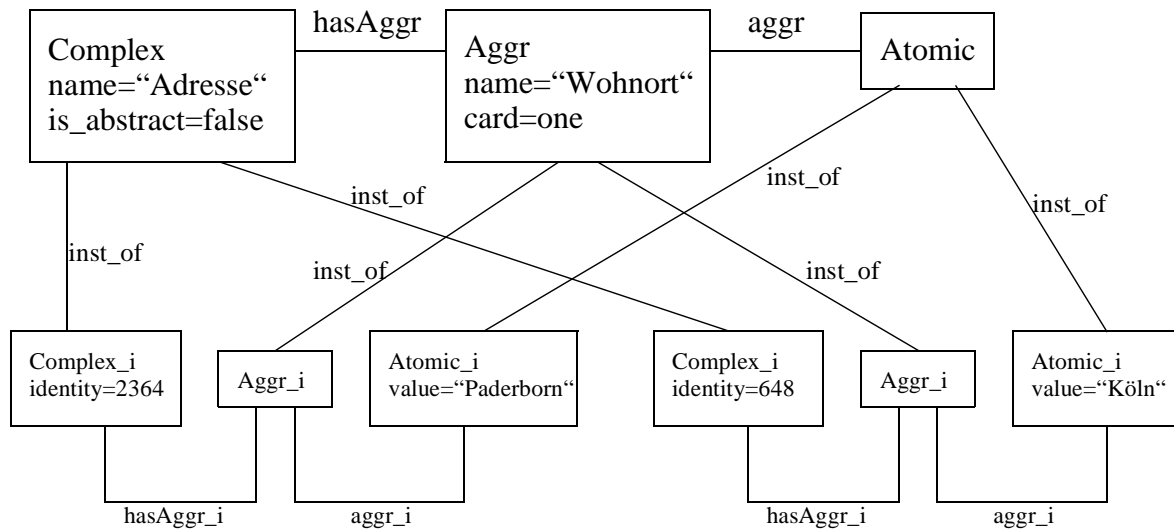
Vererbungsbeziehungen werden durch einen Knoten vom Typ *Inherit* dargestellt, der durch eine Kante vom Typ *super* mit dem Obertyp und durch eine Kante vom Typ *sub* mit dem Untertyp verbunden ist.



- Darstellung der Instanzen

Zur Darstellung der Instanzen gibt es zu jedem Schemaknotentyp einen Instanzknotentyp. Alle Instanzen eines Schemaelements werden durch je einen Knoten des entsprechenden Instanzknotentyps dargestellt und mit dem zugehörigen Schemaknoten über eine Kante vom Typ *inst_of* verbunden. Eine Ausnahme bildet die Vererbungsbeziehung. Sie drückt eine Beziehung zwischen Typen aus, nicht aber für einzelne Instanzen, und hat daher im Graphen auf Instanzebene keine direkte Entsprechung. Sie drückt sich jedoch darin aus, daß Knoten von Instanzen komplexer Typen mehrere *inst_of*-Kanten haben, wenn der zugehörige Typ einen Obertyp hat.

Als Beispiel sei hier eine Aggregation mit ihren Instanzen angegeben. Die Werte der Knotenattribute sind mit in den jeweiligen Knoten geschrieben.



Man beachte, daß die graphische Darstellung der Instanzen der in Abschnitt 4.3 angegebenen Interpretation des Schemas entspricht:

- Die Interpretation eines komplexen Typen ist eine Menge von natürlichen Zahlen, hier: {648, 2364}
- Die Interpretation eines atomaren Typen ist sein Wertebereich, hier: {„Köln“, „Paderborn“}
- Die Interpretation einer Aggregation ist eine Funktion, die jede Instanz des aggregierenden Typs auf die Menge der von ihr aggregierten Instanzen des aggregierten Typs abbildet. Die Knoten vom Typ `aggr_i` mit ihren inzidenten Kanten bilden also gewissermaßen eine graphische Darstellung dieser Funktion. Die Funktion läßt sich auch in Form einer Wertetabelle angeben:

x	f(x)
648	{„Köln“}
2364	{„Paderborn“}

Tabelle 3: Wertetabelle der Interpretationsfunktion einer Aggregation

Eigenschaften von Schema-Instanz-Graphen.

Aufgrund ihrer Konstruktion gelten für Schema-Instanz-Graphen zahlreiche Eigenschaften, die in den folgenden Kapiteln bei der Untersuchung von Transformationen auf dem Schema-Instanz-Graphen ausgenutzt werden.

- Eigenschaften der *inst_of*-Kanten
 - Jeder Knoten, der ein Instanzelement repräsentiert, also alle Knoten der Typen *Type_i*, *Atomic_i*, *Complex_i*, *CTypeAttr_i*, *Aggr_i* und *Assoc_i* haben eine Kante

- vom Typ *inst_of*, mit der sie mit ihrem zugehörigen Schemaelement verbunden sind.
- Mit Ausnahme des Typs *Complex_i* haben alle Instanzknoten nur eine *inst_of*-Kante.
- Zwischen den Schemaelementen und den Instanzelementen verlaufen ausschließlich Kanten vom Typ *inst_of*.
- Eigenschaften, die mit Aggregationen zusammenhängen
 - Knoten vom Typ *Aggr* haben genau eine Kante vom Typ *hasAggr* zu einem Knoten vom Typ *Complex*, mit der sie mit dem aggregierenden komplexen Typ verbunden sind.
 - Knoten vom Typ *Aggr* haben genau eine Kante vom Typ *aggr* zu einem Knoten vom Typ *Type*, mit denen sie mit dem aggregierten Typ verbunden sind.
 - Knoten vom Typ *Aggr_i* haben genau eine Kante vom Typ *hasAggr_i* zu einem Knoten vom Typ *Complex_i*, mit der sie mit einer Instanz des aggregierenden komplexen Typ verbunden sind.
 - Knoten vom Typ *Aggr_i* haben Kanten vom Typ *aggr_i* zu Knoten vom Typ *Type_i*, mit denen sie mit je einer Instanz des aggregierten Typs verbunden sind. Die Anzahl dieser Kanten hängt vom Wert des Attributes *card* des *Aggr*-Knotens ab, mit dem der *Aggr_i*-Knoten über seine (eindeutige, s.o.) *inst_of*-Kante verbunden ist. Ist der Wert *one*, so gibt es genau eine solche Kante. Andernfalls können es mehrere sein. Man beachte, daß die Beziehung der *Aggr_i*-Knoten zu den *Type_i*-Knoten eine andere Kardinalität haben kann, als die Beziehung der zugehörigen *Aggr*- und *Type*-Knoten. Dies wird bei der Beschreibung der Restrukturierungs-Transformationen als Graphersetzungsgesetze Abschnitt 6.2 zu beachten sein.
- Eigenschaften, die mit Assoziationen zusammenhängen
 - Knoten vom Typ *Assoc* sind mit genau einer Kante vom Typ *hasAssoc* mit einem Knoten vom Typ *Complex* verbunden.
 - Knoten vom Typ *Assoc* sind mit genau einer Kante vom Typ *assoc* mit einem anderen Knoten vom Typ *Assoc* verbunden.
 - Knoten vom Typ *Assoc_i* sind mit genau einer Kante vom Typ *hasAssoc_i* mit einem Knoten vom Typ *Complex_i* verbunden.
 - Knoten vom Typ *Assoc_i* sind mit Kanten vom Typ *assoc_i* mit je einem anderen Knoten vom Typ *Assoc_i* verbunden. Die Anzahl dieser Kanten hängt vom Wert der Attribute *total* und *functional* des *Assoc*-Knotens ab, mit dem der *Assoc_i*-Knoten über seine (eindeutige, s.o.) *inst_of*-Kante verbunden ist. Ist *total=true*, so ist es mindestens eine. Ist *functional=true*, so ist es höchstens eine. Man beachte auch hier die unterschiedliche Kardinalität der Beziehung zwischen den *Assoc_i*-Knoten verglichen mit der Kardinalität der Beziehung zwischen den *Assoc*-Knoten.

5 Abbildung konkreter Datenmodelle auf das Migrationsmetamodell

Um die in dieser Arbeit vorgestellten Redesign-Transformationen auf das Schema einer Datenbank anwenden zu können, muß das Schema im Migrationsmetamodell vorliegen. Dieses Kapitel beschreibt, wie ein Schema einer Datenbank, die in einem konkreten Modell - etwa dem relationalen Modell - vorliegt, automatisch ins M^3 überführt werden kann. Dies ist ein wichtiges Teilproblem des VARLET-Projektes und wird in anderen Diplomarbeiten ([Wadsack98], [Holle97]) ausführlich behandelt.

Zunächst muß das Schema der Ursprungsdatenbank in eine graphische Repräsentation gebracht werden. Bei relationalen Datenbanken etwa kann dies geschehen, indem Relationen und ihre Attribute jeweils durch Knoten dargestellt werden. Ein Knoten, der eine Relation repräsentiert, wird dann durch Kanten mit den Knoten für die Attribute dieser Relation verbunden. Bekannte Abhängigkeitsbeziehungen (vgl. Abschnitt 3.1) zwischen einzelnen Attributen können durch Kanten zwischen den Attributsknoten dargestellt werden.

Diese graphische Repräsentation des Schemas wird nun in eine graphische Repräsentation im M^3 überführt. Diese Überführung geschieht durch die sogenannte *initiale Abbildung*. Zur Spezifikation werden sogenannte Tripelgraphgrammatiken ([Lef95], [Schürr94]) verwendet. Diese ermöglichen es, zwei Dokumente (hier: das relationale Schema und das M^3 -Schema jeweils in graphischer Repräsentation) durch ein drittes Dokument, den sogenannten Korrespondenzgraphen, zu integrieren.

Der Zusammenhang zwischen den Komponenten des Quellschemas mit den Komponenten des Schemas im M^3 wird dann mithilfe des Korrespondenzgraphen beschrieben. Die beiden Schemata liegen dann nach der initialen Abbildung zusammen mit dem Korrespondenzgraphen in einer einzigen, aus den drei Teilen bestehenden Graphstruktur vor. Abbildung 5 zeigt das Schema eines solchen Graphen in einer an OMT ([RBPEL91]) angelehnten Notation.

Durch diese Konstruktion ist gewährleistet, daß der Zusammenhang zwischen den Komponenten des Quellschemas mit den Komponenten des Schemas nach der initialen Abbildung im M^3 nicht verlorengeht. Betrachtet man nur die Restrukturierungstransformationen, ist dieser Zusammenhang nicht erforderlich. Er ermöglicht jedoch ein weiteres wichtiges Leistungsmerkmal der VARLET-Umgebung: die inkrementelle Nachtransformation des Ursprungsschemas.

Da der Prozeß der Restrukturierung des Schemas einige Zeit andauern kann, kann es in der Praxis vorkommen, daß das Ursprungsschema während dieser Zeit verändert wird. Auch könnten Erkenntnisse aus dem Restrukturierungsprozeß eine solche Änderung wünschen lassen. Die inkrementelle Nachtransformation stellt eine Möglichkeit bereit, diese Änderungen am Ursprungsschema auf das transformierte Schema im M^3 zu propagieren. Auch eine Rückpropagation von Änderungen im M^3 in das Ursprungsschema ist möglich.

Daher ist es mit VARLET auch möglich, auf eine Datenbank eine Sicht im M^3 anzubieten, ohne die Daten zu migrieren. Der Korrespondenzgraph stellt dann die Beziehung zwischen der Sicht und der physikalischen Datenbasis her. Dies kann zur Föderierung von Datenbanken eingesetzt werden. In [Schalldach98] wird gezeigt, wie man den Korrespondenzgraphen benutzt, um eine Datenmigrationsmiddleware zu generieren, die es ermöglicht, Daten zur Laufzeit aus einer objektorientierten Sicht in eine relationale Datenbank zu überführen.

6 Restrukturierungstransformationen

In diesem Kapitel werden Aufbau, Beschreibungsformen und Eigenschaften der Restrukturierungstransformationen beschrieben. Im nächsten Kapitel wird dann ein Katalog von Transformationen angegeben, mit dem die Restrukturierung des Schemas durchgeführt werden kann.

6.1 Eigenschaften von Transformationen

Die Redesign-Transformationen, die in dieser Arbeit definiert werden, sollen ein Schema im M^3 umstrukturieren. Dabei soll es möglich sein, die Transformationen dahingehend zu klassifizieren, ob durch sie das Schema Information verliert oder nicht. Um eine Transformation bezüglich dieser Eigenschaft bewerten zu können, ist es notwendig, zu definieren, was „Informationen verlieren“ für ein Schema bedeutet. Dazu werden Begriffe aus [Hainaut91] und [Tresch95] herangezogen (vgl. auch Kapitel 2).

Transformationen und Kapazität.

Eine Restrukturierungstransformation beschreibt eine Änderung eines Schemas in dem in Kapitel 4 definierten Migrationsmetamodell. Um Eigenschaften solcher Transformationen nachzuweisen, reicht jedoch die ausschließliche Betrachtung des Schemas nicht aus. Es ist sicherzustellen, das eine zum ursprünglichen Schema gehörige Datenbasis sich in das restrukturierte Schema transferieren läßt. Daher muß eine Restrukturierungstransformation zusätzlich zu den Umformungen des Schemas auch angeben, was mit möglichen Instanzen der umgeformten Schemaelemente zu geschehen hat. Dazu kann der Begriff der Kapazität eines Datenbankschemas benutzt werden, der wie folgt definiert ist:

Es sei S^* die Menge aller Schemata in einem Datenmodell, hier dem M^3 . $S \in S^*$ sei ein solches Schema. Die zu S gehörige Datenbasis wird mit $\delta(S)$ bezeichnet. Die Menge aller möglichen Datenbasen ist die *Kapazität* von S . Sie wird mit $\kappa(S)$ bezeichnet.

Eine Redesign-Transformation beschreibt nun eine Änderung von S , die gleichzeitig Auswirkungen auf die Datenbasis $\delta(S)$ hat. Formal beschreibt eine Redesign-Transformation r also zwei Abbildungen (s, i) :

- $s: S^* \rightarrow S^*$, $s(S) = S'$ ist die Schemaabbildung oder auch *Strukturtransformation*.
- $i: \kappa(S) \rightarrow \kappa(S')$ ist die Abbildung der Datenbasis oder auch *Instanztransformation*.

Es besteht also eine direkte Abhängigkeit zwischen Struktur- und Instanztransformation, nämlich bestimmt die Strukturtransformation den Definitions- und Wertebereich der Instanztransformation.

Mit dieser Definition einer Redesign-Transformation lassen sich weitere Eigenschaften definieren:

- Ist i eine injektive Abbildung, so ist i eine *verlustfreie* Instanztransformation, da die Zustände der Ausgangsdatenbank eindeutig in einen Zustand der Zieldatenbank abgebildet werden.
- Ist i eine surjektive Abbildung, so ist i eine *vollständige* Instanztransformation, da jeder Zustand der Zieldatenbank aus einem Zustand der Ausgangsdatenbank erreicht werden kann.

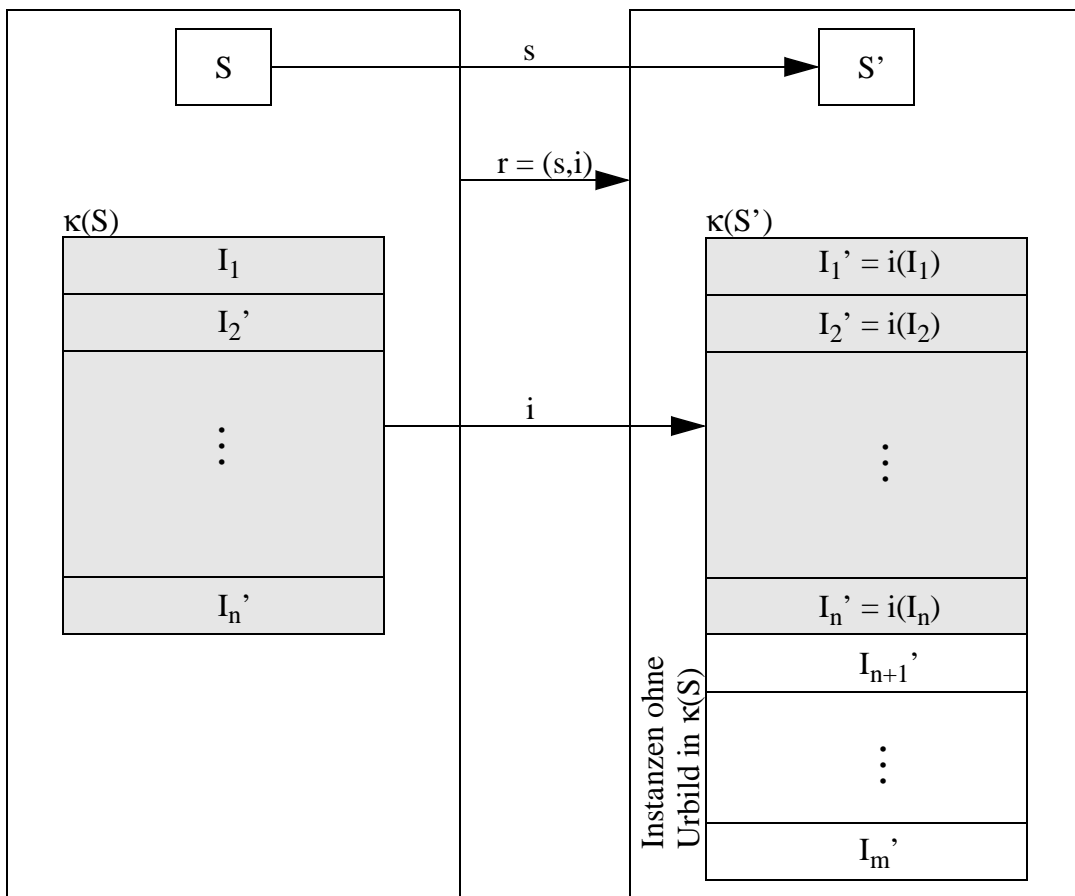
Damit können Eigenschaften von Schemata definiert werden:

- Ein Schema S' ist strukturell dominant gegenüber einem Schema S , wenn es eine verlustfreie Instanztransformation $i: \kappa(S) \rightarrow \kappa(S')$ gibt.
- Ein Schema S' ist strukturell äquivalent mit einem Schema S , wenn es eine verlustfreie und vollständige Instanztransformation $i: \kappa(S) \rightarrow \kappa(S')$ gibt.

Die folgende Definition gibt schließlich die wichtigsten Eigenschaften an, nach denen die Transformationen klassifiziert werden können:

- Eine Redesign-Transformation $r = (s, i)$ ist *kapazitätserhaltend*, wenn i bijektiv, d. h., verlustfrei und vollständig ist. Ein Schema, auf daß diese Transformation angewendet wird, ist strukturell äquivalent zum resultierenden Schema.
- Eine Redesign-Transformation $r = (s, i)$ ist *kapazitätserweiternd*, wenn i injektiv, aber nicht surjektiv, d. h., verlustfrei aber nicht vollständig ist. Das Schema nach der Transformation ist strukturell dominant zum Schema vor der Transformation.
- Eine Redesign-Transformation $r = (s, i)$ ist *kapazitätsreduzierend*, wenn i surjektiv, aber nicht injektiv, d. h., vollständig, aber nicht verlustfrei ist. Das Schema vor der Transformation ist strukturell dominant zum Schema nach der Transformation.
- Eine Redesign-Transformation $r = (s, i)$ ist *kapazitätsändernd*, wenn i weder surjektiv noch injektiv, d. h., weder vollständig noch verlustfrei ist. Das Schemata vor und nach der Transformation sind dann hinsichtlich ihrer Kapazität nicht vergleichbar.

Die folgende Abbildung verdeutlicht die Anwendung einer kapazitätserweiternden Transformation auf die Kapazität eines Schemas: Für das Schema nach der Transformation ist die Menge der möglichen Instanzen größer.


Abbildung 7: Kapazitätserweiterung

Transitivität.

Ein weiteres wichtiges Merkmal der oben definierten Redesign-Transformationen ist, daß die genannten Eigenschaften transitiv bezüglich der Hintereinanderausführung mehrerer Transformationen sind. So gilt etwa für zwei kapazitätserhaltende Transformationen $r_1 = (s_1, i_1)$ und $r_2 = (s_2, i_2)$, daß ihre Hintereinanderausführung $r = r_1 \circ r_2$ auf ein Schema S ebenfalls kapazitätserhaltend ist. Dies folgt unmittelbar aus der Konstruktion der Hintereinanderausführung. Diese ist nämlich $r = (s, i) = (s_1 \circ s_2, i_1 \circ i_2)$. Da i_1 und i_2 bijektiv sind, ist auch $i_1 \circ i_2$ bijektiv und somit r kapazitätserhaltend. Analoges gilt auch für die Hintereinanderausführung kapazitätserhalterweiternder und kapazitätsreduzierender Transformationen. Kombiniert man kapazitätserhaltende Transformationen mit kapazitätserweiternden oder kapazitätsreduzierenden, so gilt für die Hintereinanderausführung nur immer die schwächste Eigenschaft.

Die Transitivitätseigenschaft läßt sich ausnutzen, um aus einfachen Transformationen komplexere zu bauen. Es reicht dann aus, die Kapazitätseigenschaften nur für eine Menge von einfachen Transformationen nachzuweisen. Für die komplexeren gelten dann die Eigenschaften, die sich aus der Hintereinanderausführung ergeben.

6.2 Aufbau und formale Beschreibung einer Transformation

Wie im vorigen Abschnitt erläutert, besteht eine Redesign-Transformation aus zwei Teilen, der *Strukturtransformation* und der *Instanztransformation*. Die Strukturtransformation formt das Schema um; darauf Bezug nehmend beschreibt die Instanztransformation, wie die Datenbasis der Ausgangsdatenbank in die neue überführt wird.

In [Hainaut91] erfolgt die Beschreibung sowohl der Struktur- als auch der Instanztransformationen in einer algebraischen Notation im Kalkül des dort verwendeten relationalen Datenmodells (vgl. Kap. 2). Auch im Migrationsmetamodell ist eine solche Beschreibung möglich.

Dazu kann die algebraische Definition eines Schemas im M^3 aus Kapitel 4 benutzt werden. Wie in [Hainaut91] enthält die Strukturtransformation eine Vor- und eine Nachbedingung. Das Schema muß die Vorbedingung erfüllen, damit die Transformation anwendbar ist. Die Nachbedingung beschreibt dann das Schema nach der Transformation.

Die Notation dieser Bedingungen kann im Kalkül der in Abschnitt 4.2 gegebenen Definition des M^3 erfolgen. Die Vorbedingung beschreibt ein Teilschema S' . Die Aussage der Vorbedingung ist nun, daß das zu transformierende Gesamtschema S das Teilschema S' enthalten muß. Eine Transformation auszuführen bedeutet dann, das Teilschema S' aus dem Gesamtschema zu entfernen und dafür ein anderes Teilschema S'' , welches durch die Nachbedingung beschrieben wird, einzufügen.

Die Instanztransformation kann mit Hilfe der Interpretation eines Schemas im M^3 (vgl. Abschnitt 4.3) angegeben werden. Auch hier können in Form einer Vor- und Nachbedingung die Instanzmengen der von der Strukturtransformation veränderten Schemaelemente beschrieben werden.

Diese algebraische Darstellungsform hat einige Schwächen. Sie ist wenig anschaulich und daher recht aufwendig für jede Transformation zu erarbeiten. Auch in [Hainaut91] werden bildliche Darstellungen des Schemas vor und nach einer Transformation zur Illustration verwendet. Darüber hinaus ist das M^3 komplexer als das in [Hainaut91] verwendete relationale Modell, wodurch die Darstellung noch unübersichtlicher wird.

Daher werden die Restrukturierungstransformationen in dieser Diplomarbeit mit Hilfe eines graphentechnischen Ansatzes, nämlich Graphersetzungsregeln, beschrieben. Im folgenden werden Definitionen einiger grundlegender graphentechnischer Begriffe gegeben:

Graphen.

Grundlage aller folgenden Definitionen sind knoten- und kantentypisierte Graphen. Ein solcher Graph besteht aus einer Menge von Knoten, einer Menge von Kanten, zwei Funktionen, die jeweils den Knoten und Kanten einen Typ aus einer festen Menge von Knoten- bzw. Kantentypen zuordnen, sowie zwei weiteren Funktionen, die jeder Kante einen Quell- bzw. einen Zielknoten zuordnen. Formal ist ein solcher Graph ein 6-Tupel $G = (V, E, t_V, t_E, s, t)$ mit:

- $V(G) = V$ ist die endliche Menge der *Knoten* von G
- $E(G) = E$ ist die endliche Menge der *Kanten* von G
- $t_V: V \rightarrow T_V$ ist die Knotentypisierungsfunktion

- $t_E: E \rightarrow T_E$ ist die Kantentypisierungsfunktion
- $s: E \rightarrow V$ ordnet jeder Kante einen Quellknoten zu.
- $t: E \rightarrow V$ ordnet jeder Kante einen Zielknoten zu.

Für die folgenden Definitionen seien $G' = (V', E', t_V', t_E', s', t')$ und $G = (V, E, t_V, t_E, s, t)$ zwei solche Graphen.

Teilgraphen.

G' ist ein Teilgraph vom G , wenn gilt:

- Die Knotenmenge von G' ist eine Teilmenge der Knotenmenge von G : $V' \subseteq V$
- Die Kantenmenge von G' ist eine Teilmenge der Kantenmenge von G : $E' \subseteq E$
- Die Quellknoten aller Kanten von G' sind identisch mit denen der entsprechenden Kanten von G : $\forall e \in E': s'(e) = s(e)$
- Die Zielknoten aller Kanten von G' sind identisch mit denen der entsprechenden Kanten von G : $\forall e \in E': t'(e) = t(e)$
- Die Typen aller Knoten von G' sind identisch mit den Typen der entsprechenden Knoten in G : $\forall v \in V': t_V'(v) = t_V(v)$
- Die Typen aller Kanten von G' sind identisch mit den Typen der entsprechenden Kanten in G : $\forall e \in E': t_E'(e) = t_E(e)$

Graphmorphismen.

Ein Paar von Funktionen $f = (f_V, f_E)$, $f_V: V' \rightarrow V$, $f_E: E' \rightarrow E$ ist ein *Graphmorphismus* zwischen G' und G , wenn es Quell- und Zielknoten der Kanten sowie die Typen von Knoten und Kanten erhält:

- $\forall e \in E': s'(e) = s(f_E(e))$
- $\forall e \in E': t'(e) = t(f_E(e))$
- $\forall v \in V': t_V'(v) = t_V(f_V(v))$
- $\forall e \in E': t_E'(e) = t_E(f_E(e))$

Sind f_V und f_E bijektiv, so ist der Graphmorphismus auch ein *Graphisomorphismus*. Ist f_V oder f_E nicht injektiv, gibt es also zwei verschiedene Knoten v_1 und v_2 oder zwei verschiedene Kanten e_1 und e_2 in G' , für die $f_V(v_1) = f_V(v_2)$ bzw. $f_E(e_1) = f_E(e_2)$ gilt, so heißen ihre Bilder $f_V(v_1)$ bzw. $f_E(e_1)$ *identifiziert*.

Graphersetzungsregeln.

Analog zu den aus der Theorie der Stringgrammatiken bekannten Stringproduktionen lassen sich auch Produktionen für Graphen, sogenannte *Graphproduktionen* oder *Graphersetzungsregeln* definieren. In einer Stringgrammatik besteht eine Produktion (u, v) aus zwei Strings u und v . Die Anwendung einer solchen Produktion auf einen String w bedeutet, in w einen Teilstring zu finden, der mit u identisch ist und ihn durch v zu ersetzen.

Eine Graphersetzungsregel ist ähnlich aufgebaut. Aufgrund der komplexeren Struktur von Gra-

phen reicht jedoch die einfache Angabe eines zu entfernenden und eines dafür einzufügenden Teilgraphen nicht aus. Es muß darüber hinaus beschrieben werden, wie die Ersetzung durchzuführen ist. Dazu ist es im allgemeinen notwendig, einige Elemente des Graphen, die von der Regel nicht entfernt werden, als Kontext zu betrachten, und anzugeben, wie die zu entfernenden Elemente des Graphen mit diesen Elementen zusammenhängen. Dies kann mithilfe von Graphmorphismen geschehen.

Eine *Graphersetzungsregel* $p = (L, R, K, l, r)$ besteht formal auf folgenden Komponenten:

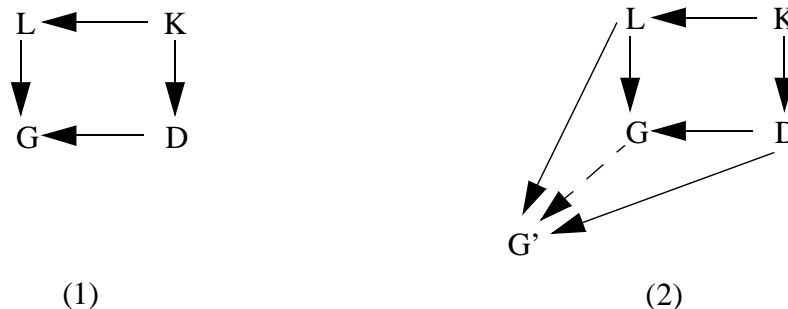
- L und R sind Graphen, die linke und die rechte Seite der Regel
- K ist ein Graph
- $l: K \rightarrow L$ und $r: K \rightarrow R$ sind injektive Graphmorphismen

Die Graphen $L_K = l(K) \subseteq L$ und $R_K = r(K) \subseteq R$ sind zu K isomorph.

Anwendung einer Graphersetzungsregel.

Zur Durchführung der Graphersetzung ist der Begriff des *gluing* erforderlich: Für zwei Graphmorphismen $K \rightarrow L$ und $K \rightarrow D$ bildet ein Graph G zusammen mit zwei Graphmorphismen $L \rightarrow G$ und $D \rightarrow G$ das *gluing von L und D über K* , wenn folgende Bedingungen erfüllt sind:

- Kommutativität: $K \rightarrow L \rightarrow G = K \rightarrow D \rightarrow G$
- Universalität: Für alle Graphen G' und Graphmorphismen $L \rightarrow G'$ und $D \rightarrow G'$, die die Bedingung $K \rightarrow L \rightarrow G' = K \rightarrow D \rightarrow G'$ erfüllen, gibt es genau einen Graphmorphismus $G \rightarrow G'$, so daß $L \rightarrow G \rightarrow G' = L \rightarrow G'$ und $D \rightarrow G \rightarrow G' = D \rightarrow G'$. Das folgende Diagramm verdeutlicht die Situation:

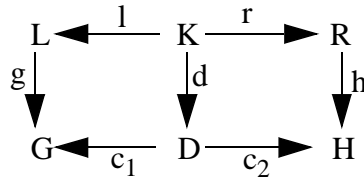


Wenn (1) die obigen Bedingungen erfüllt, wird es auch *gluing diagram* oder *pushout* genannt. Es gilt dann sogar, daß G und G' isomorph sind.

Damit kann nun die Anwendung einer Graphersetzungsregel definiert werden (vgl. [EKL91]):

Gegeben sei eine Graphersetzungsregel $p = (L, R, K, l, r)$, ein Graph G und ein Morphismus $g: L \rightarrow G$. $D = G - g(K)$ ist der Graph, der aus G entsteht, wenn man aus ihm die Elemente entfernt, deren Urbilder bezüglich g in K liegen. D heißt dann auch *Kontextgraph*. Wenn nun für l und $d: K \rightarrow D$ der Graph G zusammen mit den Morphismen g und $c_1: D \rightarrow G$ ein *gluing* bilden, so ist p durch g auf G anwendbar. Der Teilgraph von G , auf den L abgebildet wird, wird dann auch *Match* genannt.

Zur Ausführung der Regel kann dann über einen Morphismus $h: R \rightarrow H$ ein weiteres *gluing* konstruiert werden. Die folgende Abbildung verdeutlicht den Zusammenhang zwischen den beteiligten Graphen und Morphismen:



Bei der Anwendung einer Graphersetzungsregel kann es vorkommen, daß danach „baumelnde“ Kanten entstehen, also Kanten, denen kein Quell- oder Zielknoten zugeordnet ist. Um dies zu vermeiden, müssen folgende Bedingungen erfüllt sein:

- Objekte (Knoten und Kanten), die von g auf ein und dasselbe Objekt in G abgebildet werden, dürfen nicht entfernt werden, müssen also auch in K enthalten sein:
 - $\forall v_1, v_2 \in V(L) : g(v_1) = g(v_2) \Rightarrow v_1, v_2 \in V(K)$
 - $\forall e_1, e_2 \in E(L) : g(e_1) = g(e_2) \Rightarrow e_1, e_2 \in E(K)$

Diese Anforderung wird in der Literatur (z. B. [EHL97]) auch *identification condition* genannt.

- Für einen Knoten, der entfernt wird, werden auch alle inzidenten Kanten entfernt:

$$\forall v \in V(L) - V(K) : \forall e \in E(L) : s(e) = v \vee t(e) = v \Rightarrow e \in E(L) - E(K)$$

Diese Anforderung ist auch als *dangling edge condition* bekannt ([EHL97]).

Aufgrund der Symmetrie der Konstruktion ist die Anwendung einer solchen Graphersetzungsregel stets umkehrbar, und zwar durch die Regel, die entsteht, wenn man in p die linke und rechte Seite vertauscht: $p^{-1} = (R, L, K, r, l)$.

Darstellung von Restrukturierungstransformationen als Graphersetzungsregel.

In Kapitel 4 wurde erläutert, wie Datenbankschemata mit ihren Instanzen graphisch als Schema-Instanz-Graph dargestellt werden können. Im folgenden soll nun gezeigt werden, wie Restrukturierungstransformationen für Datenbanken als Graphersetzungsregeln für den Schema-Instanz-Graphen definiert werden können.

Die Darstellung wird anhand der Transformation *NewComplexType* verdeutlicht: Die Transformation *NewComplexType* dient zum Aufspalten eines komplexen Typ, so daß ein neuer komplexer Typ entsteht. Der neue Typ besitzt eine 1:1-Assoziation zum ursprünglichen Typ.

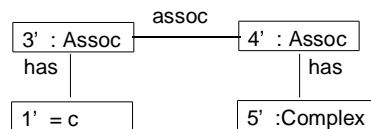
Ein einfacher Ansatz ist, nur das Schema und nicht die Instanzen zu berücksichtigen. Die Transformation *NewComplexType* läßt sich dann durch folgende Graphersetzungsregel darstellen:

transformation NewComplexType(c:Complex, newName:String, role1:String, role2:String) =

$\forall 'c \in C : 'c.name \neq newName;$

$'1 = c$

:=:



5'.name = newName

3'.name = role1

4'.name = role2

3'.total = true

4'.total = true

3'.functional = true

4'.functional = true

Zunächst einige Anmerkungen zur Notation dieser Regel: In der Kopfzeile können Parameter angegeben werden, die dann bei Anwendung der Regel mit Elementen des Wirtsgraphen identifiziert werden, wie hier ein komplexer Typ *c*. Die linke Seite der Regel fordert, daß bei Anwendung der Regel *c* als Knoten 1 verwendet wird. Als weitere Bedingung für die Anwendbarkeit wird gefordert, daß es im Wirtsgraphen keinen komplexen Typ gibt, dessen *name*-Attribut gleich dem Parameter *newName* ist. In der rechten Seite können dann Knoten der linken Seite wieder auftreten. Sie haben dann die gleiche Nummer wie auf der linken Seite. Zur Unterscheidung wird auf der linken Seite das Zeichen ' vor die Nummer geschrieben, auf der rechten Seite hinter die Nummer. Knoten, deren Nummern auf der linken Seite vorkommen, aber nicht auf der rechten, werden entfernt.

Die anderen Knoten der rechten Seite werden dem Wirtsgraphen hinzugefügt. Dabei wird dem Attribut *name* des Knoten 5' der Wert des Parameters *newName* zugewiesen. Ebenso werden die Attribute der *Assoc*-Knoten gesetzt.

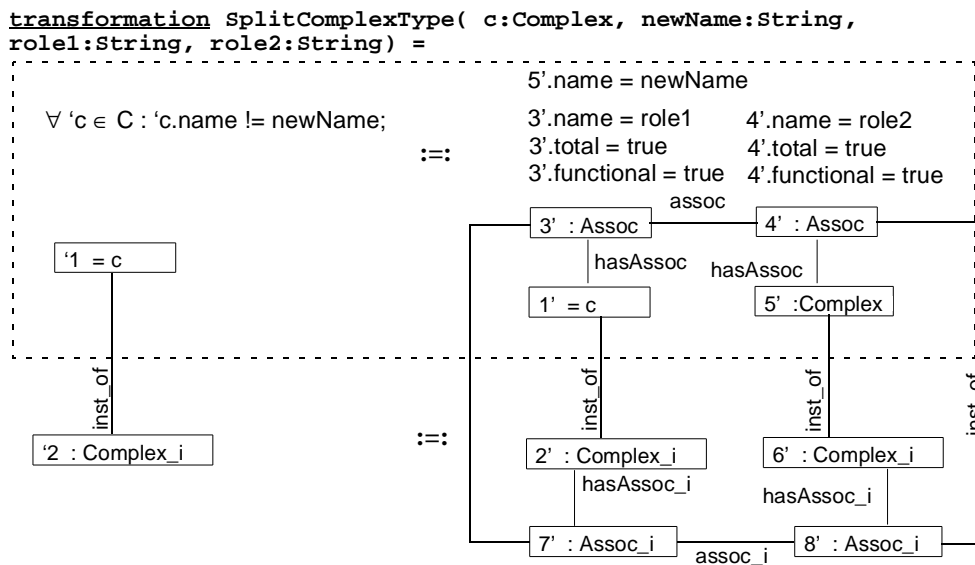
Die Notation orientiert sich an der Sprache Progres. Eine detaillierte Beschreibung dieser Sprache findet sich in [Zündorf95].

Wie bereits angedeutet, werden Instanzen des Schemas von dieser Regel in keiner Weise berücksichtigt. Um jedoch die Auswirkungen einer Transformation auf den Informationsgehalt eines Schemas untersuchen zu können, ist es von Bedeutung, auch ihre Auswirkungen auf die Instanzen zu betrachten.

Daher enthält eine Transformation zusätzlich zur Strukturabbildung noch eine Instanzabbildung, die beschreibt, wie mögliche Instanzen der von der Strukturabbildung veränderten Schemaelemente in das neu entstandene Schema zu überführen sind.

Da es zu jedem Schemaelement eine unbestimmte Anzahl von Instanzen geben kann, ist deren Beschreibung durch eine Graphersetzungsgregel nicht ohne weiteres möglich. Die nötige Erweiterung der Darstellungsform bieten *parallele Graphersetzungsgregeln*, wie sie in [Taentzer96] beschrieben werden.

Die folgende Abbildung stellt die Transformation *NewComplexType* als eine solche Regel dar.



Diese Transformation besteht aus zwei Teilen. Der obere Teil (in gestrichelten Linien eingeraht) ist die Strukturtransformation aus der vorigen Abbildung. Dazu kommt als unterer Teil die Instanztransformation.

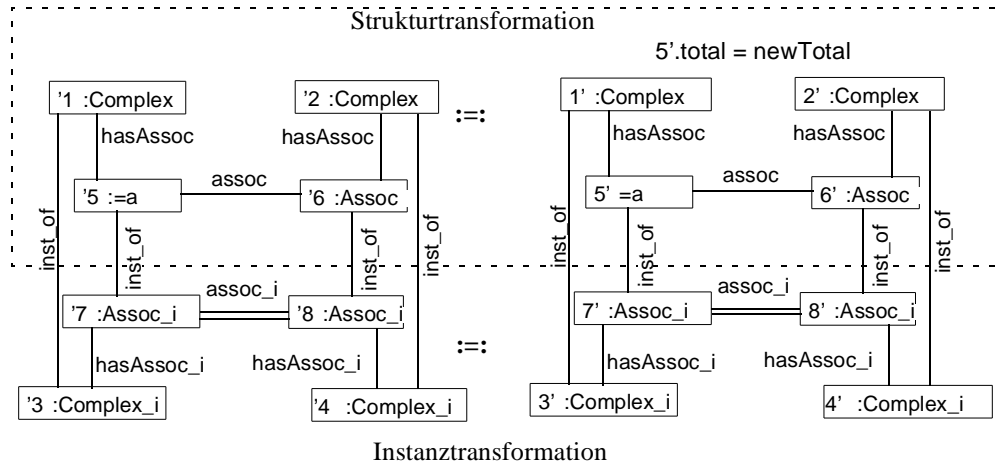
Die Besonderheit einer solchen parallelen Graphersetzungsgregel ist die Art, wie sie angewendet wird. Dies geschieht, indem im Wirtsgraphen zunächst ein Match für den oberen Teil der linken Seite gesucht wird. Dann werden alle möglichen Erweiterungen dieses Match zu Matches der gesamten linken Seite gesucht. Das heißt für die obige Regel, daß alle Knoten gesucht werden, die mit dem Knoten '1 über eine Kante vom Typ *inst_of* verbunden sind. Das sind - wie gewünscht - alle Instanzen des komplexen Typs *c*. Der durch diesen Vorgang - die sogenannte *Amalgamierung* - gefundene Teilgraph wird dann ersetzt, indem für den oberen Teil der linken Seite der obere Teil der rechten Seite eingesetzt wird, und für jedes Vorkommen des unteren Teils der linken Seite der untere Teil der rechten Seite. Die Strukturtransformation wird also nur einmal ausgeführt, die Instanztransformation für jede Instanz. Die so gewonnene Regel wird *amalgamierte* Regel genannt. Eine formale Definition der Ausführung paralleler Graphersetzungsgregeln gibt [Taentzer96].

Für einige Transformationen muß das Konzept der parallelen Graphersetzungsgregeln noch um die folgenden Konstrukte erweitert werden.

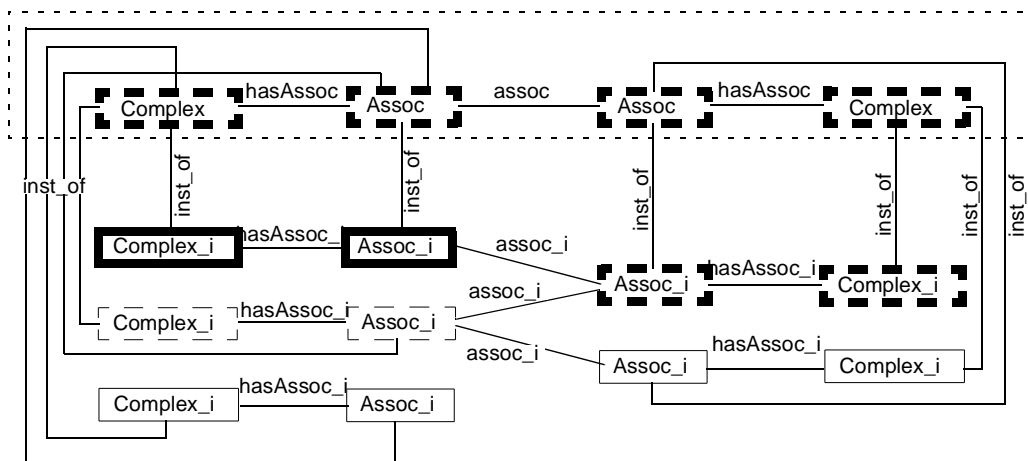
Desynchronisationskanten.

Zur Demonstration der Notwendigkeit weiterer Konstrukte sei hier die Transformation *ChangeTotality* als weiteres Beispiel angegeben. Sie ändert die Totalitätseigenschaft einer Assoziation. Man ignoriere dabei vorerst die besondere Darstellung der beiden *assoc_i*-Kanten im unteren Teil der Regel.

transformation ChangeTotality (a:Assoc; newTotal:bool; r:relation) =



Die folgende Abbildung zeigt einen Schema-Instanz-Graphen für eine linksseitig partielle Assoziation, auf den diese Transformation angewendet werden soll.



Dabei ist durch die gestrichelt und die dick gezeichneten Knoten je eine Matcherweiterung dargestellt, wie sie beim Vorgang der Amalgamierung auftreten. Dabei wird deutlich, daß es im unteren Teil Graphenelemente gibt, die an mehreren solcher Erweiterungen beteiligt sind. Dagegen werden die beiden Knoten unten links von der Amalgamierung überhaupt nicht erfaßt, da sie nicht an der Assoziation teilnehmen - was ja, da die Assoziation linksseitig partiell ist, durchaus zugelassen ist - und somit keine *assoc_i*-Kante haben.

Dennoch sollen solche Instanzen von der Transformation berücksichtigt werden. Als Lösung dafür werden daher hier sogenannte *Desynchronisationskanten* eingeführt. Diese werden in der Graphersetzungsregel als doppelte Linien dargestellt. Der Name soll ausdrücken, daß diese Kanten den Amalgamierungsvorgang in getrennte Vorgänge aufspalten, ihn also desynchronisieren.

Zur Beschreibung des Amalgamierungsvorgangs wird eine Einschränkung für die Verwendung von Desynchronisationskanten gemacht: Sie müssen den unteren Teil der Regel aufspalten, das

heißt, läßt man sie (und die *inst_of*-Kanten) weg, entstehen im unteren Teil der Regel nicht zusammenhängende Teilgraphen.

Für Graphersetzungsregeln mit Desynchronisationskanten läuft der Amalgamierungsvorgang dann in zwei Schritten ab: Zunächst wird die Desynchronisationskante entfernt und die entstehenden Teilgraphen werden getrennt amalgamiert. Dann werden alle Kanten des Wirtsgraphen, die die Desynchronisationskante „matchen“, hinzugefügt. Der so entstandene Graph ist dann das Match der Regel.

Für den obigen Schema-Instanz-Graphen bedeutet dies, daß zunächst die sechs Knoten unten links einerseits und die vier Knoten unten rechts andererseits amalgamiert werden. Danach werden die zwischen ihnen verlaufenden *assoc_i*-Kanten hinzugefügt.

Die obige Einschränkung stellt sicher, daß der Amalgamierungsvorgang wohldefiniert ist. Fordert man, daß die Desynchronisationskanten beliebig verwendet werden können, kann es eventuell zu Problemen kommen. Da aber alle in dieser Diplomarbeit definierten Transformationsregeln die Einschränkungen erfüllen, wurde dies nicht näher untersucht.

Die Desynchronisationskanten werden immer dann benötigt, wenn zwei Knotentypen auf Instanzebene bezüglich ihrer Kardinalität nicht in der gleichen Beziehung stehen, wie ihre zugehörigen Schemaknoten. Die Eigenschaften des Schema-Instanz-Graphen (vgl. Abschnitt 4.4) zeigen, für welche Fälle dies vorkommen kann: Nur die Kantentypen *aggr_i* und *assoc_i* erfordern Desynchronisationskanten.

Die Ausführung einer Regel mit Desynchronisationskanten kann im Allgemeinen nicht allein durch eine Graphersetzungsregel spezifiziert werden. Es ist zusätzlich anzugeben, wie der Wirtsgraph nach der Ausführung hinsichtlich dieser Kanten aussieht, wenn etwa die Eigenschaften einer Aggregations- oder Assoziationsbeziehung geändert werden. So ist es für die Transformation *ChangeTotality* erforderlich, wenn sie die Totalität auf *true* setzt, für einige Instanzen anzugeben, mit welchen Instanzen sie nach der Transformation in Beziehung stehen sollen, da nur so die Konsistenz des Schema-Instanz-Graphen sichergestellt werden kann. Für den obigen Schema-Instanz-Graphen muß dann eine *assoc_i* Kante für den *Assoc_i*-Knoten unten links eingefügt werden.

Die Spezifikation solcher Kanten kann mithilfe der Interpretation des Schema-Instanz-Graphen im M^3 geschehen (vgl. Abschnitt 4.3): Die Interpretation einer Aggregation bzw. einer Assoziation liefert genau die benötigten Informationen. Für Beispiele siehe etwa die Transformation *MoveAggr* in Abschnitt 7.1.

Optionale Graphenelemente und Mengenknoten.

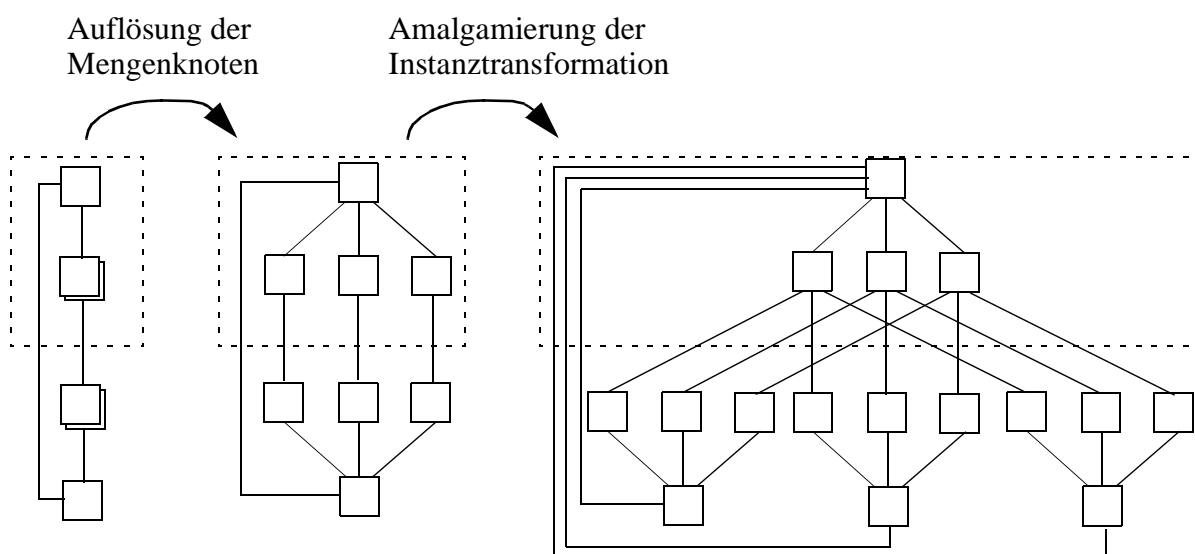
Zwei weitere Erweiterungen stellen die optionalen Graphenelemente und die Mengenknoten dar. Im Gegensatz zu den Desynchronisationskanten können sie vor der Amalgamierung aufgelöst werden und beeinflussen dann den eigentlichen Amalgamierungsvorgang nicht mehr. Sie erlauben eine allgemeinere Darstellungsform der Transformationen. Die Konzepte sind aus Progres übernommen.

Optionale Graphenelemente sind gestrichelt dargestellt. Sie können bei Anwendung der Transformation fehlen. Diese Darstellung ist also lediglich eine Kurzschreibweise, mit der es möglich

ist, zwei Transformationen - nämlich die, die entsteht, wenn man die optionalen Knoten wegläßt, und die, bei denen sie vorhanden sind - als eine einzige darzustellen. Für ein Beispiel siehe etwa die Transformation *NewSupertype* in Abschnitt 7.3.

Mengenknoten stellen eine Menge unbestimmter Größe von Knoten eines Typs dar. Sie werden als doppelt gezeichnetes Kästchen dargestellt. Sie werden zum Beispiel dann benötigt, wenn ein komplexer Typ eine unbestimmte Anzahl von Untertypen hat, und diese bei der Spezifizierung einer Transformation berücksichtigt werden müssen.

Die Amalgamierung einer Regel, in der Mengenknoten vorkommen, geschieht dann, indem zunächst die Mengenknoten aufgelöst werden und danach die Knoten der Strukturtransformation. Die folgende Abbildung verdeutlicht den Amalgamierungsvorgang.



Die Mengenknoten stellen eine Kurzschreibweise für eine unendlich große Anzahl von Regeln dar: Sie können für jede konkrete Anwendung durch eine Regel mit der entsprechenden Anzahl von Knoten ersetzt werden.

Die Kombination von Mengenknoten mit Desynchronisationskanten kann zu Problemen führen. Es wird daher verlangt, daß sie nicht gemeinsam in einer Transformationsregel verwendet werden dürfen. Dies ist für alle in dieser Arbeit spezifizierten Transformationen erfüllt.

6.3 Reversibilität von Transformation

Mit der Kapazität eines Schemas wurde in [Tresch95] ein Begriff definiert, mit dem sich die Erhaltung der Semantik einer Redesign-Transformation bewerten läßt. In [Hainaut91] wird zu diesem Zweck mit der Reversibilität ein anderer Begriff dafür definiert. Es soll nun gezeigt werden, wie sich dieser Begriff auf als Graphersetzungsregel dargestellte Transformationen des Schema-Instanz-Graphen übertragen läßt. Dabei kann auch gezeigt werden, wie die Begriffe Reversibilität und Kapazität zusammenhängen.

Zur Definition der Reversibilität wird in Analogie zu [Hainaut91] zu einer Transformation eine

Umkehrtransformation konstruiert:

Eine als Graphersetzungsregel gegebene Transformation $r = (s, i)$ ist *reversibel*, wenn es eine andere Transformation $r^{-1} = (s^{-1}, i^{-1})$ - die Umkehrtransformation gibt, die die Datenbank in ihren Zustand vor der Transformation zurückführt. Formal bedeutet das:

- r , als Funktion auf der Menge G^* aller Schema-Instanz-Graphen, ist injektiv.
- r^{-1} ist eine totale Funktion auf der Menge $r(G^*)$, also auf der Menge aller Schema-Instanz-Graphen, die durch Anwendung von r entstehen können, kann also auf jeden dieser Graphen angewendet werden.
- Die Hintereinanderausführung von r und r^{-1} ergeben die Identitätsfunktion auf G^* , also $r^{-1} \circ r = id_{G^*}$

r^{-1} ist dann die Umkehrtransformation zu r .

Die Umkehrtransformation zu einer Redesign-Transformation kann durch Austauschen der linken und rechten Seite konstruiert werden kann. Dies geschieht in Analogie zu [Hainaut91], wo zur Konstruktion der Umkehrtransformation die Vor- und Nachbedingungen, mit denen dort Transformationen beschrieben werden, ausgetauscht werden.

Die Reversibilität einer Transformation stellt sicher, daß eine mit ihr transformierte Datenbank wieder in ihren Ursprungszustand zurückgeführt werden kann. Allerdings ist damit noch nicht gewährleistet, daß die Transformation auch kapazitätserhaltend ist, denn dazu ist es ja erforderlich, daß *jede mögliche* Instanz des transformierten Schemas sich in das Schema vor der Transformation zurückführen läßt, und nicht nur die, die durch die Transformation erzeugt wurde.

Anders formuliert: Wenn r eine reversible Transformation ist, die ein Schema S mit der Kapazität $\kappa(S)$ in ein Schema S' mit der Kapazität $\kappa(S')$ transformiert, so gilt:

- jede mögliche Instanz $I \in \kappa(S)$ von S kann in eine Instanz $I' \in \kappa(S')$ überführt werden
- jede Instanz von S' , die aus der Anwendung von r auf S aus einer Instanz von S entstanden ist, kann wieder in eine Instanz von S zurücktransformiert werden (durch r^{-1}).

Wichtig ist dabei die Einschränkung im zweiten Punkt. Die Rücktransformierbarkeit ist nur für Instanzen gewährleistet, die durch Anwendung der Transformation entstanden sind. Die Instanzabbildung ist aber nicht notwendigerweise surjektiv.

Um die Rücktransformierbarkeit für jede mögliche Instanz von S' zu gewährleisten, muß eine Transformation folgende weitere Anforderung erfüllen: Eine reversible Transformation r mit Umkehrtransformation r^{-1} ist *symmetrisch reversibel*, wenn auch r^{-1} reversibel ist.

Der Zusammenhang von Reversibilität und Kapazität läßt sich also zusammenfassend so formulieren:

- Eine Transformation, die reversibel ist, aber nicht symmetrisch reversibel, ist kapazitätserweiternd.
- Eine symmetrisch reversible Transformation ist kapazitätserhaltend.

7 Katalog der Transformationen

In diesem Kapitel werden die Restrukturierungstransformationen als Graphersetzungsregel spezifiziert und hinsichtlich ihrer Auswirkungen auf die Kapazität eines Schemas untersucht.

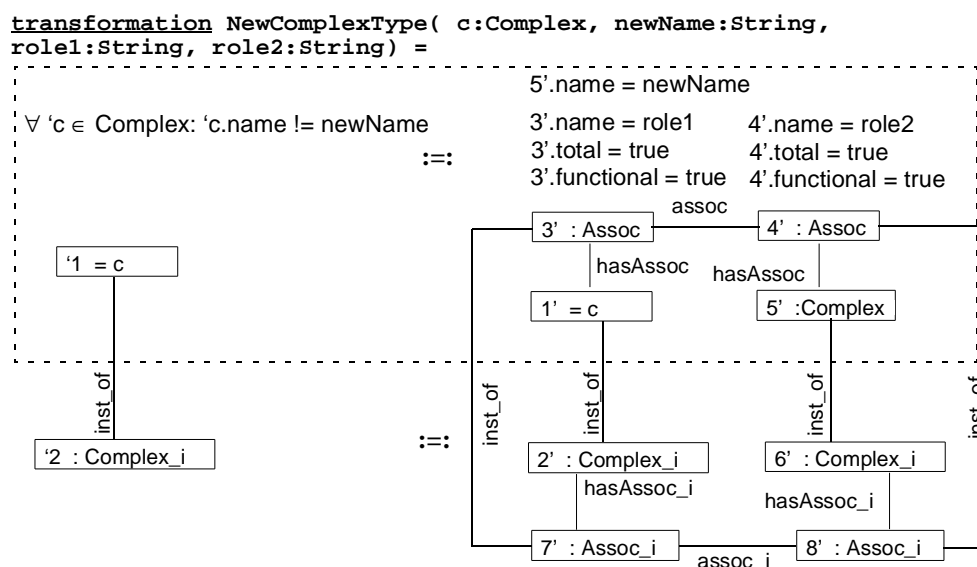
Die Transformationen lassen sich grob in drei Gruppen einteilen. Die erste Gruppe transformiert Typen, etwa, indem sie komplexe Typen aufspaltet oder Attribute zwischen komplexen Typen verschiebt. Die zweite Gruppe transformiert die Beziehungen zwischen Typen, wie zum Beispiel die Kardinalität einer Aggregation, oder durch Einfügen einer Assoziation. Die dritte Gruppe transformiert Vererbungsbeziehungen, etwa durch Einfügen eines neuen Ober- oder Untertyps oder das Verschieben eines Attributs in einen Ober- oder Untertyp.

Zur Untersuchung der Auswirkungen einer Transformation auf die Kapazität eines Schemas kann auch untersucht werden, ob sie und ihre Umkehrtransformation reversibel ist. Über den in Abschnitt 6.3 gezeigten Zusammenhang der Begriffe Kapazität und Reversibilität lassen sich daraus ihre Kapazitätseigenschaften herleiten.

Die hier vorgestellten Transformationen beschreiben jeweils nur eine sehr begrenzte Änderung, so daß sich ihre Auswirkungen recht leicht überschauen lassen. Aus diesen atomaren Transformationen lassen sich dann durch eine geeignete Hintereinanderausführung komplexere Kommandos konstruieren. Da die Kapazitätseigenschaften der Transformationen transitiv sind (vgl. Abschnitt 6.1), können auch für die zusammengesetzten Transformationen Aussagen über ihre Auswirkungen auf die Kapazität eines Schemas hergeleitet werden. Am Ende des Kapitels wird dies beispielhaft demonstriert.

7.1 Transformation von Typen

Aufspalten eines komplexen Typ.



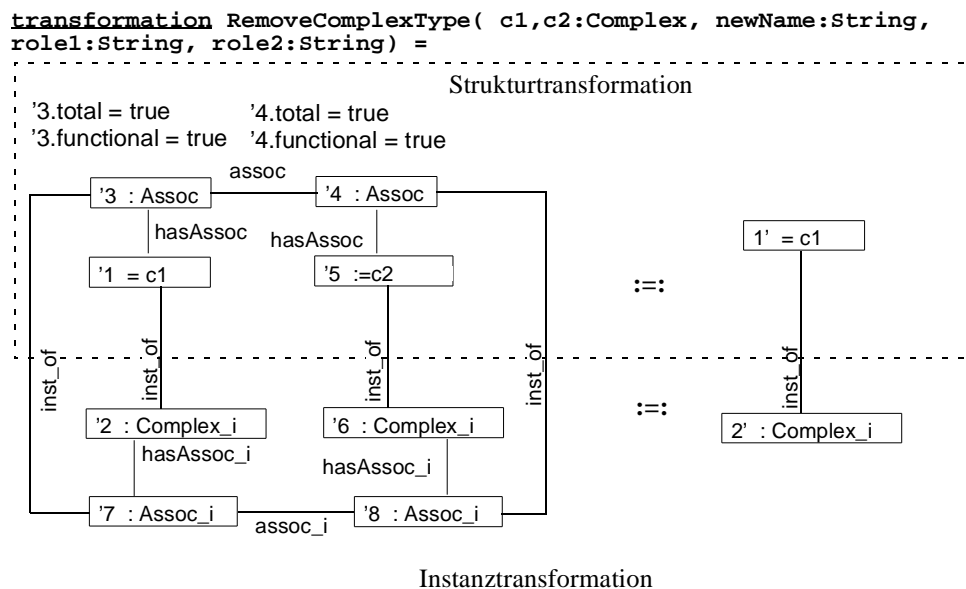
Die Transformation *NewComplexType* spaltet einen komplexen Typ auf, so daß ein neuer komplexer Typ entsteht. Der neue Typ besitzt eine 1:1-Assoziation zum ursprünglichen Typ. Er steht zunächst nicht in Beziehung zu irgendetwelchen anderen Elementen des Schemas, hat insbesondere noch keine Aggregationen oder Assoziationen. In der Regel wird der Redesigner dem Typ durch Anwendung anderer Transformationen welche hinzufügen.

Falls keine 1:1 Assoziation gewünscht wird, kann die Assoziation danach mit den Transformationen *ChangeTotality* oder *ChangeFunctionality* geändert werden.

Da eine 1:1-Assoziation eingefügt wird, kann die *assoc_i*-Kante in diesem Fall mit einer einfachen Kante statt einer Desynchronisationskante spezifiziert werden.

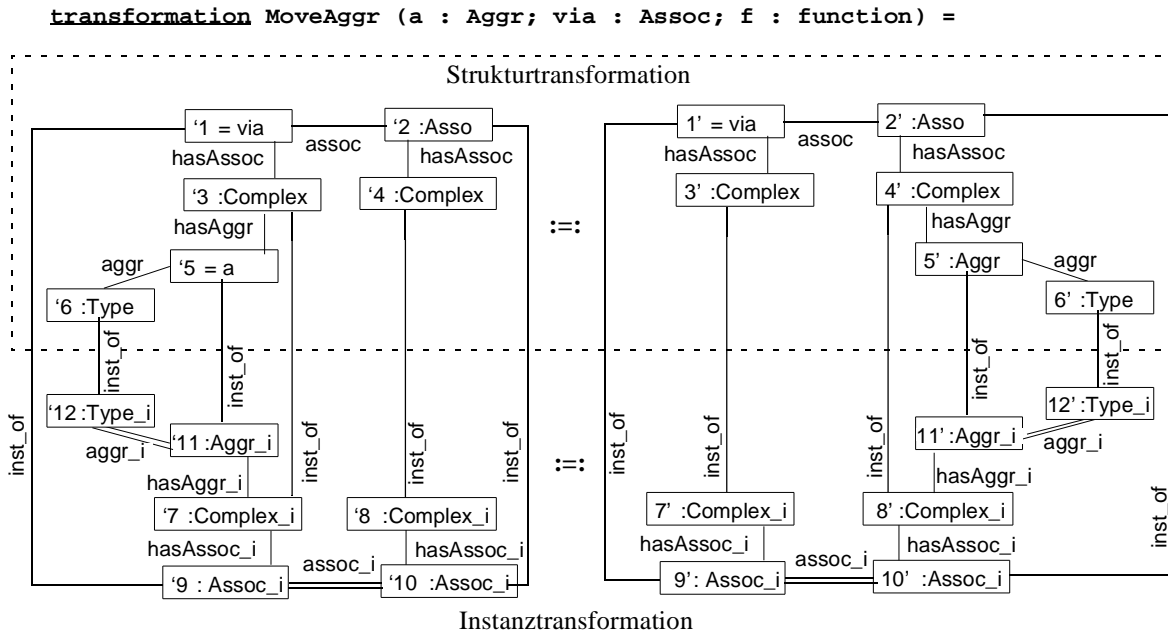
Die Transformation ist reversibel: Nach der Transformation entsteht ein neuer komplexer Typ, der keine Attribute hat, daher also keine Daten speichern kann. Aus diesem Grund ist auch die Umkehrtransformation reversibel. Somit ist *NewComplexType* symmetrisch reversibel und daher kapazitätserhaltend.

Zusammenführen zweier komplexer Typen.



Die Transformation *RemoveComplexType* vereinigt einen komplexen Typen, der keine Attribute mehr hat, mit einem anderen komplexe Typen, mit dem er über eine 1:1-Assoziation verbunden ist. Die Transformation ergibt sich direkt aus der Umkehrung der Transformation *NewComplexType*. Da *NewComplexType* kapazitätserhaltend - und damit symmetrisch reversibel ist, ist auch *RemoveComplexType* als ihre Umkehrtransformation kapazitätserhaltend.

Verschieben einer Aggregation entlang einer Assoziation.



Die Transformation **MoveAggr** verschiebt eine Aggregation von einem komplexen Typ in einen anderen. Die beiden komplexen Typen müssen dazu über eine Assoziation verbunden sein. Diese Transformation ist insbesondere auch im Zusammenhang mit der Transformation *NewComplexType* nützlich. Nach Anwendung von *NewComplexType* können einzelne Aggregationen in den neuen komplexen Typ verschoben werden.

Über die Art der Assoziation, über die die Aggregation verschoben wird, werden zunächst keine weiteren Annahmen gemacht. Sie hat jedoch Auswirkungen auf die Kapazitätseigenschaften der Transformation. Im Allgemeinen ist die Transformation kapazitätsändernd.

Da die Assoziation selbst nicht verändert wird, ist die Angabe ihrer Interpretation für diese Transformation nicht erforderlich, da sie sich ja ebenfalls nicht ändert. Es gilt also: $I_{Assoc}(1') = I_{Assoc}(1)$ und $I_{Assoc}(2') = I_{Assoc}(2)$.

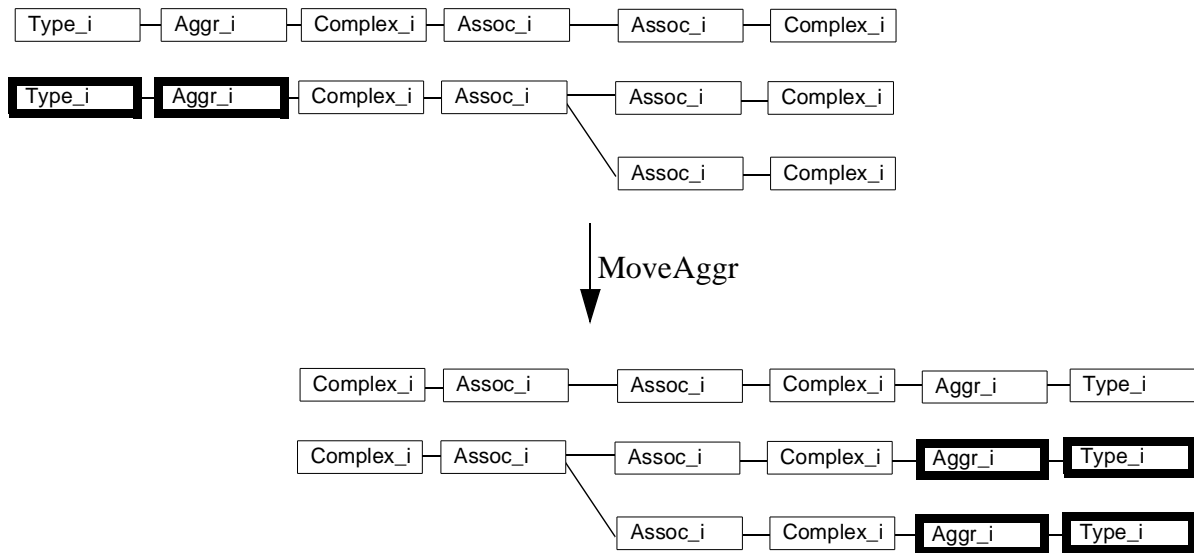
Zur Spezifikation der Desynchronisationskante zwischen den Knoten 11' und 12' ist jedoch die Angabe der Interpretation der Aggregation erforderlich. Diese wird mit dem Parameter *f* an die Transformation übergeben. Für zwei Sonderfälle läßt sie sich jedoch auch aus der Interpretation der Assoziation und der Interpretation der Aggregation vor der Transformation konstruieren. In diesen Fällen kann der Parameter *f* entfallen:

Ist die Assoziation beidseitig total und funktional, so gilt für die Interpretation der Aggregation: $I_{Aggr}(5') = I_{Assoc}(5)$. Die Instanzmenge des aggregierten Typs bleibt dann also unverändert. Für diesen Sonderfall ist die Transformation kapazitätserhaltend.

Ist die Assoziation beidseitig total und nur für Knoten 2 funktional, so läßt sich die Interpretation der Aggregation ebenfalls konstruieren: Dazu wird wiederum die Interpretation der Aggregation vor der Transformation $I_{Aggr}(5) : I_{Complex}(3) \rightarrow P(I_{Type}(6))$, sowie die Interpretation der Assoziation $I_{Assoc}(2) \subseteq I_{Complex}(4) \times I_{Complex}(3)$ benutzt.

Die Interpretation der Aggregation nach der Transformation ist dann: $I_{Aggr}(5') : I_{Complex}(4) \rightarrow I_{Complex}(6')$, $I_{Aggr}(5')(o_2) = I(5)(o_1)$. Da die Assoziation bezüglich Knoten 2 funktional ist, ist o_1 mithilfe der Interpretation der Assoziation eindeutig bestimmbar: Es ist diejenige Instanz von Knoten 3, die mit o_2 assoziiert ist.

Die folgende Abbildung illustriert die Konstruktion: Sie zeigt einen Ausschnitt eines Schema-Instanz-Graphen, auf den *MoveAggr* angewendet wird.



Da die Assoziation nur für die rechte Seite funktional ist, werden die dick gezeichnete Knoten durch die Verschiebung vervielfältigt. Die Transformation ist reversibel, aber nicht symmetrisch reversibel, da für die Umkehrtransformation nicht vorausgesetzt werden kann, daß die vervielfältigten Knoten denselben Wert repräsentieren. Für beidseitig totale, aber nur bezüglich Knoten 2 funktionale Assoziationen ist *MoveAggr* also kapazitätserweiternd. Falls die Assoziation dagegen bezüglich Knoten 1 funktional ist (und nicht bezüglich Knoten 2) ist sie kapazitätsreduzierend.

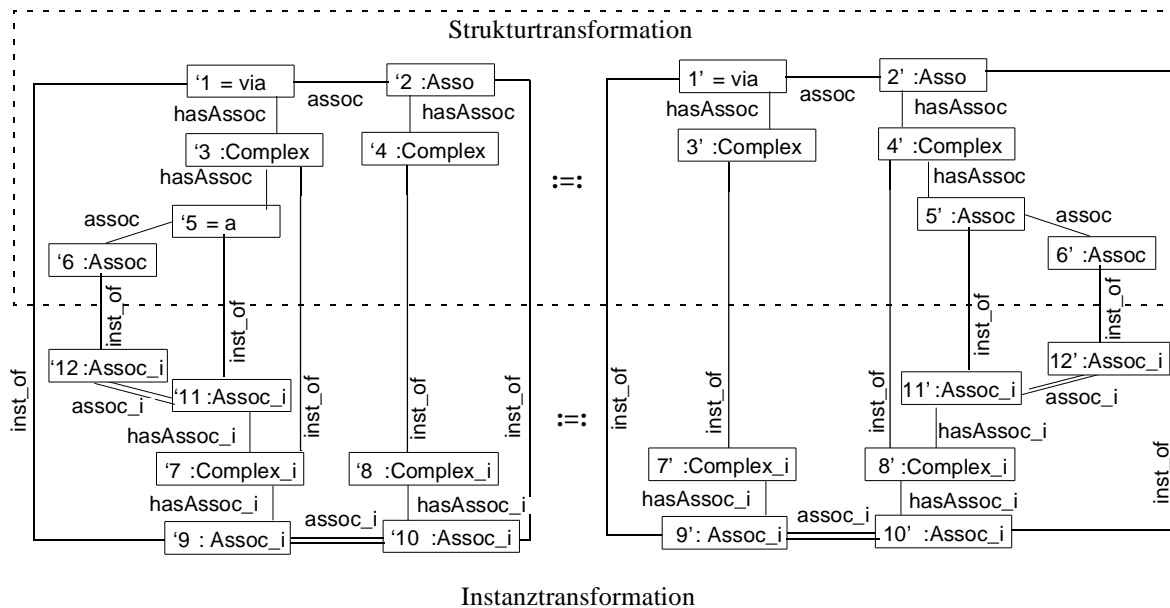
Für die anderen Fälle ist die Transformation kapazitätsändernd:

Für n:m-Beziehungen wird je nach Art der Beziehung einzelner Instanzen der komplexen Typen die Menge der Instanzen der Aggregation größer oder kleiner.

Für partielle Assoziationen ist die Transformation nicht reversibel, da es Instanzen der komplexen Typen geben kann, die nicht an der Assoziation teilnehmen. Ihre Aggregationen gehen durch die Transformation verloren.

Verschieben einer Assoziation entlang einer (anderen) Assoziation.

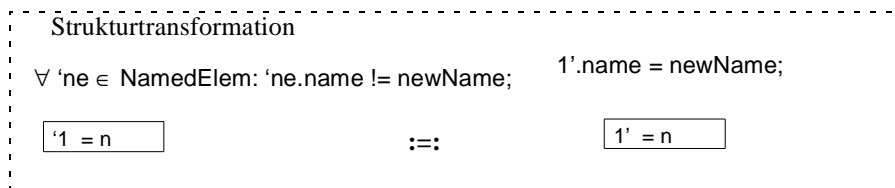
transformation MoveAssoc (a : Assoc; via : Assoc; r : relation) =



Die Transformation **MoveAssoc** verschiebt eine Assoziation von einem komplexen Typ in einen anderen. Die beiden komplexen Typen müssen dazu über eine (andere) Assoziation verbunden sein. Sie ist ganz analog zur Transformation **MoveAggr**, nur daß jetzt Assoziationen verschoben werden. Hinsichtlich der Kapazität ergeben sich die gleichen Eigenschaften: Abhängig von der Art der Assoziation, die die beiden komplexen Typen verbindet, ist die Transformation kapazitätserhaltend, -erweiternd, -reduzierend oder -ändernd.

Umbenennen von Typen.

transformation Rename (n:NamedElem, newName:String) =

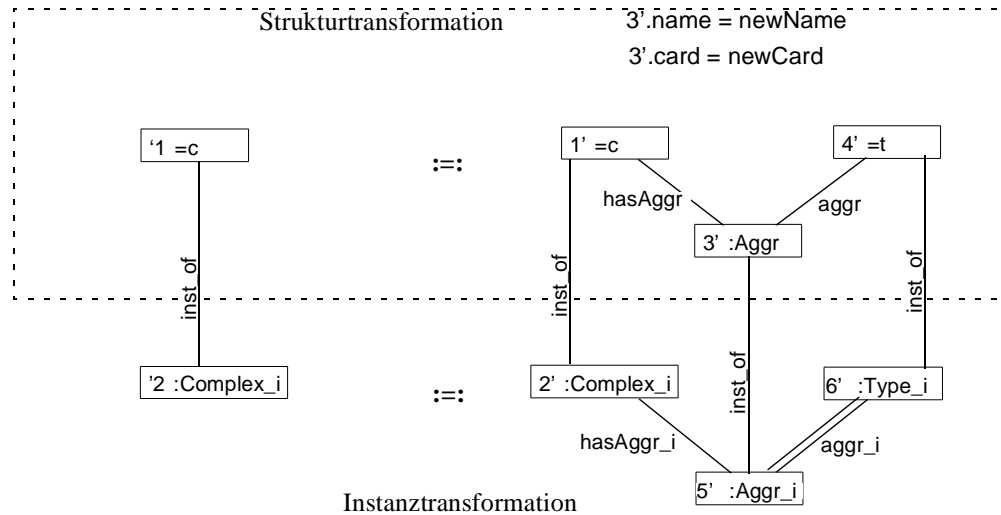


Mit der Transformation **Rename** können Schemaelemente umbenannt werden. Eine Instanztransformation ist dabei nicht erforderlich, da die Instanzen von der Umbenennung nicht betroffen sind. Daher ist die Transformation kapazitätserhaltend.

7.2 Transformation von Beziehungen zwischen Typen

Aggregieren.

```
transformation Aggregate (c: Complex; t: Type; newName:string; newCard:enum
defaultValue:void ) =
```



Die Transformation **Aggregate** fügt eine Aggregation eines Typs in einen komplexen Typ ein. Der Wert aller Instanzen des aggregierten Typs wird über den Parameter *defaultValue* festgelegt. Falls ein atomarer Typ aggregiert wird, enthält der Parameter einen Wert dieses Typs. Bei Aggregation eines komplexen Typs enthält er die Identität einer Instanz des komplexen Typs.

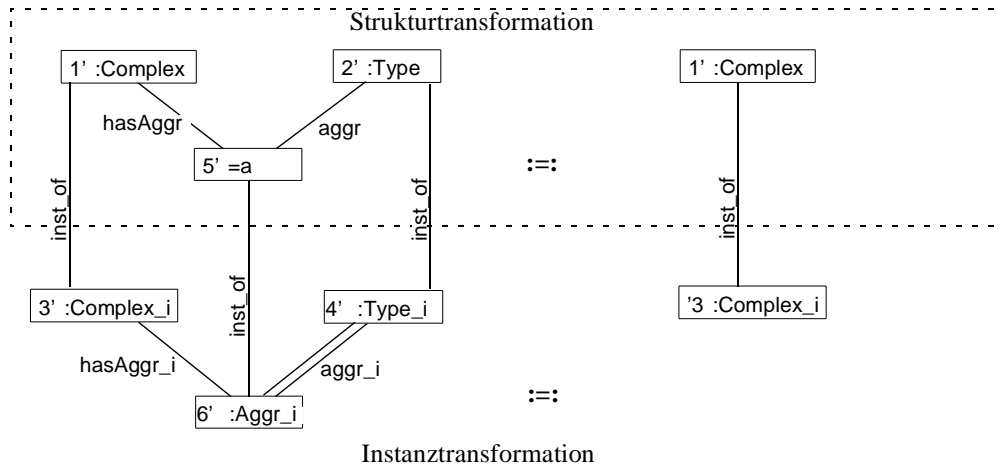
Die zur Spezifikation der *aggr_i*-Kante notwendige Interpretation des Knotens 3' ist durch den Parameter *defaultValue* gegeben. Sie lautet:

$$I_{Aggr}(3') = f \text{ mit } f: I_{Complex}(1') \rightarrow I_{Type}(4'), f(x) = \{defaultValue\}$$

Die Transformation ist kapazitätserweiternd: Ihre Änderung kann durch Entfernen der Aggregation rückgängig gemacht werden, ist also reversibel. Ihre Umkehrtransformation ist nicht reversibel: Für sie kann nicht vorausgesetzt werden, dass alle aggregierten Instanzen den Wert des übergebenen Parameters haben.

Entfernen von Aggregationen.

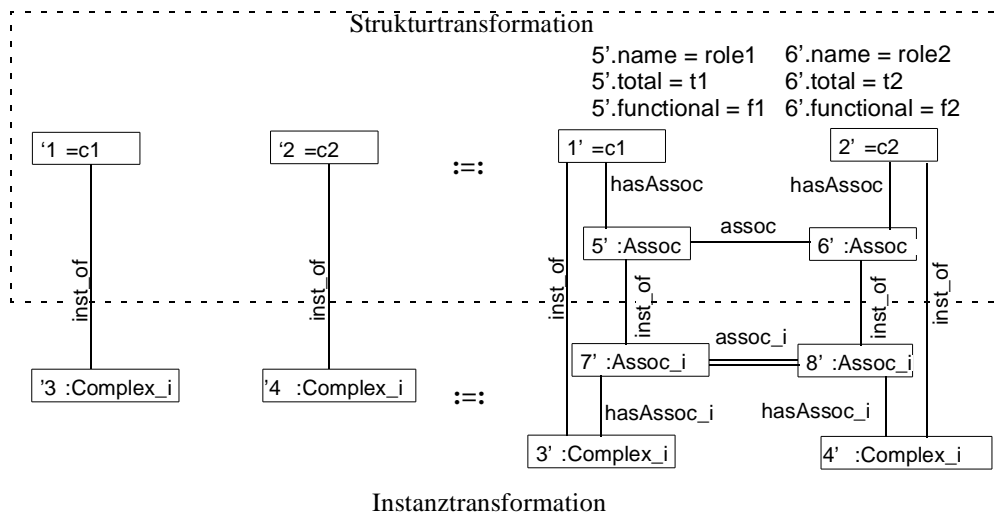
transformation Disaggregate (a:Aggr) =



Die Transformation *Disaggregate* entfernt eine Aggregationsbeziehung zwischen zwei Typen. Sie ist die Umkehrtransformation von *Aggregate* und daher kapazitätsreduzierend.

Assoziieren.

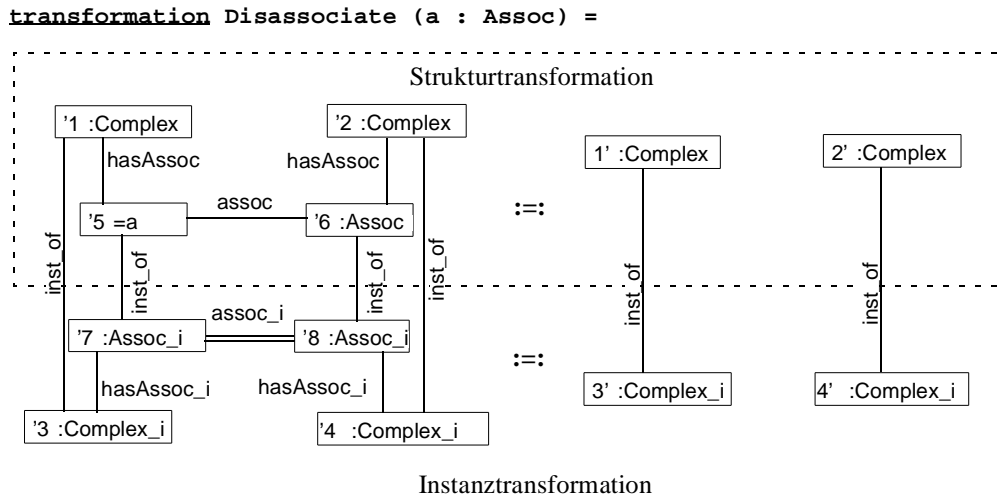
transformation Associate (c1, c2: Complex; role1, role2:string; t1, t2, f1, f2:bool; r:relation) =



Die Transformation *Associate* fügt eine Assoziation zwischen zwei komplexen Typen ein. Dabei muß spezifiziert werden, welche Instanzen der beiden komplexen Typen nach der Transformation in Beziehung stehen. Dies geschieht durch Übergabe der entsprechenden Relation im Parameter r . Die Interpretation des Knotens 7' ist dann $I_{Assoc}(7') = r$, die von Knoten 8' ist $I_{Assoc}(8') = r^{-1}$. Die Relation muß die von den anderen Parametern geforderten Eigenschaften bezüglich Totalität und Funktionalität haben.

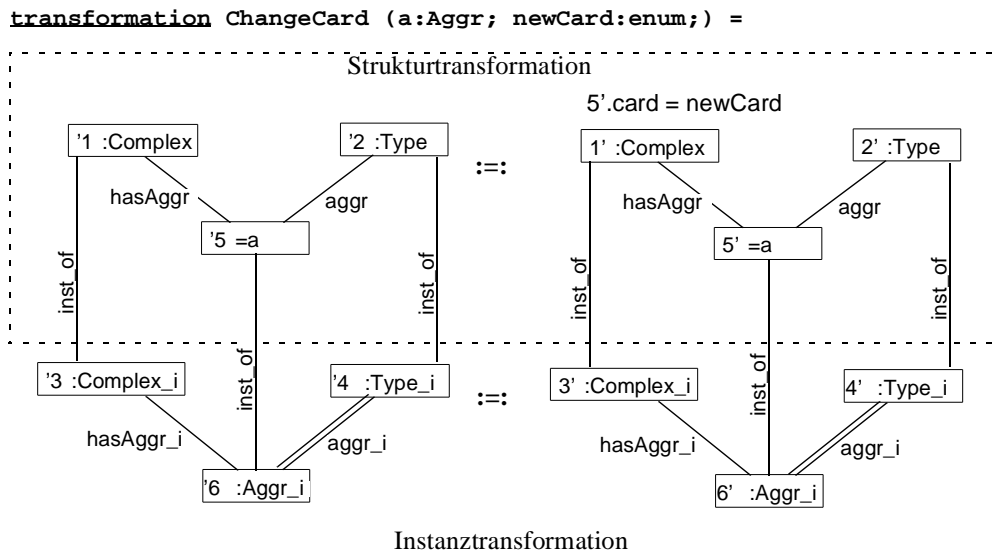
Mit der Assoziation wird eine zusätzliche Restriktion für die Menge der gültigen Instanzen des Schemas eingeführt, was die Menge der gültigen Instanzen der beteiligten komplexen Typen einschränkt. Andererseits werden durch die Assoziation dem Schema Informationen über in Beziehung stehende Instanzen hinzugefügt. Die Schemata vor und nach der Transformation sind daher hinsichtlich ihrer Kapazität nicht vergleichbar, die Transformation ist also kapazitätsändernd.

Entfernen von Assoziationen.



Die Transformation *Disassociate* entfernt eine Assoziation zwischen zwei komplexen Typen. Sie ist die Umkehrung der Transformation *Associate*. Für ihre Eigenschaften bezüglich der Kapazität kann analog argumentiert werden: Mit der Assoziation wird eine Restriktion für die Menge der Instanzen der komplexen Typen entfernt, was die Menge der gültigen Instanzen erweitert. Andererseits werden durch das Entfernen der Assoziation dem Schema Information über die in Beziehung stehenden Instanzen entzogen. Die Schemata vor und nach der Transformation sind daher hinsichtlich ihrer Kapazität nicht vergleichbar, die Transformation ist also kapazitätsändernd.

Ändern der Kardinalität von Aggregationen.



Die Transformation **ChangeCard** ändert die Kardinalität einer Assoziation, etwa von *one* zu *set* oder *list*. Ein Schema-Instanz-Graph kann in der Form, wie er in Abschnitt 4.4 eingeführt wurde, nicht zwischen *set* und *list* unterscheiden. Die Modellierung einer Listenstruktur erfordert die Definition einer Ordnung auf den zugehörigen Instanzen (vgl. Abschnitt 4.3). Im Schema-Instanz-Graphen kann eine solche Ordnung mithilfe einer Listenstruktur auf den Instanzknoten definiert werden, die mit Kanten eines zusätzlichen Kantentyps realisiert wird.

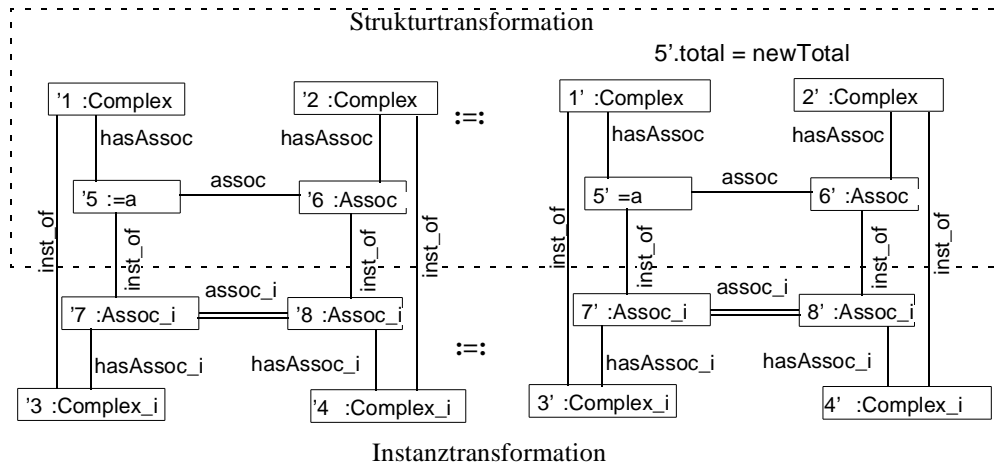
Die Transformation **ChangeCard** ist kapazitätserweiternd, wenn die Kardinalität von *one* auf *list* bzw. *set* geändert wird:

Die *aggr_i*-Kante der linken Seite ist in diesem Fall eine einfache Kante, da die Aggregation noch die Kardinalität *one* hat. Betrachtet man die rechte Seite unmittelbar nach der Transformation, so ist diese Kante auch hier noch einfach; die Eigenschaften des Schema-Instanz-Graphen lassen dann zwar mehrere Kanten zu, jedoch kommt immer nur eine vor. Die transformierte Instanz kann also rücktransformiert werden.

Für die entgegengesetzte Änderung - und damit auch für die Umkehrung der Änderung von *one* auf *set* - ist die Kante nicht degeneriert. Für Änderungen von *set* bzw. *list* auf *one* ist die Transformation also kapazitätsreduzierend.

Ändern der Totalität von Assoziationen.

```
transformation ChangeTotality (a:Assoc; newTotal:bool; r:relation) =
```



Die Transformation *ChangeTotality* ändert die Totalität von Assoziationen.

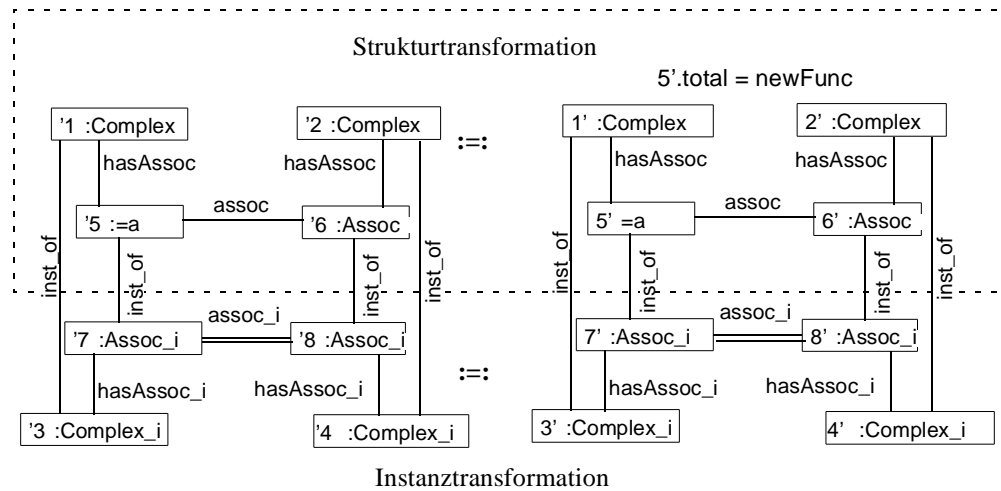
Wird die Totalität auf *true* gesetzt, so sind Instanzen, die vorher nicht an der Assoziation beteiligt waren, nicht mehr erlaubt. Für diese Instanzen werden Beziehungen - im Schema-Instanz-Graphen also Kanten - eingefügt. Dies geschieht durch Angabe der Interpretation des *Assoc*-Knotens als Parameter. Von dieser Relation wird verlangt, daß sie die bestehenden Kanten erhält, es muß also gelten: $I_{Assoc}('5) \subseteq I_{Assoc}(5) = r$. Außerdem muß die Relation natürlich total sein.

Das Setzen der Totalität auf *true* ist kapazitätsreduzierend. Ein Schema mit einer totalen Assoziation kann also weniger Daten speichern, als wenn die Assoziation nicht total ist. Das Einfügen der zusätzlichen Kanten kann in diesem Zusammenhang auch als eine Reparaturmaßnahme aufgefaßt werden, die aus ungültigen Instanzen gültige Instanzen macht, um die Konsistenz der Datenbank sicherzustellen.

Wird die Totalität auf *false* gesetzt, bleiben alle Kanten erhalten, denn jede gültige Instanz einer totalen Assoziation ist auch dann noch gültig, wenn die Assoziation nicht total ist. Der Parameter *r* kann dann entfallen. Die Änderung ist somit also reversibel, jedoch nicht symmetrisch reversibel, denn durch die Umkehrung können, wie oben erläutert, eventuell ungültige Instanzen entstehen. Das Ändern der Totalität ist also kapazitätserweiternd.

Ändern der Funktionalität von Assoziationen.

`transformation ChangeFunctionality (a:Assoc; newFunc:bool; r:relation) =`



Die Transformation *ChangeFunctionality* ändert die Funktionalität von Assoziationen.

Wird die Funktionalität auf *true* gesetzt, so sind Instanzen, die vorher mehrere Beziehungen hatten, nicht mehr erlaubt. Für diese Instanzen müssen Beziehungen - im Schema-Instanz-Graphen also Kanten - entfernt werden. Dies geschieht durch Angabe der Interpretation des *Assoc*-Knotens als Parameter. Von dieser Relation wird verlangt, daß sie keine Kanten hinzufügt, es muß also gelten: $I_{Assoc}(5) \supseteq I_{Assoc}(5') = r$. Außerdem muß die Relation natürlich funktional sein.

Wird die Funktionalität auf *false* gesetzt, bleiben alle Kanten erhalten, denn jede gültige Instanz einer funktionalen Assoziation ist auch dann noch gültig, wenn die Assoziation nicht funktional ist. Der Parameter *r* kann dann entfallen.

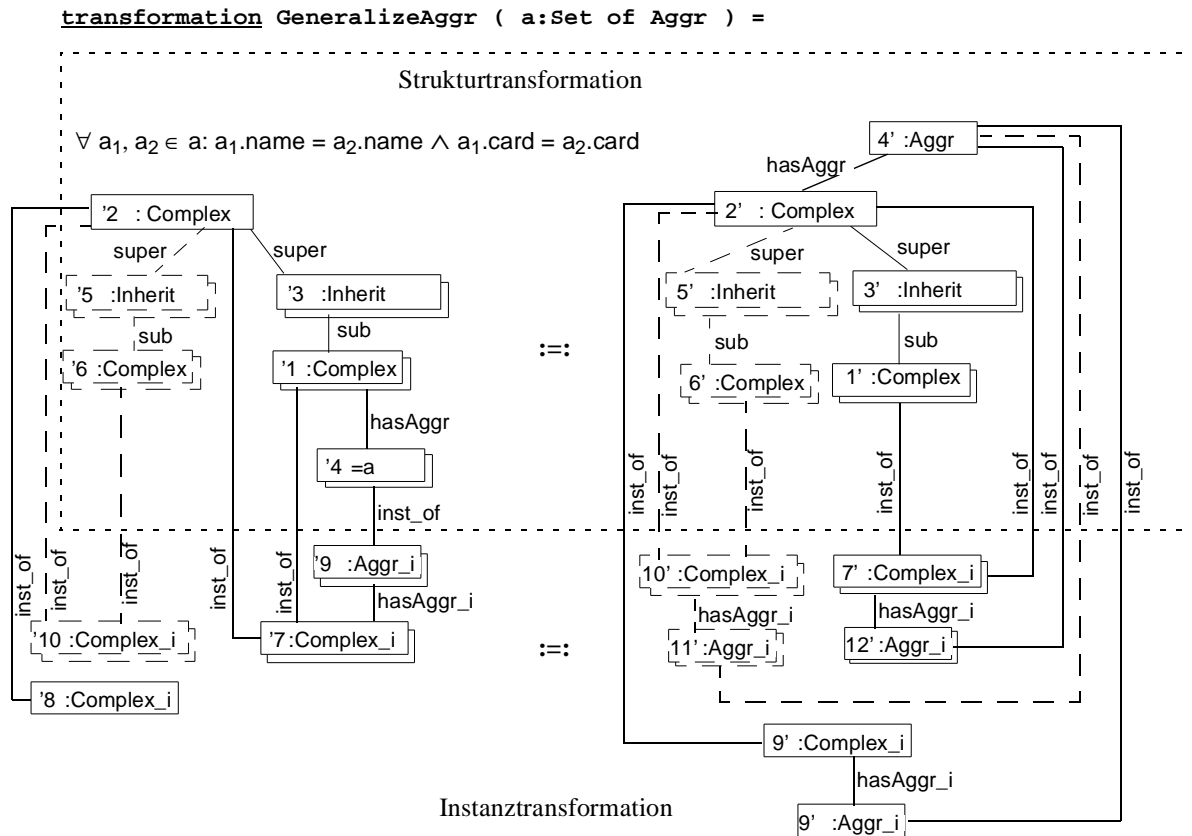
Das Ändern der Funktionalität auf *true* ist kapazitätsreduzierend, da die Funktionalität eine stärkere Restriktion der Instanzen der komplexen Typen darstellt, die Menge der gültigen Instanzen also kleiner ist. Die Änderung auf *false* ist kapazitätserweiternd, da die komplexen Typen einer nicht-funktionale Beziehung mehr gültige Instanzen haben.

7.3 Transformation von Vererbungsbeziehungen

Diese Transformationen lassen sich nochmals in zwei Gruppen einteilen. Die eine verschiebt Attribute (Aggregationen oder Assoziationen) innerhalb einer Vererbungshierarchie in Ober- oder Untertypen. Diese Transformationen sind im Allgemeinen nicht kapazitätserhaltend, da sich die Attributmenge der beteiligten komplexen Typen ändert.

Die andere Gruppe dient dazu, neue komplexe Typen als Stufen in einer Vererbungshierarchie einzufügen oder zu entfernen. Die Transformationen sind so spezifiziert, daß sie nur leere Typen - also Typen ohne Attribute - erzeugen oder löschen können. Da solche Typen keine Daten speichern können, sind diese Transformationen dann kapazitätserhaltend. Man wird sie jedoch in der Regel in Kombination mit Transformationen der anderen Gruppe verwenden müssen, um die gewünschten Restrukturierungen zu erreichen, so daß die Kombination nicht kapazitätserhaltend ist.

Verschieben einer Aggregation in einen Obertyp.



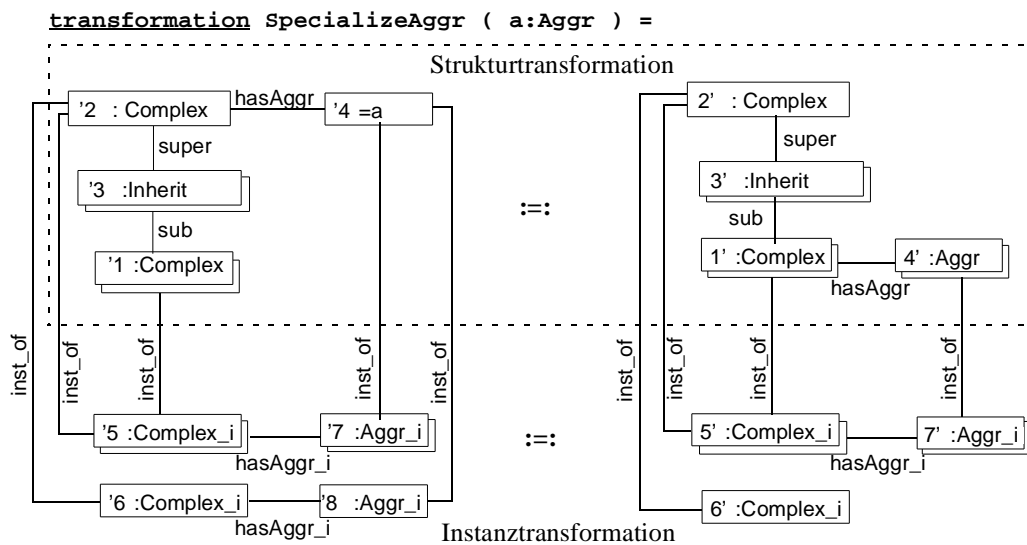
Die Transformation **GeneralizeAggr** verschiebt eine Aggregation innerhalb einer Vererbungshierarchie aus den Untertypen in ihren gemeinsamen Obertyp. Dabei wird verlangt, daß alle Aggregationen denselben Typ aggregieren, und sie in ihrem Namen und in ihrer Kardinalität übereinstimmen. Dies kann gegebenenfalls zuvor mit den Transformationen *Rename* und *ChangeCard* erreicht werden.

Die Transformation ist kapazitätserhaltend, wenn zwei Bedingungen erfüllt sind:

- Der Obertyp ist abstrakt
- Die Aggregation ist in allen Untertypen vorhanden, das heißt, die optionalen Knoten der Regel sind nicht vorhanden.

Andernfalls ist die Transformation kapazitätserweiternd: Nach der Transformation hat der Obertyp ein weiteres Attribut und folglich ist seine Instanzmenge größer, es sei denn, der Typ ist abstrakt, hat also keine Instanzen. Ist das Attribut vor der Transformation nicht in allen Untertypen vorhanden, so bekommen diese Untertypen über die Vererbungsbeziehung ein neues Attribut, was die Instanzmenge ebenfalls vergrößert.

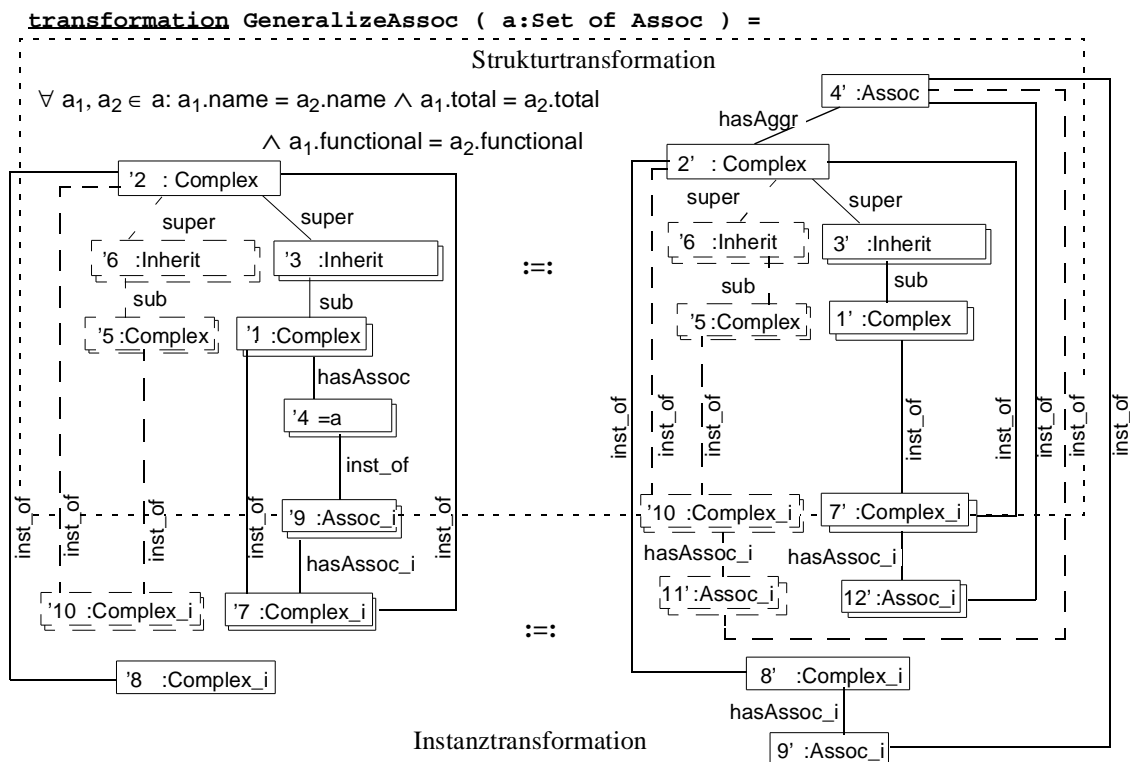
Verschieben einer Aggregation in einen Untertyp.



Die Transformation *SpecializeAggr* verschiebt eine Aggregation innerhalb einer Vererbungshierarchie aus einem Obertyp in alle seine Untertypen. Die Transformation ist kapazitätserhaltend, wenn der Obertyp abstrakt ist. Andernfalls ist sie kapazitätsreduzierend, denn der Obertyp verliert die Aggregation, so daß seine Instanzmenge kleiner wird.

Diese Transformation ist ein Sonderfall der Umkehrtransformation von *GeneralizeAggr*, nämlich für den Fall, daß bei *GeneralizeAggr* die optionalen Knoten fehlen.

Verschieben einer Assoziation in einen Obertyp.



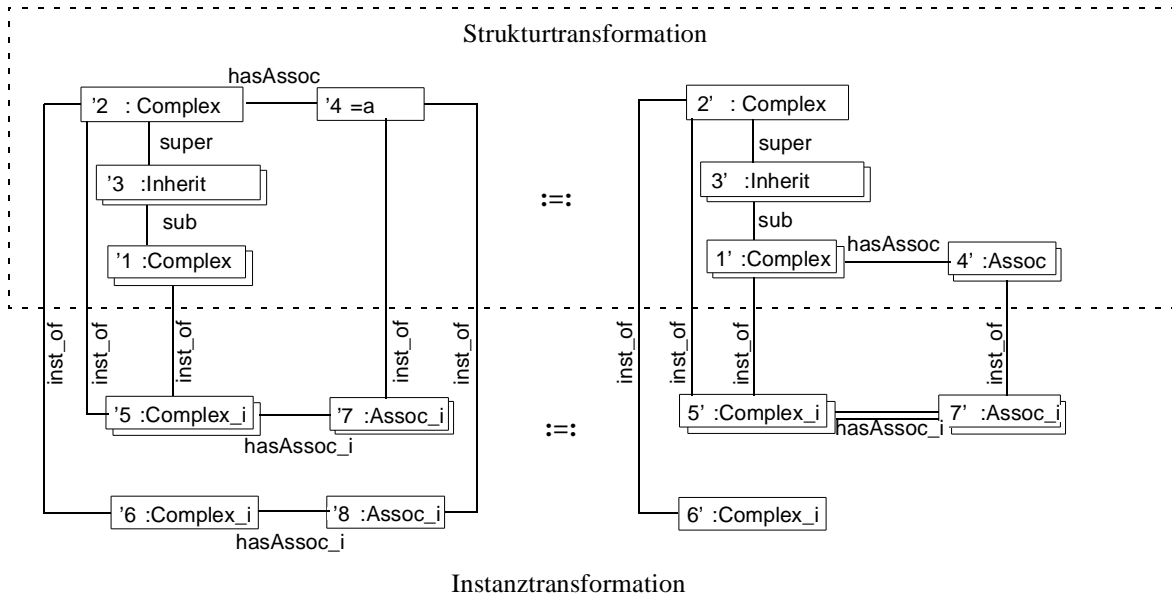
Die Transformation *GeneralizeAssoc* verschiebt eine Assoziation innerhalb einer Vererbungshierarchie aus den Untertypen in ihren gemeinsamen Obertyp. Analog zu *GeneralizeAggr* wird dabei verlangt, daß alle Assoziationen denselben Typ als Assoziationspartner haben, und sie in ihrem Namen, ihrer Totalität und ihrer Funktionalität übereinstimmen. Dies kann gegebenenfalls zuvor mit den Transformationen *Rename*, *ChangeTotality* und *ChangeFunctionality* erreicht werden.

In Analogie zu *GeneralizeAggr* ist diese Transformation im Allgemeinen kapazitätserweiternd und für den folgenden Sonderfall kapazitätserhaltend:

- Der Obertyp ist abstrakt.
- Die Assoziation ist in allen Untertypen vorhanden, das heißt, die optionalen Knoten der Regel sind nicht vorhanden.

Verschieben einer Assoziation in einen Untertyp.

`transformation SpecializeAssoc (a:Assoc) =`

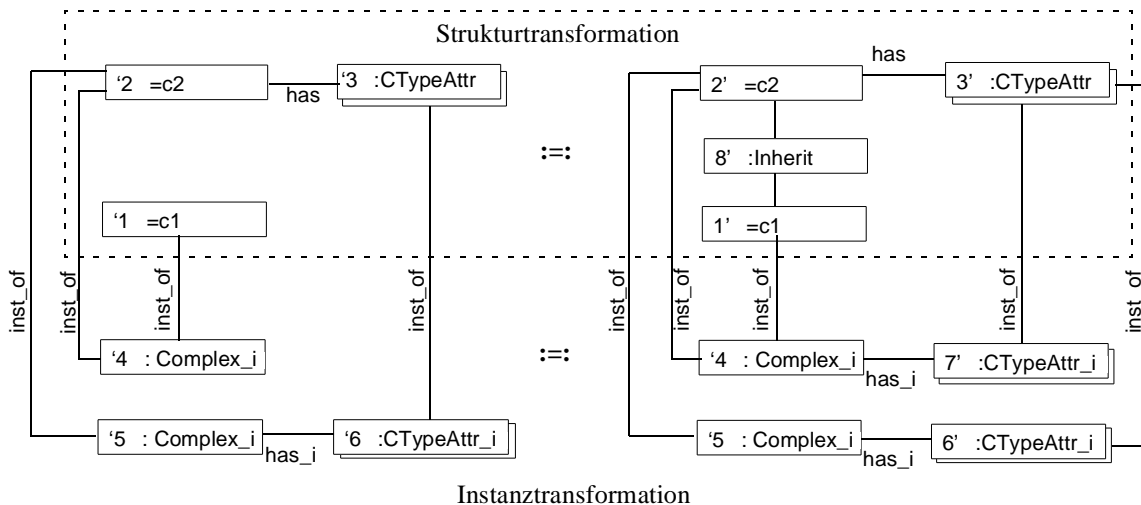


Die Transformation *SpecializeAssoc* verschiebt eine Assoziation innerhalb einer Vererbungshierarchie aus einem Obertyp in alle seine Untertypen. In Analogie zu *SpecializeAggr* ist diese Transformation im Allgemeinen reduzierend und für den Sonderfall, daß der Obertyp abstrakt ist, ist sie kapazitätserhaltend.

Sie ist ein Sonderfall der Umkehrtransformation von *GeneralizeAssoc*, nämlich für den Fall, daß bei *GeneralizeAssoc* die optionalen Knoten fehlen.

Vererbungsbeziehung einfügen.

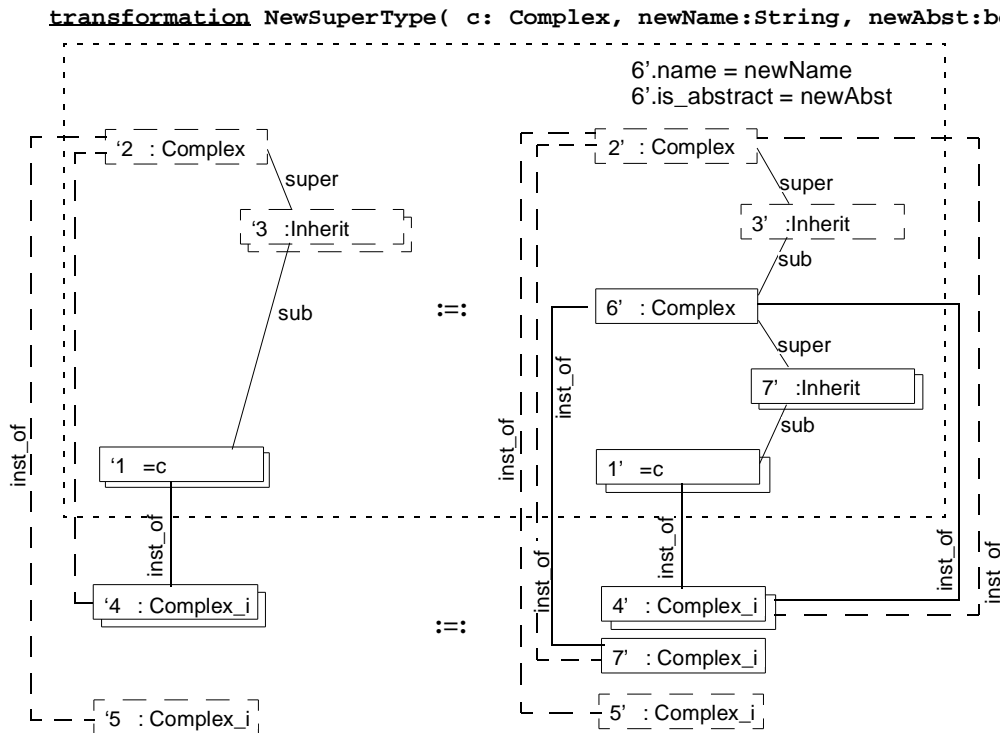
`transformation AddInherit(c1, c2: Complex) =`



Die Transformation *AddInherit* fügt eine Vererbungsbeziehung zwischen zwei komplexen Typen ein.

Dabei werden dem Untertyp die Attribute des Obertyps als neue Attribute hinzugefügt. Daher ist die Transformation kapazitätserweiternd, denn die Menge der möglichen Instanzen wird größer.

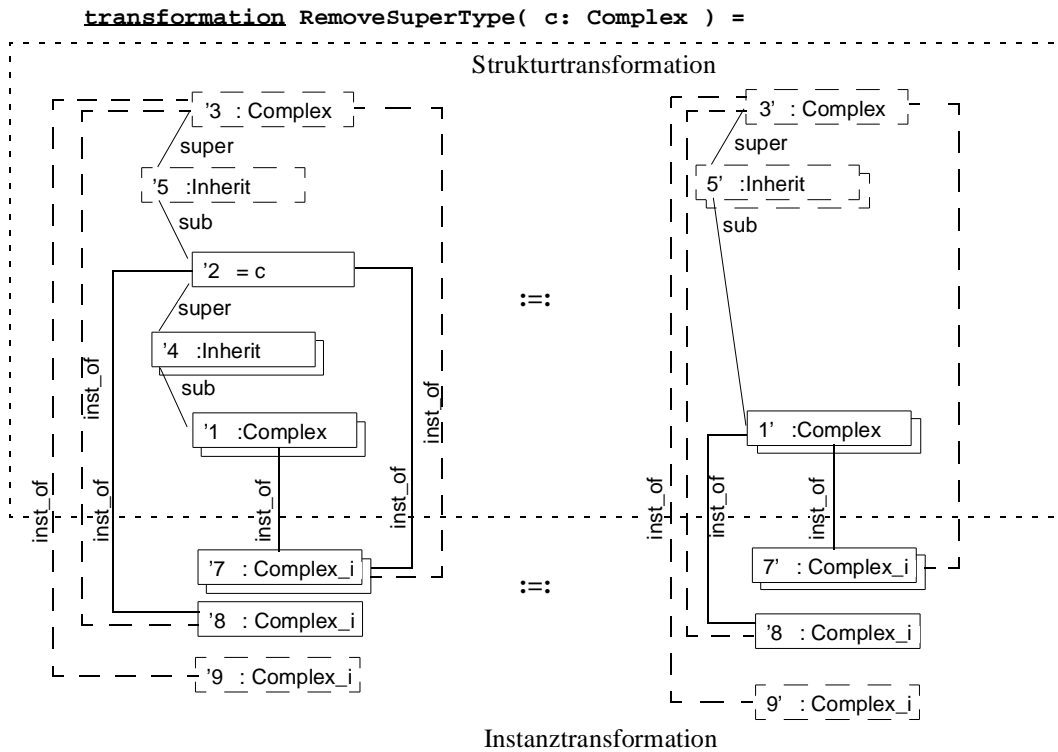
Neuen Obertyp einführen.



Die Transformation *NewSupertype* führt zu einem oder mehreren gegebenen komplexen Typen einen neuen komplexen Typ als gemeinsamen Obertyp ein. Falls die Typen zuvor schon einen Obertyp hatten, muß dieser für alle derselbe sein. Dann ist dieser Typ nun Obertyp des neu eingefügten Typs. Als weiterer Parameter kann angegeben werden, ob der neue Typ abstrakt ist, oder nicht.

Diese Transformation ist kapazitätserhaltend, da der neu eingefügte komplexe Typ keine Attribute hat und somit keine Daten speichern kann. Erst wenn man dem Typ Attribute gibt, etwa durch *GeneralizeAssoc*, ändert sich die Kapazität.

Obertyp entfernen.

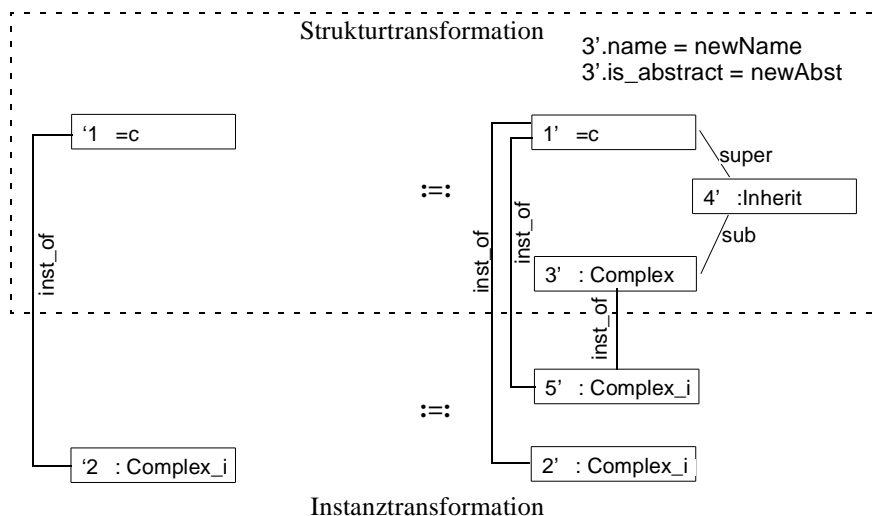


Die Transformation **RemoveSuperType** entfernt einen komplexen Typ und die Vererbungsbeziehungen zu seinen Untertypen. Dabei wird verlangt, daß der entfernte Typ keine Aggregationen und Assoziationen mehr hat. Diese können gegebenenfalls zuvor mit den Transformationen *SpecializeAggr* und *SpecializeAssoc* in die Untertypen oder mit *GeneralizeAggr* und *GeneralizeAssoc* in den Obertyp (sofern vorhanden) verschoben werden. Falls der entfernte Typ einen Obertyp hatte, ist dieser nun Obertyp der Untertypen des entfernten Typs.

Die Transformation ist kapazitätserhaltend, denn weil der entfernte Typ keine Attribute hat, verändert sich die Instanzmenge nicht.

Neuen Untertyp einführen.

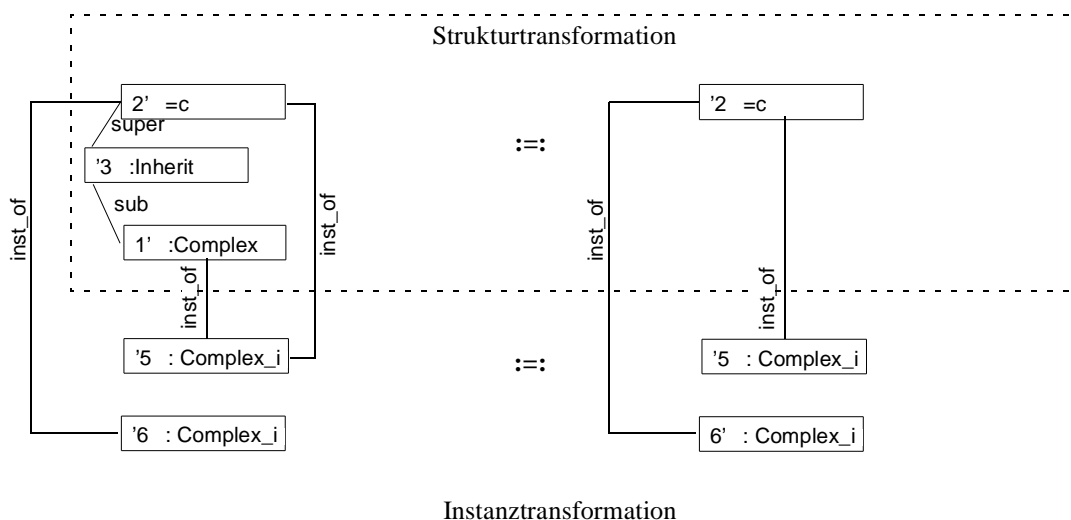
`transformation NewSubType(c: Complex, newName:String, newAbst:bool) =`



Die Transformation *NewSubtype* führt zu einem komplexen Typ einen neuen komplexen Typ als Untertyp ein. Als weiterer Parameter kann angegeben werden, ob der neue Typ abstrakt ist oder nicht. Diese Transformation ist kapazitätserhaltend, da der neu eingefügte komplexe Typ keine Attribute hat und somit keine Daten speichern kann.

Untertyp entfernen.

`transformation RemoveSubType(c: Complex) =`



Die Transformation *RemoveSubtype* entfernt einen komplexen Typ und die Vererbungsbeziehungen zu seinem Obertyp. Der entfernte Typ darf keine Aggregationen und Assoziationen haben. Weiterhin wird verlangt, daß der zu entfernende Typ selbst keine Untertypen hat. Falls ein Typ entfernt werden soll, der Untertypen hat, kann die Transformation *RemoveSupertype* ver-

wendet werden.

Die Transformation ist kapazitätserhaltend, denn weil der entfernte Typ keine Attribute hat, konnte er keine Daten speichern.

7.4 Anwendungsbeispiel

Um zu illustrieren, wie die Transformationen benutzt werden können, wird hier ein Anwendungsbeispiel gegeben. Daran soll auch gezeigt werden, wie die Transitivität der Kapazitätseigenschaften (vgl. Abschnitt 6.1) ausgenutzt werden kann, um aus den hier vorgestellten Transformationen komplexere Kommandos zusammzusetzen.

Das Beispielschema enthält einen komplexen Typ „Person“ mit den Aggregation „Name“, „Strasse“ und „Wohnort“. Der komplexe Typ soll durch Transformation aufgespalten werden, so daß ein neuer Typ Adresse entsteht. In diesen sollen die Aggregationen „Straße“ und „Wohnort“ verschoben werden.

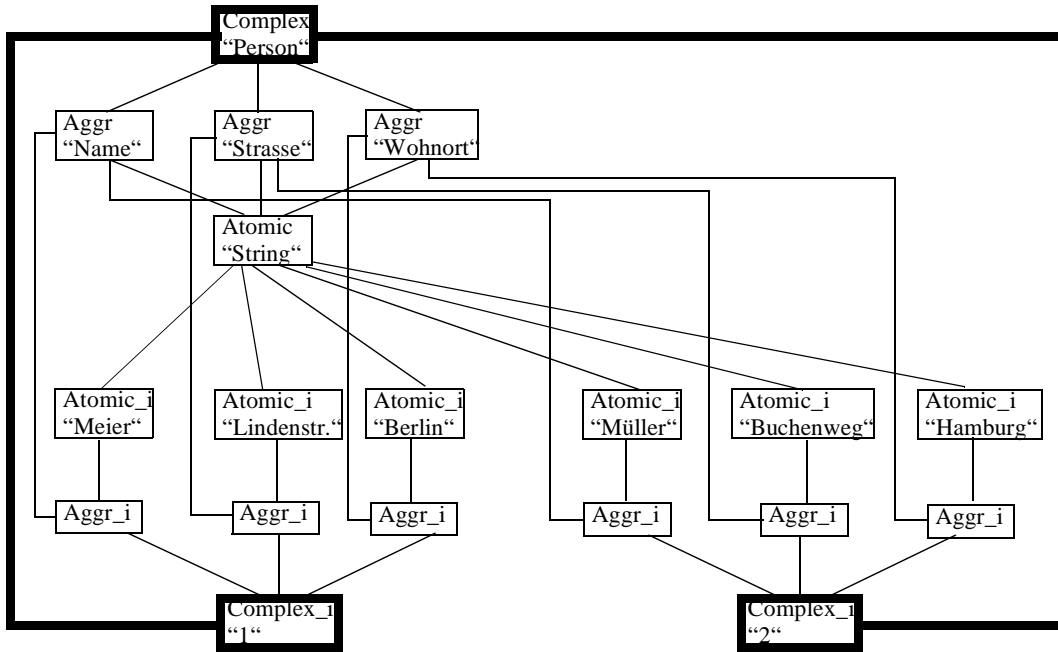
Im M^3 kann das Schema so beschrieben werden: $S = (\text{Complex}, \text{Atomic}, \text{Aggr}, \text{Assoc}, \text{Inherit})$ mit:

- $\text{Complex} = \{ \text{Person} \}$
- $\text{Atomic} = \{ \text{String} \}$
- $\text{Aggr} = \{$
 $(\text{Person}, \text{Name}, \text{one} \rightarrow \text{String}),$
 $(\text{Person}, \text{Strasse}, \text{one} \rightarrow \text{String}),$
 $(\text{Person}, \text{Wohnort}, \text{one} \rightarrow \text{String}) \}$
- $\text{Assoc} = \{ \}, \text{Inherit} = \{ \}$

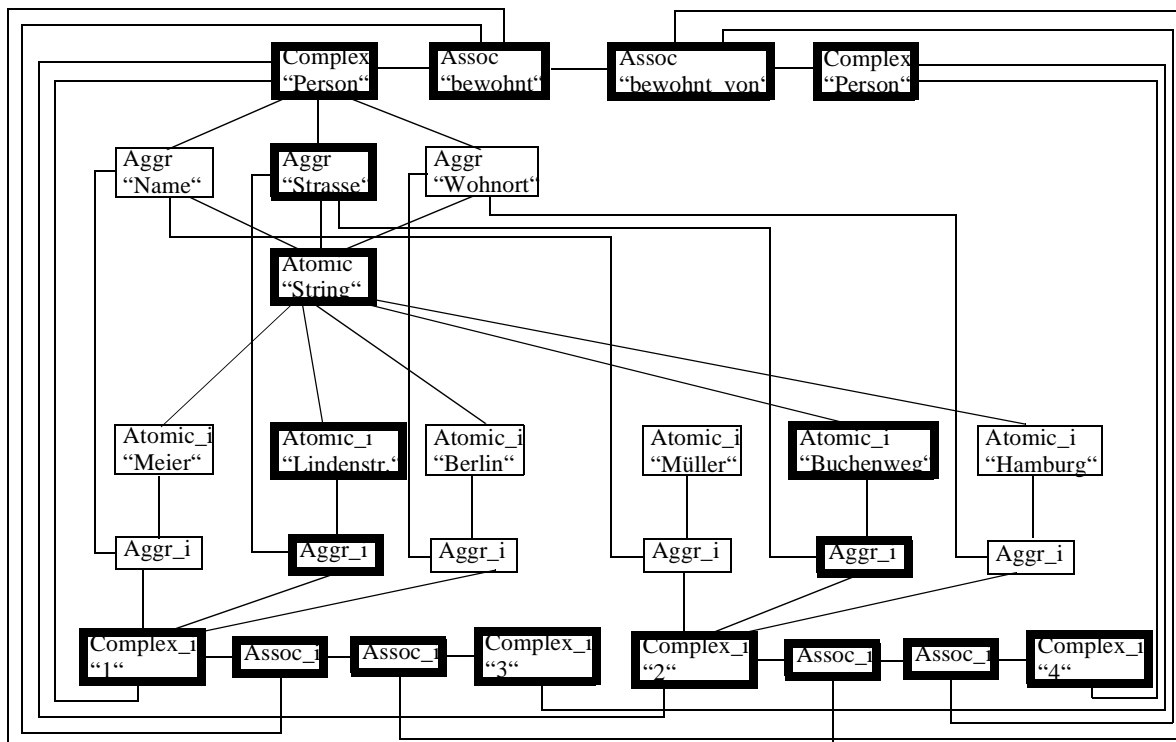
Die Interpretation dieses Schemas, also seine Instanzmenge, sei:

- $I_{\text{Complex}}(\text{Person}) = \{ 1, 2 \}$
- $I_{\text{Atomic}}(\text{String}) = \{ \text{Meier}, \text{Müller}, \text{Lindenstraße}, \text{Buchenweg}, \text{Berlin}, \text{Hamburg} \}$
- $I_{\text{Aggr}}((\text{Person}, \text{Name}, \text{one} \rightarrow \text{String})) = f_1: I_{\text{Complex}}(\text{Person}) \rightarrow P(I_{\text{Atomic}}(\text{String})), f_1(1) = \{ \text{Meier} \}, f_1(2) = \{ \text{Müller} \}$
 $I_{\text{Aggr}}((\text{Person}, \text{Strasse}, \text{one} \rightarrow \text{String})) = f_3: I_{\text{Complex}}(\text{Person}) \rightarrow P(I_{\text{Atomic}}(\text{String})),$
 $f_3(1) = \{ \text{Lindenstraße} \}, f_3(2) = \{ \text{Buchenweg} \}$
 $I_{\text{Aggr}}((\text{Person}, \text{Wohnort}, \text{one} \rightarrow \text{String})) = f_4: I_{\text{Complex}}(\text{Person}) \rightarrow P(I_{\text{Atomic}}(\text{String})),$
 $f_4(1) = \{ \text{Berlin} \}, f_4(2) = \{ \text{Hamburg} \}$

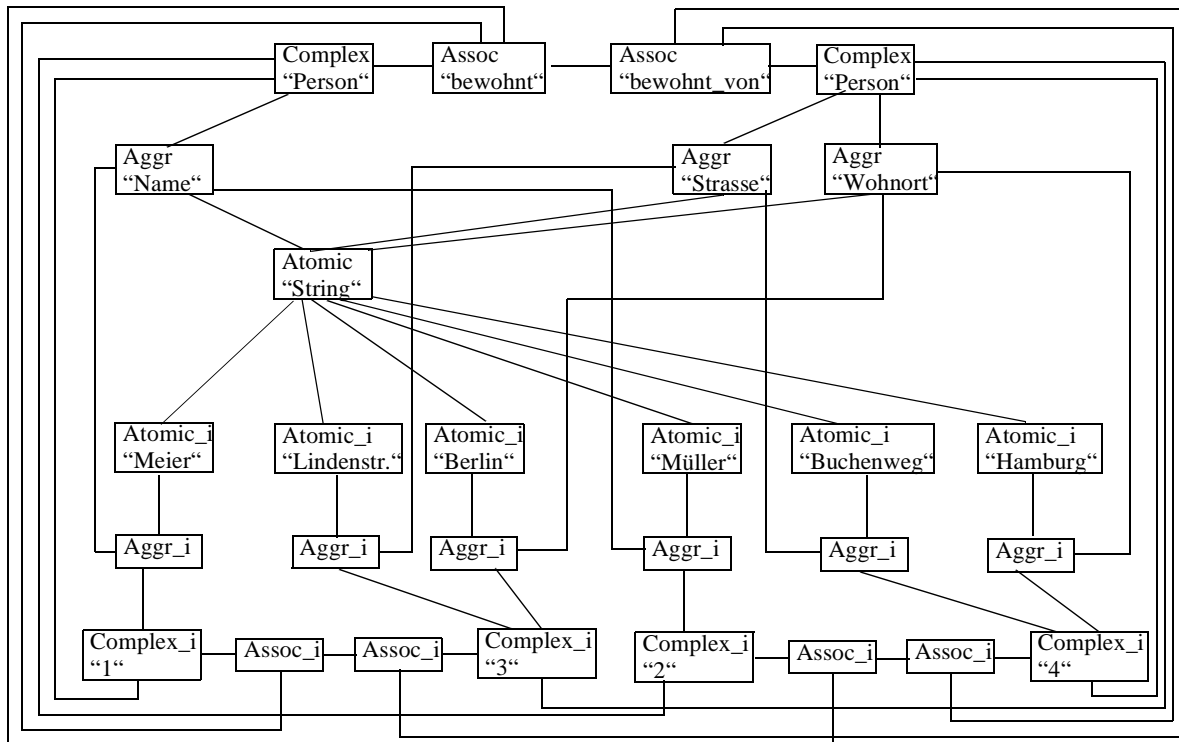
Die folgende Abbildung zeigt den Schema-Instanz-Graphen für diese Datenbank. (Kantenbeschriftungen wurden wegen der Übersichtlichkeit weggelassen.) Dabei ist der markierte Teilgraph ein Match für die Transformation *NewComplexType*.



Wendet man darauf die Transformation *NewComplexType*("Person", "Adresse", "wohnt_in", "bewohnt_von") an, entsteht der folgende Schema-Instanz-Graph:



Der Teilgraph aus den markierten Knoten (und den sie verbindenden Kanten) ist ein Match für die Transformation *MoveAggr*. Wendet man diese zweimal an (einmal für das obige Match und einmal für das Match, bei dem die Aggregation „Wohnort“ statt „Strasse“ verwendet wird), so entsteht der folgende Schema-Instanz-Graph:



Die verwendeten Transformationen sind kapazitätserhaltend. Wegen der Transitivität der Kapazitätseigenschaften ist auch ihre Komposition kapazitätserhaltend. Auf diese Weise können aus den atomaren Transformationen komplexere Kommandos zusammengesetzt werden. Die obige Transformationen läßt sich in allgemeinerer Form etwa so notieren:

```

transformation SplitComplexType( c:Complex; newName:String;
role1,role2:String; aggrs:Set of Aggr; assocs:Set of Assoc) = {
  NewComplexType( c, newName, role1, role2);
  for a in aggrs do MoveAggr( a, c.role1 );
  for a in assocs do MoveAggr( a, c.role1 );
}

```

Die Umkehrung dieser Transformation kann durch Verwendung von *MoveAggr*, *MoveAssoc* und *RemoveComplexType* realisiert werden.

8 Zusammenfassung und Ausblick

Bei der Umstellung und Anpassung bestehender Informationssysteme auf neue Technologien stellt die Anpassung der verwendeten Datenbanken ein besonders wichtiges Problem dar.

Die bestehenden Systeme basieren dabei oft auf unterschiedlichen Technologien. Für ein Werkzeug zur Unterstützung solcher Migrations- und Integrationsprozesse ist es daher wichtig, einerseits möglichst viele Legacy-Systeme zu berücksichtigen, andererseits aber auch von konkreten Systemen abstrahieren zu können. Auf Datenbanksysteme gemünzt bedeutet dies, die Struktur von Datenbanken unabhängig von konkreten Systemen und Modellen beschreiben und bearbeiten zu können.

Für diese Aufgaben wurde in dieser Arbeit mit dem Migrationsmetamodell ein Rahmenwerk geschaffen. Es wurde zunächst formal definiert, und aufbauend auf dieser Definition wurde mit den Schema-Instanz-Graphen eine Möglichkeit geschaffen, eine Datenbank mit ihrem Inhalt graphisch zu modellieren.

Zur Bearbeitung der so modellierten Datenbank wurden Restrukturierungstransformationen als Graphersetzungsregeln für die Schema-Instanz-Graphen spezifiziert. Zur Bewertung dieser Transformationen wurden mit den Begriffen Kapazität eines Schemas und Reversibilität einer Schematransformation zwei aus der Literatur bekannte Konzepte auf die Schema-Instanz-Graphen und ihre Transformationen angewendet. Dabei konnte gezeigt werden, wie diese - bisher nur unabhängig voneinander betrachteten - Konzepte zusammenhängen.

Bei der Darstellung von Redesign-Transformationen als Graphersetzungsregeln für den Schema-Instanz-Graphen zeigte sich, daß eine solche Beschreibung zwar schwierig - es mußten komplexe Graphenelemente wie Mengenknoten und Desynchronisationskanten eingeführt werden - aber immerhin für wesentliche Elemente, insbesondere die im VARLET-Projekt wichtigen Schemaelemente, möglich ist. Lediglich im Fall der Instanzmengen von Aggregationen und Assoziationen mußte auf eine algebraische Definition dieser Konstrukte zurückgegriffen werden. Weiterhin wurde dabei gezeigt, wie sich das Konzept der parallelen Graphersetzungsregeln mit den komplexen Graphenelementen verbinden läßt. Ein formaler Nachweis der Verträglichkeit wurde allerdings noch nicht geführt.

Schließlich wurde ein Katalog von grundlegenden Redesign-Transformationen spezifiziert und ihre Eigenschaften mithilfe der genannten Begriffe diskutiert. Dabei wurde demonstriert, wie die grundlegenden Transformationen zu komplexeren Kommandos zusammengesetzt werden können.

Die Redesign-Transformationen wurden in einer an Progres orientierten Notation spezifiziert. Daher können sie leicht in die Varlet-Umgebung, die ebenfalls auf Progres basiert, integriert werden. Es soll in Varlet nur die Strukturtransformation realisiert werden, da für die Instanztransformation ja die gesamte Datenbank mit ihren unter Umständen sehr großen Datenbeständen als Schema-Instanz-Graph in interne Datenstrukturen eingelesen werden müßte. Daher muß bei der Implementierung auch die für einige Transformationen recht komplexe algebraische Konstruktion der Instanztransformation durch die Interpretation der M^3 -Schemata nicht berücksichtigt werden. Die Erkenntnisse über die Kapazitätseigenschaften, die in dieser Arbeit aus diesen Konstruktionen gewonnen wurden, können aber in Varlet benutzt werden, um den Anwender über die Konsequenzen der Strukturtransformationen, die er durchführt, zu informie-

ren.

Der Mechanismus, die grundlegenden Transformationen zu komplexeren zusammzusetzen, kann genutzt werden, um daraus eine Bibliothek häufig durchgeführter Redesign-Schritte zu realisieren. Auch eine solche Bibliothek könnte dann in Varlet verwendet werden.

Es existieren Ideen, den Nachweis der Kapazitätseigenschaften anhand der graphischen Eigenschaften der Redesign-Transformationen nachzuweisen. Dazu wurden in [JZ98] Kriterien angegeben, anhand derer sich die Reversibilität einer Transformation nachweisen läßt. Die Kriterien wurden bisher jedoch nur für Transformationen formuliert, die sich mit einfachen Graphenelementen darstellen lassen. Ob sie sich auch auf komplexe Graphenelemente verallgemeinern lassen, ist noch unklar. Insbesondere die für einige Transformationen notwendige algebraische Spezifikation der Instanztransformation scheint für diese Art von Kriterien nur schwer zugänglich zu sein, da sie sich im Allgemeinen nicht graphisch formulieren läßt.

Literaturverzeichnis

- [BCN92] C. Batini, S. Ceri, S. B. Navathe. *Conceptual Database Design - An Entity-Relationship Approach*. Benjamin/Cummings Publishing. 1992
- [Bewermeyer98] B. Bewermeyer: *Cliché-Erkennung in relationalen Datenbankanwendungen*, Diplomarbeit, Universität-Gesamthochschule Paderborn, September 1998
- [BP96] M. Blaha and W. Premerlani. *A Catalog of Object Model Transformation*. Presented at *3rd Working Conference on Reverse Engineering, Monterey, California*. November 1996
- [Cattell97] R.G.G. Cattell et al. *Object Database Standard: ODMG 2.0*, Morgan Kaufmann Publishers, Inc. 1997
- [DD97] C. J. Date, H. Darwen: *A Guide to the SQL Standard*. Fourth Edition, Addison-Wesley, 1997
- [DT95] C. Delobel, D. Tallot: *Algorithmic Specification for the Relational/Object Oriented Transformation*, SYNDAMMA Esprit III-9006, Reference: SYN-D-SPC-3.1.O.2-1.1, December 18, 1995
- [Ehrig79] Hartmut Ehrig. *Introduction to the Algebraic Theory of Graph Grammars*. In: *G. Goos and J.Hartmanis (Ed.) Lectunr Notes in Computer Science 73, Graph-Grammars and Their Application to Computer Science and Biology*. Springer-Verlag, Berlin, Heidelberg, New York 1979
- [Ehrig87] H. Ehrig. *Tutorial Introduction to the Algebraic Approach of graph grammars*. In: *H. Ehrig, M. Nagl, G. Rozenberg, (Ed.) Proc. of the 3rd International Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*. Springer Verlag 1987.
- [EHL97] H. Ehrig, R.Heckel, M. Löwe. *Algebraic Approaches to Graph Transformation*. In: *G. Rozenberg (Ed.) Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific Publishing 1997.
- [EKL91] H. Ehrig, M. Korff, M. Löwe. *Tutorial Introduction to the Algebraic Approach of graph grammars based on double and single pushouts*. In: *H. Ehrig, H.-J. Kreowski, G. Rozenberg, (Ed.) Proc. of the 4th International Workshop on Graph Grammars and their Application to Computer Science, LNCS 532*. Springer Verlag 1991.
- [EN94] R. Elmasri, S. B. Navathe: *Fundamentals of Database Systems, Second Edition*, Benjamin/Cummings Publishing Company, 1994
- [Fagin77] R. Fagin. *Multivalued dependencies and a new normal form for relational databases*. In: *ACM TODS, Vol 2, No 3*, 1977.
- [Fong97] J. Fong. *Converting Relational to Object-Oriented Databases*. In *SIGMOD Record*, Vol. 26, No. 1, March 1997.

- [FV93] C. Fahrner and G. Vossen. Transforming Relational Database Schemas into Object-Oriented Schemas according to ODMG-93. In *Proc. of the 4th Int. Conf. of on Deductive and Object-Oriented Databases 1995*, 1995.
- [Hainaut91] J.-L. Hainaut. Entity-generating Schema Transformations for Entity-Relationship Models. In: *Proceedings of the 10th Entity-Relationship Conference, San Mateo*, 1991
- [HCDM92] J.-L. Hainaut, M. Cadelli, B. Decuyper, O. Marchand. TRAMIS: a transformation-based database CASE tool. In: *Proceedings of the 5th International Conference on Software Engineering and its Applications, Toulouse*, December 1992
- [HTJC93] J.-L. Hainaut, C. Tonneau, M. Joris and M. Chandelon. Schema transformation techniques for database reverse engineering. In *Proc. of the 12th Int. Conference of the Entity-Relationship Approach, Arlinton-Dallas*, December 1993.
- [Hohenstein93] Uwe Hohenstein. Formale Semantik eines erweiterten Entity-Relationship-Modells. B. G. Teubner Verlagsgesellschaft, Stuttgart, Leipzig. 1993
- [Holle97] J. Holle. Ein Generator für integrierte Werkzeuge am Beispiel der object-relationalen Datenbankschemamigration. Diplomarbeit, Universität-Gesamthochschule Paderborn, Juli 1997.
- [JJ94] M.A. Jeusfeld and U.A. Johnen. An Executable Meta Model for Re-Engineering of Database Schemas. In: *Proc. of the 13th Int. Conference of the Entity-Relationship Approach, Manchester*, December 1994.
- [JSZ96] J. Jahnke, W. Schäfer and A. Zündorf. A Design Environment for Migrating Relational to Object Oriented Database Systems. In: *Proc. of the 1996 Int Conference on Software Maintenance (ISCM'96)*. IEEE Computer Society, 1996.
- [JSZ97] J. Jahnke, W. Schäfer and A. Zündorf. Generic Fuzzy Reasoning Nets as a Basis for Reverse Engineering Relational Database Applications. In: *Proc. of European Software Engineering Conference (ESEC/FSE)*, number 1302 in LNCS. Springer, September 1997.
- [JW98] J. Jahnke, J. Wadsack. Towards Integration of Analysis and Redesign Activities in Information System Reengineering. Technischer Bericht, Universität-Gesamthochschule Paderborn. 1998.
- [JZ98] J. Jahnke, A. Zündorf. Using Graph Grammars for Building the Varlet Database Reverse Engineering Environment. Accepted for *1998 Intl. Workshop on Theory and Applications of Graph Grammars, Paderborn, Germany*. November 1998.
- [Lef95] M. Lefering. Integrationswerkzeuge in einer Softwareentwicklungsumgebung. Informatik, Verlag Shaker, 1995.
- [Nagl79] Manfred Nagl. A Tutorial and Bibliographical Survey on Graph Grammars. In: *G. Goos and J.Hartmanis (Ed.) Lecture Notes in Computer Science 73, Graph-Grammars and Their Application to Computer Science and Biology*. Springer-Verlag, Berlin, Heidelberg, New York 1979

- [Nickel97] U. Nickel. Konzeption einer objektorientierten Bibliothek für gemischt analog/digitale Schaltkreise. Diplomarbeit. Universität-Gesamthochschule Paderborn. 1997
- [ONTOS96] ONTOS Object Integration Server for Relational Databases 2.0: Schema Mapper User's Guide, ONTOS, Inc., 1996
- [OpenDM96] OpenDM/ODMG-93 1.0 Technical Reference Manual, C-LAB, Paderborn, 1996
- [Radeke95] E. Radeke: Federation and Migration Among Database Systems, Dissertation, Universität-Gesamthochschule Paderborn, 1995
- [RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, N. J. 07632, 1991
- [RH97] S. Ramanathan and J. Hodges. Extraction of Object-Oriented Structures from Existing Relational Databases. In: *SIGMOD Record*, Vol. 26, No. 1, March 1997.
- [Rozenberg97] Grzegorz Rozenberg, editor. Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific, Singapore, 1997.
- [RS97] J. Rehers, A. Schürr. Defining and Parsing Visual Languages with Layered Graph Grammars. In: *Journal of Visual Languages and Computing*, Vol. 8, No. 1. Academic Press, London, 1997.
- [Schalldach98] Heike Schalldach, Integration von JAVA-Anwendungen mit relationalen Informationssystemen, Diplomarbeit, Universität-Gesamthochschule Paderborn, 1998.
- [Schiefer93] Bernhard Schiefer. Eine Umgebung zur Unterstützung von Schemaänderungen und Sichten in objektorientierten Datenbanksystemen. Dissertation. Karlsruhe, 1993.
- [Schürr94] A. Schürr. Specification of Graph Translators with Tripel Graph Grammars. In: G. Tinhofer (ed.): *Proc. WG 94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany*, Juni 1994. LNCS 903, Berlin, Springer Verlag (1994), 151-163.
- [SL90] A. Sheth, J. Larson: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases, In: *ACM Computing Surveys*, Band 22, Nr.3, S. 183-236, September 1990
- [SST97] G. Saake, C. Türker, I. Schmitt: Objektdatenbanken: Konzepte, Sprachen, Architekturen, International Thomson Publishing GmbH, Bonn, 1997
- [Taentzer96] G. Taentzer. Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems. Dissertation. Technische Universität Berlin. 1996
- [Tresch95] Markus Tresch. Evolution in Objekt-Datenbanken. B. G. Teubner Verlagsgesellschaft, Stuttgart, Leipzig. 1995
- [Ullman88] J. D. Ullman. Database and Knowledge-Base Systems. Computer Science Press. 1988

[Wadsack98] J. Wadsack. Inkrementelle Konsistenzerhaltung in der transformationsbasierten Datenbankmigration. Diplomarbeit, Universität-Gesamthochschule Paderborn, 1998.

[Zündorf95] A. Zündorf. PROgrammierte GRaphErsetzungsSysteme (Spezifikation, Implementierung und Anwendung einer integrierten Entwicklungsumgebung), Dissertation RWTH Aachen. Deutscher Universitätsverlag (1995)