Abbildungen zwischen Schemata unterschiedlicher Datenmodelle*

Susanne Braun

Technische Universität Kaiserslautern, Lehrgebiet Datenverwaltungssysteme

1 Einleitung

In den letzten Jahrzehnten wurde in vielen Bereichen komplexe und sehr spezialisierte Software entwickelt. Wie der Name Informatik schon sagt, beschäftigt sie sich hautpsächlich mit der Verarbeitung, Verwaltung und Bereitstellung von Informationen. Es gibt die unterschiedlichsten Datenhaltungssysteme, angefangen beim Dateisystem über Datenbanksysteme bis hin zum World Wide Web. Hinzu kommt Anwendungssoftware, welche die Daten gemäß ihrem Anwendungsbereich strukturiert, und zur Speicherung dieser Daten die Datenhaltungssysteme benutzt. Gleichzeitig besteht der Bedarf, die Informationen, die in den unterschiedlichen Datenhaltungssystemen vorliegen, miteinander verknüpfen zu können.

Beispiel 1. Im Automobilbau kommen eine Vielzahl an Softwaresystemen zur Verwaltung von Fahrzeugdaten zum Einsatz – Computer-Aided-Design-Systeme (CAD-Systeme), Produktdaten-Managementsysteme, Enterprise-Resource-Planning-Systeme, Dokumenten-Managementsysteme, um nur einige zu nennen. Das Produktmanagement würde enorm profitieren, wenn ein einheitlicher, transparenter Zugriff auf alle Daten möglich wäre, die in Zusammenhang mit einem Fahrzeug stehen.

Viele Lösungsansätze basieren darauf, die Schemata der beteiligten Datenhaltungssysteme in ein globales Schema, das „Zielschema“, zu integrieren. Für die Schemaintegration wird eine Abbildung, zwischen den Datenmodellen der „Quellsysteme“ und dem Datenmodell, in dem das Zielschema modelliert wird, benötigt.

In dieser Arbeit werden zwei Teilthemen schwerpunktmäßig behandelt. Ein Schwerpunkt wird auf Abbildungen zwischen relationalen Daten und objektorientierten Daten liegen. Das Entity-Relationship-Modell wird heute in einigen Anwendungsbereichen als semantisch nicht reichhaltig genug angesehen, um bestimmte Konzepte der realen Welt zufriedenstellend nachbilden zu können. Die Fähigkeit zur Umstrukturierung relationaler Daten gemäß den Modellierungskonstrukten anderer Datenmodelle ist von Interesse, weil relationale Datenbanksysteme eine bewährte Technologie sind und die meisten Daten in relationaler

* Arbeit im Rahmen des Seminars „Mastering the Information Explosion – Information Integration and Information Quality“

Form vorliegen. Durch den Einsatz von objektrelationaler Middleware wird versucht, die Vorzüge objektorientierter Modellierungstechniken und die von relationalen Datenbanksystemen miteinander zu verbinden.

Der zweite Schwerpunkt liegt auf Abbildungen zwischen relationalen Daten und XML-Daten. Zusammen mit dem World Wide Web hat die Extensible Markup Language (XML) starke Verbreitung gefunden. Auch in diesem Kontext gilt, dass aus Sicht der Anwendung XML die gewünschte Repräsentationsform der Daten darstellt. Aus Performancegründen sollten Daten, für die eine hohe Änderungsrate zu erwarten ist, jedoch nach wie vor in relationaler Form gespeichert werden.

Im folgenden Abschnitt soll eine Einführung in die grundlegenden Begriffe im Zusammenhang mit Schemaintegration und -transformation erfolgen. In Abschnitt 3 wird eine Klassifikation der Probleme und Konflikte vorgestellt, die bei der Schemaintegration und -transformation auftreten können. Mit der Zielsetzung, integrierte objektorientierte Schemata zu erstellen, werden in Abschnitt 4 Techniken zur Abbildung relationaler Daten auf objektorientierte Daten vorgestellt. Abschnitt 5 konzentriert sich auf die Untersuchung von Abbildungen zwischen relationalen Daten und XML-Daten.

2 Grundlagen

Der Prozess der Integration von Daten, die nach unterschiedlichen Datenmodellen strukturiert sind, erfolgt im Allgemeinen in 4 Stufen. Dies ist in der 5-Ebenen-Schema-Architektur nach Sheth und Larson [13] anschaulich dargestellt. Um die Elemente dieser Referenzarchitektur wieder ins Gedächtnis zu rufen, erfolgt in den nächsten Unterabschnitten eine kurze Wiederholung.

2.1 Die 5-Ebenen Schema-Architektur nach Sheth und Larson

In Abbildung 1 ist die 5-Ebenen-Schema-Architektur graphisch dargestellt. In die Föderation gehen n potenziell heterogene Datenbanksysteme ein. Jedes solche Komponenten-Datenbanksystem verfügt über ein *lokales Schema*. Dies ist das konzeptionelle Schema, das von den lokalen Anwendungen genutzt wird. Dementsprechend liegt es in dem Datenmodell des Datenbankmanagementsystems der Komponenten-Datenbank vor.

Das *Komponentenschema* entsteht durch Transformation des lokalen Schemas in das *kanonische Datenmodell*. Die Wahl eines sinnvollen kanonischen Datenmodells spielt für den gesamten Integrationsprozess eine entscheidende Rolle. Wählt man ein Datenmodell, das über weniger semantische Modellierungskonstrukte verfügt als das Datenmodell eines Komponentensystems, so können im föderierten Schema unter Umständen nicht alle semantischen Zusammenhänge des lokalen Schemas ausgedrückt werden. Wählt man im Gegenzug ein mächtiges Datenmodell für das föderierte Schema, so bietet es sich an, die lokalen Schemata mit Semantik anzureichern. Die hinzugekommene Semantik kann jedoch

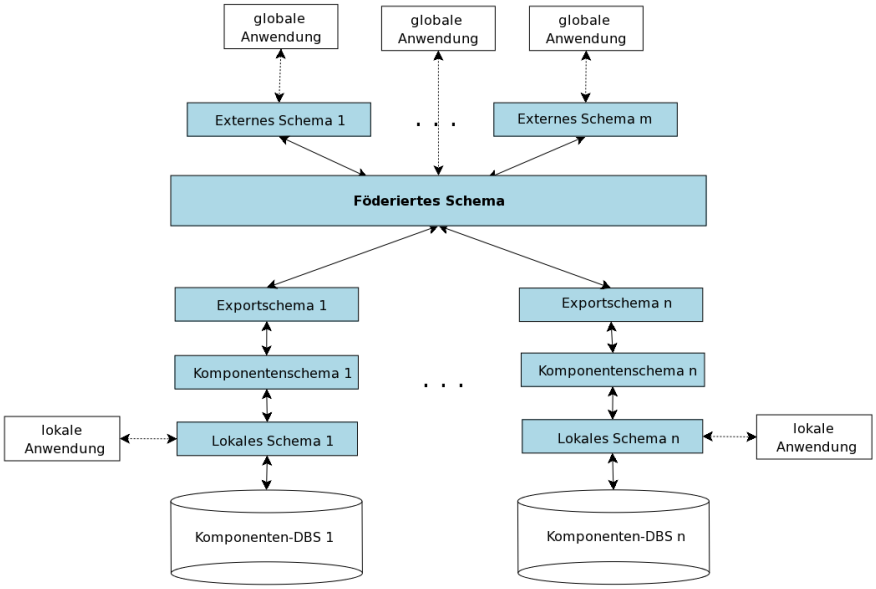


Abb. 1. Die 5-Ebenen-Schema-Architektur eines föderierten Datenbanksystems.

nach wie vor nicht in den lokalen Schemata dargestellt werden, und damit auch a priori nicht in den Komponentensystemen gespeichert werden.

Je nach Autonomität eines Komponentensystems teilt dieses einen größeren oder einen kleineren Teil seiner Daten mit dem föderierten Datenbanksystem. Diejenigen Daten, die in dem Schema der Föderation enthalten sein können, werden durch das *Exportschema* festgelegt. Vergleichbar ist dies mit dem Sichtenkonzept in SQL, das es ermöglicht den Zugriff auf bestimmte Daten einzuschränken.

Das *föderierte Schema* entsteht durch Integration der Exportschemata und entspricht dem Schema des föderierten Datenbanksystems. Das *externe Schema* ist wiederum mit dem Sichtenkonzept in SQL vergleichbar, und ermöglicht es, für unterschiedliche Benutzergruppen angepasste Sichten auf das föderierte Schema zu definieren.

Diese Architektur wird auch als Referenz-Architektur bezeichnet, weil sie die Architektur aller existierenden föderierten Datenbanksysteme umfasst. Die Unterschiede zwischen den Systemen sind darauf zurückzuführen, dass zwei oder mehr Ebenen der Architektur zusammenfallen.

Die verfügbaren Systeme unterstützen nur bestimmte Datenmodelle und damit auch nur bestimmte Datenbanksysteme. Häufig wird eine Anfragesprache zur Verfügung gestellt, die es erlaubt, aufbauend auf diesen Datenmodellen Anfragen zu stellen. Damit ist es theoretisch ausreichend, das föderierte Schema als Vereinigung aller Komponentenschemata zu betrachten. Eine Transformation

der lokalen Schemata in das kanonische Datenmodell wäre damit unnötig, weil das kanonische Datenmodell auch einfach der Vereinigung der beteiligten Datenmodelle entspräche. Letztere Vorgehensweise wird auch als sprachgekoppelt bezeichnet und bei so genannten „lose gekoppelten“ Systemen verfolgt.

2.2 Das Konzept der Prozessoren

Prozessoren sind Programme, die Daten oder Anfragen transformieren können. Viele Techniken, die in Integrationssystemen zum Einsatz kommen, kann man sich abstrakt als einen Prozessor vorstellen (vgl. [13]).

Transforming Processor. Ein Transforming Processor übersetzt entweder Anfragen einer Sprache in eine andere Sprache oder Daten eines bestimmten Formats in ein anderes Format. In unserem Fall ist eine Anfragesprache an eine beschränkte Anzahl von Datenmodellen gebunden, die von ihr unterstützt werden. Man kann sich die Abbildung einer Anfrage auf ein Datenmodell D', das von von einer Anfragesprache A nicht unterstützt wird, durch ein Paar von Transformationsprozessoren vorstellen. In Abbildung 2 ist ein solches Paar dargestellt. Der erste Prozessor übersetzt von der Anfragesprache A in die Anfragesprache A' von D'. Dazu benutzt er die Abbildungsvorschriften zwischen dem Schema S und dem Schema S'. Der zweite transformiert die Ergebnisse von A' zurück in die ursprüngliche Repräsentation des Datenmodells D. Ein generischer Transformationsprozessor holt sich die Definition der Abbildung dynamisch aus einer Datenstruktur.

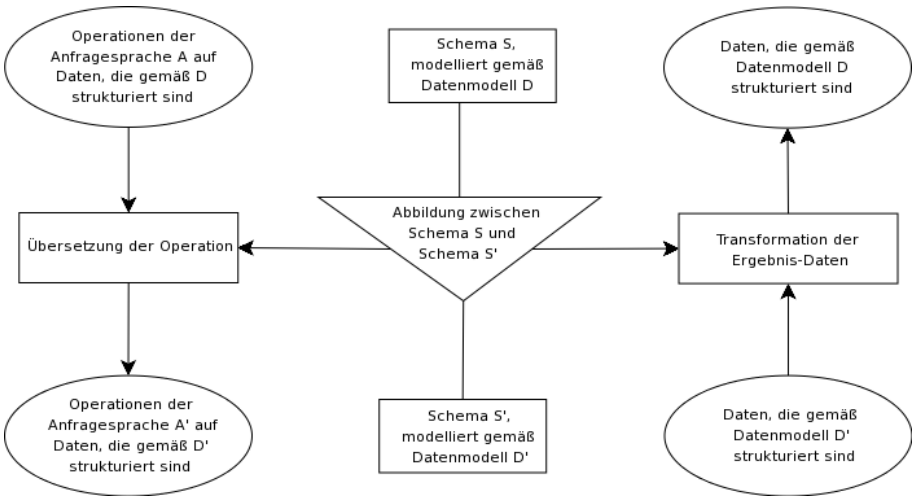


Abb. 2. Paar von Transformationsprozessoren zur Realisierung einer Abbildung zwischen unterschiedlichen Datenmodellen

Filtering Processor. Durch einen Filtering Processor (vgl. Abbildung 3) kann die Einhaltung von Constraints und referentieller Integrität realisiert werden. Per Definition kontrolliert ein Filtering Processor die Zulässigkeit von Operationen und Anfragen bzw. die Zugriffsrechte auf Daten.

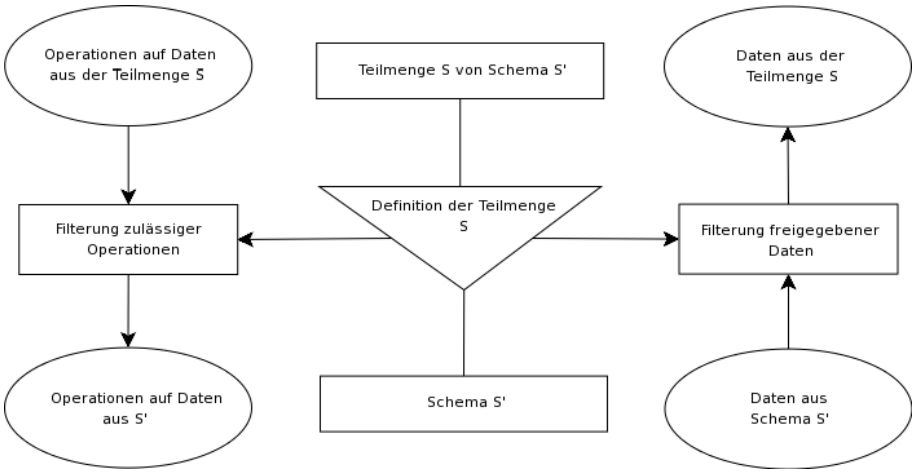


Abb. 3. Paar von Filterprozessoren zur Realisierung von semantischer Integrität und Zugriffsrechten

Constructing Processor. Ein Constructing Processor (vgl. Abbildung 4) kann Anfragen/Operationen in einzelne Teile zerlegen, so dass jeder Teil von einem anderen Prozessor als Eingabe akzeptiert und weiterverarbeitet werden kann. Die Teil-Ergebnisse der Anfrage/Operation, die von den einzelnen Prozessoren zurückgeliefert werden, werden von ihm zusammengeführt und in einem einheitlichen Datenmodell repräsentiert. In Abbildung 4 ist ein abstrakter Constructing Processor dargestellt.

Damit stellen die drei Prozessorarten Abstraktionen von Systemkomponenten eines föderierten Datenbanksystems dar. In Abbildung 5 ist das Zusammenspiel zwischen den Schemata der fünf Ebenen und der Prozessoren graphisch dargestellt.

2.3 Erweiterungen der 5-Ebenen Schema-Architektur

Auf den ersten Blick mag diese Architektur nicht allgemein genug für Anwendungsfälle erscheinen, in denen auch Nicht-Datenbanksysteme zu integrieren sind. Es sollen beispielsweise (CAD-)Daten aus proprietären Dateisystemen integriert werden, oder Informationen aus Webseiten ausgelesen werden. Für diese

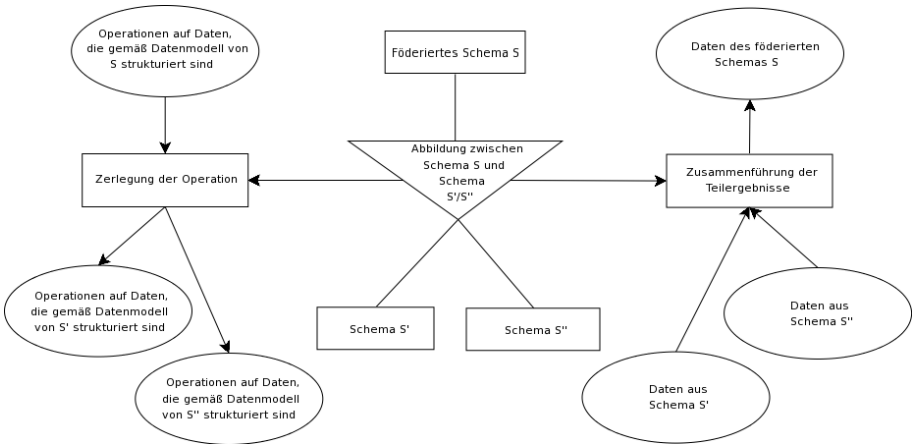


Abb. 4. Schemaintegration und Zerlegung von Anfragen durch einen Constructing Processor

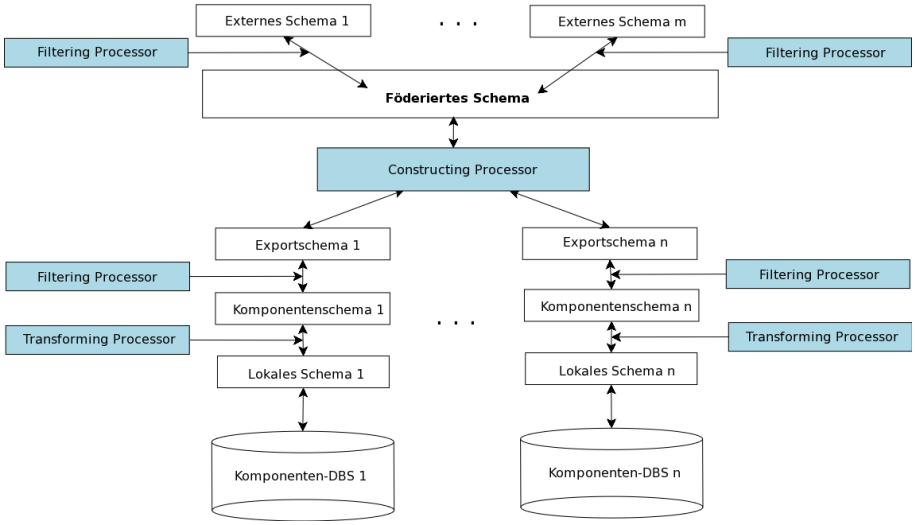


Abb. 5. Zusammenhang zwischen den fünf Schema-Ebenen und Prozessoren

Daten ist aber gar kein Schema vorhanden. Eine mögliche Lösung bietet der Einsatz von Wrappern. Ein Wrapper entspricht im Wesentlichen einem Transforming Processor, der nach außen eine relationale Schnittstelle zur Verfügung stellt. Der Zugriff auf die Daten kann so erfolgen, als ob sie in Relationen organisiert wären. Im Allgemeinen wird nur eine Teilmenge von SQL unterstützt, zumindest sollte aber ein `SELECT * FROM T` möglich sein. Für weiterführende Informationen zur Wrapper-Technologie sei auf die Literatur [10] verwiesen.

Für Integrationssysteme, wie sie im folgenden vorgestellt werden, benötigt man noch ein weiteres Schema für die Metadaten der Integration. In [13] wird dieses mit „auxiliary schema“ bezeichnet. Dieses beinhaltet dann unter anderem die Abbildungsvorschriften.

2.4 Zusammenfassung

Um die Integration mehrerer Quellschemata vornehmen zu können, muss eine Abbildung zwischen den Objekten des Zielschemas und den Objekten der Quellschemata gegeben sein. Dies beinhaltet eine Abbildung zwischen potenziell heterogenen Datenmodellen. Zur Definition einer Abbildung, die beispielsweise von einem generischen Transforming Processor benutzt werden kann, muss es einen Formalismus (eine Abbildungssprache) geben, der es erlaubt, die Abbildung zu spezifizieren.

Erfolgt die Datenintegration in einem föderierten Schema, und nicht im Anwendungsprogramm, so erhöht sich die Datenunabhängigkeit. Dem Anwendungsprogrammierer wird es ermöglicht, deklarative Anfragen wie „`Gib mir alle Sportwagen`“ zu stellen. In der zwischengeschalteten Middleware ist eine mithilfe der Abbildungssprache formulierte Abbildung zwischen dem Objekttyp `sportwagen` auf die reale Repräsentation der Daten, die einen Sportwagen charakterisieren, vorhanden. Kann die Definition der Abbildungsvorschriften ebenfalls deklarativ erfolgen, so erhöht sich Flexibilität und Wartungsfreundlichkeit des föderierten Systems.

3 Problematik der Schemaintegration und -transformation

Bei der Schemaintegration und -transformation können Konflikte auftreten. Dabei unterscheidet man strukturelle Konflikte und semantische Konflikte. Semantische Konflikte treten auf der Ebene des Schemas auf, und sind umso seltener, je höher die semantische Kongruenz zwischen den beteiligten Schemata ist. Mit semantischer Kongruenz ist „die Übereinstimmung des betrachteten Ausschnitts der realen Welt“ [11] in den beteiligten Schemata gemeint, wohingegen strukturelle Kongruenz „die Übereinstimmung in der Art der Datenrepräsentation“ [11] der beteiligten Schemata bezeichnet. Interessiert man sich für die Kongruenz zwischen Zielschema und Quellschema, so spricht man von vertikaler (semantischer oder struktureller) Kongruenz. Analog bezeichnet horizontale Kongruenz

die Kongruenz zwischen den beteiligten Quellschemata.

Im folgenden Abschnitt werden diese Kongruenzen im Zusammenhang mit möglichen Konflikten untersucht. Die beschriebene Klassifikation geht auf [11,6] zurück. Neben dieser gibt es auch noch andere Klassifikationsansätze, z.B. in [2] oder [8]; es handelt sich dabei aber um gröbere Einteilungen.

3.1 Elementare Abbildungsprobleme

An diese Stelle sollen die elementaren Abbildungsprobleme analysiert werden. Elementar deshalb, weil es sich um inhärente Konflikte der Schematransformation handelt. Viele der beschriebenen Probleme würden schon auftreten, würde man die Transformation innerhalb eines Datenmodells durchführen. Deshalb werden die neutralen Begriffe Objekt und Objekttyp verwendet.

Semantische Kongruenz. Bei einer Schemaintegration kann davon ausgegangen werden, dass keine vertikale semantische Inkongruenz vorliegt, da sonst eine Integration keinen Sinn machen würde. Die betrachteten Weltausschnitte des Zielschemas (Z) und der Quellschemata (Q_i) sind also identisch. In den allermeisten Fällen werden aber die Anwendungsbereiche nicht identisch, sondern nur überlappend sein.

Beispiel 2. Die Ingenieure und Konstrukteure eines Automobilherstellers sind an dem gleichen Objekt der Realität interessiert, einem bestimmten Fahrzeug. Beide Personengruppen kommen aus unterschiedlichen Anwendungsbereichen, und der Aufbau eines Fahrzeugs gliedert sich für beide nach unterschiedlichen Gesichtspunkten. So orientiert sich die Produktstruktur bei den Konstrukteuren am konstruktiven Aufbau des Fahrzeugs, wohingegen sich die Produktstruktur bei den Ingenieuren nach dem funktionalen Aufbau eines technischen Systems gliedert.

Bidirektionale Abbildungen. Bei einer bidirektionalen Abbildung muss nicht nur eine Abbildungsvorschrift von $Q_i \rightarrow Z$ vorhanden sein, sondern auch eine von $Z \rightarrow Q_i$. Sollen also Änderungen, die auf den Objekten des Zielschemas gemacht werden, persistent gespeichert werden, so müssen sie in die Objekte der Quellschemata Q_i propagiert werden. Ist die Ursprungsvorschrift $Q_i \rightarrow Z$ nicht eindeutig umkehrbar, so spricht man vom View-Update-Problem. Es kann auch vorkommen, dass für die Abbildung $Z \rightarrow Q_i$ eine andere gewünscht wird als die, die sich als Umkehrung der Abbildung $Q_i \rightarrow Z$ ergibt.

Beispiel 3. In Abbildung 6 entsteht der Zielobjekttyp `Teil` durch eine Verbundoperation auf den Entities `TEIL` und `PERS`. Wird nun auf dem Zielobjekttyp eine Änderung des Attributes `erstellt_von` vorgenommen, so würde die Anwendung der Umkehrabbildung zur Folge haben, dass der Name des Konstrukteurs geändert wird. Im Allgemeinen wird man mit dieser Operation aber bezwecken wollen, das der Konstrukteur des Teils geändert wird. Mit anderen Worten, der

Fremdschlüssel in `TEIL` soll auf den Primärschlüssel derjenigen Person aus `PERS` zeigen, deren Name mit dem geänderten Wert in `erstellt_von` übereinstimmt.

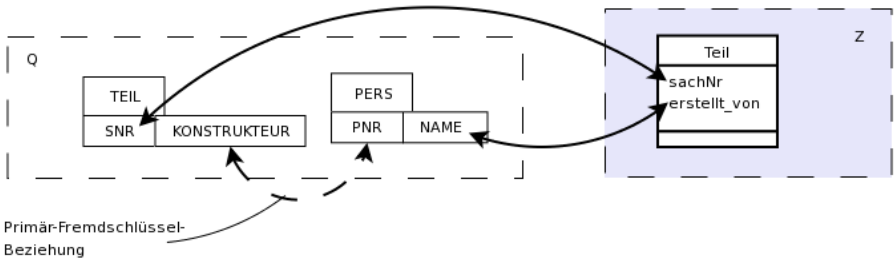


Abb. 6. Verbundoperation auf Objekten eines Quellschemas Q

Attributkorrespondenzen. Eine Attributkorrespondenz drückt einen Zusammenhang zwischen Attributen des Zielschemas und Attributen eines Quellschemas aus. Im einfachsten Fall entspricht jedes Attribut des Zielschemas genau einem Attribut des Quellschemas. Dann spricht man von einer 1:1-Attributkorrespondenz.

Eine 1:n-Attributkorrespondenz liegt dann vor, wenn ein Attribut eines Quellschemas mit mehreren Attributen des Zielschemas in Zusammenhang steht. Zum Beispiel dient die E-Mail-Adresse gleichzeitig als Benutzername eines Accounts.

Analog wird bei der n:1-Attributkorrespondenz ein Attribut des Zielschemas aus n Attributen des Quellschemas abgeleitet. Eine n:1-Attributkorrespondenz kann auch eine bedingte Abbildung ausdrücken:

Beispiel 4. Ein Hersteller verwaltet seine Produktbeschreibungen in mehreren Sprachen. Je nach Produkt wird `beschreibung` im Zielschema auf `beschreibung_deutsch` oder `beschreibung_englisch` abgebildet.

Entsprechend wird mit einer n:m-Attributkorrespondenz ein komplexer Zusammenhang zwischen n Attributen eines Quellschemas und m Attributen des Zielschemas ausgedrückt. Die Umrechnung von kartesischen Koordinaten in Zylinderkoordinaten eines Punktes ist so ein Beispiel.

Objekttypkorrespondenzen. In analoger Weise können Korrespondenzen auf Objekttypebene betrachtet werden. Bei der 1:n-Objekttypkorrespondenz beispielsweise stehen die Objekte eines Quellschema-Typs T_Q in Zusammenhang mit den Objekten mehrerer Typen T_{Z_i} des Zielschemas.

Bei relationalen, nicht-normalisierten Schemata kann es vorkommen, dass mehrere Konzepte durch nur einen Objekttyp repräsentiert werden. Dann ist es

sinnvoll, diesen Objekttyp im Zielschema in mehrere Teile zu zerlegen, um ein besseres Abbild der Realität zu modellieren und Anomalien zu beseitigen.

Ein weiteres Beispiel sind kontextabhängige Abbildungen. Dabei werden Zielobjekte nach Kontext unterschiedlich auf Objekte der Quellschemata abgebildet.

Vernetzung von Objekttypen. Besteht eine Vernetzung der Quellkonstrukte, so sollen die entsprechenden Referenzen, wie beispielsweise Primär-Fremdschlüssel-Beziehungen, auch im Zielschema gültig sein und müssen entsprechend transformiert werden. Im Zuge einer semantischen Anreicherung können im Zielschema neue Referenzen hinzukommen.

Schemakardinalität. Die beschriebenen Korrespondenzen wurden anhand eines Quellschemas Q und des Zielschemas Z erläutert. Sind n Quellschemata $Q_1, \ldots, Q_n$ zu integrieren, so müssen Objekte, die aufgrund der horizontalen semantischen Kongruenz in mehreren Quellsystemen vorliegen, identifiziert werden, da sie im Zielschema selbstverständlich nur einmal repräsentiert sein dürfen. Die Änderungsoperationen müssen jedoch konsistent in allen Quellsystemen erfolgen. Objekte eines Objekttyps, die über mehrere Komponentensysteme verteilt sind, und unterschiedlich repräsentiert sind, müssen im Zielschema einheitlicht dargestellt werden.

Bezeichner. Ohne standardisierte Bezeichnungen für bestimmte Konzepte kann es vorkommen, dass in den verschiedenen Quellschemata das gleiche Konzept unterschiedliche bezeichnet wird (Synonyme) oder unterschiedliche Konzepte den gleichen Bezeichner haben (Homonyme).

Datentypkorrespondenzen. Aufgrund der Heterogenität der beteiligten Quellsysteme müssen die Datentypen der Quellsysteme gegebenenfalls auf Datentypen des Zielsystems abgebildet werden.

Zusammenfassung. Ein Datenintegrationssystem sollte es ermöglichen, die betrachteten Korrespondenzen zwischen den Objekten aus Z und den Objekten der Q_i, bzw. den Objekten der Q_i untereinander, zu spezifizieren. Um bei bidirektionalen Abbildungen eine globale Überprüfung von referentieller Integrität zu ermöglichen, müssen die Beziehungen zwischen den Objekten bekannt sein. Das Bekanntmachen der Zusammenhänge in dem Datenintegrationssystem hat noch einen weiteren positiven Nebeneffekt. Dadurch erfolgt nämlich gleichzeitig eine Dokumentation der semantischen Abhängigkeiten und Zusammenhänge [11]. Insbesondere sollte eine Abbildungssprache über eine ausreichende Mächtigkeit verfügen, so dass die Zusammenhänge ausdrückbar sind, und die Auflösung eines Konflikts spezifizierbar ist. Aus diesem Grund wurde eine ausführliche Klassifikation der Konfliktarten vorgestellt, da sie die Grundlage für die Entwicklung einer Abbildungssprache mit ausreichender Mächtigkeit darstellen.

3.2 Probleme bei der Umsetzung des objektorientierten Paradigmas

Ein generelles Problem beim Einsatz von Objektorientierung im Zusammenhang mit Datenbanken ist die Tatsache, dass es kein einheitliches Datenmodell gibt, auf das man sich beziehen könnte. Im folgenden beschränke ich mich daher auf Konzepte, die allgemein anerkannt sind.

Abbildung mengenwertiger Attribute. Im Gegensatz zu normalisierten Relationen dürfen Objekte über mengenwertige Attribute verfügen. Um die Elemente eines mengenwertigen Attributes auf einwertige Attribute abzubilden, muss der sogenannte Unnest-Operator angewendet werden.

Abbildung von Spezialisierungs- und Generalisierungsbeziehungen. Ist ein objektorientiertes Schema vorhanden, so kann eine Generalisierungshierarchie automatisch auf Relationen abgebildet werden: durch vertikale Partitionierung, volle Redundanz oder durch das Hausklassenmodell. Liegen die Daten bereits relational vor, und soll durch den Einsatz von Middleware eine objektorientierte Schnittstelle angeboten werden, so muss definiert werden, welche Tupel eines Entities auf welche (Sub-)Typen der Hierarchie abzubilden sind.

Realisierung der Objektidentität. Das objektorientierte Paradigma sieht vor, dass jedes Objekt über eine systemweit eindeutige Identität und eine Lebensdauer verfügt. Objekte mit übereinstimmenden Attributwerten sind nicht identisch und können anhand einer OID unterschieden werden. Die Identifikation von Tupeln hingegen, erfolgt anhand der Werte eines Primärschlüssels. Der Wert dieses Schlüssels wird aus einem oder mehreren Attributwerten gebildet. Im Sinne der Objektorientierung sollte die Identifikation eines Objektes jedoch unabhänigig von Attributwerten erfolgen. Hinzu kommt, dass ein Primärschlüssel nur innerhalb einer Relation und nicht für alle Tupel der Datenbank eindeutig sein muss.

Da die Objekte aber aus relationalen Daten abgeleitet werden, kann die Identität nur anhand von relationalen Attributwerten festgestellt werden. Aufgrund der Autonomität der Quellsysteme können Änderungsoperationen auf den relationalen Attributwerten nicht verboten werden. Trotzdem muss sichergestellt werden, dass ein Objekt des betrachteten Weltauschnittes bei unterschiedlichen Zugriffen jedesmal die gleiche Identität besitzt.

3.3 Probleme bei der Abbildung auf XML-Daten

Erhaltung der Reihenfolge von Elementen. In XML spielt die Reihenfolge der Elemente eine Rolle und ist zu erhalten. Relationale Datenbanksysteme hingegen sind mengenorientiert und kennen keine Reihenfolge.

Abbildung einer hierarchischen Struktur und mengenwertiger Elemente. Die Elemente eines XML-Dokumentes werden hierarchisch geschachtelt und können in beliebiger Kardinalität (+, *) auftreten. Analog zu objektorientierten, mengenwertigen Attributen sind diese auf einwertige Attribute von Relationen abzubilden. Im Gegenzug gibt es sehr viele Möglichkeiten, relationale Daten in einer hierarchischen Struktur anzuordnen (exponentiell in der Anzahl der Spalten, vgl. [9]).

4 Abbildungen zwischen relationalen Daten und objektorientierten Daten

Dieser Abschnitt stellt zwei Systeme vor, die es erlauben, virtuelle integrierte objektorientierte Schemata auf relationalen Quellschemata zu erstellen.

4.1 Pegasus

Systemarchitektur. Pegasus [1,12] ist konzipiert als eigenständiges Datenbankmanagementsystem, das es erlaubt, zusätzlich zu seiner eigenen nativen Datenbank externe, autonome Datenbanken zu integrieren. Der Prototyp unterstützt sowohl relationale als auch objektorientierte Datenbanken. Das kanonische Datenmodell basiert auf dem Datenmodell des objektorientierten Datenbanksystems IRIS. Dieses verfügt über drei wesentliche Konstrukte: Typen, Objekte und Funktionen. Insbesondere werden Attribute durch „get-Funktionen“ repräsentiert. Eine Erweiterung von SQL, „HOSQL“, dient als Data Definition und Data Manipulation Language. wobei nur Lesezugriffe auf das integrierte Schema möglich sind, und somit insbesondere keine bidirektionalen Abbildungen unterstützt werden.

Importieren von Komponenten-Schemata. Um auf die Schemata der Quellsysteme zugreifen zu können, müssen deren Relationen importiert werden. Dazu werden von Pegasus zwei Konstrukte zur Verfügung gestellt: Das Importieren von Typen und das Importieren von Funktionen. Ein importierter Typ bzw. eine importierte Funktion bezieht sich immer auf eine Relation eines Quellschemas. Es wird vorausgesetzt, dass die importierten Typen/Funktionen disjunkt sind; dies kann durch entsprechende Namensgebung erreicht werden.

In Abbildung 7 ist ein objektorientiertes Zielschema dargestellt, in der eine Klasse `Funktionsgruppe` modelliert ist. `Funktionsgruppe` besitzt ein mengenwertiges Attribut `teile`, in der alle Teile, die zu einer Funktionsgruppe gehören, enthalten sind. Eine Funktionsgruppe sei durch ihre Bezeichnung eindeutig identifiziert. Die Definition des Zielobjekttyps `Funktionsgruppe` kann durch die Import-Operationen wie in Beispiel 5 erfolgen.

Beispiel 5. Definition eines Objekttyps.

```
1   REGISTER Relational IBM/DB2 DATASOURCE DB1 'braun@sunflower' AS q;

2   CREATE TYPE Funktionsgruppe AS
3   IMPORTED FROM Relational DATASOURCE q RELATION TEIL
4   PRODUCING BY (FUNKTIONSGRUPPE)
5   FUNCTIONS( bezeichnung STRING UNIQUE AS MATCHING FUNKTIONSGRUPPE );

6   CREATE FUNCTION teile(Funktionsgruppe f) -> SETTYPE (STRING t)
7   AS IMPORTED FROM Relational DATASOURCE q RELATION TEIL(
8      bezeichnung(f) AS MATCHING FUNKTIONSGRUPPE;
9      t AS MATCHING SNR );
```

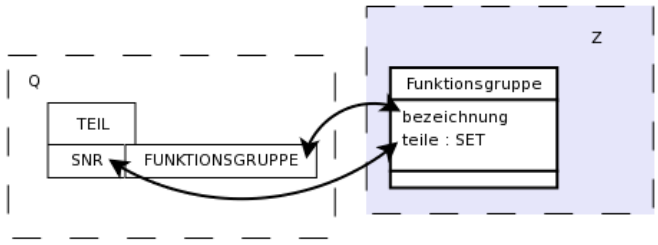


Abb. 7. Abbildung eines Aggregates

Durch die `PRODUCING BY`-Anweisung wird für jeden Wert von `TEIL.FUNKTIONS-GRUPPE` genau eine Instanz im Zielschema erzeugt (vgl. Zeile 4). Innerhalb der Typ-Definition von `Funktionsgruppe` kann nur die Definition des Attributes `bezeichnung` erfolgen (Zeile 5), nicht aber die Definition des Attributes `teile`, weil es keine 1:1-Abbildung zwischen dem mengenwertigen Attribut `teile` und dem Attribut `TEIL.SNR` gibt.

Durch das Importieren eines Objekttyps ist immer eine 1:1-Objekttypkorrespondenz gegeben. Dabei können Attribute, die im Zielschema nicht benötigt werden, herausprojiziert werden. Ein Objekttyp des Zielschemas kann nicht direkt als Verbundobjekt zweier Objekttypen des Quellschemas importiert werden. Um eine Verbundoperation zwischen `TEIL` und `PERS` (vgl. Abbildung 6) auszuführen, müssen zunächst beide Relationen importiert werden (siehe Beispiel 6, Zeile 1–10). Anschließend kann, unter Zuhilfenahme einer benutzerdefinierten Funktion (Zeile 15–21), der gewünschte Objekttyp gebildet werden.

Beispiel 6. Datenbank-übergreifende Verbundoperation.

```
1   CREATE TYPE Teil_q AS
2   IMPORTED FROM Relational DATASOURCE q RELATION TEIL
3   PRODUCING BY (SNR)
```

```
4   FUNCTIONS( snr_q STRING UNIQUE AS MATCHING SNR,
5              konstrukteur_q STRING AS MATCHING KONSTRUKTEUR );

6   CREATE TYPE Pers_q AS
7   IMPORTED FROM Relational DATASOURCE q RELATION PERS
8   PRODUCING BY (PNR)
9   FUNCTIONS( pnr_q STRING UNIQUE AS MATCHING PNR,
10             name_q STRING AS MATCHING NAME );

11  CREATE TYPE Teil AS
12  IMPORTED FROM Relational DATASOURCE q RELATION TEIL
13  PRODUCING BY (SNR)
14  FUNCTIONS( sachNr STRING UNIQUE AS MATCHING SNR );

15  CREATE FUNCTION erstellt_von(Teil t) ->  STRING k
16  AS HOSQL
17  SELECT k
18  FOR EACH Teil_q tq, Pers_q pq
19  WHERE sachNr(t) = snr_q(tq)
20  AND konstrukteur_q(tq) = pnr_q(pq)
21  AND name_q(pq) = k;
```

Die beschriebene Vorgehensweise ist mehr oder weniger umständlich. Dies liegt wohl daran, das solche Abbildungen in Pegasus nicht vorgesehen sind. Aus der Literatur geht nicht eindeutig hervor, ob das mehrfache Importieren einer Relation überhaupt zulässig ist. Theoretisch ist es auf jeden Fall machbar.

Objektidentität. Jedes Objekt des Zielschemas bekommt eine global eindeutige ID. Um die Eindeutigkeit der OID sicherzustellen, wird sowohl der Objekttyp als auch der Wert, der durch Auswerten der `PRODUCING BY`-Anweisung entsteht, zur OID-Bildung verwendet.

Horizontale semantische Kongruenz. Die Grundidee zur Integration eines Konzeptes, das über mehrere Quellsysteme verteilt ist, liegt in der Generalisierung von Typen der Quellschemata. Werden beispielsweise die Angestellten eines Unternehmens abteilungsweise in eigenen Datenbanksystemen verwaltet, so müssen zunächst die Angestellten-Relationen der einzelnen Abteilungen importiert werden. Durch anschließende Definition eines Supertyps, der die importierten Relationen umfasst, kann die Integration der Angestellten-Gruppen erfolgen. Der Vorgang ist in Beispiel 7 skizzenhaft dargestellt.

Beispiel 7. Horizontale semantische Kongruenz.

```
1   REGISTER Relational IBM/DB2 DATASOURCE DB1 'braun@sunflower' AS q1;
2   REGISTER Relational Oracle DATASOURCE DB2 'huba@violin' AS q2;

3   CREATE TYPE Informatiker AS
```

```
4   IMPORTED FROM Relational DATASOURCE q1 RELATION PERS
5   PRODUCING BY (PNR)
6   FUNCTIONS( pnr STRING UNIQUE AS MATCHING PNR ... );

7   CREATE TYPE Ingenieure AS
8   IMPORTED FROM Relational DATASOURCE q2 RELATION PERS
9   PRODUCING BY (PNR)
10  FUNCTIONS( pnr STRING UNIQUE AS MATCHING PNR ... );

11  CREATE TYPE Personal AS
12  AS COVERING SUPERTYPE OF Informatiker, Ingenieure;

13  CREATE FUNCTION pnr(Personal) -> STRING AS NULL; ...
```

Die Instanzen eines Supertyps entsprechen genau der Vereinigungsmenge der Instanzen seiner Subtypen. Insbesondere müssen alle Funktionen eines Supertyps abgeleitete Funktionen oder Dummy-Funktionen wie in Zeile 13 sein. Beim Aufruf einer solchen Dummy-Funktion wird dynamisch die entsprechende Funktion des Subtypen aufgerufen.

Gibt es einen Angestellten, der sowohl als Ingenieur, als auch als Informatiker angestellt ist, so kann die Identität der korrespondierenden Objekte beispielsweise anhand der Personalnummer festgestellt werden. Durch folgende Anweisung, können die beiden Objekte von Pegasus wie ein einziges Objekt behandelt werden.

```
DEFINE OBJECT IDENTITY ON (Informatiker if | Ingenieur ig)
BY Informatiker.pnr(if) = Ingenieur.pnr(ig);
```

4.2 BRIITY

Die Abbildungssprache von BRIITY (Bridging Heterogeneity) [11,6] ist stark an SQL angelehnt und zeichnet sich insgesamt durch ihre Deskriptivität aus. Sie ermöglicht Abbildungen zwischen relationalen und objektorientierten Schemata in beide Richtungen (objektorientierte und relationale Datenbanksysteme sowie relationale und objektorientierte Zielschemata werden unterstützt). BRIITY ist den bisher vorgestellten Verfahren überlegen, insbesondere ermöglicht es die Darstellung von vernetzten Zielobjekttypen und die Definition bidirektionaler Abbildungen. BRITTY wurde so konzipiert, dass ein möglichst hoher Grad an Technologieunabhängigkeit gewährleistet werden kann. Die Sprache erlaubt es „Elemente verschiedener Schemata in Beziehung zueinander zu setzen, die denselben Sachverhalt der realen Welt evtl. auf sehr unterschiedliche Weise darstellen“ [11]. In den kommenden Unterabschnitten werden die wichtigsten Sprachkonstrukte von BRIITY vorgestellt und anhand von Beispielen erläutert, die im Wesentlichen auf [11] zurückgehen.

Der eingeführte Begriff eines Objekttyps entspricht in BRIITY einem Entity, entsprechend bildet die *Entity-Mapping Section* das Herzstück der Abbildungssprache.

Objekttyp- und Attributtypkorrespondenzen. In der Entity-Mapping Section können die Klassen (oder Entities) des Zielschemas definiert werden. In Abbildung 8 entsteht die Klasse Teil des objektorientierten Zielschemas durch eine Verbundoperation aus den Entities TEIL und PERS des relationalen Quellschemas. In Beispiel 8 ist die zugehörige, in BRIITY formulierte Abbildungsvorschrift gegeben.

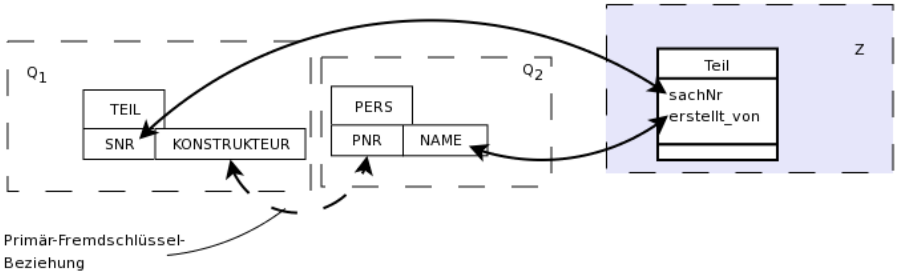


Abb. 8. Datenbank-übergreifende Verbundoperation

Beispiel 8. Datenbank-übergreifende Verbundoperation.

```
1 SCHEMA_DECLARATION
2    MAP SCHEMA  integrierte_sicht
3    ALIAS       z
4    DEF FILE    /usr/sauter/schemas/integrierte_sicht.exp
5    FROM DATABASE teile_db
6       SYSTEM     db2
7       HOST       131.246.173.1
8       ALIAS      q1
9    AND DATABASE  pers_db
10      SYSTEM     oracle
11      HOST       131.246.173.2
12      ALIAS      q2

13 ENTITY_MAPPING
14   MAP  Teil
15   FROM _t := q1.TEIL, _p := q2.PERS;
16      ON_RETRIEVE
17         sachNr = _t.SNR;
18         erstellt_von = _p.NAME;
19         WHERE _t.KONSTRUKTEUR = _p.PNR AND
20               _t.SNR > 100;
       ...
21   END_MAP;
```

```
    ...
22 END_ENTITY_MAPPING;
```

Dem Entity-Mapping geht immer eine Schema-Declaration Section voraus, in der die zur Bildung des föderierten Schemas benötigten Datenbanksysteme aufgeführt werden. Dabei können systemspezifische Angaben gemacht und Aliase definiert werden.

Ausgangspunkt für eine `MAP`-Anweisung in der `ENTITY_MAPPING`-Section ist ein Objekttyp (`Teil`, Z. 14) des Zielschemas. Alle einen Objekttyp betreffenden Abbildungsvorschriften werden in seinem `MAP`-Anweisungsteil vorgenommen. Dadurch erfolgt eine Anlehnung an das objektorientierte Paradigma. Ein Objekttyp des Zielschemas wird abgeleitet aus einem oder mehreren Objekttypen der Quellschemata (`q1.TEIL` und `q2.PERS`, Z. 15). In der retrieve-Klausel (`ON_RETRIEVE`, Z.15-19) wird spezifiziert, wie der Zugriff auf die Attributwerte des Objekttyps erfolgt. Dabei müssen die Werte aus Attributwerten der Entities, die in der from-Klausel (Z. 15) aufgelistet werden, gebildet werden können. In Zeile 17 und 18 werden die entsprechenden Attributkorrespondenzen definiert. Der Zuweisungsteil einer Attributkorrespondenz kann auch ein geschachtelter `IF-THEN-ELSE`-Ausdruck sein (siehe Beispiel 9).

Beispiel 9. Bedingte Abbildung von Attributen.

```
12 ENTITY_MAPPING
13   MAP  Teil
14   FROM _t := q1.TEIL, _p := q2.PERS;
15      ON_RETRIEVE
16         sachNr = IF(_t.SNR < 100)
17                      THEN 'A' + TO_VARCHAR(_t.SNR)
18                      ELSE IF(_t.SNR < 200)
19                         THEN 'B' + TO_VARCHAR(_t.SNR % 100)
20                         ELSE TO_VARCHAR(Schema._t.SNR);
       ...
21   END_MAP;
    ...
22 END_ENTITY_MAPPING;
```

Die Objektidentität. Um zu verhindern, dass das gleiche Objekt mehrfach im Zielschema repräsentiert wird, gibt es, analog zu der `PRODUCED-BY`-Anweisung, die `IDENTIFIED_BY`-Anweiung. Damit nicht für jedes Tupel aus `TEIL` eine Instanz `Funktionsgruppe` im Zielschema angelegt wird, kann die `IDENTIFIED_BY`-Anweisung wie in Beispiel 10 (Zeile 6) verwendet werden.

Beispiel 10. Objektidentität.

```
1   MAP  Funktionsgruppe
2   FROM _t := q.TEIL;
3      ON_RETRIEVE
```

```
4         bezeichnung = _t.FUNKTIONSGRUPPE;
5         teile = ...
6         IDENTIFIED_BY (_t.FUNKTIONSGRUPPE);
       ...
7   END_MAP;
```

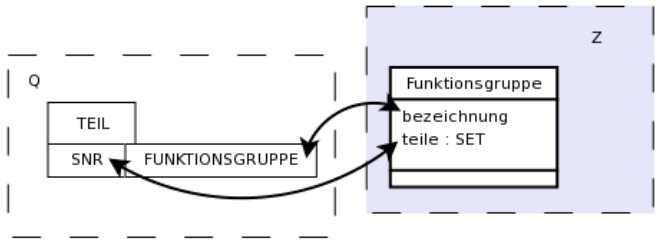


Abb. 9. Identitätsabbildung und Nest-Operation

Wird die `IDENTIFIED_BY`-Anweisung weggelassen, so werden alle Primärschlüsselattribute, bzw. OIDs der Entities, die in der `FROM`-Klausel aufgelistet werden, zur OID-Bildung herangezogen.

Wäre das Quellschema in diesem Beispiel objektorientiert, so müsste Zeile 6 entsprechend `IDENTIFIED_BY (OID(_t.FUNKTIONSGRUPPE))` lauten.

Der Nest-Operator. Um aus dem einwertigen Attribut `SNR` der Relation Teil ein mengenwertiges Attribut `teile` ableiten zu können, benötigt man einen Nest-Operator. Dieser soll die Sachnummern bestimmter Tupel aus `TEILE` zusammenfassen. Das Zusammenfassen erfolgt analog zu SQL mittels einer `GROUPED_BY`-Anweisung. In Beispiel 11 ist die Verwendung des `NEST`-Operators dargestellt. Dabei werden alle Sachnummern von Teilen zusammengefasst, die in dem Attribut `FUNKTIONSGRUPPE` übereinstimmen. Optional kann, ebenfalls analog zu SQL, eine `ORDER_BY`-Klausel verwendet werden um eine sortierte Liste zu erhalten. Mit der ebenfalls optionalen `WHERE`-Klausel können schließlich unerwünschte Tupel aus der Gruppierung herausgefiltert werden. Es ist erlaubt den `NEST`-Operator zu schachteln. Zur Abbildung von Aggregaten objektorientierter Quellschemata, auf relationale Zielschemata gibt es einen `UNNEST`-Operator.

Beispiel 11. Der Nest-Operator.

```
1   MAP  Funktionsgruppe
2   FROM _t := q.TEIL;
3      ON_RETRIEVE
4         bezeichnung = _t.FUNKTIONSGRUPPE;
5         teile = NEST( _t.SNR)
6                 WHERE ...
```

```
7                ORDER BY ...
8                GROUPED_BY _t.FUNKTIONSGRUPPE
9          IDENTIFIED_BY (_t.FUNKTIONSGRUPPE);
       ...
10  END_MAP;
```

Vernetzung von Objekttypen. In einem relationalen Schema werden Beziehungen zwischen Entities wertebasiert über Primär-Fremdschlüssel-Beziehungen modelliert. In objektorientierten Systemen wird dieser Sachverhalt durch Referenzen dargestellt. Ein referenzierendes Objekt kennt die Identitäten (OIDs) seiner referenzierten Objekte. Greift ein referenzierendes Objekt auf ein von ihm referenziertes Objekt zu, so wird anhand der OID das zugehörige Objekt bestimmt, und in den Hauptspeicher geladen. In Abbildung 10 ist ein relationales Quellschema und ein objektorientiertes Zielschema abgebildet. Im Zielschema gibt es eine Referenz von `Teil` auf `Person` über das Attribut `erstellt_von`. Die Definition einer Referenz erfolgt in BRIITY mithilfe der `CASCADED_MAP`-Anweisung.

Beispiel 12. Abbildung von Referenzen

```
1   MAP  Person
2   FROM _p := q.PERS;
3      ON_RETRIEVE
4         name = _p.NAME;
5         IDENTIFIED_BY (_p.PNR);
...
6   END_MAP;

7   MAP  Teil
8   FROM _t := q.TEIL;
9      ON_RETRIEVE
10        erstellt_von = CASCADED_MAP Person
11                       WITH_ID _t.KONSTRUKTEUR
12                       WHERE ...
13        IDENTIFIED_BY (_t.SNR);
       ...
14  END_MAP;
```

Durch die `WITH_ID`-Anweisung wird die OID des referenzierten Objektes zugewiesen. Im Beispiel speichert die Relation `TEIL` in dem Attribut `KONSTRUKTEUR` die Personalnummer des Konstrukteurs. Auf diesen Wert verweist die WITH_ID-Anweisung in Zeile 11. Weil in der Abbildungsvorschrift von Person außerdem definiert wurde, dass eine Person durch `PERS.PNR` identifiziert wird, enthält `erstellt_von` OIDs von Objekten vom Typ `Person`.

Objektidentität bei horizontaler semantischer Kongruenz. In Abbildung 11 ist ein Beispiel für die Verteilung desselben Konzeptes über mehrere

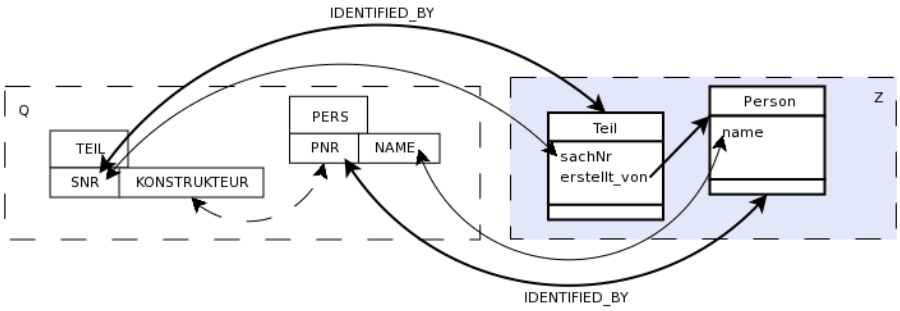


Abb. 10. Abbildung von Primär-und Fremdschlüsselbeziehungen auf Referenzen

Quellsysteme gegeben. Ein Hersteller möchte die Stammdaten von Einzelteilen, die über Zulieferer bezogen werden, in das eigene System integrieren. Zur einheitlichen Darstellung der Teile im Zielschema sind die beiden folgenden Probleme zu lösen: Erstens muss die Abbildung von `Teil` zweimal definiert werden, und dann die Vereinigungsmenge gebildet werden. Zweitens muss eine Sechskantschraube, die von mehreren Zulieferern bezogen wird, aber nur eine Sachnummer hat, unterschieden werden können. Das erste Problem wird gelöst, indem die Objektmenge des Zielobjekttyps in Partitionen zerlegt wird. Um die Teile, die in den Quellsystemen über die gleiche logische Identität verfügen, unterscheiden zu können, werden ihnen globale Identitäten zugewiesen. Im Listing von Beispiel 13 ist die Syntax beispielhaft dargestellt.

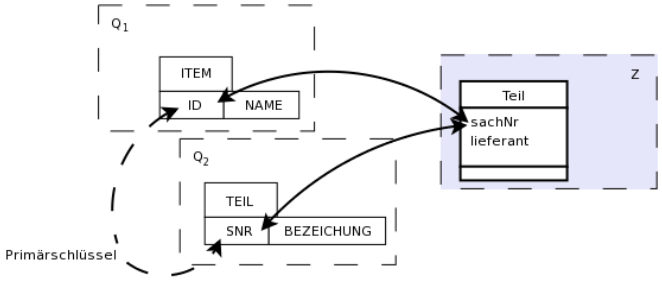


Abb. 11. Globale Objektidentität und horizontale semantische Kongruenz

Beispiel 13. Globale Objektidentität und horizontale semantische Kongruenz

```
1   MAP  Teil
2   FROM _t := q1.TEIL IN_PARTITION par_q1;
```

```
3        _i := q2.ITEM IN_PARTITION par_q2;
4      PARTITION par_q1:
5         ON_RETRIEVE
6            sachNr = _t.SNR;
7            lieferant = 'Firma x';
8            IDENTIFIED_BY (_t.SNR);
9            GLOBAL_IDENTITY(_t.SNR, 'Firma x');
           ...
10     PARTITION par_q2:
11        ON_RETRIEVE
12           sachNr = _i.ID;
13           lieferant = 'Firma y';
14           IDENTIFIED_BY(_i.ID);
15           GLOBAL_IDENTITY(_i.ID, 'Firma y');
           ...
16   END_MAP;
```

Man beachte den Unterschied zwischen `IDENTIFIED_BY` und `GLOBAL_IDENTITY`. Die `IDENTIFIED_BY`-Anweisung bezieht sich auf die Identitäten im Quellschema, und entsprechend deren Instanziierung im Zielschema. Auch das Propagieren von Änderungsoperationen ist nur möglich, falls ein Objekt des Zielschemas als ein Objekt des Quellschemas identifiziert werden kann. `GLOBAL_IDENTITY` bezieht sich auf die Objektidentitäten des föderierten Schemas. Wird jedoch die `GLOBAL_IDENTITY`-Anweisung weggelassen, dann entspricht die globale Identität des Objektes derjenigen, die durch die `IDENTIFIED_BY`-Anweisung festgelegt wird.

Kontextabhängige Abbildungen. Bei einer kontextabhängigen Abbildung wird ein Zielobjekttyp, je nach Kontext unterschiedlich auf Objekttypen der Quellschemata abgebildet. In Abbildung 12 werden die Instanzen des Objekttyps `Punkt` je nach Kontext, `ende_1` oder `ende_2`, unterschiedlich auf die Attribute in `q.LEITUNG` abgebildet. Zur Realisierung einer solchen Abbildung wird wiederum auf das Konzept der Partitionierung eines Zielobjekttyps zurückgegriffen.

Beispiel 14. Kontextabhängige Abbildung eines Zielobjekttyps

```
1   MAP  Punkt
2   FROM _l := q.LEITUNG IN_PARTITION endpunkt1;
3        _l := q.LEITUNG IN_PARTITION endpunkt2;
4      PARTITION endpunkt1:
5         ON_RETRIEVE
6            xKoord = _l.ENDE1_X;
7            yKoord = _l.ENDE1_Y;
8            IDENTIFIED_BY(_l.ID);
           ...
9     PARTITION endpunkt2:
10        ON_RETRIEVE
11           xKoord = _l.ENDE2_X;
```

```
12          yKoord = _l.ENDE2_Y;
13          IDENTIFIED_BY(_l.ID);
          ...
14  END_MAP;
...
15  MAP Kabel
16  FROM _l := q.LEITUNG;
17     ON_RETRIEVE
18        ende_1 = CASCADED_MAP Punkt
19                 PARTITION endpunkt1;
20        ende_2 = CASCADED_MAP Punkt
21                 PARTITION endpunkt2;
21        IDENTIFIED_BY(_l.ID);
23  END_MAP;
```

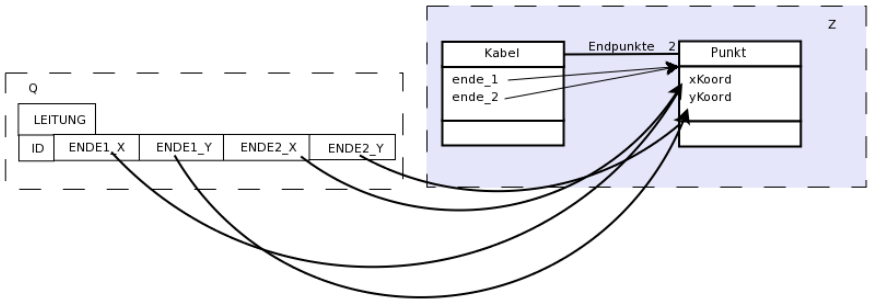


Abb. 12. Kontextabhängige Abbildung

Weil die zu einem Kabel gehörigen Punkte die gleiche OID wie das Kabel selbst haben, kann die `WITH_ID`-Anweisung wegfallen. Denn `ende_1` und `ende_2` sind dann durch die jeweilige Partition `endpunkt1` oder `endpunkt2` eindeutig identifizierbar.

Generalisierungshierarchien. In BRIITY gibt es ein Subtyping-Konzept, das es ermöglicht, Abbildungsvorschriften von einem Supertyp zu erben. Dies ist bequem, genügt jedoch nicht der Problematik, wie sie in Abschnitt 3.2 beschrieben wurde. Soll ein Entity des Quellschemas im Zielschema auf mehrere Typen einer Generalisierungshierarchie abgebildet werden, so ist dies in BRIITY nur eingeschränkt möglich. Denn die Zugehörigkeit einer Instanz des Quellschemas zu einem bestimmten Objekttyp des Zielschemas kann nur über ein Prädikat in der `WHERE`-Klausel gesteuert werden. Es werden insbesondere keine Metadaten verwaltet, die besagen, ob ein Objekt mit einer bestimmten OID zu einem bestimmten (Sub-)Typ gehört. Werden Änderungsoperationen auf diesen Objekten vorgenommen, so kann sich dadurch theoretisch auch der Typ ändern [11].

Somit steht der Typ eines Objektes erst zum Zeitpunkt des Zugriffs fest, und ergibt sich aus der Auswertung eines Prädikates. In der Objektorientierung ist aber ein Objekt normalerweise solange an einen Typ gebunden, bis dieser durch ein explizites Casting geändert wird.

Insgesamt unterstützt BRIITY das Subtyping-Konzept nur insoweit, wie sich die Typzugehörigkeit anhand eines oder mehrer Attribute eindeutig festmachen lässt.

In Abbildung 13 ist die Abbildung des Entities `EINZELTEIL` auf die Hierarchie `Verbindungselemente`, `Schraube`, `Sechskantschraube` dargestellt. Im folgenden Beispiel ist eine zugehörige Abbildungsvorschrift gegeben. Dabei wurde die starke Annahme gemacht, dass die Sachnummern schön partitioniert sind, was zur Folge hat, dass sich die entsprechenden Auswahlprädikate leicht formulieren lassen.

Beispiel 15. Bidirektionale Abbildung

```
1   MAP Verbindungselement
2   FROM _t := q.TEIL;
3      ON_RETRIEVE
4          sachNr = _t.SNR;
5          IDENTIFIED_BY (_t.SNR);
6          WHERE _t.SNR > 100
7          AND _t.SNR <= 300;
       ...
8   END_MAP;

9   MAP Schraube SUBTYPE_OF(Verbindungselement)
10  FROM _t := q.TEIL;
11     ON_RETRIEVE
12         IDENTIFIED_BY (_t.SNR);
13         WHERE _t.SNR > 100
14         AND _t.SNR <= 200;
       ...
15  END_MAP;

16  MAP Sechskantschraube SUBTYPE_OF(Schraube)
17  FROM _t := q.TEIL;
18     ON_RETRIEVE
19         IDENTIFIED_BY (_t.SNR);
20         WHERE _t.SNR > 200
21         AND _t.SNR <= 300;
       ...
22  END_MAP;
```

Bidirektionale Abbildungen. Zur Definition der Abbildungen von Z nach Q_i gibt es `ON_UPDATE`-, `ON_INSERT`- und `ON_DELETE`-Klauseln. Die `ON_INSERT`- und `ON_DELETE`-Klauseln sind anzuwenden, wenn im Zielschema Objekte eingefügt

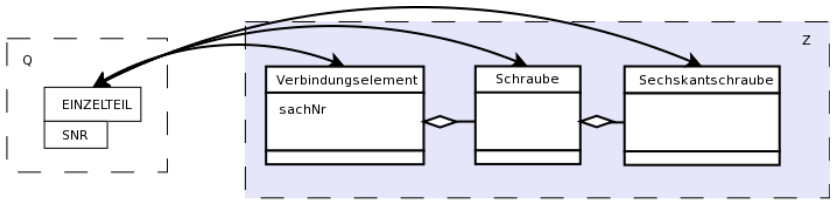


Abb. 13. Abbildung einer Generalisierungshierarchie

bzw. gelöscht werden. Die `ON_UPDATE`-Klausel wird zur Propagierung von Änderungen auf bereits existierenden Objekten des Zielschemas verwendet. Innerhalb einer Klausel kann eine Definition des Zustandes vorgenommen werden, in der eine Quelldatenbank nach erfolgter Änderungsoperation vorliegen soll.

Ruft ein globales Anwendungsprogramm eine Änderungsoperation auf, so wird der Zustand der Quelldatenbank mit der Zustandsdefiniton in der entsprechenden Klausel verglichen. Liegt der Zielzustand noch nicht vor, so werden die entsprechenden Operationen auf der Quelldatenbank ausgeführt, um den Zielzustand zu erreichen.

Die Beschreibung eines Zustandes einer Quelldatenbank wird mit der `ASSIGN`-Anweisung eingeleitet. In Beispiel 16 ist die Verwendung der `ASSIGN`-Anweisung anhand der `ON_INSERT`-Klausel dargestellt. Innerhalb einer `ASSIGN`-Anweisung kann gefordert werden, dass ein bestimmtes Objekt (`IS_INSTANCE`) vorhanden, oder nicht vorhanden (`NOT_IS_INSTANCE`) ist. Wird ein neues Objekt vom Typ `Funktionsgruppe` erzeugt (vgl. Abbildung 9), so muss für jedes Element des mengenwertigen Attributes `teile` (Zeile 6) eine korrespondierende Instanz in `q.TEIL` vorhanden sein, sonst wird das entsprechende Tupel in der Quelldatenbank erzeugt (Zeile 7–9).

Beispiel 16. Bidirektionale Abbildung

```
1   MAP Funktionsgruppe
2   FROM q.TEIL;

3      ON_RETRIEVE ...

4      ON_UPDATE ...

5      ON_INSERT
6         FOR EACH_ELEMENT_VALUE _t OF Funktionsgruppe.teile DO
7             ASSIGN IS_INSTANCE(
8                 q.TEIL : SNR = _t,
9                          FUNKTIONSGRUPPE = Funktionsgruppe.bezeichnung
10            );
11        END_FOR;
```

```
12     ON_DELETE ...

13  END_MAP;
```

Das Einfügen oder Löschen von Objekten eines Objekttyps kann auch explizit, durch die `RESTRICTED`-Anweisung, verboten werden. Zu jeder `ASSIGN`-Anweisung kann optional eine `WHERE`-Klausel definiert werden, um die Zustandssicherung auf bestimmte Objekte des Zielschemas einzuschränken. Durch Definition entsprechender `ASSIGN`-Anweisungen lässt sich referentielle Integrität sicherstellen.

5 Abbildungen zwischen relationalen Daten und XML-Daten

Relationale Daten lassen sich relativ einfach, durch eine flache Abbildung, als XML-Daten darstellen [7,9]. Die Abbildung erfolgt dann in drei Hierarchiestufen: das Wurzelelement bezeichnet das Schema, für jedes Tupel einer Relation gibt es ein untergeordnetes Element und jeder atomare Wert eines Tupels wird auf ein XML-Attribut, oder auf ein untergeordnetes Element des Tupelelementes abgebildet. In Abbildung 14 ist ein relationales Schema dargestellt. In der Relation `BAUGRUPPE_TEIL` seien die Tupel wie in Abbildung 15 dargestellt enthalten. Eine flache Abbildung dieser Tabelle ist in Beispiel 17 gegeben.

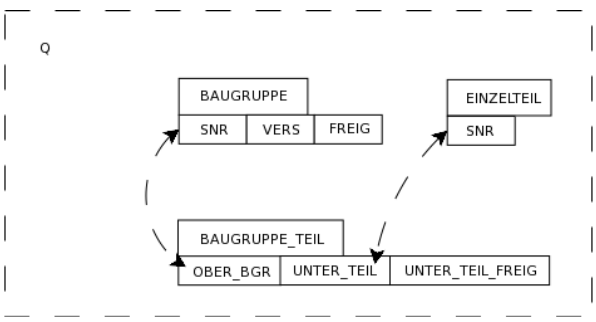
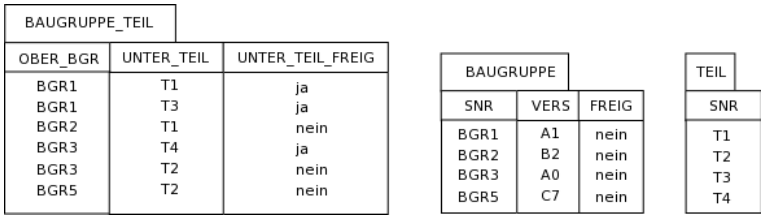


Abb. 14. Relationales Quellschema Q

Beispiel 17. Flache Abbildung einer Relation.

```
<Q>
    <BAUGRUPPE_TEIL>
        <OBER_BGR>BGR1</OBER_BGR>
        <UNTER_TEIL>T1</UNTER_TEIL>
        <UNTER_TEIL_FREIG>ja</UNTER_TEIL_FREIG>
    </BAUGRUPPE_TEIL>
```



BAUGRUPPE_TEIL

OBER_BGR	UNTER_TEIL	UNTER_TEIL_FREIG
BGR1	T1	ja
BGR1	T3	ja
BGR2	T1	nein
BGR3	T4	ja
BGR3	T2	nein
BGR5	T2	nein

BAUGRUPPE

SNR	VERS	FREIG
BGR1	A1	nein
BGR2	B2	nein
BGR3	A0	nein
BGR5	C7	nein

TEIL

SNR
T1
T2
T3
T4

Abb. 15. Die Relationen des Schemas Q

```
    <BAUGRUPPE_TEIL>
        <OBER_BGR>BGR1</OBER_BGR>
        <UNTER_TEIL>T3</UNTER_TEIL>
        <UNTER_TEIL_FREIG>ja</UNTER_TEIL_FREIG>
    </BAUGRUPPE_TEIL>
    ...
</Q>
```

Die XML-Schemabeschreibung kann mit einer Document Type Definition (DTD) oder XML Schema erfolgen. Da DTDs außer Zeichenketten keine Datentypen kennen, ist XML Schema der DTD im Allgemeinen vorzuziehen. Auch Primär- und Fremdschlüssel-Beziehungen werden von XML Schema direkt unterstützt.

Bei der flachen Abbildung sind die Primär- und Fremdschlüssel-Beziehungen nur implizit in den Element- und Attribut-Werten vorhanden. Ein Anwendungsprogrammierer, der mit einem solchen XML-Dokument arbeitet, muss, um die Daten korrekt interpretieren zu können, das relationale Schema kennen. Durch DTDs können Beziehungen über `ID` und `IDREF` nachgebildet werden. Dies hat jedoch den Nachteil, dass Attributwerte vom Typ `ID` für das gesamte XML-Dokument eindeutig sein müssen. Da Primärschlüssel nur innerhalb einer Relation eindeutig sein müssen, können Primärschlüssel nicht direkt als `ID` übernommen werden. Die `key`- und `keyref`-Elemente in XML Schema kennen diese Einschränkung nicht.

5.1 Die Ausgabe relationaler Daten als XML-Daten im SQL/XML-Standard

Die SQLX-Group [3] beschreibt im Rahmen von SQL/XML [5,4] eine standardisierte Ausgabe relationaler Daten als XML-Daten. Da SQL benutzerdefinierte Typen erlaubt, wäre die Abbildung der Spalten auf XML-Attribute nicht ausreichend. Die Tabelle `BAUGRUPPE` aus Abbildung 15 würde nach SQL/XML wie in Beispiel 18 auf XML abgebildet werden.

Beispiel 18. Standard-XML-Ausgabe einer Relation nach SQL/XML.

```
<Q>
    ...
    <BAUGRUPPE>
        <row>
            <SNR>BGR1</SNR>
            <VERS>A1</VERS>
            <FREIG>nein</FREIG>
        </row>

        <row>
            <SNR>BGR2</SNRR>
            <VERS>B2</VERS>
            <FREIG>nein</FREIG>
        </row>
        ...
    </BAUGRUPPE>
    ...
</Q>
```

Zu jedem XML-Dokument wird ein zugehöriges XML-Schema-Dokument generiert. Dazu musste SQLX die SQL-Bezeichner auf XML-Namen abbilden [3]. Dies erfolgt im Wesentlichen durch eine Umwandlung in Unicode. Dabei müssen SQL-Bezeichner, die kein gültiger XML-Name sind gesondert behandelt werden. Beispielsweise sind alle Worte, die das Präfix „xml“ haben kein XML-Name. Bei der Übersetzung wird daher das Präfix „_xFFFF_“ vorangestellt.

Mit der Zielsetzung, möglichst viele Informationen des relationalen Schemas zu erhalten, werden die SQL-Datentypen auf XML-Schema-Datentypen abgebildet. Dabei wird der XML-Schema-Datentyp so weit wie möglich eingeschränkt, um eine möglichst hohe Übereinstimmung mit dem SQL-Datentyp zu erreichen. Würden bei der Abbildung eines SQL-Datentyps trotzdem Informationen verloren gehen, so kann das `<xsd:anntotions>`-Element verwendet werden, um zusätzliche Angaben zu machen.

Das XML Schema zu Beispiel 19 ist in folgendem Listing gegeben.

Beispiel 19. Generiertes XML-Schema zu einer Standard-XML-Ausgabe.

```
<xsd:schema>

    <xsd:simpleType name="CHAR_4">
        <xsd:restriction base=xsd:string>
            <xsd:length value="4"/>
        </xsd:restriction>
    </xsd:simpleType>
```

```
    <xsd:complexType name="RowType.Catalog.Q.BAUGRUPPE">
        <xsd:sequence>
            <xsd:element name="SNR" type="CHAR_4"/>
            <xsd:element name="VERS" type="CHAR_4"/>
            <xsd:element name="FREIG" type="CHAR_4"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="TableType.Catalog.Q.BAUGRUPPE">
        <xsd:sequence>
            <xsd:element name="row" type="RowType.Catalog.Q.BAUGRUPPE"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="BAUGRUPPE" type="TableType.Catalog.Q.BAUGRUPPE"/>

</xsd:schema>
```

Die beschriebene standardisierte Abbildung von relationalen Daten entspricht einer flachen Abbildung. In Abschnitt 5.4 wird beschrieben, wie, beispielsweise im Rahmen einer Schemaintegration, XML-Sichten auf der Standard-XML-Ausgabe definiert werden können.

5.2 Erzeugen von XML-Daten mit SQL/XML

Der SQL/XML-Standard [5,4] führt außerdem einen XML-Datentyp ein. Um Instanzen dieses Typs erzeugen zu können, wird SQL um zusätzliche Funktionen erweitert. Dabei können die folgenden Funktionen auf die Tupel der Ergebnisrelation einer SQL-Anfrage angewendet werden:

- `XMLELEMENT` und `XMLATTRIBUTE`: Zum Erzeugen von XML-Elementen und XML-Attributen.
- `XMLFOREST`: Erzeugt eine Sequenz von attributlosen XML-Elementen.
- `XMLCONCAT`: Erzeugt eine Sequenz aus einer Eingabe von XML-Elementen.
- `XMLAGG`: Wie bei gewöhnlichen Aggregatfunktionen wird zunächst eine Gruppierung der Tupel der Ergebnisrelation vorgenommen. Auf jedem Tupel einer Gruppierung können, analog zur Ergebnisrelation, die XML-Funktionen angewendet werden. Das dabei entstehende Teil-Dokument ist ein Subelement desjenigen Elementes, in dem `XMLAGG` aufgerufen wurde.

Im Listing aus Beispiel 20 ist eine erweiterte SQL-Anfrage und das erzeugte XML-Dokument angegeben.

Beispiel 20. Erweiterte SQL-Anfrage nach SQL/XML.

```
SELECT XMLELEMENT (
    NAME "Baugruppe",
    XMLFOREST (
        B.SNR AS "SNr",
        B.VERS AS "Version",
        B.FREIG AS "Freigabe"
    ),
    XMLAGG(
        XMLELEMENT (
            NAME "Einzelteil",
            XMLATTRIBUTES (BT.FREIG AS "Freigabe")
            XMLELEMENT (
                NAME "SNr", T.SNR
            )
        )
    )
)
FROM BAUGRUPPE AS B, BAUGRUPPE_TEIL AS BT, TEIL AS T
WHERE B.SNR = BT.OBER_BGR
AND T.SNR = BT.UNTER_TEIL
GROUP BY B.SNR
```

Dazugehörige Ausgabe:

```
<Baugruppe>
    <SNr>BGR1</SNr>
    <Version>A1</Version>
    <Freigabe>nein</Freigabe>
    <Einzelteil Freigabe="ja">
        <SNr>T1</SNr>
    </Einzelteil>
    <Einzelteil Freigabe="ja">
        <SNr>T3</SNr>
    </Einzelteil>
</Baugruppe>

<Baugruppe>
    <SNr>BGR2</SNr>
    <Version>B2</Version>
    <Freigabe>nein</Freigabe>
    <Einzelteil Freigabe="nein">
        <SNr>T1</SNr>
    </Einzelteil>
</Baugruppe>
...
```

Die Rückgabewerte von `XMLELEMENT, XMLFOREST, XMLCONCAT` und `XMLAGG` sind Werte des XML-Datentyps.

5.3 Erzeugen und Speichern von XML-Dokumenten mit DB2 XML Extender

Der XML Extender von DB2 erlaubt es, eine Abbildung zwischen relationalen Daten und XML-Daten zu spezifizieren. Die Definition der Abbildung erfolgt in XML und wird als Document Access Defintion (DAD) bezeichnet. Zur Erstellung einer DAD können zwei Notationen (SQL Composition und RDB Node) verwendet werden. Die DAD aus Beispiel 21 wurde mit SQL-Composition-Notation erstellt.

Eine DAD gliedert sich in zwei Teile. Im ersten Teil (Zeile 6–15) werden die relevanten Daten über eine SQL-Anfrage ausgewählt. Im zweiten Teil (Zeile 18–28) wird die Struktur des XML-Dokumentes und die eigentliche Abbildung auf die relationalen Daten festgelegt.

Beispiel 21. DAD des DB2 XML Extender

```
1  <?xml version="1.0"?>
2  <!DOCTYPE DAD SYSTEM "dad.dtd">
3  <DAD>
4     <validation>no</validation>
5     <Xcollection>
6     <SQL_stmt>
7         SELECT
8          B.SNR AS snr,
9          COUNT(
10             SELECT BT.UNTER_TEIL
11             FROM BAUGRUPPE_TEIL AS BT
12             WHERE BT.OBER_BGR = B.SNR
13         ) AS anz
14        FROM BAUGRAUPPE AS B;
15    </SQL_stmt>

16    <prolog>?xml version="1.0"?</prolog>
17    <doctype>!DOCTYPE baugruppe SYSTEM "baugruppe.dtd"</doctype>
18         <root_node>
19               <element_node name="Baugruppe">
20               <attribute_node name="SNr">
21                   <column name="snr"/>
22               </attribute_node>
23               <element_node name="Anz_Einzelteile">
24                   <text_node>
25                       <column name="anz"/>
26                   <text_node>
27               </element_node>
28        </root_node>
29   <Xcollection>
30 </DAD>
```

Dazugehörige Ausgabe:

```
<Baugruppe SNr="BGR1">
    <Anz_Einzelteile>2</Anz_Einzelteile>
</Baugruppe>

<Baugruppe SNr="BGR2">
    <Anz_Einzelteile>1</Anz_Einzelteile>
</Baugruppe>
...
```

Um zu einer DAD gehörige XML-Daten zu speichern, kann die stored procedure `DB2XML.dxxShredXML` genutzt werden. Um die einzelnen Werte des XML-Dokumentes auf Relationen verteilen zu können, dient ebenfalls die Abbildungsvorschrift der korrespondierenden DAD.

Nach meinem Wissen gibt es bei DADs keine Möglichkeit zur Anwendung einer Aggregatfunktion analog zu `XMLAGG`. Das heißt, die Darstellung der 1:n-Beziehung zwischen `Baugruppe` und `Einzelteil` ist nicht durch eine Vater-Kind-Beziehung (vgl. Beispiel 20) möglich.

5.4 XML-Sichten auf relationalen Daten mit XQuery

Anhand von Beispiel 20 wurde deutlich, dass XML-Daten zwar wunderbar intuitiv und bis zu einem gewissen Grad auch selbstbeschreibend sind. Es ist jedoch auch ersichtlich, dass die Daten im XML-Dokument sehr viel mehr Redundanz aufweisen als in relationaler Form. In jeder Baugruppe, in der ein Einzelteil verbaut wird, wird es separat aufgelistet. Zum anderen ist die Auswertungsrichtung der Beziehung auf Baugruppe → Einzelteil festgelegt.

Durch Festlegen von Auswertungsrichtungen bleibt wenig Spielraum für Anfrageoptimierung. Durch die eingeführte Redundanz kommt es zu Änderungsanomalien. Deshalb ist es bei Daten, die eine hohe Änderungsrate aufweisen, vorzuziehen, diese weiterhin in relationaler Form zu speichern. Es genügt der Anwendung, die Daten als XML-Daten zur Verfügung zu stellen. Diese Vorgehensweise wird auch als „Publishing-Paradigm“ bezeichnet.

Auch unter Ausnutzung von `ID, IDREF` und `key` bzw. `keyref` kommen die Vorteile eines XML-Dokumentes nicht voll zum tragen. In XML können Beziehungen zwischen Daten auf intuitive Weise durch Vater-/Sohn-Elemente dargestellt werden. Liegt eine 1:n-Beziehung zwischen Enitity E_1 und Entitiy E_2 vor, so können die Tupel von E_2 als Subelemente der Elemente von Entity E_1 modelliert werden. Bei einer n:m-Beziehung zwischen E_1 und E_2 könnten ebenso die Tupel von E_2 als Subelemente der Elemente von E_1 dargestellt werden. Die andere Richtung der Beziehung kann über `key` in E_1 und `keyref` in E_2 modelliert werden. Die n:m-Beziehung zwischen `EINZELTEIL` und `BAUGRUPPE` ist in Beispiel 22 unter Ausnutzung von Vater-/Sohn-Beziehungen dargestellt.

Beispiel 22. Darstellung von Beziehungen durch hierarchische Strukturen.

```
<Baugruppen>
    <Baugruppe>
```

```
        <SNr>BGR1</SNr>
        <Version>A2</Version>
        <Freigabe>nein</Freigabe>
        <Einzelteil Freigabe=ja>
            <SNr>T1</SNr>
            <In_BGR>BGR1</In_BGR>
            <In_BGR>BGR2>/In_BGR>
        </Einzelteil>
        <Einzelteil Freigabe=ja>
            <SNr>T3</SNr>
            <In_BGR>BGR1</In_BGR>
        </Einzelteil>
    </Baugruppe>

    <Baugruppe>
         <SNr>BGR2</SNr>
         <Version>B1</Version>
         <Freigabe>nein</Freigabe>
         <Einzelteil Freigabe=nein>
            <SNr>T1</SNr>
            <In_BGR>BGR1</In_BGR>
            <In_BGR>BGR2>/In_BGR>
        </Einzelteil>
    </Baugruppe>
...
</Baugruppen>
```

Dieses XML-Dokument gibt die Entities `BAUGRUPPE`, `EINZELTEIL`, sowie die Beziehung zwischen ihnnen auf intuitive Weise wieder. Auch in diesem Dokument liegen jede Menge Datenelemente redundant vor, und der Zugriff auf ein Einzelteil kann nur über eine Baugruppe erfolgen. Dies stört aber nicht weiter, wenn dieses Dokument lediglich als Sicht für bestimmte Anwendungen auf das Entity Baugruppe vorgesehen ist.

Mit der Standard-XML-Ausgabe aus Abschnitt 5.1 als Eingabe kann diese Sicht leicht mit XQuery definiert werden. Die Erstellung von XML-Sichten mit XQuery auf Grundlage der Standard-XML-Ausgabe wurde ebenfalls für den SQL/XML-Standard vorgeschlagen [5]. Die XML-Sicht aus Beispiel 22 könnte dann wie folgt erstellt werden.

Beispiel 23. Definition einer XML-Sicht mit XQuery.

```
CREATE VIEW BAUGRUPPEN AS

<Baugruppen>{

FOR $B IN TABLE("BAUGRUPPE")/BAUGRUPPE/row RETURN
<Baugruppe>{

    <SNr>{data($B/SNR)}</SNr>,
```

```
    <Version>{data($B/VERS)}</Version>,
    <Freigabe>{data($B/FREIG)}</Freigabe>,

    FOR $BT IN TABLE("BAUGRUPPE_TEIL")/BAUGRUPPE_TEIL/row
        $T IN TABLE("TEIL")/TEIL/row
    WHERE $B/SNR = $BT/OBER_BGR
    AND $T/SNR = $BT/UNTER_TEIL RETURN
    <Einzelteil Freigabe="{data($BT/UNTER_TEIL_FREIG)}">
       <SNr>{data($T/SNR)}</SNr>

       FOR $BT2 IN TABLE("BAUGRUPPE_TEIL")/BAUGRUPPE_TEIL/row
       WHERE $BT2/UNTER_TEIL = $T.SNR RETURN
       <In_BGR>{data($BT2/OBER_BGR)}</In_BGR>

    <Einzelteil/>

}</Baugruppen>
```

Das Schöne bei dieser Vorgehensweise ist, dass sowohl die Erstellung als auch die Anfrageoperationen auf einer Sicht mit dem gleichen Konzept, nämlich XQuery, erfolgt. Folgende XQuery-Anfrage würde die Ausgabe wie in Listing 24 ergeben: `VIEW("BAUGRUPPEN")\Baugruppen[SNr=BGR1]`

Beispiel 24. Ergebnis einer XQuery-Anfrage

```
<Baugruppe>
    <SNr>BGR1</SNr>
    <Version>A2</Version>
    <Freigabe>nein</Freigabe>
    <Einzelteil Freigabe=ja>
        <SNr>T1</SNr>
        <In_BGR>BGR1</In_BGR>
        <In_BGR>BGR3</In_BGR>
     </Einzelteil>
     <Einzelteil Freigabe=ja>
        <SNr>T3</SNr>
        <In_BGR>BGR1</In_BGR>
     </Einzelteil>
</Baugruppe>
```

Analog zum SQL-Sichtenkonzept können XML-Sichten auf XML-Sichten erstellt werden.

6 Zusammenfassung

Unter anderem wurde die Abbildungssprache von BRIITY vorgestellt, die als sehr mächtig einzustufen ist. Positiv zu bewerten ist vor allem die Deklarativität und die Möglichkeit zur expliziten Definition einer Umkehrabbildung. Es werden sehr komplexe Abbildungsvorgänge unterstützt, vor allem wenn man an

kontextabhänigige Abbildungen denkt. Die Spezifikation einer Abbildung ist jedoch immer nur insoweit möglich, wie sie durch ein Prädikat formulierbar ist [11]. Dies wurde in Abschnitt 4.2 bei der Untersuchung der Abbildbarkeit von Generalisierungshierarchien besonders deutlich.

Trotzdem liegt die Ursache dieses Problems nicht unbedingt daran, dass kein geeignetes Prädikat vorhanden ist. Wie soll eine Abbildung auch anders definiert werden, als unter Zuhilfenahme von Prädikaten? Das eigentliche Problem liegt in der Autonomität der Quellsysteme und in den Eigenschaften relationaler Daten.

Aufgrund der Autonomität können keine zusätzlichen Metadaten, wie beispielsweise Typangaben in den Quellsystemen gespeichert werden. Deshalb kann auch die Korrespondenz zwischen Objekten des Zielschemas und den Tupeln der Quellschemata nur anhand von Prädikaten über Attributwerten definiert werden. Wegen der Mengenwertigkeit von Relationen ist auch folgendes nicht möglich: „Tupel eins korrespondiert mit dem Objekt der OID 4711“.

Desweiteren läßt sich theoretisch auch nicht sicherstellen, dass ein Objekt der Miniwelt während seiner gesamten Lebensdauer immer die gleiche OID besitzt. Dieses Problem ist in Pegasus genauso wie in BRIITY nicht gelöst, und kann wahrscheinlich auch nicht gelöst werden.

Insgesamt ist es mithilfe von Pegasus möglich, virtuelle integrierte objektorientierte Schemata zu erstellen, deren Daten aus relationalen und objektorientierten Quellsystemen stammen. Jedoch ist die Flexibilität bei der Erstellung der Objekttypen des Zielschemas eingeschränkt. Die Integration der Konzepte der Quellsysteme erfolgt hauptsächlich durch die Bildung von Supertypen. Sollen Objekttypen des Zielschemas durch Verbundoperationen aus mehreren Objekttypen des Quellschemas abgeleitet werden, so muss dies über die Definition benutzerdefinierter Funktionen erfolgen. Konflikte werden also hauptsächlich innerhalb von Funktionen aufgelöst. Aus der Literatur ist nicht eindeutig hervorgegangen, ob Referenzen im objektorientierten Sinn definiert werden können. Auch kontextabhängige Abbildungen werden a priori nicht unterstützt.

In Kapitel 5 habe ich mich auf die reine Abbildung zwischen XML-Daten und relationalen Daten konzentriert. Systeme oder Verfahren zur Integration von mehreren Quellschemata wurden nur insoweit untersucht, als das relationale Daten in XML-Daten transformiert werden können.Gegebenenfalls kann diese Transformation auch gemäß eines globalen Schemas erfolgen. Sind die Daten eines Konzeptes jedoch über mehrere Quellsysteme verteilt, so muss diese Integration noch in der Anwendung erfolgen. Das gleiche gilt für kontextabhängige Abbildungen.

In Abschnitt 3 wurde deutlich, dass viele Probleme des Abbildungsvorgangs nicht unbedingt an bestimmte Datenmodelle gebunden sind, sondern bereits auftreten, wenn man sich innerhalb eines Datenmodells bewegt.

Literatur

1. Rafi Ahmed, Philippe De Smedt, Weimin Du, William Kent, Mohammad A. Ketabchi, Witold Litwin, Abbas Rafii, Ming-Chien Shan: The Pegasus Heterogeneous Multidatabase System. IEEE Computer 24(12): 19-27 (1991)
2. Stefan Conrad: Schemaintegration Integrationskonflikte, Lösungsansaätze, aktuelle Herausforderungen. Inform., Forsch. Entwickl. 17(3): 101-111 (2002)
3. Andrew Eisenberg, Jim Melton: SQL/XML and the SQLX Informal Group of Companies. SIGMOD Record 30(3): 105-108 (2001)
4. Andrew Eisenberg, Jim Melton: SQL/XML is Making Good Progress. SIGMOD Record 31(2): 101-108 (2002)
5. John E. Funderburk, Susan Malaika, Berthold Reinwald: XML programming with SQL/XML and XQuery. IBM Systems Journal 41(4): 642-665 (2002)
6. Theo Härder, Günter Sauter, Joachim Thomas: The Intrinsic Problems of Structural Heterogeneity and an Approach to Their Solution. VLDB J. 8(1): 25-43 (1999)
7. Klettke Meike, Meyer Holger: XML & Datenbanken, Konzepte, Sprachen und Systeme. dpunkt.verlag Heidelberg (2003)
8. Evaggelia Pitoura, Omran A. Bukhres, Ahmed K. Elmagarmid: Object Orientation in Multidatabase Systems. ACM Comput. Surv. 27(2): 141-195 (1995)
9. Dongwon Lee, Murali Mani, Frank Chiu, Wesley W. Chu: NeT & CoT: translating relational schemas to XML schemas using semantic constraints. CIKM: 282-291 (2002)
10. Mary Tork Roth, Peter M. Schwarz: Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. VLDB: 266-275 (1997)
11. Günter Sauter: Interoperabilität von Datenbanksystemen bei struktureller Heterogenität: Architektur, Beschreibungs-und Ausführungsmodell zur Unterstützung der Integration und Migration. Infix-Verlag Sankt Augustin (1998)
12. Ming-Chien Shan, Rafi Ahmed, Jim Davis, Weimin Du, William Kent: Pegasus: A Heterogeneous Information Management System. Modern Database Systems, ACM Press: 664-682 (1995)
13. Amit P. Sheth: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. VLDB: 489- (1991)