Zum Einsatzpotential von ORDBMS in Entwurfsumgebungen

Wolfgang Mahnke [*], Hans-Peter Steiert [*]

[*] Universität Kaiserslautern, FB Informatik, Postfach 3049, 67653 Kaiserslautern
{mahnke, steiert}@informatik.uni-kl.de

Zusammenfassung:

Die Hersteller relationaler Datenbankverwaltungssysteme (RDBMS) haben ihre Produkte in der neuesten Generation um typisch objektorientierte Konzepte erweitert. Diese Systeme werden daher als Objekt-Relationale DBMS (ORDBMS) bezeichnet. Mit der neuen Funktionalität sollen neue Anwendungsbereiche, beispielsweise technische Entwurfsanwendungen, besser unterstützt werden. Wir untersuchen ein solches System auf seine Eignung zur Verwaltung von Entwurfsdaten eines großen Software-Entwicklungsprojekts. Die daraus resultierenden Anforderungen sind typisch für Entwurfsanwendungen. In unserem Aufsatz präsentieren wir unsere Erfahrungen mit der Abbildung von objektorientiert modellierten Produktdaten auf das erweiterte Datenmodell des ORDBMS. Als Beispiel dient uns hier das Metamodell der UML (Unified Modeling Language). UML ist eine objektorientierte Modellierungssprache, die sich bei der Daten- und Systemmodellierung immer mehr durchsetzt. Der zweite Aspekt unseres Aufsatzes beschäftigt sich mit der Anbindung externer Daten. In Entwurfsumgebungen fallen oft Daten an, die bisher nicht von RDBMS verwaltet werden konnten, da die Werkzeuge einen Zugriff auf das Dateisystem verlangen. Wir stellen dar, wie diese externen Daten unter die Kontrolle des ORDBMS gebracht und über die SQL-Schnittstelle zugänglich gemacht werden können.

1 Einleitung

Hersteller Objekt-Relationaler Datenbankverwaltungssysteme (object relational database management systems, ORDBMS) versprechen, objektorientierte Konzepte in die bewährten relationalen Datenbanksysteme zu integrieren. Stonebraker [12] spricht gar von der „*next greate wave*“, die über uns hereinbrechen wird. Wir werden in diesem Artikel untersuchen, inwieweit es sich dabei um eine positive Entwicklung oder um eine unheilverkündende Prophezeiung handelt.

In diesem Aufsatz gehen wir auf zwei Aspekte näher ein, die im Umfeld von ORDBMS immer wieder als Hauptargumente aufgeführt werden: Datenmodellierung und Integration externer Datenquellen.

Gegenüber RDBMS bieten ORDBMS vor allen Dingen ein erweitertes und erweiterbares Typsystem. Dieses gestattet es dem Benutzer, auch komplexer strukturierte

Typen zu definieren. Eine andere Erweiterung stellen die benutzerdefinierten Funktionen dar. Sie erlauben es, eigene Operationen für benutzerdefinierte Typen bereitzustellen und in den Datenbankoperationen zu benutzen. Weiterhin können diese Funktionen eingesetzt werden, um Anwendungslogik auf den Datenbankserver zu verlagern. Ein kurzer Überblick über die neue Funktionalität der ORDBMS zur Modellierung von Daten und Anwendungen wird in Kapitel 2 gegeben.

Als Anwendungsbeispiel dient uns die Verwaltung von UML-Modellen. Die Modellierungssprache UML (Unified Modeling Language, [9,10]) wird immer häufiger zur graphischen Modellierung von objektorientierten Systemen verwendet. Wir entwikkeln ein UML-Repository, in dem UML-Modelle verwaltet werden können. Dieses dient zur Unterstützung des CASE-Prozesses, sowohl während der Entwicklung der Systeme als auch als Basis für eine mögliche Wiederverwendung. Die aus diesem Anwendungsbereich resultierenden Anforderungen sollten sich auf andere Bereiche aus dem CAx übertragen lassen (Kapitel 3). Inwieweit sich die neue Funktionalität der ORDBMS bei der Abbildung eines komplexen Produktdatenmodells auf ein Datenbankschema eines kommerziellen Produkts ausnutzen läßt, ist in Kapitel 4 beschrieben.

ORDBMS bieten die Möglichkeit, DBMS-spezifische Funktionalität an eigene Bedürfnisse anzupassen bzw. passend zu erweitern. Dies kann beispielsweise genutzt werden, um einen Zugriff auf externe Daten mit der Anfragesprache SQL zu ermöglichen. Wir werden dies in Kapitel 5 diskutieren und an einem Beispiel vertiefen.

Eine genaue Abgrenzung zu OODBMS unterbleibt aus Platzgründen. Sie ist insofern problematisch, als daß die existierenden OODBMS eine sehr heterogene Funktionalität anbieten. Daher ist vor dem Hintergrund eines praktischen Einsatzes immer nur ein Vergleich mit einem konkreten System oder mit der von allen Systemen gemeinsam angebotenen Funktionalität sinnvoll. Insbesondere sind die vom ODMG-Standard [1] vorgegebenen Konzepte sehr unterschiedlich weit realisiert, so daß auch dieser nur als grobe Richtschnur dienen kann. Bei den ORDBMS entwickelt sich der SQL-Standard [6] dagegen augenblicklich fast langsamer als die Realisierungen der Herstellern. Er kann damit als eine gemeinsame Basis betrachtet werden, die von den großen Herstellern zumindest in naher Zukunft angeboten werden wird. Daher steht das Einsatzpotential von ORDBMS im Mittelpunkt unserer Betrachtungen. Ein Vergleich mit RDBMS wird dort herangezogen, wo neue Funktionalität als solche kenntlich gemacht werden soll.

2 Objekt-Relationale Datenbankverwaltungssysteme

Stonebraker [12] sieht in ORDBMS die DBMS-Technologie der nächsten Jahre. Diese Aussage begründet sich zum einen aus dem Trend, moderne Anwendungen mit objektorientierten Methoden zu entwickeln. Der Bruch zwischen Datenmodell und Programmierstil eines RDBMS und der meist objektorientierten Middleware bzw. den objektorientierten Programmiersprachen (OOPL) wird dabei zunehmend als Hindernis empfunden. Zum anderen fordern neue Anwendungsgebiete (WWW, Internet) in verstärktem Maße, neue Datentypen (Video, Bilder, Audio, (Hyper-)Text) mit dem

DBMS zu verwalten. Hierzu sind Erweiterungen herkömmlicher RDBMS notwendig. Stonebraker [12] leitet daraus folgende Forderungen ab:

Erweiterung um neue Basisdatentypen: Die Menge der von kommerziellen RDBMS bereitgestellten Basisdatentypen ist eingeschränkt. Insbesondere bei Ingenieuranwendungen sind jedoch vielfältigere Basisdatentypen wünschenswert, beispielsweise Datentypen wie „Punkt“ und „Fläche“ mit den zur Verarbeitung notwendigen Operationen (*distance*, *includes*). ORDBMS lassen sich um solche Datentypen erweitern, die dann wie gewöhnliche Basisdatentypen zur Definition von Attributen in Tabellen Verwendung finden können. Dazu müssen neben anwendungsspezifischen Operationen, wie *distance*, auch einige Funktionen implementiert werden, die dem DBMS die Verwaltung erlauben. Weiterhin lassen sich neue Indexstrukturen für die neuen Datentypen implementieren, beispielsweise R-Bäume [3], mit denen ein effizienterer Zugriff auf räumliche Objekte möglich wird.

Komplexe Typen: Komplexe Typen sind aus (mehreren) benutzerdefinierten oder elementaren Typen mit Hilfe von Typkonstruktoren (Feld, Liste, Menge) zusammengesetzt. Sie lassen sich zur Definition von Tabellen verwenden, die als Container zur Aufnahme von Tupeln mit dem zuvor definierten Typ dienen. Ebenso können einzelne Attribute in Tabellen als komplexe Typen definiert werden.

Eine andersartige Erweiterung sind Referenzen. Diese lassen sich in komplexen Typen dazu verwenden, Verweise auf Tupel in (anderen) Tabellen aufzunehmen. In Anfragen werden Referenzen ähnlich der Syntax objektorientierter Programmiersprachen verfolgt. Damit können Beziehungen in Anfragen berücksichtigt werden, ohne Joins formulieren zu müssen.

Ein ORDBMS erlaubt es weiterhin, benutzerdefinierte Funktionen zu registrieren, die komplexe Typen als Parameter akzeptieren. Diese lassen sich in Anfragen benutzen, um aufwendigere Auswertungen oder Manipulationen durchzuführen.

Vererbung: Eines der wichtigsten objektorientierten Konzepte ist die Vererbung. Sie erlaubt eine bessere Strukturierung des Entwurfs und unterstützt Wiederverwendung. ORDBMS bieten Vererbungshierarchien für Typen und Tabellen. Anfragen auf Supertabellen liefern dabei auch alle Tupel, die in der Subtabelle gespeichert sind. Es wird Polymorphie unterstützt, d. h., benutzerdefinierte Funktionen werden in Anfragen spät gebunden, die Implementierung also abhängig vom Typ des konkreten Tupels ausgewählt. Dazu erlauben es die Systeme, an einen Funktionsnamen mehrere Implementierungen zu binden und die konkrete Implementierung anhand der Parameter auszuwählen. Nach [12] sollte ein ORDBMS auch Mehrfachvererbung unterstützen.

Regeln: Regelsysteme können vielfältig genutzt werden, beispielsweise zur Integritätssicherung, zum Protokollieren von Aktionen oder zum Anstoßen von Anwendungslogik. Solche Systeme sind zum Teil bereits in RDBMS integriert. Für ORDBMS werden allerdings Erweiterungen gefordert. So sollen beispielsweise Aktionen nicht nur durch andere Aktionen angestoßen werden können, sondern diese auch vollständig ersetzen können. Weiterhin ist ein Benachrichtigungsmechanismus wünschenswert, der Anwendungen über das Eintreten eines Ereignisses asynchron informiert.

Diese Liste neuer Anforderungen erweitert sich zusätzlich um folgende Punkte [7]:

Zugriff auf externe Daten: Trotz des Erfolges von RDBMS wird noch immer der überwiegende Teil der Daten nicht in solchen Systemen gespeichert. Dafür gibt es im wesentlichen drei Gründe. Erstens ist die Migration von Altanwendungen hin zu DBMS oft zu kostspielig. Zweitens werden insbesondere im Umfeld des Internet viele „kleine" Anwendungen genutzt, beispielsweise Email-Programme. Einzeln betrachtet steht bei diesen Anwendungen der Aufwand bei der Verwendung eines DBMS in keinem Verhältnis zum Nutzwert. Erst durch die Vielzahl solcher Anwendungen und die Summe der so anfallenden Daten wird es interessant, sie mit Hilfe eines DBMS zu verwalten. Drittens ist die reale Anwendungswelt nicht homogen, häufig werden Werkzeuge und Anwendungen von verschiedenen Herstellern bezogen, bei denen jeder seine eigene Strategie zur Verwaltung der Daten verfolgt.

ORDBMS lassen sich in einem solchen heterogenen Umfeld als Integrationsplattform heranziehen. Zu den Anwendungen hin bieten sie einen homogenen Zugriff über die bekannte und standardisierte SQL-Schnittstelle. Um auf externe Daten zugreifen zu können, wird eine Schnittstelle angeboten, die es erlaubt, Wrapper zu implementieren. Das ORDBMS greift bei der Verarbeitung von Anfragen auf die Wrapper in gleicher Weise zu wie auf interne Komponenten. Die Aufgabe der Wrapper besteht darin, Daten externer Quellen so aufzubereiten, daß dem ORDBMS eine Sicht angeboten wird, die dem Zugriff auf intern verwaltete Daten entspricht.

Plug-Ins: Eigentlich ist unter dem Begriff „Plug-In", oder in Produkten gesprochen ‚DataBlades' (Informix), ‚Cartridges' (Oracle) und ‚Extenders' (IBM), kein eigenständiges Konzept zu verstehen. Ein „Plug-In" ist vielmehr eine Sammlung von benutzerdefinierter Datentypen, Indexstrukturen etc. Als Paket bildet diese eine DBMS-gestützte Komponente, die eine anwendungsspezifische Erweiterung der Funktionalität bereitstellt.

Die Entwicklung der einzelnen Produkte verläuft mit sehr unterschiedlicher Schwerpunktbildung. Teilweise sind die oben aufgeführten Punkte auch schon in den Standard SQL:1999 eingeflossen.

3 Anwendungsbeispiel: UML-Repository

In [12] werden auch Entwurfsanwendungen als solche Anwendungen genannt, die sich mit ORDBMS besonders gut unterstützen lassen. Unser Forschungsprojekt SENSOR[1] beschäftigt sich nun im Rahmen des SFB 501 mit der Frage, inwieweit diese Aussage zutrifft. Eine Aufgabe unserer Arbeitsgruppe ist es, ein UML-Repository zu entwickeln, in dem sich UML-Modelle ablegen und verwalten lassen. Dieses Repository soll als Grundlage für eine Kooperation der Entwickler und als Basis für die Wiederverwendung von Entwürfen dienen. Wir wollen es hier als Beispielanwendung für obige Frage benutzen.

[1] Teilprojekt A3 Supporting Software Enngineering Processes by Object-Relational Database Technology des Sonderforschungsbereichs 501 Entwicklung großer Systeme mit generischen Methoden, mit Unterstützung der Deutschen Forschungs Gemeinschaft.

Die Semantik von UML-Modellen ist in [10] mit verschiedenen Techniken spezifiziert. Zunächst werden graphische UML-Modelle eingesetzt, um das UML-Metamodell zu beschreiben. Da die graphische Modellierung nicht ausreicht, ergänzen Invarianten die Spezifikation. Diese sind in OCL formuliert. OCL ist eine deskriptive, objektorientierte Sprache, mit der sich Integritätsbedingungen ausdrücken lassen [8]. Beides wird noch mit einer textuellen Beschreibung vervollständigt.

Das in [9] beschriebene UML-Modell des UML-Metamodells dient uns als Ausgangspunkt. Es enthält eine Vielzahl typischer objektorientierter Modellierungskonstrukte. So wird bei der Modellierung intensiv Vererbung benutzt. Neben einfacher Vererbung kommt dabei auch Mehrfachvererbung vor. Weiterhin sind die Beziehungen, im UML-Sprachgebrauch Assoziationen genannt, fein spezifiziert, beispielsweise durch Angabe von Kardinalitäten. Weiterhin treten geordnete Assoziationen und solche mit Aggregationssemantik auf. Das Modell des UML-Metamodells stellt damit ein Beispiel für einen typischen objektorientierten Entwurf eines Produktdatenmodells dar, wie er in technischen Entwurfsanwendungen häufig anzutreffen ist.

Wir haben nun dieses UML-Modell als Datenbankschema eines ORDBMS implementiert. Dabei wollten wir erreichen, daß der Programmierer an der SQL-Schnittstelle die im UML-Modell modellierte Sicht angeboten bekommt. Um die Integrität der Daten im Repository zu sichern, wurden die OCL-Constraints auf SQL-Integritätsbedingungen abgebildet. Diese Abbildungsvorgänge sollen in Abschnitt 4.3 genauer erläutert werden.

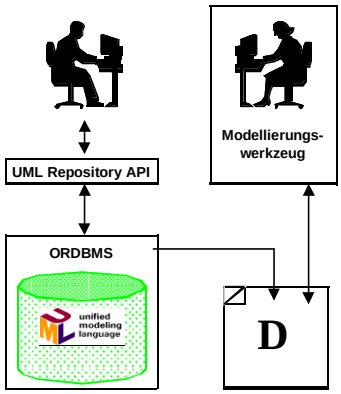


Abbildung 1: Überblick

Da wir für die graphische Modellierung am Markt erhältliche Werkzeuge einsetzen wollen, mußten wir eine Umgebung schaffen, in der sich diese Daten in das Repository integrieren lassen. Unser Hauptziel war dabei nicht, alle Daten in das ORDBMS zu kopieren, sondern sie weiterhin unter der Kontrolle des Werkzeuges zu belassen. Zugleich soll aber mit eigenen Anwendungen über die SQL-Schnittstelle des ORDBMS auf diese Daten in gleicher Weise zugegriffen werden, als wären sie im DBS gespeichert. Eine Kooperation der Werkzeuge ist nur in seltenen Fällen zu erwarten. Wir wollen diesen Zugriff beispielsweise dazu verwenden, um vorgegebene Entwurfsrichtlinien mit OCL zu formulieren und mit Hilfe des Datenbanksystems auf den Daten externer Werkzeuge zu überprüfen. Einen Überblick gibt Abbildung 1.

Aus unserer Sicht stellt das hier beschriebene UML-Repository eine typische Entwurfsanwendung dar: Es müssen komplexe Strukturen verwaltet werden, bei deren Modellierung objektorientierte Konzepte eingesetzt wurden. Diese Daten sollen zusammen mit den Daten von externen Werkzeugen verwaltet werden. Wir haben es also mit einer heterogenen Umgebung zu tun, was ebenfalls typisch ist. Weiterhin ist eine Unterstützung für die Kontrolle von Datenintegrität und Entwurfsrichtlinien vorgesehen.

Die weiteren Abschnitte werden unsere Erfahrungen bei der Entwicklung des Repository auf Basis eines kommerziellen ORDBMS präsentieren. Da die Anforderungen anderer Anwendungen aus dem CAx-Bereich ähnlich ausfallen, halten wir diese für übertragbar.

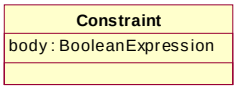
4 Modellierung

Im folgenden wird die Modellierungsmächtigkeit von ORDBMS am Beispiel des UML-Metamodells untersucht. Dieses wird auf das Schema eines ORDBMS abgebildet. Dabei helfen objekt-relationale Erweiterungen, die beispielsweise Vererbungshierarchien auf Tabellenebene und benutzerdefinierte Routinen (UDR) erlauben. Bei der Abbildung werden wir im besonderen auf die Mehrfachvererbung eingehen, die von Seite der ORDBMS nicht unterstützt wird. Außerdem werden wir die Integritätssicherung in Entwurfsdaten ansprechen. Im UML-Repository müssen dazu die OCL-Constraints [8] in SQL-Statements übersetzt werden, um sie durch das ORDBMS zu kontrollieren.

4.1 Abbildung des UML-Metamodels auf ein ORDB-Schema

Zur Beschreibung des UML-Metamodells als UML-Modell [9] werden im wesentlichen Klassen, Vererbungsbeziehungen sowie allgemeine Beziehungen verwendet. Sonderformen wie bspw. die AssociationClass werden dabei auf eine Klasse mit Beziehungen zu den verbundenen Klassen abgebildet.

Unser Ansatz besteht darin, Klassen auf getypte Tabellen abzubilden. Die Attribute der Klassen werden auf Attribute des Typs der Tabelle abgebildet, wobei ORDBMS die Möglichkeit vorsehen, eigene Datentypen zu definieren, so daß beispielsweise auch mengenwertige Attribute gespeichert werden können. Die Methoden der Klasse können als benutzerdefinierte Funktionen des ORDBMS definiert werden. ORDBMS bieten die Möglichkeit, auf getypten Tabellen Vererbungsbeziehungen zu definieren, so daß die Vererbungsbeziehungen des UML-Metamodells i. d. R. direkt auf diese abgebildet werden können. Einzige Ausnahme bildet die Mehrfachvererbung (siehe Abschnitt 4.2). Eine Abbildung der Klasse „Constraint“ des UML-Metamodells auf Typ und Tabelle ist in Abbildung 2 dargestellt. Links davon ist der entsprechende Ausschnitt des UML-Metamodells aufgezeigt.




```
CREATE ROW TYPE Constraint_ty
(body    BooleanExpression)
UNDER ModelElement_ty;

CREATE TABLE Constraint_ta
OF TYPE Constraint_ty
(PRIMARY KEY (ID))
```

Abbildung 2: Die UML-Metaklasse „Constraint"

Beziehungen zwischen Klassen können nicht durch Referenzen realisiert werden, da das von uns verwendete ORDBMS diese nicht unterstützt, obwohl sie Teil des Standards SQL:1999 sind. Ansätze, für mehrwertige Beziehungen statt Referenzen Primärschlüssel in mengenwertigen Attributen zu speichern, haben sich als sehr schwierig erwiesen. Für die Einträge solcher Attribute können in dem von uns verwendeten ORDBMS keine Fremdschlüssel definiert werden, so daß diese über Trigger auf beteiligten Tabellen abgebildet werden müssen. Da uns dieser Verwaltungsaufwand zu hoch erscheint, verwenden wir in diesem Fall die klassische Abbildung, wie bei relationalen DB-Schemata [2].

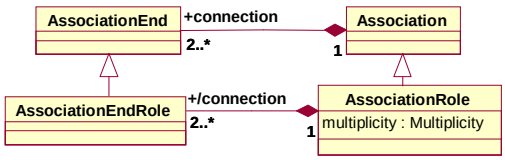


Abbildung 3: Ausschnitt des UML-Metamodells

Um eine korrekte Sicht auf die einzelnen Klassen zu bekommen, ist es teilweise erforderlich, geerbten Attributnamen einen anderen Namen zu geben. Wie in Abbildung 3 zu sehen ist, müßte das Attribut „Association" der Tabelle „AssociationEnd_ta" für die Subtabelle „AssociationEndRole_ta" „AssociationRole" heißen. Aus diesem Grund und zur Realisierung der Mehrfachvererbung (siehe Abschnitt 4.2) definieren wir für jede Klasse Sichten, über die auf die Tabellen zugegriffen wird. Es gibt verschieden Sichten für SELECT-, INSERT-, UPDATE- und DELETE-Anweisungen. Dies ist erforderlich, da eine Sicht alleine nicht alle Anforderungen erfüllen kann. Damit beim Aufruf von Funktionen an der SQL-Schnittstelle polymorphes Verhalten gewährleistet ist, muß für SELECT-, UPDATE- und DELETE-Statements neben den Attributen auch noch der gesamte ROW TYPE zurückgeben werden[2]. Auf dieser Sicht ist allerdings keine INSERT-Anweisung möglich. Entsprechend wird für jede Anweisungsart eine eigene Sicht definiert. Damit das „ONLY"-Statement verwendet werden kann, müssen weitere Sichten definiert werden, da ein „`SELECT * FROM ONLY(`*`view`*`)`" alle Tupel der Sicht liefert, unabhängig von der

[2] Die Verwendung von getypten Sichten ist zwar möglich, bei Anfragen auf einer Tabellenhierarchie wird jedoch immer der gleiche Typ (der, der Sicht) zurückgegeben, so daß keine Polymorphie ausgenutzt werden kann.

Tabellenhierarchie. Die Definitionen der Sichten der Klasse „Constraint“ wird beispielhaft an der SELECT-Sicht gezeigt:

```
CREATE VIEW Constraint_vi_s AS
SELECT *, Constraint_ta AS as_row
FROM Constraint_ta;
```

Das vorgestellte Konzept der Abbildung von Klassen auf Typen, Tabellen und Sichten (Views) wird im folgenden TTV^n genannt.

Da das von uns eingesetzte ORDBMS keinen Objektidentifikator (OID) anbietet, verwenden wir als Primärschlüssel einen eigenen Basisdatentyp, der Informationen liefert, aus welcher Tabelle und von welchem Typ das Tupel ist. Dieser OPAQUE DATA TYPE ist intern dreigeteilt und enthält die Tabellen-ID, die Typ-ID sowie eine Tupel-ID, die das Tupel in der Tabelle eindeutig bestimmt. Diese OID wird beim Einfügen eines Tupels automatisch generiert. Dazu benötigte Informationen werden aus dem Metadatenkatalog der Datenbank gewonnen. Daher kann aus der OID eindeutig das Tupel einer getypten Tabelle bestimmt werden.

4.2 Mehrfachvererbung

Obwohl Stonebraker in [12] fordert, daß ein ORBDMS Mehrfachvererbung beherrschen sollte, unterstützt das von uns verwendete ORDBMS in seiner aktuellen Version keine Mehrfachvererbung bei Typ- bzw. Tabellenhierarchien. Auch der Standard SQL:1999 [6] sieht keine Mehrfachvererbung vor. Nun ist die Verwendung von Mehrfachvererbung durchaus umstritten, kommt allerdings in praktischen Beispielen vor. So enthält unser Anwendungsbeispiel Klassen, die, durchaus sinnvoll von zwei anderen abgeleitet sind. Außerdem ist sie als Modellierungsmittel in wichtigen Standards enthalten, beispielsweise dem im CAD-Bereich verbreiteten STEP-Standard [5]. Die Abbildungsproblematik läßt sich daher nicht einfach ignorieren.

Demnach ergibt sich bei der in Abschnitt 4.1 vorgestellten Abbildung des UML-Metamodells auf ein ORDB-Schema in Problem und es ist eine Lösung nötig, bei der eine Mehrfachvererbung in der Tabellenhierarchie simuliert wird. Im folgenden werden die Anforderungen an eine solche Lösung genannt sowie die Einschränkungen aufgezählt, die sinnvollerweise gemacht werden müssen, um eine entsprechende Semantik zu erhalten. Anschließend wird auf die Lösungsmöglichkeiten eingegangen. Zur einfacheren Verständlichkeit verwenden wir ein einfaches, abstraktes Beispiel, das die Problematik verdeutlicht (siehe Abbildung 4).

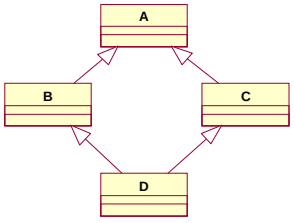


Abbildung 4: Beispiel einer Tabellenhierarchie mit Mehrfachvererbung

4.2.1 Anforderungen an Lösungen zur Simulation von Mehrfachvererbungen

Zunächst wird die Semantik einer Hierarchie mit Mehrfachvererbung festgelegt. Da das von uns verwendete ORDBMS Tabellenhierarchien mit einfacher Vererbung unterstützt, orientieren sich die im folgenden vorgestellten Anforderungen an eine simulierte Mehrfachvererbung an der von diesem System vorgegebenen Semantik.

Auswahl von Tupeln: Bei jedem SELECT-Statement müssen sowohl die qualifizierten Tupel der Tabelle als auch die der Subtabellen jeweils genau einmal ausgewählt werden. Nur wenn sich die Anweisung explizit nur auf eine Tabelle bezieht („`SELECT * FROM ONLY (table)`“) werden die Daten in den Subtabellen nicht betrachtet. Ein „`SELECT table FROM table`“ liefert nicht die einzelnen Attribute von *table*, sondern alle Daten eines Tupels von *table* als Typ. Die Tupel aus Subtabellen tragen den Typ der jeweiligen Tabelle und beinhalten auch die Daten der Attribute, die nicht im Typ von *table* definiert wurden (Polymorphie).

Einfügen von Tupeln: Ein INSERT eines Tupels in eine Subtabelle führt immer dazu, daß das eingefügte Tupel anschließend in allen Supertabellen sichtbar ist.

Ändern von Tupeln: UPDATE-Statements auf einer Supertabelle wirken sich auch auf die Tupel der Subtabellen aus. Dabei wird eine Änderung für jedes qualifizierte Tupel genau einmal durchgeführt. Dies gilt insbesondere für „relative“ UPDATE-Statements (z. B. `Gehalt:=Gehalt+10`).

Löschen von Tupeln: Ähnlich dem Ändern der Tupel muß beim Löschen darauf geachtet werden, daß auch qualifizierte Tupel der Subtabellen gelöscht werden.

Polymorphie: Eine auf einem Supertyp definierte UDR ist auch mit einem Tupel des Subtyps aufrufbar. Wurde die UDR für den Subtyp überschrieben, wird die entsprechende Implementierung des Subtyps ausgeführt.

Casts: Cast-Operatoren auf den Typen der Supertabellen müssen auch für die Typen der Subtabellen vorhanden sein.

Integritätsbedingungen: Integritätsbedingungen, die auf den Supertabellen definiert wurden, gelten auch für die entsprechenden Subtabellen. Integritätsbedingungen der Supertabellen können auf den Subtabellen nicht eingeschränkt, sondern lediglich erweitert werden. Eine Fremdschlüsselbeziehung zu einer Supertabelle muß auch Tupel der Subtabellen referenzieren dürfen. Ein auf der Tabelle A definierter Check-Constraint (z. B. `Gehalt > 10.000`) gilt auch für Tupel aus den Tabellen B, C und D. Er kann auf den Subtabellen weiter eingeschränkt werden, z. B. `Gehalt > 15.000`. In den Subtabellen kann ein neuer Primärschlüssel definiert werden. Der Primärschlüssel der Supertabelle wird als Attribut mit der Eigenschaft „`UNIQUE NOT NULL`“ übernommen. Es wird ausschließlich über die Typhierarchie festgelegt, ob ein Attribut obligatorisch ist. Innerhalb der Tabellenhierarchie kann das nicht geändert werden. Die Eindeutigkeit (`UNIQUE`) von Attributen kann in Subtabellen neu eingeführt, aber nicht zurückgenommen werden.

Trigger: Trigger, die auf den Supertabellen definiert wurden, müssen, sofern sie nicht von den Subtabellen überschrieben wurden, auch bei Tupeln der Subtabellen ausgeführt werden[3].

4.2.2 Einschränkungen bei der Modellierung/Abbildung

Damit eine Lösung für die Mehrfachvererbung bei Typ- bzw. Tabellenhierarchien möglich ist, müssen folgende Einschränkungen bei der Modellierung gemacht werden:

Attributnamen dürfen lediglich so vergeben werden, das sich keine Mehrdeutigkeiten ergeben. Beispielsweise dürfen in den Tabellen B und C keine neuen Attribute definiert sein, die den identischen Namen haben, da sie sonst in D doppelt vorkommen würden.

Die UDRs des Typs einer Tabelle müssen so definiert werden, daß der Aufruf für jeden Typ eindeutig ist. Zum Beispiel dürfen für die Typen der Tabellen B und C keine neuen UDRs definiert sein, die ansonsten die gleiche Signatur besitzen, solange in D die entsprechende UDR nicht überschrieben wird, da sonst nicht eindeutig ist, welche UDR für ein Tupel aus D aufgerufen werden soll.

Trigger können lediglich dann erstellt werden, wenn für jede Tabelle eindeutig ist, welcher Trigger aufgerufen werden muß. Sollte beispielsweise sowohl in B als auch in C ein INSERT-Trigger definiert sein, und in D nicht, wäre nicht eindeutig, welcher Trigger bei einem INSERT auf D ausgeführt werden sollte.

4.2.3 Lösungsmöglichkeiten

Im folgenden werden verschiedene Lösungsmöglichkeiten für das oben geschilderte Problem der Mehrfachvererbung von Tabellenhierarchien vorgestellt und ihre Vor- und Nachteile diskutiert. Dabei wird die in Abschnitt 4.1 vorgestellte TTV^n-Abbildung als Grundlage vorausgesetzt, d. h., eine Anfrage (SELECT, INSERT, UPDATE oder DELETE) erfolgt immer über eine Sicht. Weiterhin müssen bei der Typdefinition von D alle Attribute aus dem Zweig von C inklusive Typ und Integritätsbedingungen (NOT NULL) mit aufgenommen werden.

Diese Lösungsmöglichkeiten sind im einzelnen:

I. Verwendung von Sichten zur Simulation
II. Duplizieren einzelner Tupel und Wartung durch Trigger
III. Verwendung des VTI (Virtual Table Interface [4])

---

[3] Diese Forderung ergibt sich aus der Trigger-Semantik des verwendeten ORDBMS, bei dem pro Ereignis (z. B. Einfügen oder Löschen) lediglich ein Trigger innerhalb einer Tabelle ausgeführt werden kann. In einer Tabellenhierarchie bedeutet das, daß Trigger ähnlich Methoden überschrieben werden, so daß weiterhin lediglich ein Trigger pro Ereignis ausgeführt wird.

Lösungsansatz I: Verwenden von Sichten

Idee: Durch die Verwendung von Sichten soll die Mehrfachvererbung auf der Tabellenhierarchie nachgebildet werden. Die Definition der Sicht (UNION VIEW) auf die Tabelle C könnte folgendermaßen aussehen:

```
-- Hilfssicht, da verwendetes ORDBMS kein UNION in Subquery zuläßt
CREATE VIEW c_d AS
   SELECT * FROM c_ta
   UNION
   SELECT [alle Attribute von C] FROM d_ta;

CREATE VIEW c_vi AS
   SELECT c_d.*,
         (SELECT a_ta FROM a_ta WHERE c_d.ID = a_ta.ID) AS as_row
   FROM c_d;
```

Diese Lösung bietet sich wegen der in Abschnitt 4.1 beschriebenen TTV^n-Abbildung an, da für jeden Zugriff auf Tabellen sowieso Sichten definiert werden, die nur bei der Tabelle C (und allen möglicherweise zwischen A und C in der Hierarchie liegenden Tabellen) erweitert werden müssen. Die erste Sicht vereinigt die Tupel aus C und D und die zweite sorgt dafür, daß Tupel aus Subtabellen vollständig und mit korrektem Typ im Ergebnis erhalten sind (im Attribut `as_row`).

Funktionalität \ Tabelle	A	B	C	D
SELECT * FROM *table*	++	++	+	++
ONLY (*table*)	o	o	o	o
SELECT *table* FROM *table*	++	++	+	++
INSERT INTO *table*	++	++	++	++
UPDATE / DELETE *table*	++	++	NA[a]	++
SELECT f(*table*) FROM *table*	++	++	-[d]	-[d]
CAST*: table_typ*::x und x::*table_typ*	++	++	++	-[d]
FOREIGN KEY	++	++	--[b]	++
CHECK ...	++	++	-[c]	++
UNIQUE	++	++	-[c]	++
Trigger	++	++	-[c]	++

„**o**“ = Syntax ändert sich (durch VTT^n-Abbildung);
„**++**“ = funktioniert;
„**+**“ = leichte Einschränkungen: z. B muß Sicht erweitert werden;
„**-**“ = schwere Einschränkungen z. B. UDRs mehrfach schreiben;
„**--**“ = schwer realisierbar; „**NA**“ = kann nicht realisiert werden;

Tabelle 1: Merkmale des Lösungsansatzes I

Allerdings hat dieser Lösungsansatz schwerwiegende Nachteile (siehe Tabelle 1). Es kann keine Sicht für ein UPDATE oder DELETE auf C definiert werden, bei der sich die Operationen auch auf Tupel von D auswirken (a). Zum anderen können keine Fremdschlüsselbeziehungen auf C definiert werden (b), die auch Tupel der Tabelle D berücksichtigen, da diese nicht auf Sichten definiert werden können. Als Ausweg bliebe nur die aufwendige Realisierung mit Check-Constraints bzw. Trigger bei allen referenzierenden und referenzierten Tabellen. Die Check-Constraints und Trigger von C (und Tabellen zwischen C und A) müssen zusätzlich auch auf D definiert werden (c). Ebenso ist es notwendig, UDRs des Typs von C (inkl. Casts) auf den Typen von

D und A (bei A mit einer Fehlermeldung beim Aufruf) zu definieren (d). Letzteres ist erforderlich, um auf der Sicht von C diese UDRs auszuführen, da die Sicht über die Tabelle A das getypte Tupel liefert.

Lösungsansatz II: Duplizieren und Wartung durch Trigger

Idee: Um die Wirkung der vereinigenden Sicht zu erreichen, wird jedes Tupel der Tabelle D zusätzlich in der Tabelle C gespeichert (dupliziert). Dabei sind die Duplikate in C als solche erkennbar. Die Wartung der Abhängigkeiten erfolgt über Trigger auf den Tabellen C und D (siehe Abbildung 5). In Tabelle 2 werden die Möglichkeiten dieser Lösung aufgezeigt.

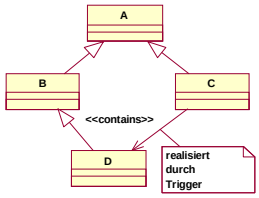


Abbildung 5: Duplizieren und Wartung durch Trigger

Hier müssen bei Anfragen auf A (und Tabellen oberhalb von A) die Duplikate in den Sichten gefiltert werden (a). Beim Einfügen in D sowie UDPATE- und DELETE-Operationen auf C und D ergibt sich ein Mehraufwand durch die Trigger (b). Die Rückgabe des richtigen Typs bei einer Anfrage auf C erweist sich als schwierig, ist jedoch mit folgender Sicht möglich (c):

```
CREATE VIEW c__vi_select AS
      SELECT c_ta.*,
            (SELECT a_ta FROM a_ta WHERE a_ta.ID=c_ta.ID) AS as_row
      FROM c_ta
```

Daraus folgt aber, daß alle UDRs des Typs von C wie bei Lösungsansatz I zusätzlich auf A definiert werden müssen. Ebenso müssen alle UDRs von C (inkl. Cast-Ope-

Funktionalität \ Tabelle	A	B	C	D
SELECT * FROM *table*	+[a]	++	++	++
ONLY (*table*)	o[a]	o	o	o
SELECT *table* FROM *table*	+[a]	++	-[c]	++
INSERT INTO *table*	++	++	++	+[b]
UPDATE / DELETE *table*	NA[e]	++	+[b]	+[b]
SELECT f(*table*) FROM *table*	+[a]	++	-[c]	-[d]
CAST*: table_typ*::x und x::*table_typ*	++	++	++	-[d]
FOREIGN KEY	++	++	++	++
CHECK ...	++	++	++	++
UNIQUE	++	++	++	++
Trigger	++	++	+[b]	+[b]

„**o**“ = Syntax ändert sich (durch VTT[n]-Abbildung);
„**++**“ = funktioniert;
„**+**“ = leichte Einschränkungen: z. B muß Sicht erweitert werden;
„**-**“ = schwere Einschränkungen z. B. UDRs mehrfach schreiben;
„**--**“ = schwer realisierbar; „**NA**“ = kann nicht realisiert werden;

Tabelle 2: Merkmale der Lösungsansatzes II

ratoren) für den Typ der Tabelle D definiert werden (d). UPDATE- bzw. DELETE-Operationen auf A (oder Tabellen oberhalb von A) können nicht direkt ausgeführt werden (e), da das verwendete System in einem UPDATE- oder DELETE-Trigger keine UPDATE- oder DELETE-Operationen zuläßt, die auf die gleiche Tabelle zugreifen. (Bei einem Zugriff auf A sind C und D Teil der gleichen Tabelle). Statt dessen sind ein Statement auf ONLY(A) und weitere auf den einzelnen Subtabellen (in diesem Fall B und C) erforderlich.

Lösungsansatz III: Verwendung des VTI

Bei den eben geschilderten Lösungsansätzen hat sich gezeigt, daß eine Lösung unter Verwendung von SQL bzw. den objekt-relationalen Erweiterungen der ORDBMS lediglich unter Einschränkungen möglich ist, wie beispielsweise das Aufgliedern eines UPDATE-Statements in mehrere Statements. Der ORDBMS-Hersteller Informix bietet jedoch die Möglichkeit an, über das VTI (Virtual Table Interface [4]) Tabellen auf den unteren Ebenen des ORDBMS zu simulieren. Die internen Zugriffsfunktionen für Tabellen werden dabei durch eigene Implementierungen ersetzt. Obwohl das VTI dafür konzipiert wurde, externe Daten zu integrieren, kann es auch verwendet werden, um bspw. „UPDATEABLE UNION“-Sichten zu implementieren. Bei einer Lösung unter Verwendung des VTI lassen sich Sichten und Tabellen durch simulierte Tabellen ersetzen. Dazu gibt es eine Vielzahl von Möglichkeiten, von denen wir hier lediglich den Fall beschreiben wollen, der sich als am vielversprechendsten erwiesen hat.

Idee: Die Tabelle C wird in der Vererbungshierarchie durch eine simulierte Tabelle C(VTI) ersetzt. In einer Tabelle C‘ werden die Tupel von C(VTI) gespeichert. Ein Zugriff auf C(VTI) liefert alle Werte von C‘ sowie alle Tupel der Tabelle D (siehe Abbildung 6). Tupel aus D werden dabei als Duplikate gekennzeichnet, damit sie beim Zugriff auf A und möglichen Tabellen über A durch Sichten eliminiert werden können. Um bei Zugriffen auf C(VTI) und Tabellen zwischen C(VTI) und A die Typinformation von D zu erhalten, muß das vollständige Tupel wie bei Lösungsansatz I und II über eine Teilanfrage auf A als zusätzliches Attribut (`as_row`) mitgeliefert werden.

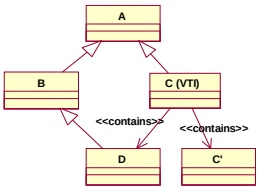


Abbildung 6: Verwendung des VTI (Fall 3)

In Tabelle 3 sind die Merkmale dieses Lösungsansatzes aufgeführt. Es ist zu sehen, das die gesamte Funktionalität erfüllt ist. Duplikate werden bei Zugriffen auf A per Sicht eliminiert (a), der Zugriff auf C erfolgt über das VTI (b), beim „ONLY“-Statement werden noch die Duplikate eliminiert (c). Ein Zugriff auf den Typ der Tabelle D über C erfordert eine Subanfrage auf A (siehe Lösungsansatz I und II) und deshalb die Definition aller UDRs von C auf A (d). Trigger, UDRs, und Check-Constraints von C müssen zusätzlich auf D definiert werden (e). Probleme bereiten die Fremdschlüssel

Funktionalität \ Tabelle	A	B	C	D
SELECT * FROM *table*	+[a]	++	+[b]	++
ONLY (*table*)	o[a]	o	o[b,c]	o
SELECT *table* FROM *table*	+[a]	++	-[b,d]	++
INSERT INTO *table*	++	++	+[b]	++
UPDATE / DELETE *table*	+[a]	++	+[b]	++
SELECT f(*table*) FROM *table*	+[a]	++	-[d]	-[e]
CASTs*: table_typ*::x und x::*table_typ*	++	++	+	-[e]
FOREIGN KEY	++	++	--[f]	++
CHECK ...	++	++	-[e]	++
UNIQUE	++	++	-[f]	++
Trigger	++	++	-[e]	++

„**o**“ = Syntax ändert sich (durch VTT[n]-Abbildung);
„**++**“ = funktioniert;
„**+**“ = leichte Einschränkungen: z. B muß Sicht erweitert werden;
„**-**“ = schwere Einschränkungen z. B. UDRs mehrfach schreiben;
„**--**“ = schwer realisierbar; „**NA**“ = kann nicht realisiert werden;

Tabelle 3: Merkmale der Lösungsansatz III

und Unique-Constraints auf C (f). Da in D Tupel ohne Kontrolle des VTI geändert oder gelöscht werden können, müssen die Fremdschlüsselbeziehungen mit Triggern auf D gewährleistet werden. Im Gegensatz zu Lösungsansatz I können die Fremdschlüssel in den referenzierenden Tabellen jedoch normal definiert werden, da diese über die C(VTI)-Tabelle überprüft werden.

Bewertung der Lösungsansätze

Von Lösungsansatz I müssen wir abraten, da keine UPDATE- oder DELETE-Operationen auf C in einem Statement durchgeführt werden können. In Lösungsansatz II sind diese Operationen für die Tabelle A (und Tabellen über A) nicht verfügbar. Dies liegt allerdings an den (nicht ganz nachvollziehbaren) Einschränkungen des von uns verwendeten ORDBMS bzgl. UPDATE- und DELETE-Operationen in Triggern. Damit ist dieser Lösungsansatz in anderen Systemen möglich, sofern diese einfache Vererbungsbeziehungen unterstützen. Allerdings müssen für alle Tabellen der Vererbungshierarchie unterhalb der Mehrfachvererbung die Tupel redundant gespeichert werden (zumindest die Attribute eines Zweiges). Lösungsansatz III ist lediglich als Lösungsmöglichkeit in einem bestimmten System zu sehen, die nicht einfach übertragen werden kann. Allerdings werden hier alle Anforderungen unterstützt, wenn auch teilweise mit großem Aufwand.

Es ist zu beachten, daß die vorgestellten Lösungsmöglichkeiten voraussetzen, daß eine Tabelle A existiert, da an diese Anfragen zu Typinformation gestellt werden. Sollte keine Tabelle A existieren, kann diese eingefügt werden. In diesem Fall wäre Lösungsansatz II auch in dem von uns verwendeten ORBDMS möglich, da kein UPDATE oder DELETE auf dieser Tabelle zu erwarten wäre.

Um die große Anzahl an Abhängigkeiten zu berücksichtigen, halten wir bei allen drei Lösungsansätzen eine Generierung des Datenbankschemas für sinnvoll.

4.3 Abbildung von OCL-Constraints auf SQL

Je komplexer die zu verwaltenden Datenstrukturen sind, desto schwieriger wird es, die Integrität der gespeicherten Daten zu sichern. Die modellinhärenten Integritätsbedingungen der Datenmodelle reichen im allgemeinen nicht aus, um die Semantik auf Anwendungsebene vollständig zu erfassen. Deshalb müssen zusätzliche Integritätsbedingungen formuliert werden. Wie bereits erwähnt, ist dies auch beim UML-Metamodell geschehen. Neben der Strukturmodellierung als UML-Modell wurden sogenannte Invarianten mit OCL formuliert. Diese Invarianten sind im UML-Repository als SQL-Integritätsbedingungen implementiert. Dabei haben wir von der Möglichkeit gebraucht gemacht, benutzerdefinierte Funktionen zu registrieren und in der Anfragesprache zu nutzen.

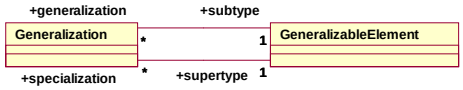


Abbildung 7: Ausschnitt UML-Metamodell

Abbildung 7 zeigt einen Ausschnitt aus dem UML-Metamodell. Die eine Klasse repräsentiert Modellierungselemente, die an Vererbungsbeziehungen teilnehmen dürfen, die andere die Beziehung selbst. Wie üblich ist auch in UML ein Zykel in den Vererbungsbeziehungen nicht erlaubt. Dies wird durch die nachfolgende, in OCL formulierte Invariante gesichert:

```
context GeneralizableElement inv:
   not self.allParents->includes( self )
```

Man beachte, daß im Constraint eine Methode „allParents“ aufgerufen wird. Diese ist ebenfalls in [10] definiert. Sie berechnet die transitive Hülle aller Eltern. Um die Invariante in eine SQL-Integritätsbedingung zu überführen, definieren und implementieren wir eine benutzerdefinierte Funktion, die genau dies berechnet. Es resultiert der nachfolgende SQL-Check-Constraint:

```
CHECK NOT EXISTS(
        SELECT *
        FROM generalizable_element_vi_s ge
        WHERE ge NOT IN all_parents( ge ) )
```

Die Modellierungselemente sind in der Tabelle „generalizable_element“ gespeichert. Der Zugriff darauf erfolgt über die passende Sicht „generalizable_element_vi_s“. Die UDR „all_parents“ liefert die oben beschriebene Menge. Die implizite „for all“-Semantik der Invariante wird durch „not exists not“-Konstrukt nachgebildet.

Wie man an dem Beispiel erkennen kann, treten in Entwurfsumgebungen Integritätsbedingungen auf, die komplexe Berechnungen beinhalten. Im Gegensatz zu den RDBMS erlauben es ORDBMS, solche Berechnungen in Anfragen zu benutzen. Diese Erweiterung ermöglicht also eine an die Bedürfnisse von Entwurfsumgebungen angepaßte Integritätssicherung.

Wir nutzen dies nicht nur, um globale Integritätsbedingungen zu sichern, sondern auch, um die Kooperation der Entwickler zu kontrollieren und den Entwicklungspro-

zeß zu steuern. So lassen sich Integritätsbedingungen einsetzen, um das Erreichen des Ziels eines Entwurfsschrittes zu überprüfen. Im Rahmen der Bearbeitung können weiterhin phasenweise geltende Anforderungen an das Entwurfsobjekt zugesichert werden, beispielsweise um Entwicklungsstände einzufrieren. Der korrekte Einsatz von Entwurfswerkzeugen läßt sich mit deskriptiv formulierten Vor- und Nachbedingungen kontrollieren [11].

Diese Vorteile lassen sich nur nutzen, wenn Integritätsbedingungen einfach deskriptiv formuliert und zentral gewartet werden können. Ersteres erlaubt OCL, für letzteres sind jedoch weitere Vorkehrungen notwendig. Da die Abbildung von UML-Modellen schwierig ist, erweist sich auch das manuelle Übersetzen eines OCL-Constraints in eine SQL-Integritätsbedingung als aufwendig und fehleranfällig. Neben einem umfangreichen Verständnis der beiden Sprachen OCL und SQL muß der betreffende Entwickler auch noch mit der Abbildung des UML-Modells vertraut sein. Deshalb streben wir eine automatische Übersetzung von OCL nach SQL an.

Der OCL-Constraint wird dazu eingelesen und in eine interne Darstellung gebracht, die wir Übersetzungsgraph nennen. Dieser besteht aus zwei Arten von Knoten, Übersetzungsknoten und Metadatenknoten. Die Übersetzungsknoten implementieren den SQL-Generierungsalgorithmus. Dazu sind Metadaten notwendig, die von den Metadatenknoten bereitgestellt werden. Ein Beispiel für einen Übersetzungsgraphen ist in Abbildung 8 zu sehen. Er resultiert aus dem obigen OCL-Constraint.

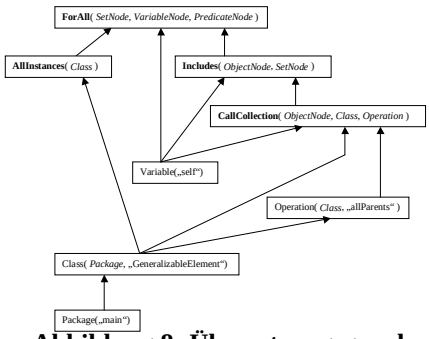


Abbildung 8: Übersetzungsgraph

Der Übersetzungsvorgang basiert auf SQL-Templates und Konstruktionsregeln, welche die Übersetzungsknoten bereitstellen bzw. implementieren. Der Knoten „ForAll“ nutzt folgendes SQL-Template:

```
NOT EXISTS(    SELECT *
               FROM ($setnode$) AS $variablenode$
               WHERE NOT ( $predicatenode$ ) )
```

Die Parameter sind durch „$“-Zeichen eingeschlossen und werden im Rahmen der Übersetzung durch SQL-Fragmente ersetzt, die von den zugehörigen Subknoten im Graphen bereitgestellt werden. Bei manchen Regeln wird der Übersetzungsvorgang außerdem durch Metadaten gesteuert. So benötigt der Knoten „AllInstances“ ein Objekt der Klasse „Class“, das alle notwendigen Informationen über die Klasse „GeneralizableElement“ bereitstellt (Name der Klasse, Name der Datenbanktabelle, usw.). Das Ergebnis des Übersetzungsvorgangs ist allerdings komplizierter struktu-

riert als das oben gezeigte manuell erstellte Beispiel. Es stellt damit auch größere Herausforderungen an den Optimierer des ORDBMS. Ob diese ähnlich leistungsfähig sind wie einige der Optimierer heutiger Systeme, ist noch zu prüfen.

Augenblicklich entwickeln wir einen Übersetzer, der nach obigem Verfahren arbeitet. Einige OCL-Konstrukte lassen sich momentan noch nicht nach SQL übertragen, beispielsweise der generische Iterator-Operator. Wir sind jedoch optimistisch, dieses Problem mit den Erweiterungsmöglichkeiten des ORDBMS zu lösen.

5 Integration externer Daten

Der Wunsch zur Integration externer Daten ist in den vorangegangenen Abschnitten ausreichend motiviert worden. Abbildung 9 zeigt die bei der Integration möglichen Alternativen. Das Dokument D soll Daten repräsentieren, die in einer nicht kooperativen Datenquelle gespeichert sind, beispielsweise im Dateisystem. Die Daten können als Folge von Bytes gelesen werden. Für eine Anwendung ergeben sich die folgenden Wege, um auf diese Daten zuzugreifen:

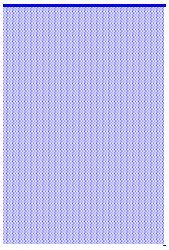


Abbildung 9: Verschiedene Zugriffsarten

Direkter Zugriff (A): Bei diesem Szenario greift die Anwendung direkt auf die Daten zu.

Zugriff über ein Verwaltungsprogramm (B): Hier greift die Anwendung über die Schnittstelle eines Verwaltungprogramms zu. Die Rolle dieses Verwaltungsprogramms, im Bild der externe Server, kann beispielsweise von einem anderen DBMS, einem Dokumentenverwaltungssystem oder einem speziellen Dateisystem übernommen werden. Die angebotene Schnittstelle erlaubt es, anstelle der Binärdaten Anwendungsobjekte zu lesen und zu schreiben.

Zugriff über das ORDBMS (C): In diesem Fall wird auf die Daten über die SQL-Schnittstelle des ORDBMS zugegriffen. Dazu ist dort ein Datenbankschema erzeugt worden. Die zugehörigen Daten können dabei auf zwei Wegen integriert werden, wobei auch Mischformen möglich sind:

- **Kommunikation mit einem externen Server (C1):** Bei dieser Form der Integration kommuniziert das ORDBMS mit einem externen Server, der eine passende Kommunikationsschnittstelle anbietet. Analyse und Aufbereitung der Daten erfol-

gen dabei durch den externen Server. Das ORDBMS liest und schreibt Anwendungsobjekte. Über die Erweiterungsschnittstelle wird lediglich Datenkonvertierungs- und Kommunikationslogik in das ORDBMS integriert.

- **Direkter Zugriff und Aufbereitung durch das DBMS (C2):** Hier greift das ORDBMS direkt auf die Binärdaten zu. Diese werden von der in das DBMS integrierten Logik eingelesen, interpretiert, aufbereitet und weitergereicht.

5.1 Einbettung in die transaktionsorientierte Verarbeitung

Bei der Integration von Datenquellen mit lesendem und schreibendem Zugriff in ein ORDBMS muß auch die transaktionsorientierte Verarbeitung bedacht werden.

Wenn Datenzugriffe über den Weg A zu erwarten sind, so ist keine transaktionsorientierte Verarbeitung möglich, da diese Zugriffe nicht unter der Kontrolle des ORDBMS liegen und die Datenquelle sich nicht kooperativ verhält. Dieser Weg wird daher im folgenden nicht mehr betrachtet.

Bei Zugriffen über das ORDBMS (C) muß man die beiden Integrationsmöglichkeiten unterscheiden. Erfolgt die Integration über einen externen Server, so müssen dieser und das ORDBMS zusammen eine verteilte Transaktion durchführen. Dies setzt insbesondere voraus, daß der externe Server selbst schon lokale Transaktionen unterstützt. Dann sind auch gleichzeitige Zugriffe anderer Anwendungen über den externen Server unproblematisch. Die verteilte Transaktion ist dabei für die Anwendung transparent. Will die gleiche Anwendung in einer Transaktion sowohl über das ORDBMS als auch über den externen Server zugreifen, so muß die verteilte Transaktion auf Anwendungsebene sichtbar sein, da nur sie alle beteiligten Komponenten kennt und diesen den Transaktionskontext bekannt machen kann. Beide Systeme müssen diese Kooperation unterstützen.

Integriert man die externen Daten direkt in das ORDBMS, so gelten nicht automatisch die Transaktionseigenschaften. Da die Datenquelle keine Vorkehrungen für Isolation und Recovery trifft, muß dies bei der Integration geschehen. Grundvoraussetzung für eine eigene Fehlerbehandlung ist, daß die integrierte Logik alle den Transaktionskontext betreffenden Zustandsänderungen vom ORDBMS mitgeteilt bekommt. Dies gilt auch für den Fall, daß der Server vollständig abstürzt. Um Transaktionen gegeneinander zu isolieren, ist eine geeignete Schnittstelle zur Sperrverwaltung des ORDBMS wünschenswert. In einigen Fällen kann dies aber auch über die übliche SQL-Schnittstelle simuliert werden. Bei gleichzeitigem Zugriff über die Wege B und C muß der externe Server mit dem DBMS kooperativ arbeiten. Die Vorkehrungen beider Systeme für Isolation und Fehlerbehandlung müssen abgeglichen werden. Greift die gleiche Anwendung auf beiden Wegen zu, so ist zusätzlich wiederum eine auf Anwendungsebene sichtbare verteilte Transaktion notwendig.

Bei dem von uns verwendeten ORDBMS lassen sich an der Integrationsschnittstelle noch keine verteilten Transaktionen durchführen. Wenn Transaktionen notwendig sind, muß eine direkte Integration durchgeführt und der alleinige Zugriff über das ORDBMS sichergestellt werden. Je nach Anwendung lassen sich dann Maßnahmen

für Isolation und Fehlerbehandlung treffen. Diese sind jedoch anwendungsabhängig, da das System keine passende Schnittelle zur Sperrverwaltung oder zur Recovery-Komponente anbietet.

5.2 Entwicklungsaufwand bei der Integration

Im ersten Fall (C1) müssen die Rohdaten in der integrierten Logik interpretiert werden. Dazu ist mehr Anwendungslogik im ORDBMS notwendig, was die Realisierung, insbesondere mit Blick auf die Fehlersuche, aufwendig macht. Positiv ist zu vermerken, daß man bei der Integration nicht auf die Kooperation externer Komponenten angewiesen ist. Außerdem entfallen gegenüber der vorangegangenen Lösung Kommunikationskosten, was sich positiv auf das Leistungsverhalten auswirken dürfte.

Der Vorteil der Integration über einen externen Server (C2) besteht darin, daß der Datenzugriff auf einem höheren Niveau erfolgt, da an der Schnittstelle bereits Anwendungsobjekte sichtbar sind. Die Realisierung der Integration beschränkt sich dadurch auf Kommunikation- und Datenkonvertierung, was den Entwicklungsaufwand weiter reduziert. Außerdem lassen sich bestehende Verwaltungsprogramme und deren Funktionalität ausnutzen. Im Falle einer Dokumentenverwaltung wäre es beispielsweise nicht sinnvoll, spezielle Indexstrukturen und Suchalgorithmen nochmals zu implementieren.

5.3 Anwendungsbeispiel: UML-Server

Im Falle unseres UML-Repository wollen wir auf Daten eines Werkzeugs zur graphischen Modellierung, die im Dateisystem gespeichert sind, zugreifen. Der Zugriff über die SQL-Schnittstelle soll rein lesend erfolgen, da er nur dazu dient, Auswertungen auf den UML-Modellen durchzuführen oder die Daten in das ORDBMS zu übertragen.

Der Zugriff auf die Daten erfolgt also auf zwei Arten. Da sich die Modellierungswerkzeuge üblicherweise nicht kooperativ verhalten, gehen wir hier von einem direkten Zugriff aus (A). Unsere eigenen Anwendungen nutzen die SQL-Schnittstelle zum Zugriff auf die Daten über das ORDBMS. Um die Realisierung so einfach wie möglich zu halten, haben wir uns dafür entschieden, die Integration über einen selbst entwickelten externen Server zu implementieren (C2). Da wir vom ORDBMS aus nur lesend auf die Daten zugreifen, ist eine verteilte Transaktion nicht notwendig. Damit die beiden Arten des Zugriffes nicht kollidieren können, müssen besondere Vorsichtsmaßnahmen getroffen werden, beispielsweise das Kopieren der Daten in einen nur dem ORDBMS zugänglichen Bereich. Dies ist jedoch unabhängig von der Art der Integration.

Der externe Server ist grob in drei Ebenen aufgeteilt: Ebene der Kommunikation, Ebene der Projektion und Selektion, Ebene des Dateizugriffs. Damit verschiedene Dateiformate zugegriffen werden können, definiert die unterste Ebene eine Schnittstelle zu Wrapper-Komponenten, welche die Analyse der Dateien übernehmen. In

Abbildung 10 sind Wrapper für Dateien im XMI (XML Metadata Interchange) und im proprietären RationalRose-Format angedeutet. Die extrahierten Daten werden dann an die nächste Ebene zur Aufbereitung weitergegeben. Um den Datenaustausch zwischen ORDBMS und externem Server zu reduzieren, können auf dieser Ebene Prädikate ausgewertet werden. Weiterhin werden hier nur die benötigten Daten aus den Tupeln projiziert. Auf oberster Ebene wird die Kommunikation mit dem ORDBMS realisiert. Hier werden einlaufende Anfragen entgegengenommen, deren Auswertung angestoßen und das Ergebnis an das ORDBMS übertragen.

Abbildung 10: Wrapper

Das von uns verwendete ORDBMS bietet nur wenig Unterstützung für die Transaktionsverarbeitung. Ein verteiltes 2-Phasen-Commit mit dem externen Server war nicht möglich, aber auch nicht notwendig. Die Implementierung der integrierten Komponenten (Zugriffsfunktionen) hat sich als schwierig erwiesen, da die Schnittstellen des ORDBMS schwierig zu benutzen sind. Vor allem die Fehlersuche ist aufwendig, da die üblichen Werkzeuge nicht verwendet werden können. Das System war während der Entwicklung aufgrund der Eingriffe nicht stabil, was den Aufwand erhöht. Diesbezüglich hat es sich also als sinnvoll erwiesen, einen Großteil der Logik in den externen Server zu integrieren. Da ein rein lesender Zugriff geplant war und besondere Maßnahmen bei ansonsten direktem Zugriff immer notwendig sind, stellt die mangelnde Integration in die Transaktionslogik bei uns keinen Nachteil dar.

6 Zusammenfassung und Ausblick

Ziel dieses Aufsatzes war eine Untersuchung, inwieweit ORDBMS technische Entwurfsanwendungen besser unterstützen als RDBMS. Als Anwendungsbeispiel diente uns ein UML-Repository, in dem UML-Modelle verwaltet werden.

Die vorgestellte VTT^n-Abbildung ist gut geeignet, das UML-Metamodell auf ein ORDBMS-Schema abzubilden. Insbesondere die Vererbungsbeziehungen sind dabei hilfreich. Die Probleme der Mehrfachvererbung sind mit dem VTI von Informix lösbar. Für den Fall, das dies (oder ähnliche Möglichkeiten) nicht verwendbar ist, bleibt der Ansatz des Duplizierens mittels Trigger. Die Einschränkung der fehlenden Möglichkeit eines UPDATEs auf den Supertabellen ist zudem lediglich bei dem von uns verwendeten ORDBMS aufgetreten. Andere Systeme erlauben ein UPDATE in einem UPDATE-Trigger auf der gleichen Tabelle. Hier zeigt sich, das ein Standard

notwendig ist, damit die Hersteller eine einheitliche Syntax und Semantik unterstützen.

ORDBMS bieten eine adäquate Unterstützung für die beschriebenen besonderen Anforderungen an die Sicherung semantischer Integrität: Integritätsbedingungen lassen sich deskriptiv formulieren, zentral kontrollieren und können komplexe Berechnungen beinhalten. Insbesondere letzteres wird erst mit benutzerdefinierten Funktionen möglich, die im ORDBMS registriert sind und sich in Anfragen einsetzen lassen.

Die Integration externer Daten ist ein in Entwicklungsumgebungen typisches Problem. Das ORDBMS dient uns als Integrationsplattform, um einheitlich mit eigenen Anwendungen mittels der Anfragesprache SQL auf Daten zuzugreifen, die von mehreren heterogenen Quellen in einem proprietären Format bereitgestellt werden. Leider erfüllt die verfügbare Funktionalität des ORDBMS noch nicht alle Wünsche. Die Integration ist deshalb sehr aufwendig.

Insgesamt hat sich gezeigt, daß die Anforderungen unseres Anwendungsbeispiels von ORDBMS besser unterstützt werden als von rein relationalen Systemen. Es konnte jedoch anhand der Mehrfachvererbung verdeutlicht werden, daß die Entwicklung von OR-Systemen noch am Anfang steht. Dies betrifft nicht so sehr das Fehlen der Mehrfachvererbung selbst. Vielmehr würde sie sich leichter simulieren lassen, wenn die verschiedenen Konzepte orthogonal voneinander einsetzbar wären. Beispielsweise sollte man eine Sicht auf eine getypte Tabelle in einer Vererbungshierarchie definieren können, bei der in Anfragen auch Tupel aus Subtabellen den korrekten Typ tragen. Auch die schwierige Benutzbarkeit hat uns mehrfach behindert. Vieles läßt sich realisieren, selten jedoch so, wie man es zunächst erwartet. Wir hoffen, daß die Entwicklung der ORDBMS soweit voranschreitet, das ihre Verwendung einfacher wird. Die bisherige Entwicklung ist es zwar ein großer Schritt für die Hersteller, aber nur ein kleiner in die richtige Richtung.

7 Literatur

1. Cattell, R.G.G., Barry, D.K.:The Object Database Standard: ODMG 2.0. Morgan Kaufmann, 1997.
2. Date, C. J.: An Introduction to Database Systems. 6. ed, Addison Wesley, 1995
3. Guttmann, A.: R-Trees: A Dynamic Index Structure for Spacial Searching. Oric. ACM SIGMOD Conf. Boston, 1984.
4. Informix: Guide to the Virtual Table Interface. Informix Press, 1997.
5. ISO TC184/SC4/WG5, Product Data Representation and Exchange – Part 11: EXPRESS Language Reference Manual. http://www.nist.gov/sc4/, 1994
6. ISO Final Committee Draft – Database Language SQL. ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public/, 1999.
7. Mattos, N. M., Kleewein, J., Tork Roth, M., Zeidenstein, K.: From Object-Relational to Federated Databases. Proc. Datenbanken in Büro, Technik und Wissenschaft (BTW99), 1999.
8. OMG, Object Constraint Language Specification. Version 1.1, OMG Document ad/97-08-08, 1997.
9. OMG, UML-Notation Guide. Version 1.1, OMG Document ad/97-08-05, 1997.
10. OMG, UML-Semantics. Version 1.1, OMG Document ad/97-08-04, 1997.
11. Ritter, N.: DB-gestützte Kooperationsdienste für technische Entwurfsanwendungen. „infix“-Verlag, Sankt Augustin, 1997
12. Stonebraker, M., Brown, M.: Object-Relational DBMSs: Tracking the next greate Wave. Morgan Kaufmann Series in Data Management Systems, 1999.