# Diplomarbeit:

# Entwurf und Implementierung eines generischen Anfrageübersetzers zur Integration heterogener Informationsquellen

Sebastian Jekutsch

Prof. Dr. J. Calmet
Institut für Algorithmen und Kognitive Systeme (IAKS)
Universität Karlsruhe

28. Juni 1996

## Kurzfassung

Im Umgang mit Informationssystemen aller Art tritt häufig der Fall auf, daß ein Datum aus der einen Datenbank mit einem Datum aus einer anderen Wissensquelle verknüpft werden muß. Haben die zugrundeliegenden Informationsquellen verschiedene Architekturen und Zugriffsweisen, so muß diese Verknüpfung der Benutzer selbst erledigen. Bei größeren Datenmengen ist dies nicht mehr möglich. Ein Mediatorprogramm assistiert dem Benutzer, indem es einen einheitlichen Zugang zu einer Reihe heterogener Wissensquellen bietet und Verknüpfungen zwischen diesen automatisieren kann. Der Benutzer arbeitet fortan nur noch mit dem Mediator. Zwischen dem Mediator und den Wissensquellen muß jeweils ein Übersetzer die unterschiedlichen Anfragesprachen und Datenformate in die einheitliche Mediatordarstellung übersetzen. Einen solchen Übersetzer für jede einzelne Wissensquelle von neuem zu implementieren bedeutet einen immensen Aufwand.

Die vorliegende Arbeit beschreibt einen Weg, diesen Aufwand zu minimieren. Indem ein in den meisten Fällen verwendbares Gerüst entworfen, beschrieben und implementiert wird, reduziert sich die Arbeit des Übersetzer-Schreibens auf Wissensquellen-spezifische Details. Die Spezifikation der Anfrageübersetzung erfolgt in wesentlichen Teilen deskriptiv, was die Erstellung, Fehlersuche und Wartung erleichtert. Der Ansatz nutzt die Ergebnisse der Compilertechnik, geht jedoch von einer neu entwickelten Darstellung von Anfragen aus (Kastendarstellung), die speziell für deren Übersetzung geeignet ist. Verschiedene Anwendungen zeigen die Mächtigkeit des Rahmens auf, auch für Fälle, in denen eine Wissensquelle eine an sie gestellte Anfrage gar nicht ohne äußere Hilfe bearbeiten kann.

Neben der neuartigen Darstellung von Anfragen bietet diese Arbeit erstmals eine Integration von Modussystemen (Bindungsmuster) und Anfrage-Bearbeitungsregeln zur Spezifizierung des Anfrageverhaltens einer Wissensquelle. Durch das hier zum ersten Mal beschriebene Verfahren, semantische Aktionen gemäß den Fähigkeiten einer Informationsquelle zu implementieren oder nicht, wird die Erstellung eines Anfrage-Übersetzers deutlich vereinfacht. Anfragen können konjunktiv sein und dürfen auch Negationen enthalten. Eine neuartige Modularisierung des Übersetzers unterstützt die Wiederverwendung von vormals erstelltem Code.

Für Mami und Vati

## Danksagung

Zu danken habe ich: Peter für seine Neugierde und Zustimmung; Joachim vor allem für Sachen, die die Diplomarbeit gar nicht direkt betrafen; Silvia für die Herausarbeitung eines (leider immer noch nicht gelösten) Problems; Morio für einige gute Gedanken, die ich leider nicht sofort verstand; und einigen Bekannten und Freunden – insbesondere Martin, danke für den Tee und das Korrekturlesen! – die ich bequatschen konnte.

## Erklärung

Hiermit erkläre ich, daß ich die Diplomarbeit ohne unzulässige Hilfe erstellt und alle verwendeten Literaturquellen angegeben habe.

Karlsruhe, den 28.Juni 1996

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

*Mediation simplifies, abstracts, reduces, merges and explains data. A mediator is a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications.*

Gio Wiederhold in [Wie92]

In der Einleitung wird die vorliegende Arbeit genauer beschrieben und in einen bestehenden Projektrahmen eingeordnet. Zunächst ist zu klären, was ein Mediator ist und wie er eingesetzt wird. Nach der Einordnung der Arbeit im zweiten Abschnitt wird eine Inhaltsübersicht dieser Ausarbeitung gegeben. Im letzten Abschnitt werden zentrale Begriffe kurz erläutert. Diese Einleitung dient somit als Motivation und Vorbereitung für den Kern der Arbeit, der mit dem zweiten Kapitel beginnt.

## 1.1 Mediatoren

Ein Mediator ist ein Informationssystem, das andere Wissensquellen nutzt, um Informationen geben zu können. Oder: Informationsquellen können einheitlich in einem Mediator zusammengefaßt werden. Abbildung 1.1 zeigt die Grobarchitektur einer Mediatorumgebung und verbildlicht beide Sichtweisen. Die erste ist mediatororientiert: Der Mediator *nutzt* die Wissensquellen. Die zweite geht von den Quellen aus: Der Mediator *vereinigt* die Wissensquellen. Ein Mediator entspricht beiden Sichtweisen. Sie werden im folgenden genauer beschrieben.

### 1.1.1 Nutzen von Domain-Experten

Wissensbasierte Systeme oder Expertensysteme kranken häufig daran, daß ihr Wissens- oder Diskursbereich (auch Domäne genannt) klein ist. Eine große Wissensbasis zu erstellen und zu warten ist schwierig. Um ein größeres System zu erstellen, kann man versuchen, auf kleine stabile Expertensysteme zurückgreifen, sie also zu benutzen. Größere wissensbasierte Systeme auf diese Art zu erstellen hat einige Vorteile (siehe auch [PL91]):

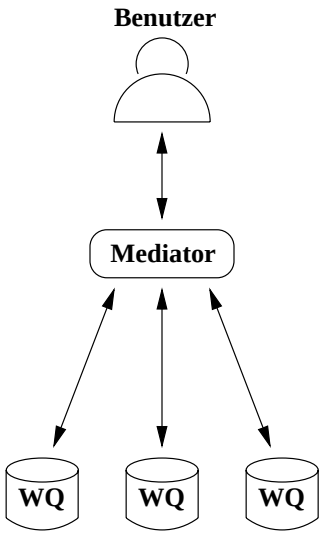



Abbildung 1.1: Grundmediatorarchitektur

- Je nach Diskursbereich sollten verschiedene Inferenzmethoden und -techniken (siehe z.B. [Bib93]) angewendet werden. Ebenfalls bieten sich bei gleichem Diskursbereich selbst innerhalb ein und derselben Wissensrepräsentation für verschiedene zu erledigende Inferenzen auch verschiedene Kodierungen an ([vHPS94, Mye94]). Jedes Problem hat seine optimalen Wege und Mittel, gelöst zu werden. Die Expertensysteme werden also einzeln in ihrer geeigneten, speziellen Art und Weise realisiert und in einem größeren System zusammengefaßt. Hier entsteht natürlich das Problem der korrekten Verwendung und Verbindung mehrerer unabhängig entstandener und unterschiedlich entwickelter Expertensysteme.

- Schon bestehende Systeme können wiederverwendet werden, ohne nochmal erstellt, modifiziert oder kopiert werden zu müssen.

- Wie oben schon erwähnt werden auf diese Weise die Wissensbasen klein gehalten, was vorteilhaft in bezug auf ihre Erstellung und Wartung ist. Natürlich verursacht es Aufwand, die einzelnen Wissensbasen zu einer neuen zusammenzufügen. In sofern wird es keine Arbeitserleichterung geben. Die Situation ähnelt der beim modularen oder objektorientierten Programmieren.

- Neue Expertensysteme können nachträglich hinzugefügt werden, ohne den Aufbau des Gesamtsystems wesentlich ändern zu müssen. Dies ist eine zentrale Eigenschaft von Multi-Agenten-Systemen aus dem Gebiet der verteilten künstlichen Intelligenz.

- Mehrere unabhängig entstandene kleine Experten mit gleichem Diskursbereich erhöhen die Robustheit und Fehlertoleranz. Es kann sehr sinnvoll sein, zu einem Problem mehrere Quellen zu fragen und aus eventuellen Widersprüchen neue Schlüsse zu ziehen. Man nutzt somit mehrere Perspektiven.

- Experten, die unabhängig voneinander als Prozesse laufen, können parallel arbeiten.

Das Problem, aus mehreren kleinen Domänen (Mikrowelten) eine stimmige, reichere, größere (Makrowelt) zu kompilieren, darf jedoch nicht unterschätzt werden.

### 1.1.2 Integration von heterogenen Wissensquellen

Häufig ist es notwendig, Informationen von mehreren verschiedenen Quellen abzurufen, um zu der eigentlich benötigten Information zu gelangen. Um z.B. von einer Stadt in eine andere zu kommen, muß man Daten über die Fluglinien, über Zugverbindungen und eventuell auch über Busfahrpläne haben. Um zu entscheiden, ob man ein Aktienpaket verkauft oder nicht, muß man sowohl in einer Datenbank nach den alten Notierungen suchen, als auch bei einem Online-Dienst Daten über den aktuellen Kurswert abrufen. Es wäre in solchen Fällen sehr hilfreich, wenn der Benutzer dazu nicht von einem Ort an den anderen gehen und sich Zwischenresultate notieren muß. Er sollte ein Werkzeug zur Hand haben, welches verdeckt, daß tatsächlich mehrere Wissensquellen angesprochen werden müssen, um den Dienst zu erbringen. Die einzelnen Wissensquellen müssen zu diesem Zweck integriert werden. Das Problem dabei ist häufig die Heterogenität der einzelnen Quellen. Die Quellen an sich bleiben so wie sie sind (autonom). Deshalb wird dieser Ansatz auch als *föderiert* bezeichnet [SL90].

Ein wesentliches Problem ist die Heterogenität. Die Informationsquellen unterscheiden sich nicht nur darin, daß gleiche Informationen verschieden dargestellt werden (relationale oder objektorientierte Datenbanken, Dateien, Expertensystemen, Katalogen, etc. ), sondern – und dies wird häufig übersehen – selbst bei gleicher Darstellung kann eine Wissensquelle andere Informationen enthalten als eine andere. Sie ergänzen sich nicht nur, sondern können sich auch widersprechen. Eine Integrationsmethode muß dafür Konfliktlösungen bieten.

### 1.1.3 Mediatoren als Vermittler

Ein Mediator integriert verschiedene Wissensquellen, aber er fügt weiteres, vermittelndes (daher sein Name) Wissen hinzu. Er ist sozusagen ein Expertensystem zur Integration heterogener und sich eventuell sogar widersprechender Wissensquellen. Er gibt die Informationen, die die Wissensquellen bieten, nicht nur an den Benutzer weiter, sondern zieht selbständig Schlüsse daraus. So könnte er automatisch den schnellsten und den billigsten Weg zwischen zwei Städten ermitteln, ohne daß der Benutzer sich die einzelnen Datenbasen separat anschauen muß. Oder er leitet den Verkauf der Aktien automatisch in die Wege, wenn gewisse Daten nach bestimmten Regeln dafür sprechen. Somit liegt seine Aufgabe *zwischen* den beiden bislang genannten Gesichtspunkten:

- Er ist kein eigenständiges wissensbasiertes System, das – wenn es nötig oder sinnvoll erscheint – andere Expertensysteme „zu Rate zieht“. Sein Zweck ist vielmehr einzig und allein, die anderen Wissensquellen zu integrieren, um neue Schlüsse aus ihren Informationen ziehen zu können. Das heißt auch, daß die Wissensbasis des Mediators sehr stark von den Wissensquellen abhängt.

  Ein wesentlicher Unterschied zwischen einem Mediator und einem Expertensystem besteht auch darin, daß ein Expertensystem von sich aus initiativ wird, d.h. dem Benutzer Fragen stellt, wenn noch wichtiges Vorwissen fehlt. Ähnlich wie eine Datenbank geht

ein Mediator so nicht vor. Dennoch ist es möglich, den Benutzer selbst als eine Wissensquelle zu modellieren, womit ein entsprechendes Verhalten realisiert werden kann.

- Er integriert nicht nur die Wissensquellen zu einer virtuellen einheitlichen Quelle, sondern fügt Wissen hinzu,

  1. um neue Informationen zu kompilieren, die sich erst aus dem Zusammenschluß der Einzelinformation ergeben können (horizontale Integration).
  2. um Konflikte und Widersprüche zwischen zwei Wissensquellen nach diskursabhängigen Regeln auflösen zu können (vertikale Integration).

  Für solche Aufgaben ist eine normale Datenbankfunktionalität nicht ausreichend.

Dabei ist es nicht zwingend, daß der Mediator *sämtliches* Wissen der einzelnen Wissensquellen integriert. Es wird häufig der Fall sein, daß er auf einen Teilbereich spezialisiert ist. Ein anderer Mediator kann mit anderem Schwerpunkt genau dieselben Wissensquellen ansprechen wollen. Ein Mediator fungiert auf diese Weise als Informationsfilter und -aufbereiter. Ziel ist es nicht, nur die Daten, die zwei oder mehrere Wissensquellen bieten, im Sinne eines Data Warehouses[1] nebeneinander zu stellen. Je unübersichtlicher das Daten-Angebot wird, desto eher werden *Informationen* verlangt, nicht nur *Daten*.

Mediatoren, die andere Wissensquellen nutzen, dürfen nicht mit Multi-Agenten-Systemen aus dem Gebiet der verteilten künstlichen Intelligenz [vM92] verwechselt werden. Im Gegensatz zu den Agenten haben die Wissensquellen niemals die Initiative, sondern dienen nur als Server. Man verbindet also nicht zwei Wissensquellen miteinander, damit sie gemeinsam aushandelnd ein Problem lösen. Insbesondere sind die Wissensquellen sehr verschieden; sie verstehen sich gar nicht untereinander und können jede nur in einer eigenen speziellen Sprache angesprochen werden. Insbesondere verstehen nicht alle Wissensquellen dieselben Anfragen, wie es bei Multi-Agenten-Systemen der Fall ist. Die Aktivität in einem Mediatorsystem behält also stets der Mediator, der seinerseits Pläne erstellen kann, wie und in welcher Reihenfolge er die Wissensquellen anspricht und was er mit deren Antworten macht. Der Mediator ist auch kein Blackboard, denn die Wissensquellen lesen oder schreiben nicht von oder auf dieses. Nur der Mediator „liest“ von den Wissensquellen.

Beim Einsatz von Mediatoren entstehen gewisse sicherheitsrelevante und rechtliche Fragen. Zum einen ist zu klären, ob es überhaupt erlaubt ist, zwei unabhängig entstandene Informationsquellen zu koppeln. Nicht selten entsteht daraus die Möglichkeit, den Datenschutz zu verletzen. Zum anderen muß dem Benutzer klar sein, *daß* der Mediator die zugrundeliegenden Informationen filtert und aufbereitet und somit auch verfälscht und nur eingeschränkt wiederspiegelt. Der Mediator wird die Wissensquellen immer auf bestimmte Weise interpretieren, um sie miteinander koppeln zu können. Ein Mediator ist somit ein typischer Fall von Informationstechnik, die Entscheidungen abnimmt. Dabei sollte er aber eigene Entscheidungen nicht völlig unmöglich machen. Eine Erklärungskomponente wäre hier vonnöten, wie man sie auch in Expertensystemen einsetzt. Außerdem sollte die Möglichkeit bestehen, auch direkten, ungefilterten Zugriff auf die Wissensquellen zu erhalten. Aber auch in diesem Fall besteht ein Problem darin, daß die Darstellung der Informationen einer Wissensquelle

[1] Ein Data Warehouse ist gemäß [Inm96] „a subject-oriented, integrated, non-volatile, timevariant collection of data organized to support management needs“.

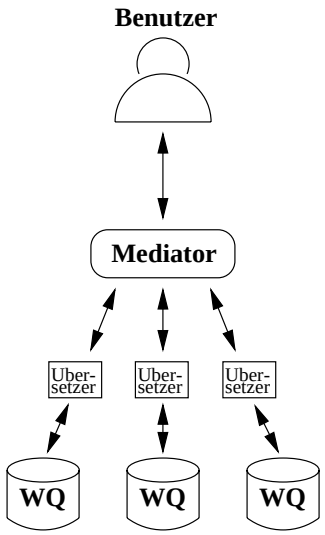



Abbildung 1.2: Mediatorarchitektur mit Übersetzern

im Mediator eine andere ist als ihre ursprüngliche. Diese Darstellungen müssen ineinander übersetzt werden; hier können schwer zu entdeckende Fehler entstehen und Informationen verlorengehen.

## 1.2 Einordnung der Arbeit

Die Notwendigkeit einer Übersetzung wurde soeben erwähnt, und genau hier setzt die vorliegende Arbeit an: Die einzelnen Wissensquellen, die der Mediator integriert, können nicht nur auf verschiedenen Hardware-Plattformen mit ihren verschiedenen Datenformaten und Zugriffsarten eingerichtet sein, sondern sie können auch strukturell gesehen heterogen sein oder gar keinen Bezug zueinander haben. Damit der Mediator aufgrund eines Auftrags vom Benutzer die einzelnen Wissensquellen ansprechen kann, muß er ihre Sprachen „sprechen“ können. Aus diesem Grund arbeitet zwischen dem Mediator und jeder einzelnen Wissensquelle ein Übersetzer, wie Abbildung 1.2 zeigt. Der Übersetzer muß Anfragen, die der Mediator an eine Wissensquelle stellt, in diejenige Form übersetzen, die die Wissensquelle versteht. Zusätzlich muß er auch die Antwort der Wissensquelle in eine Form bringen, mit der der Mediator arbeiten kann.

Inhalt dieser Arbeit ist der Entwurf einer Umgebung zur vereinfachten Erstellung solcher Anfrage-Übersetzer für verschiedenste Wissensquellen. Das System basiert auf dem Projekt KOMET (Karlsruhe Open MEdiator Technology), in welchem eine generische Entwicklungsumgebung für Mediatoren entworfen wird. Die Architektur des Mediators ist also vorgegeben. Ziel ist es, ein Gerüst zu finden, Wissensquellen der verschiedensten Art auf möglichst einfache Weise für den KOMET-Mediator nutzbar zu machen. Dabei sollten die Wissensquellen nicht geändert zu werden brauchen. Im Idealfall werden sie als Black-Box behandelt. Der Übersetzer sollte außerdem Wissensquellen für den Mediator einheitlich genug erscheinen lassen, so daß der Entwickler des Mediators nur in Betracht ziehen muß, *was* eine Wissensquelle weiß, nicht *wie* man ihr dieses Wissen entlockt. Selbstverständlich muß aber der

Übersetzerentwickler einiges über eine Wissensquelle wissen. Die entwickelte Methode sollte unterscheiden zwischen der Art einer Wissensquelle (z.B. relationale Datenbank, Informationssystem), das diese Art realisierende Softwarepaket (Oracle, OPAC) und der speziellen Datenbasis, die damit verwaltet wird (Uni-Datenbank, Uni-Bibliothek). Insbesondere sollten für zwei Wissensquellen möglichst viele Teile des Übersetzers wiederverwendbar sein, wenn sie derselben Art angehören. Die spezielle Datenbasis – sofern sie überhaupt vom Softwarepaket trennbar ist – sollte wenig Einfluß auf den Übersetzer haben. Damit wird erreicht, daß es z.B. einen Übersetzer für Oracle-Wissensquellen gibt, der für alle Oracle-Datenbanken einsetzbar ist. Der Übersetzerentwickler sollte dabei eine möglichst angenehme Entwicklungsumgebung vorfinden. Im Idealfall werden klare Entwurfsschritte vorgegeben. Es sollte so einfach wie möglich sein, die Korrektheit des Übersetzerprogramms zu testen und einzuschätzen.

Die Probleme bei der Entwicklung eines Übersetzers bestehen darin, daß die Wissensquellen heterogen sein können, sowohl in der Art ihrer Anfragesprachen, als auch in der Art ihres Zuganges. Ein Rahmenwerk dafür zu schaffen bedeutet, den kleinsten gemeinsamen Nenner zu finden und die nötigen Erweiterungen klar zu strukturieren.

Bislang gab es bei KOMET noch keine Unterstützung dieser Art. Nicht betrachtet werden telematische Aspekte für den Fall, daß der Mediator über das Netz auf die Wissensquellen zugreifen muß. Plattform-Heterogenitäten, d.h. die Kommunikation zwischen zwei verschiedenen Rechnern, wird ebenfalls nicht betrachtet. Ein Aspekt davon ist auch die Heterogenität der Repräsentation eigentlich gleicher Information, also die Aufgabe der Datenkonvertierung, die in der vorliegenden Arbeit eher am Rande bedacht wird. Die Aufgabe, die Datenstruktur einer Wissensquelle der des Mediators anzugleichen (Schemaintegration), wird nicht schwerpunktmäßig behandelt; siehe dazu aber Abschnitt 2.3.1.

## 1.3 Inhaltsübersicht

Die Ausarbeitung ist wie folgt gegliedert: Im zweiten Kapitel wird die Architektur der Mediatorumgebung KOMET erläutert, um die Aufgabe des Übersetzers exakt zu bestimmen. Einige in diesem Zusammenhang wichtige Probleme werden etwas eingehender erläutert. Kapitel 3 ist der Hauptteil dieser Arbeit. In ihm wird anhand einer Reihe von Beispielen das entwickelte Rahmenwerk für den Übersetzer begründet, erläutert und formal definiert. Anhand realistischer und realisierter Szenarien wird die Anwendung der Entwicklungsumgebung beschrieben. Das darauffolgende Kapitel 4 ist ein Art Benutzerhandbuch für den Übersetzerentwickler. Es erläutert die vorgenommene Implementation und die Arbeiten, die er vorzunehmen hat, um einen lauffähigen Übersetzer fertigzustellen. Das abschließende fünfte Kapitel bietet eine Zusammenfassung, ein Vergleich mit anderen Arbeiten und einen Ausblick.

Der Rest dieses Kapitels ist noch einigen Definitionen von Begriffen vorbehalten, die zentral für den Hauptteil dieser Ausarbeitung sind. Sie dienen zur Klärung und Präzisierung, werden aber nicht formal durchgeführt. Man greife dazu auf die dort genannten Lehrbücher zurück.

## 1.4 Begrifflichkeiten

Die Wissensquellen werden meistens Datenbanken sein. Die Begriffe, die beim Datenbankentwurf benutzt werden, sind auch auf Wissensquellen allgemein anwendbar, etwa Computer-Algebrasysteme oder unstrukturierte Quellen (z.B. http-Seiten im World Wide Web).

Alles, was Daten abspeichert und zugreifbar macht, ist ein **Datenbanksystem (DBS)**. Ein Software-Paket für ein DBS ist ein **Datenbankverwaltungssystem (DBMS)** (z.B. Oracle) und die Daten selbst heißen **Datenbank oder Datenbasis (DB)**. Die Strukturierung einer DB, die ein DBMS bietet, kann in zwei Ebenen abstrahiert werden: Ein Daten**modell** legt die Mittel fest, mit denen die Daten repräsentiert und beschrieben werden können, ein Daten**schema** definiert mit Hilfe des Modells die Gegenstände eines bestimmten Gegenstandsbereichs (Domäne). Die DB legt also die Daten mit Hilfe der vom Schema definierten Strukturen fest, deren Semantik und Syntax das Modell beschreibt. Mit einem Modell wird das *Wie*, also die **Anfragesprache**, mit dem Schema das *Was* der möglichen **Anfragen** an das DBS festgelegt, also die *Anfragesprache* insgesamt. Die DB schließlich bestimmt natürlich die **Antworten** auf eine Anfrage. Eine **Sicht** ist ein Schema, das innerhalb ein und desselben Modells mithilfe eines anderen Schemas definiert wird.

Im Falle des **relationalen Modells** hat man es mit einer Menge von eindimensionalen Tabellen (= Relation) zu tun. Das Schema ist hierbei die Menge der Tabellenköpfe (Tabellenname, Spaltenname und deren Wertebereiche), und die Datenbank ist die Menge aller Tabelleneinträge (Tupel). Man siehe dazu auch [LL95]. Das relationale Modell ist zentral für diese Arbeit. Die Tabellen werden im mathematischen Sinne als **Relationen** verstanden, mit der Definition (dem Inhalt), die die Datenbank angibt und mit dem Definitionsbereich, den das Schema angibt. Es ist zu unterscheiden zwischen einer Relation, die durch eine Menge von Tupeln, und einer solchen, die durch eine Mehrfachmenge von Tupeln beschrieben werden. Bei letzterer kann ein Tupel mehrmals in einer Relation auftauchen, was mathematisch nicht üblich, umso mehr aber im Datenbankbereich nützlich ist.

Relationen kann man wie bei Mengen gewohnt **vereinigen**, **schneiden** und zwischen zweien die **Differenz** und das **Kreuzprodukt** bilden. Eine **Selektion** filtert aus einer Relation diejenigen Tupel heraus, die eine gegebene Bedingung nicht erfüllen. Die entstehende Relation ist dann eine Teilmenge der ursprünglichen Relation. Eine Kombination aus Kreuzprodukt und anschließender Selektion wird auch **Verbund** (engl. Join) genannt. Eine Projektion bildet aus einer Relation eine neue Relation, die nur noch die gewünschten Spalten (**Attribute**) der Tabelle enthält, die Anzahl und die verbleibenden **(Attribut-)Werte** der Tupel aber nicht ändert. Die entstehende Relation ist keine Teilmenge der ursprünglichen Relation mehr, weil sich ihr Schema (=Definitionsbereich) geändert hat. Eine Kombination eines Verbunds mit anschließender Projektion taucht in dieser Arbeit häufig auf. Alle genannten Operationen bilden eine oder zwei Relationen in eine Relation ab. Dieses System bildet die **relationale Algebra**. Für näheres siehe Abschnitt 3.3.2 und z.B. [AHV95].

Der Mediator in KOMET basiert auf dem relationalen Modell, genauer gesagt auf einem **logikbasierten Modell** [Ull88]. Das relationale Modell wird in KOMET erweitert um komplexe Attribut-Domänen [Trc96]. Das klassische relationale Modell betrachtet hingegen nur atomare Attributwertebereiche, also Werte ohne Struktur, wie es ganze Zahlen, Zeichen, aber z.B. nicht Listen sind. Zusätzlich ist die Zugehörigkeit eines Tupels zu einer Relation in

KOMET keine eindeutige sondern eine graduelle. Eine nähere Erläuterung ist in Abschnitt 2.2 zu finden. Dies ist im relationalen Modell nicht vorgesehen. Zum Modell im Mediator siehe Abschnitt 2.2.

Eine Anfragesprache für das relationale Modell ist mit den genannten Operatoren Relationen schon gegeben. Um z.B. alle Nachnamen von volljährigen Mitgliedern eines Vereins zu erhalten, muß man aus der Mitglieder-Relation alle über achtzehn Jahre herausselektieren und nur den Nachnamen projizieren. Die algebraischen Operatoren werden in den realen Systemen unterschiedlich syntaktisch ausgedrückt, in den meisten kommerziellen relationalen Datenbanksystemen z.B. als SQL-Anfrage. Viele Anfragesprachen sind aber mächtiger als die relationale Algebra. In dieser Arbeit wird eine neue **Kastendarstellung** vorgestellt, die der Aufgabe der Anfrageübersetzung angepaßt ist, aber zur relationalen Algebra äquivalent bleibt. Aus der Semantik des erwähnten logikbasierten Modells heraus kommt der zentrale Begriff der **konjunktiven Anfrage**, der in Abschnitt 3.3 definiert wird. Zur Mächtigkeit von Anfragesprachen siehe [AHV95].

Das logikbasierte Modell bietet mit Hilfe von **Klauseln** (Regeln) die Möglichkeit, Konstruktionsregeln für nicht real vorhandene Relationen anzugeben. Die Berechnung von neuen Relationen aus bekannten Relationen ist eine Form von **Inferenz**. Diese Klauseln entsprechen den schon erwähnten Sichten und den konjunktiven Anfragen, bieten aber darüber hinaus auch die Möglichkeit, rekursive Inferenzen zu formulieren, um so z.B. alle Vorfahren einer Person zu ermitteln, wenn sämtliche Eltern-Kind-Beziehungen bekannt sind.

Häufig kann man auf Relationen nur eingeschränkt zugreifen, weil sie als Funktion errechnet werden müssen oder ungeeignet gespeichert sind. So enthält z.B. die Relation „Plus“ das Tupel (4,2,2), weil $4 = 2 + 2$ ist. Ein Taschenrechner – ein Beispiel für eine Wissensquelle – kann aber eine Anfrage wie „Gib mir alle Zahlenpaare, die in der Summe 4 ergeben“ nicht beantworten. Man muß stets die letzten beiden Teile des gesuchten Tupels, also die Summanden, vorgeben oder **binden**, um das Ergebnis, also die Summe, zu erhalten. Niemals kann man die Summe binden und die Summanden frei lassen, wie in der Beispielanfrage gefordert. Nur das **Bindungsmuster** *(frei,gebunden,gebunden)* (oder kurz $(f, g, g)$) ist also erlaubt. Für den Fall, daß alle Bindungsmuster erlaubt wären – wie etwa grundsätzlich bei relationalen Datenbanken – kann man abkürzend $(*, *)$ schreiben.

# Kapitel 2

# Mediatorarchitektur

In diesem Kapitel werden die grundsätzliche Mediatorarchitektur und seine Komponenten erklärt. Eine Mediatorumgebung besteht aus dem Mediator, dessen Modell in Abschnitt 2.2 erläutert wird, mehreren Wissensquellen, deren Eigenschaften in Abschnitt 2.4 besprochen werden, und pro Wissensquelle einem Übersetzer (bestehend aus Schema-(2.3.1) und Modell-(2.3.2) Übersetzer).

## 2.1 Mediatorarchitektur in KOMET

Die Grobarchitektur wurde schon in Abbildung 1.2 dargestellt: Eine Mediatorumgebung besteht aus dem Mediator selbst, einer Reihe von Wissensquellen und je Wissensquelle einem Übersetzer. Die Wissensquellen mit jeweils zugehörigem Übersetzer sind unabhängig voneinander, daher ist im folgenden nur noch von *der* Wissensquelle und *dem* Übersetzer die Rede.

Da die Daten im Mediator in einem noch zu erläuternden **Mediatormodell** vorliegen, die Daten in den Wissensquellen aber in einem anderen **lokalen Modell**, ist eine Übersetzung der Modelle vonnöten. Die lokalen Wissensquellenmodelle werden übersetzt in ein **einheitliches Datenmodell** (engl. Common Data Model). In KOMET sind das einheitliche Datenmodell und das Mediatormodell identisch. Mit Hilfe des einheitlichen Datenmodells wird die Wissensquelle für den Mediator ansprechbar. Ebenso muß das **lokale Schema** einer Wissensquelle in das **föderative Schema**, d.h. in das alle lokalen Schema integrierende Mediator-Schema, übersetzt werden. Beide Übersetzungen werden im Übersetzer vorgenommen, aber dennoch voneinander strikt getrennt. Abbildung 2.1 verdeutlicht die so entstehenden Schnittstellen. Das lokale Schema, im lokalen Modell definiert, ist vorgegeben. Das lokale Schema wird mittels des einheitlichen Datenmodells neu formuliert. Dies leistet der **Modellübersetzer**. Das so entstehende Schema, das zum lokalen Schema äquivalent ist, ist das **Exportschema**. Dieses muß nun innerhalb des einheitlichen Datenmodells = Mediatormodells in das föderatives Schema übersetzt werden. Dies erledigt der **Schemaübersetzer**. Ein föderatives Schema ist nicht nur von dieser Wissensquelle abhängig, sondern auch von allen anderen, ebenso wie von den Aufgaben, die der Mediator erledigen muß.

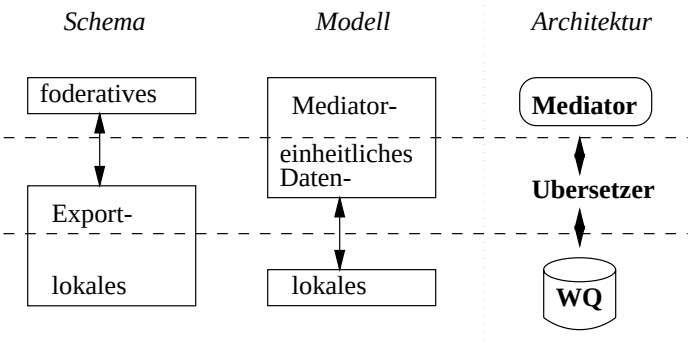



Abbildung 2.1: Übersetzung des Modells und des Schemas in KOMET

Diese Architektur ist eine Spezialisierung einer heterogenen, föderativen Datenbank-Architektur, wie sie etwa in [SL90, LL95] dargelegt wird. Die Gleichsetzung von Mediatormodell und einheitlichem Datenmodell ist naheliegend: Dies spart nicht nur einen weiteren Übersetzungsschritt, sondern ermöglicht es auch, die Schemaübersetzung mit Hilfe des Mediators zu realisieren, da diese ja innerhalb des Mediatormodell geschieht. Die Mächtigkeit des Mediators in KOMET ist dabei sehr hilfreich. Da der Mediator die Möglichkeit hat, andere Wissensquellen anzusprechen, kann dies auch im Rahmen der Schemaübersetzung geschehen. So kann, wenn z.B. eine Wissensquelle einen Geldbetrag in Dollar speichert, der Mediator ihn aber in D-Mark erwartet (ein Schemakonflikt), eine Wissensquelle angesprochen werden, die den aktuellen Wechselkurs speichert. Aufgrund des logikbasierten Mediator-Modells sind zusätzlich rekursive Übersetzungsvorschriften möglich.

## 2.2 Mediator

Der Mediator ist detailiert in [CJK+96] beschrieben. An dieser Stelle soll in für diese Arbeit notwendigem Umfang auf das vorgegebene Mediatormodell eingegangen werden.

Das Mediatormodell ist relational. Die Attributtypen sind jedoch nicht atomar, sondern können auch komplex, d.h. selbst wieder Tupel über andere Typen sein. Auch rekursiv definierte Typen, wie etwa Listen, sind so definierbar. Siehe [Trc96] für eine genauere Erläuterung des KOMET-Typsystems. Die Datenbasis ist in Form von Fakten der Art $p(a_1, \ldots, a_p) : [\mu]$ gegeben, wobei die $a_i$ nicht gebunden sein müssen (non-ground fact). $(a_1, \ldots, a_p)$ bezeichnet dabei ein Tupel in der Relation $p$. Ein solches Tupel ist zusätzlich mit einer Annotation $\mu$ versehen. Dies ist ein Wert aus einem (mathematischen) Verband, der den Grad der Zugehörigkeit des Tupels zur Relation angibt.

Relationen können nicht nur mittels Fakten, sondern alternativ oder zusätzlich mittels Klauseln (oder Sichten) der Form

$$p_0(x_0) : [\mu_0] \leftarrow c_1(y_1), \ldots, c_n(y_n) \;||\; p_1(x_1) : [\mu_1] \;\&\; \ldots, \&\; p_m(x_m) : [\mu_m]$$

definiert werden. Hier wird ein Teil von $p_0$ oder ganz $p_0$ aufgrund der Werte der Relationen $p_1, \ldots, p_m$ und der **Constraint-Relationen** $c_1, \ldots, c_n$ definiert. Constraint-Relationen werden, im Gegensatz zu den anderen Relationen, nicht auf die bislang beschriebene Art

und Weise definiert, sondern durch eine externe, d.h. nicht in der Datenbasis gegebene, Berechnungsvorschrift. Dies sind in KOMET stets Berechnungen, die die Wissensquellen ausführen. Eine Constraint-Relation ist eindeutig einer Wissensquelle (in diesem Zusammenhang auch Domäne genannt) zugeordnet. Sie ist bei dieser explizit vermerkt, z.B. ruft $Uni :: student(699952, X)$ in $X$ alle Studenten mit der angegebenen Matrikelnummer aus der Uni-Wissensquelle ab. Die $y_1, \ldots, y_n, x_1, \ldots, x_m$ sind Termtupel mit meistens überschneidenden Variablenmengen. Man bemerke, daß Annotationen bei den Constraint-Relationen nicht mehr auftauchen. Die Relationen können, mit Ausnahme der Relation im Kopf einer Klausel, negiert auftreten. Eine Relation kann im Kopf und im Rumpf ein und derselben Klausel auftauchen. Diese Relation ist dann rekursiv definiert. Zur exakten modelltheoretischen Semantik siehe [LNS96, JL87].

Die Schnittstelle zu den Wissensquellen besteht also aus den Constraint-Ausdrücken in einer Klausel. Da mehrere Constraint-Relationen derselben Wissensquelle (oder Domäne) in einer Konjunktion auftauchen können, stellt der Mediator im allgemeinen eine **konjunktive Anfrage** an eine Wissensquelle.

Eine solche mittels Klauseln regelbasiert dargestellte Datenbasis besitzt einige Vorteile, wenn man einen Mediator als ein Integrations-Expertensystem ansieht, denn wissensbasierte Systeme werden in vielen Fällen regelbasiert implementiert. Das Mediatormodell in KOMET besitzt eine modellbasierte Semantik, im Gegensatz etwa zu Produktionssystemen, die häufig angewendet werden. Die Annotationen bieten ein zusätzliches Konstrukt, um Unschärfe, Unsicherheit oder auch Zeitinformationen zu formulieren. Sie können auch herangezogen werden, um die Quelle einer Information an ein gespeichertes Tupel zu binden. Mit dieser Hilfe können Widersprüche zweier Wissensquellen innerhalb des Mediators regelbasiert gelöst werden. Siehe dazu auch die Beispiele in [CDJS96].

## 2.3 Übersetzer

Der Übersetzer teilt sich, wie in Abschnitt 2.1 dargelegt, in die Schema- und die Modellübersetzung. Diese Trennung vereinfacht die Entwicklung des Übersetzers, da man beide Übersetzer weitestgehend unabhängig voneinander entwickeln und warten kann. Zudem kann der Modellübersetzer auch für andere Mediatoren, die die gleiche Wissensquelle nutzen, wiederverwendet werden. Unter diesem Gesichtspunkt erscheint es logisch, den Modellübersetzer als einen eigenen Serverprozeß für verschiedene Mediatoren der Wissensquelle vorzuschalten.

Der Übersetzer muß Konflikte zwischen dem Mediator und einer Wissensquelle lösen. Diese Konflikte (Heterogenitäten) können in drei Arten eingeteilt werden:

**Syntaktische Konflikte** tauchen aufgrund verschiedener Darstellungs- und Zugriffsarten auf. Dies entspricht der Tatsache, daß sich das Mediatormodell vom lokalen Modell unterscheidet. Diese Art von Konflikt kann *nur vom Übersetzer* gelöst werden, nicht vom Mediator, da dieser ja eine der Konfliktparteien ist. Die syntaktischen Konflikte zu lösen ist die Aufgabe des Modellübersetzers.

**Semantische Konflikte** tauchen aufgrund verschiedener Speicherungen und Schemata auf. Diese Konflikte können auch vom Mediator gelöst werden, da er selbständig Schemata

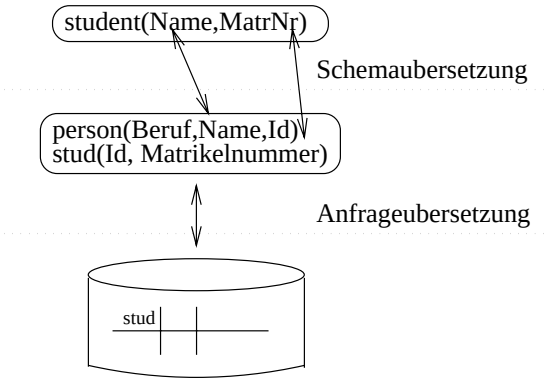



Abbildung 2.2: Die beiden Übersetzungen an einem kleinen Beispiel

ineinander überführen kann, und zwar mit Hilfe von Sichten in Form von Klauseln. Die Grenze zu den syntaktischen Konflikten ist allerdings fließend. So kann z.B. eine Datentypkonvertierung als semantischer (verschiedene Maßeinheiten) oder als syntaktischer (verschiedene Kodierung) Fehler angesehen werden.

**„Pragmatische Konflikte“** sollen in Unterscheidung zu den semantischen Konflikten jene Konflikte heißen, die zwischen zwei Wissensquellen in den Datenwerten auftauchen, z.B. wenn zwei Wissensquellen bei der gleichen Anfrage auch ohne semantische Konflikte verschiedene Antworten liefern, wenn sie sich also widersprechen. Diese Konflikte können *nur vom Mediator* gelöst werden, nicht vom Übersetzer, da letzterer immer nur *eine* Wissensquelle „sieht“, also insbesondere keine Inter-Wissensquellen-Konflikte lösen kann. Diese Konflikte sind in der Regel nicht einfach zu lösen. Der Einsatz von KI-Techniken scheint an dieser Stelle häufig angebracht.

Diese Einteilung der Konflikte unterstützt zusätzlich die Trennung von Schema- und Modellübersetzung, so daß nun vier Instanzen entstanden sind; neben diesen beiden noch der Mediator selbst und natürlich die Wissensquelle. Die beiden Übersetzerteile werden nun beleuchtet.

### 2.3.1 Schemaübersetzer

Obschon die Schemaübersetzung nicht das eigentliche Thema dieser Arbeit ist, wurde ihr Wesen erst durch die Abgrenzung zur hier im Mittelpunkt stehenden Modellübersetzung deutlich.

Die Schemaübersetzung oder -integration findet in KOMET mittels Sichtenklauseln im Mediatorcode statt. Die Erstellung dieser Klauseln wird z.B. in [Wei96] behandelt. Eine völlig automatische Erzeugung solcher Klauseln wird wohl nur mit einem semantisch deutlich reicheren Exportschema denkbar, als dem in KOMET benutzten. Die Art der Erzeugung hängt auch vom Schwerpunkt des Mediators ab, also davon, ob er mehr als Nutzer oder als Integrator von Wissensquellen auftaucht. Im ersten Fall wird man von „oben“ (top-down), also vom Mediatorschema ausgehen, da dieses vorgegeben ist, im zweiten Fall von „unten“ (bottom-up), d.h. von den Wissensquellen, da nur deren Schemata vorgegeben sind.

#### Top-Down-Schemaintegration

Ist die Aufgabe des Mediators, also das zu lösende Problem, vorgegeben, und werden die Wissensquellen nur ge*nutzt*, um dieses Problem zu lösen, so hat man die Schemata der Wissensquellen *und* das des Mediators geben und versucht sie zu verbinden. Die Schemaintegration würde in diesem Fall immer einzeln Mediator $\leftrightarrow$ Wissensquelle vorgenommen. In den seltensten Fällen müßte das gesamte Schema einer Wissensquelle herangezogen werden. Welcher Teil wichtig ist, gibt ja schon das Mediatorschema vor. Je kleiner und spezialisierter dieses ist, desto eher wird man Top-Down vorgehen. Eine solche Vorgabe ist aber nicht immer möglich, zumal bei der Absicht des reinen Nutzens einer Wissensquelle mehr die prinzipielle Problemlösungskompetenz der Wissensquelle bekannt ist, nicht die konkrete Ausprägung des anwendbaren Schemas. Ein nachträgliches Hinzufügen einer Wissensquelle ist hier ohne Neubetrachtung der anderen Wissensquellen möglich.

#### Bottom-Up-Schemaintegration

Ist es jedoch Aufgabe des Mediators, die Wissensquellen zu *integrieren*, so würde direkt aus den einzelnen vorgegebenen lokalen Schemata das föderative Mediatorschema entstehen. Der Konstruktionsprozeß der Sichtenklauseln ist also ein anderer als bei der Top-Down-Integration. In diesem Fall würden die gesamten lokalen Schemata integriert, und zwar jede Wissensquelle mit jeder anderen. Eine solche vollständige Schemaintegration ist meistens ein umfangreiches Unterfangen. Sie ist auch viel schwieriger speziell für eine gewünschte Inferenz auf Mediatorebene herzustellen als eine anwendungsgeleitete, teilweise Top-Down-Integration [vHPS94].

In der Realität wird für die Schemaintegration eine Zwischenform gewählt werden müssen, wobei die Top-Down- und Bottom-Up-Integration als Extremfälle gelten. Je kleiner und früher bekannt die Mediatordomäne ist, desto eher bietet sich ein Top-Down-Vorgehen an, je umfassender der Mediator die Quellen integriert, desto eher ist das Bottom-Up-Vorgehen geeignet.

### 2.3.2 Modellübersetzer

Die Modellübersetzung ist das Thema dieser Arbeit. Auf eine autonome, unveränderliche Wissensquelle, wie sie im Rahmen von KOMET vorausgesetzt wird, ist es nur möglich, mittels der von ihr gebotenen Anfrageschnittstelle zuzugreifen[1]. Ein Modellübersetzer ist also ein Anfrage- und ein Antwortenübersetzer. Eine Antwortenübersetzung erschöpft sich meist in Daten(format)konvertierungen. Eine Anfrageübersetzung ist wesentlich umfangreicher. Auf sie wird ab Kapitel 3 eingegangen.

[1] Die Möglichkeit, auf Wissensquellen auch schreibend zuzugreifen wird im Rahmen von KOMET noch nicht betrachtet.

## 2.4 Wissensquelle

Für den Mediator sind Wissensquellen Instanzen, die Constraint-Relationen auswerten können. Diese Constraint-Relationen sind ebenfalls im relationalen Mediatormodell definiert. Der Mediator sieht eine Wissensquelle demnach als eine Erweiterung seiner Wissensbasis an. Eine Wissensquelle muß dementsprechend vom Übersetzer aufbereitet werden. Der Mediator schickt eine Konjunktion von Constraint-Ausdrücken zur Wissensquelle. Ein solches „Schicken“ entspricht einer Anfrage an eine Wissenquelle.

Eine Wissensquelle kann dementsprechend alles sein, was eine Relation oder eine Konjunktion von Relationen berechnen kann. Darunter fällt vieles, wie z.B. Datenbanken, Informationssysteme, Expertensysteme, Funktionsbibliotheken, aber auch andere Mediatoren oder sogar der menschliche Benutzer.

Für den Mediator muß spezifiziert werden, welche Constraint-Relationen eine Wissensquelle evaluieren kann. Er muß neben den Namen der Relationen auch die einzelnen Attributtypen kennen. Dies ist im Exportschema definiert. Um eine Wissensquelle adäquat ansprechen zu können, also eine Anfrageübersetzung korrekt bewerkstelligen zu können, bedarf es eventuell noch weiterer Informationen, die der Mediator nicht zwingend nutzen wird. So gibt es bei vielen Wissensquellen für jedes Attribut einer Relation einen eigenen Namen. Zur optimierten Anfrageübersetzung ist eventuell auch die Größe einer jeden Relation interessant. Diese Zusatzinformation ist ebenfalls im Exportschema zu finden.

Da der Mediator die Wissensquelle als einen „Anbieter“ von (Constraint-) Relationen sieht, muß das lokale Modell in das relationale Modell übersetzt werden. Die Möglichkeit einer solchen Übersetzung ist eine notwendige Vorbedingung, die bei stärkeren (z.B. objektorientierten) oder schwächeren (z.B. Textdateien) lokalen Modellen sorgfältig zu prüfen ist. An zwei Beispielen wird eine solche problematische Modellkonvertierung vorgeführt, jedoch beinhaltet diese Arbeit keine Anleitung für ein allgemeines Vorgehen, wie man dieses Problem löst.

# Kapitel 3

# Anfrageübersetzung

*Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity, forced without rhyme or reason by many human institutions and systems to which his interfaces must conform.*

F. P. Brooks in „No Silver Bullet“, IEEE Computer, April 1987

In diesem Kapitel wird der neue Ansatz zur Anfrageübersetzung erläutert. Nach einer einleitenden Darlegung der Grundidee, eine Anfrageübersetzung mit Techniken des Compilerbaus anzugehen, werden in einem „didaktischen Teil“ nacheinander anhand komplexer werdender Beispiele die notwendigen Konstrukte vorgestellt, die dann im „formalen Teil“ präziser und auf Basis der relationalen Algebra definiert werden. Im letzten Abschnitt, dem „Anwendungsteil“ werden vier typische Wissensquellenarten, namentlich eine relationale Datenbank, ein Computeralgebrasystem, eine objektorientierte Datenbank und eine HTML-Seite mit bekannter Struktur, herangezogen, um die Nützlichkeit dieses Ansatzes zu prüfen.

## 3.1 Grundlegendes Konzept

Anfragen an eine Wissensquelle beschreiben, wie die angefragten Daten aussehen (deskriptive Anfrage) oder wie die Wissensquelle an die angefragten Daten gelangt (imperative Anfrage). In beiden Fällen wird der Wissensquelle eine Beschreibung gegeben. Der Mediator liefert eine Anfrage an den Übersetzer, jedoch in einer anderen Beschreibungssprache. Diese ineinander zu übersetzen ist eine ähnliche Aufgabe, wie das Übersetzen einer höheren Programmiersprache in eine maschinennahe. Daher bietet es sich an, Compilerbau-Techniken für die Anfrageübersetzung zu nutzen. Im folgenden kommt daher zunächst ein kurzer Überblick dieser Techniken. Eine kurze, aber brauchbare Zusammenfassung steht in [Vie89]. Ausführlicheres bietet [ASU86].

Die meistens angewendete Technik ist die syntaxgesteuerte Übersetzung. Dort wird die Übersetzung von der Syntax der Quellsprache gesteuert, welche als kontextfreie Grammatik gegeben ist. Jede Produktionsregel der Grammatik enthält eine sogenannte semantische Regel

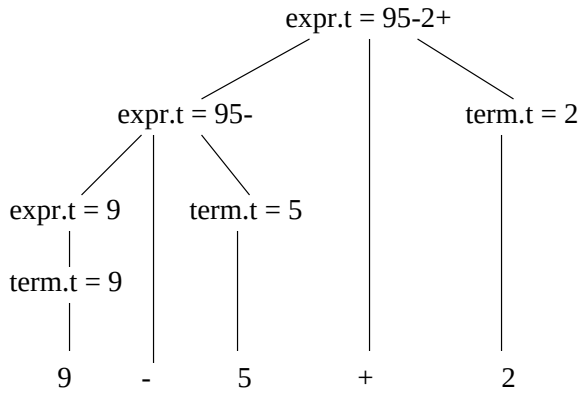



Abbildung 3.1: Beispiel zur syntaxgesteuerten Übersetzung

(attributierte Grammatik). Der zu übersetzende Quellstring wird geparst, also ein Parsebaum erstellt, welcher dann in Tiefensuche durchlaufen und an jedem Knoten die entsprechende semantische Aktion ausgeführt wird. Diese Aktionen bestehen in ihrer Reinform (gemäß [Knu68]) aus Zuweisungen an synthetisierte Attribute. Dies sind String-Variablen, die an Nichtterminale geknüpft sind. Der Wert eines Attributs an einem Knoten hängt von den Werten der Attribute an den darunter liegenden Knoten ab, genauso wie eben ein Nichtterminal in einer Produktion eine Reihe neuer Nichtterminale erzeugen kann. Der Wert eines bestimmten Attributs in der Wurzel ist das Übersetzungsergebnis. In folgenden wird ein Beispiel zur Übersetzung eines Infix- in einen Postfixausdruck gezeigt. In geschweiften Klammern stehen die semantischen Aktionen. || ist das Zeichen für die Konkatenation zweier Zeichenketten. Indizes an den Nichtterminalen besorgen eine Numerierung, auf die in den Aktionen Bezug genommen wird. $expr_1.t$ bezeichnet z.B. das Attribut $t$ des ersten $expr$-Symbols. Der Ablauf einer Übersetzung wird in Abbildung 3.1 gezeigt. Das Beispiel stammt aus [ASU86].

$$
\begin{aligned}
expr_1 &\to expr_2 + term && \{expr_1.t := expr_2.t \mid\mid term.t \mid\mid '+'\} \\
expr_1 &\to expr_2 - term && \{expr_1.t := expr_2.t \mid\mid term.t \mid\mid '-'\} \\
expr &\to term && \{expr.t := term.t\} \\
term &\to 0 && \{term.t :=' 0'\} \\
&\dots \\
term &\to 9 && \{term.t :=' 9'\}
\end{aligned}
$$

Am Beispiel lassen sich die semantischen Regeln auch einfacher als ein Emittieren von Strings realisieren. Wird z.B. die Regel

$$expr \to expr + term$$

angewendet, so genügt die semantische Aktion

$$\{emit('+')\}.$$

[Vie89] enthält noch einige Ideen in diese Richtung. Für komplexere Übersetzungen ist es eventuell noch notwendig, ererbte Attribute zu nutzen, deren Wert abhängig von Vaterknoten ist. Darauf wird hier nicht eingegangen. Für näheres siehe man [ASU86].

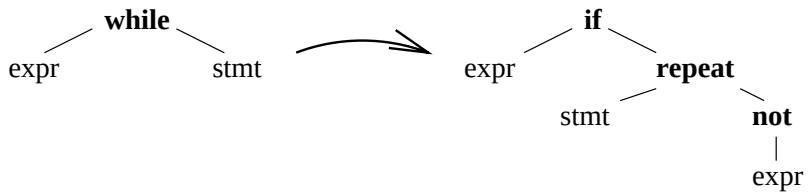



Abbildung 3.2: Beispiel zur Baumtransformation

Eine andere Art der Übersetzung ist die Baumtransformation. Wieder ist der Parsebaum gegeben, diesmal wird er aber nicht durchlaufen und dabei Attribute errechnet oder Strings emittiert, sondern der Baum selbst wird mittels parametrisierter Regeln so umgeformt, daß der entstehende Baum passend zur Zielsprache ist. Eine Regel könnte z.B. sein, eine while-Anweisung in eine repeat-Anweisung zu übersetzen, wie Abbildung 3.2 zeigt. (Das Beispiel stammt aus [Vie89].) Solche Transformationen werden häufig als Vortransformationen benutzt, d.h. der Quell-Parsebaum wird zur Anwendung einer syntaxgesteuerten Übersetzung geeignet umgeformt. Als alleinige Übersetzungstechnik ist sie zu umständlich. Das hat sich auch bei den Versuchen im Rahmen dieser Arbeit gezeigt. Abschnitt 3.2.10 geht auf diesen Punkt ein.

Zur Anfrageübersetzung wird in dieser Arbeit ebenfalls eine Kombination aus Transformation und semantischen Aktionen benutzt. Wegen der Eigenarten von Anfragen, die eine wesentlich andere, einheitlichere Form haben als Ausdrücke einer höheren Programmiersprache, werden *keine* fertigen Compiler-Compiler genutzt (z.B. Yacc), da diese meist spezialisiert sind auf Programmiersprachenübersetzungen. Eine eigene Entwicklung bietet außerdem die Möglichkeit der effizienteren und spezifischeren Übersetzung. Dies fängt schon bei der Repräsentation der Anfrage an. Als Basis zur Übersetzung wird kein Anfragestring herangezogen, sondern eine spezielle graphische Kastendarstellung der Anfrage, die unter anderem in den folgenden Abschnitten vorgestellt wird.

## 3.2 Konstrukte

In diesem Abschnitt werden Schritt für Schritt anhand einer Reihe von Anforderungen und Problemen die dem Übersetzungsvorgang zugrunde liegenden Konstrukte erläutert. Am Ende ist dann der gesamte Ansatz vorgestellt und die Verständnisgrundlage für den Rest der Ausarbeitung gelegt. Hier sollen nur die Ideen vorgestellt werden, ohne zuviele formale Definitionen. Diese werden erst im anschließenden Kapitel geliefert.

### 3.2.1 Kastendarstellung einer konjunktiven Anfrage

Übersetzt werden immer konjunktive Anfragen, nicht einzelne Relationen. Ein konkretes Beispiel ist

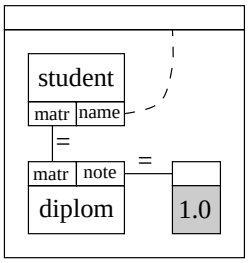
$$ans(Name) \quad \leftarrow \quad student(Matrnr, Name) \ \& \ diplom(Matrnr, 1.0). \tag{3.1}$$

Hier werden die Namen aller Studenten angefragt, die eine 1.0 im Diplom haben. *ans* steht für „answer“. Die Verbindung zwischen den Relationen *student* und *diplom* findet über die

Matrikelnummer statt. Im folgenden werden Relationennamen immer klein und Variablen immer groß geschrieben. Man kann eine solche Anfrage äquivalent umformen in eine Anfrage, bei der jede Relation ihre eigenen Variablen hat und die Gleichheit von Variablen explizit dargestellt wird. Wenn man hinzu noch Grundterme wie die 1.0 im Beispiel als einelementige Relation '1.0', d.h. als die Menge $\{\langle 1.0\rangle\}$, repräsentiert, läßt sich Anfrage 3.1 auch wie folgt ausdrücken:

$$\begin{aligned} ans(Name) \quad \leftarrow \quad & student(Matrnr1, Name)\ \&\ diplom(Matrnr2, Note1)\ \& \\ & '1.0'(Note2)\ \&\ Matrnr1 = Matrnr2\ \&\ Note1 = Note2. \end{aligned}$$

Die Kastendarstellung dieses Beispiels ist folgende:



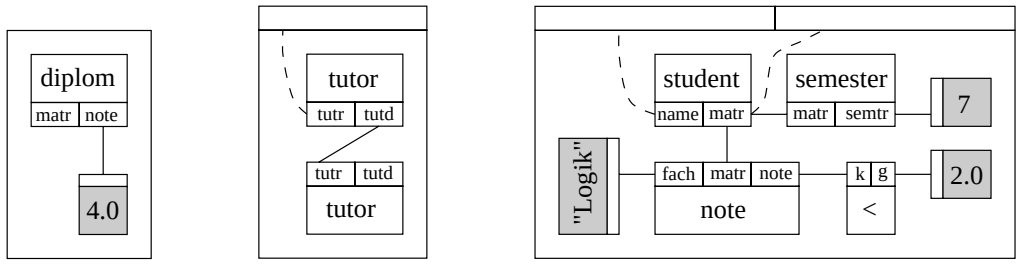

Dabei repräsentiert ein **Kasten** (Rechteck) eine Relation. Ein Kasten hat für jede der Attribute der Relation eine **Stelle** (kleine Rechtecke an den großen). Die Stellen sind untereinander mit Kanten verbunden, wenn in der konjunktiven Anfrage die Variablen, die diese beiden Stellen darstellen, gleichgesetzt sind. Die Kanten sind entsprechend mit einem '=' markiert. Der Kasten der Relation '1.0' ist dunkel schattiert dargestellt, da dies eine Relation ist, die nicht die Wissensquelle gespeichert hat, sondern der Übersetzer direkt kennt. Diese Art von Kasten wird **M-Kasten** genannt. M steht für „materialisiert". Die anderen Kästen werden im Vergleich dazu auch **N-Kästen** genannt; N für „namentlich". Die ganze Darstellung selbst ist auch ein Kasten. Da dieser Kasten mehrere Kästen enthält, wird er **K-Kasten** genannt; K steht für „komplex". Ein K-Kasten enthält Kästen mit ihren Stellen und eine Menge von markierten Kanten. Von der Stelle „name" des „student"-Kastens geht eine gestrichelte Kante von innen an die Stelle des K-Kasten, der die ganze Anfrage, also die Relation *ans*, repräsentiert. Dies bedeutet, daß die entsprechende Stelle bzw. Variable im Kopf der Anfrageklausel auftaucht, also projiziert wird. Die gestrichelten Kanten bezeichnen also Projektionen und die =-Kanten Verbunde. Dies ist kurz gesagt die Semantik der Kastendarstellung. Eine Anfrage wird also repräsentiert als die (K-)Kastendarstellung der Ergebnisrelation *ans*. Genauer wird dies in Abschnitt 3.3 ausgeführt.

Nachfolgend kommen noch drei Beispiele mit ihren Kastendarstellungen. Nicht markierte Kanten sind =-Kanten. Man beachte, daß es keine Beziehung zwischen den hier verwendeten Variablennamen und den Namen der Stellen gibt. Die Variablen in den Klauseln sind nur geeignet gewählt, die Namen der Kästen und Stellen sind aber jene, die auch die Wissensquelle benutzt.

$$ans() \quad \leftarrow \quad diplom(X, 4.0). \tag{3.2}$$

$$ans(Tutor) \quad \leftarrow \quad tutor(Tutor, Tutand)\ \&\ tutor(Tutand, X). \tag{3.3}$$

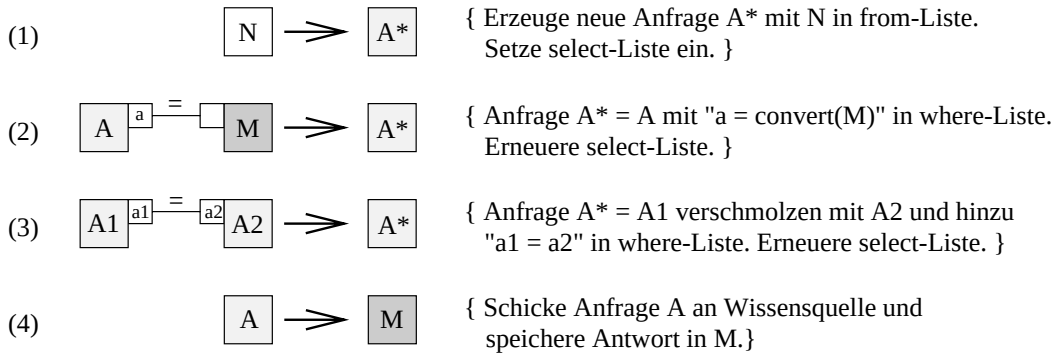
$$\begin{aligned} ans(Name, Matrnr) \quad \leftarrow \quad & student(Name, Matrnr)\ \&\ semester(Matrnr, 7)\ \& \\ & note(\text{“}Logik\text{“}, Matrnr, Note)\ \&\ <(Note, 2.0). \end{aligned} \tag{3.4}$$




### 3.2.2 Übersetzung einer Kastendarstellung

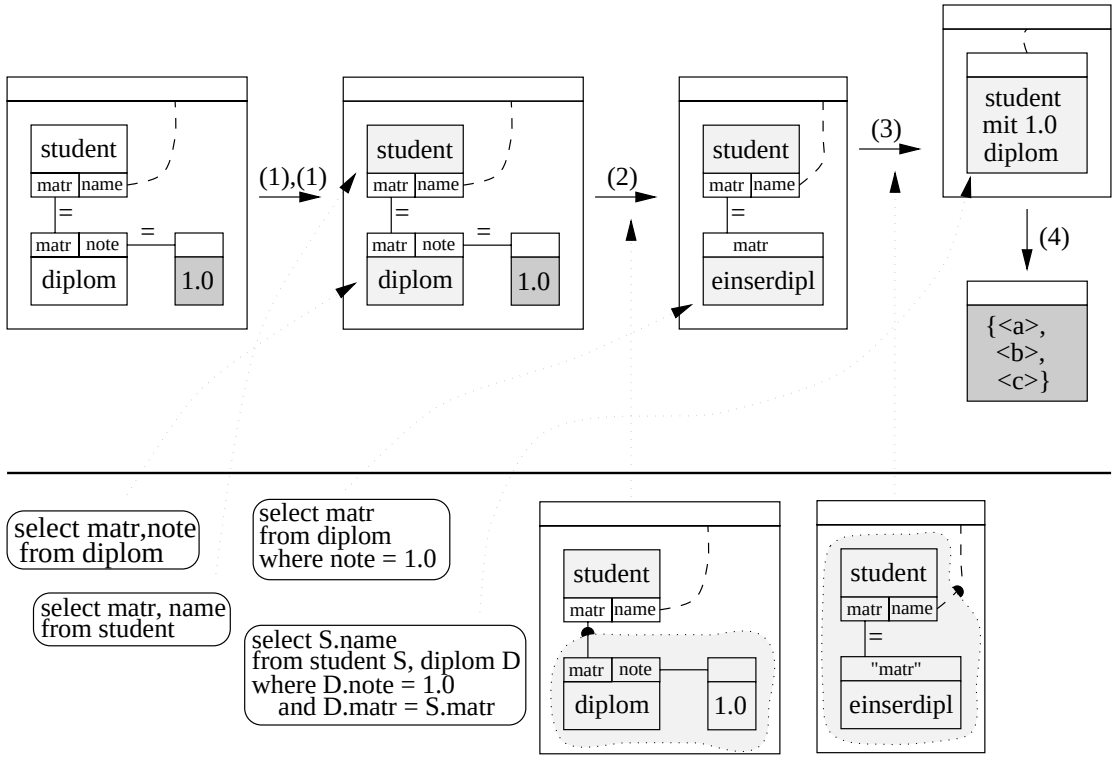
Am Beispiel der Übersetzung der Anfrage 3.1 in eine äquivalente SQL-Anfrage soll das Prinzip des Übersetzungsvorgangs erläutert werden. Zunächst wird jede vom Mediator kommende Anfrage in die Kastendarstellung konvertiert und nur noch mit dieser gearbeitet. Die Übersetzung wird gesteuert durch eine Reihe von Regeln, die entweder einen Kasten in einen anderen Kasten transformieren (**Kasten-Regeln**) oder eine Kante eliminieren, indem die durch sie verbundenen Kästen zu einem neuen Kasten verschmelzen (**Kanten-Regeln**). Das Ergebnis einer Regel ist also immer ein Kasten. Während der Übersetzung entsteht eine neue Kastenart, nämlich abgearbeitete Kästen oder **A-Kästen**, hell schattiert dargestellt. Pro Regel gibt es eine (semantische) Aktion, die ein beliebiges Programm ist, das alle interessanten Parameter der an der Regel beteiligten Konstrukte erhält. Diese Aktionen erzeugen Anfragestrukturen und werden als Attribut dem Ergebniskasten – meist einem A-Kasten – mitgegeben. Dies ist genau die Art von synthetisierten Attributen, die auch in der Compilertechnik benutzt werden. Die Regeln entsprechen einer rückwärts angewendeten Grammatik für die Kasten-Sprache.

Die zu erzeugende SQL-Anfrage soll möglichst allgemein gespeichert werden, so daß sie für verschiedene SQL-Schnittstellen geeignet ist. (Zu SQL siehe auch [LL95].) Das Grundgerüst einer SQL-Anfrage ist ein „select-from-where“-Konstrukt. Im folgenden wird daher so übersetzt, daß man als Ergebnis drei Listen bekommt, nämlich die entsprechenden select-, from- und where-Listen. Die semantischen Aktionen sind hier nur natürlichsprachlich beschrieben. Die Regeln sind die folgenden:



Die Regeln haben eine bestimmte Reihenfolge. Die oben beginnend erste anwendbare Regel wird angewendet. Die Übersetzung endet, wenn keine Regel mehr anwendbar ist.

Die Regeln wandeln die Kastendarstellung selbständig um, d.h. die neue Kastendarstellung wird unabhängig von den Aktionen erstellt. Die erste Regel heißt **Instanziierung**, die vierte Regel **Materialisierung**. Bei jeder Regel wird ein bestimmter Teil der Darstellung in einen neuen Kasten gewandelt. Welche Stellen dieser entstehende Kasten hat und welche neuen Kantenverbindungen entstehen, kann automatisch ermittelt werden, indem die nach außen zeigenden Kanten entsprechend eingerichtet werden. Die folgende Grafik verdeutlicht diese schrittweise Anwendung der Regeln. Dabei sind die Namen der Nicht-N-Kästen und -Stellen willkürlich gewählt.




Bei der Anwendung einer Regel wird der Teil der Darstellung, der die Prämisse[1] enthält, umschlossen. Die dabei geschnittenen Kanten bzw. die so noch als wichtig erkannten Stellen müssen beibehalten werden. Das ist nichts anderes als eine Projektion dieser Stellen (=Attribute). Bei einer Regelanwendung werden diese Stellen als Parameter für die Aktion übergeben. Genau diese Information nutzt die SQL-Übersetzung, um die select-Liste aufzubauen. Dies ist mit „Erneuere select-Liste“ in den Aktionen gemeint. Die Grafik verdeutlicht im unteren rechten Viertel dieses Vorgehen bei der Anwendung der Regeln (2) und (3). Ebenfalls gezeigt werden unten links die Zwischenergebnisse, die die semantischen Regeln als Attribute (Merkmale) in den A-Kästen erzeugt haben. Die genaue SQL-Übersetzung muß auf diverse andere Besonderheiten achten, z.B. können, wie in diesem Beispiel geschehen, zwei gleiche

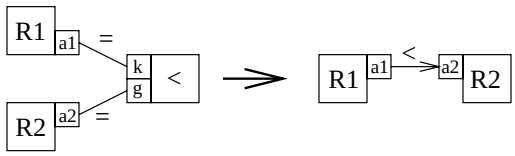
[1]Die **Prämisse** einer Regel ist der (immer links vom Pfeil dargestellte) Teil, der die Quelle der Regelanwendung angibt. Der andere Teil, also das was für die Prämisse eingesetzt werden soll, heißt **Conclusio**

Attributnamen oder wie in Anfragebeispiel 3.3 zwei gleiche Relationennamen auftauchen, die unterschieden werden müssen.

### 3.2.3 Transformation der Kastendarstellung

Im Anfragebeispiel 3.4 wird die Relation $<$ benutzt. Eine SQL-Datenbank kennt aber eine Relation mit diesem Namen nicht. Statt dessen muß diese Relation zu einer Ordnungsselektion werden. Das $<$ taucht also in einer SQL-Anfrage nicht in der from-Liste auf, sondern als Selektion in der where-Liste. Die bisherige Übersetzung steckt es aber fälschlicherweise in die from-Liste. Dies wird verhindert, wenn die ursprüngliche Kastendarstellung noch mittels Transformationsregeln ohne semantische Aktionen verändert werden kann. Diese Regeln werden noch *vor* der eigentlichen Übersetzung angewendet. Sie entsprechen den in Abschnitt 3.1 erwähnten Baumtransformationsregeln. Man bemerke aber, daß die Kastendarstellung keine Baumform hat, sondern ein Graph ist.

Im vorliegenden Fall 3.4 wäre folgende Regel die Lösung:




Die Regel wandelt alle $<$-Relationen in eine $<$-Kante. Diese Kante hat also *keine* $=$-Markierung. Daher müssen auch bei den Kantenregeln entsprechende $<$-Kanten auftauchen. Man kann den Typ der Kante in einer Kantenregel variabel lassen und diesen der Aktion als weiteren Parameter übergeben. Außerdem ist diese Kante wegen der unsymmetrischen $<$-Relation gerichtet. Im allgemeinen ist jede Kante gerichtet.

Natürlich ist diese Transformation auch für andere Vergleiche außer $<$ sinnvoll, z.B. $\leq$,$\neq$,$>$ und so weiter. Auch komplexere Vergleiche, etwa bei der Textsuche, sind denkbar. Die Gleichheit $=$ bekommt ihren Sonderstatus aufgrund der Tatsache, daß sie als einzige essentiell für konjunktive Anfragen ist, wie sie vom Mediator kommen.

### 3.2.4 Cursor

Regel (3) bei der vorgestellten Übersetzung in SQL ist durchführbar, weil relationale Datenbanken auch tatsächlich mehrere Relationen auf einmal in Form einer konjunktiven Anfrage bearbeiten können. Dies ist nicht bei allen Wissensquellen der Fall. Man nehme z.B. ein Recherchesystem in einer Bibliothek. Dort steht in der Regel eine Relation „Buch“ mit vielen Attributen zur Verfügung. Man kann sich zwar alle Bücher herausgeben lassen, die bestimmte einfache Attribut-Eigenschaften haben, aber nicht z.B. alle Bücher deren Autorenname der Titel von einem Buch ist. Diese Anfrage würde bei der vereinfachten Relation $buch : Autor \times Titel$ wie folgt aussehen:

$$ans(Autor, Titel) \quad \leftarrow \quad buch(Autor, Titel) \;\&\; buch(X, Autor). \tag{3.5}$$

Das Problem ist, daß eine solche Anfrage nicht mit einer einmaligen Angabe der „Buch“-Relation formulierbar ist, wie es das Bibliotheksystem verlangt. Der Benutzer müßte statt

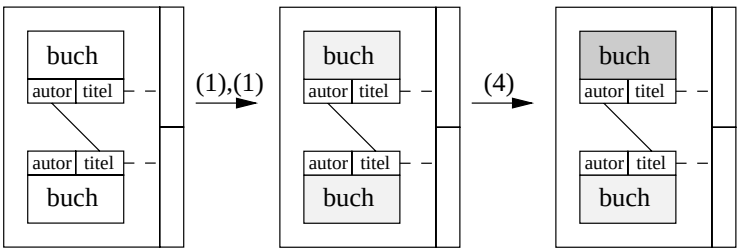
dessen umständlich Einträge in riesigen Listen vergleichen. Das Vorgehen an sich ist aber nicht kompliziert, und es wäre nicht schwer, diese Anfrage anhand der Datenbasis des Informationssystems zu beantworten. Der Übersetzer sollte diese Arbeit übernehmen.

Die Wissensquelle kann also keine zwei Relationen, und sei es auch ein und dieselbe, miteinander verbinden. Aus diesem Grund kann Regel (3) aus der Übersetzung in SQL nicht von der Wissensquelle bearbeitet werden. Taucht ein Verbund zwischen zwei Relationen auf, so muß der Übersetzer dies erledigen. In diesem Ansatz ist das Problem schon damit gelöst, daß Regel (3) einfach weggelassen wird! Es wird also, wenn die Regeln (1) und (2) nicht mehr anwendbar sind, sofort Regel (4) angewendet, das heißt, die erste bislang konstruierte Anfrage wird abgeschickt. Der entstehende M-Kasten ist aber nicht das Endergebnis, denn es ist danach wieder Regel (2) anwendbar. Man beachte, daß der so entstandene M-Kasten aus mehr als nur einer Zahl besteht.

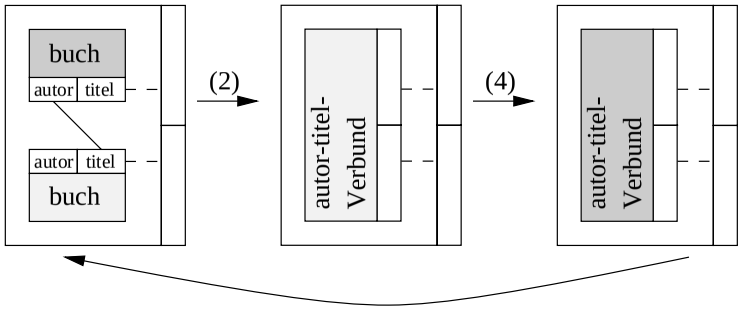
Regel (3) ist bei einer Wissensquelle grundsätzlich anwendbar oder niemals anwendbar. Die Implementation, die in Abschnitt 4 beschrieben wird, weicht von dieser Idee ab und erlaubt es, eine Regel auch abhängig von den beteiligten Kästen abzulehen oder auszuführen.

Um einen M-Kasten, der nicht nur aus einer Konstante besteht, auszulesen, muß es eine einheitliche Schnittstelle zu ihnen geben. So wie ein A-Kasten eine Anfragestruktur speichert (als Attribut hat), so hat ein M-Kasten einen **Cursor** als Attribut. Ein M-Kasten steht wie schon erwähnt für eine materialisierte Relation, d.h. eine Relation, deren Tupel auslesbar sind, so als seien sie eines nach dem anderen gespeichert. Ein Cursor ist ein Zeiger auf die Tupel in einem M-Kasten. Es ist möglich, den Tupelwert, auf den der Cursor zeigt, zu lesen, den Cursor auf das nächste Tupel zu setzen und zu testen, ob es überhaupt noch ein nächstes Tupel gibt. Auf diese Weise ist es möglich, alle Tupel eines M-Kastens Stück für Stück zu erhalten. Bei relationalen Datenbanken ist dieses Cursorkonzept bekannt; für andere Wissensquellen muß es nachgebildet werden. Es ist allgemein genug, daß es bei jeder Wissensquelle anwendbar sein sollte.

Wie bearbeitet der Übersetzer nun diese Anfrage? Im Beispiel 3.5 ist nach erstmaliger Anwendung der Regel (4) folgendes passiert:




Man kann nun Regel (2) anwenden, hat hier aber – im Gegensatz zu den bisherigen Fällen – im M-Kasten nicht nur einen Wert (Konstante aus der Anfrage) sondern eine ganze Reihe von Tupeln. Die Semantik der Anfrage verlangt, daß Regel (2) nacheinander mit allen Werten des M-Kastens angewendet werden muß. Man hat also zunächst folgende Regelanwendungen:




Der Cursor des nun in zweiter Ebene entstandenen Ergebnis-M-Kastens muß vom Mediator ausgelesen werden. Ist der Cursor am Ende der Tupelliste, so heißt es in diesem Fall aber *nicht*, daß alle Antworttupel gefunden wurden. Der M-Kasten muß – da er auf einem anderen M-Kasten basiert – nochmal in diejenigen Kästen zurückgewandelt werden, in dem der Cursorwert (in der bislang *Convert* genannten Routine) ausgelesen wurde. Hier nun wird der nächste Cursorwert geholt und der äußere M-Kasten erneut erzeugt, also eine weitere Anfrage abgeschickt, diesmal aber mit anderen Konstanten. Die Wissensquellenantwort darauf repräsentiert wiederum einen Cursor, den der Mediator ausliest. Dieser Zyklus wird rekursiv solange wiederholt, bis alle Cursor erschöpfend durchiteriert wurden. Erst dann ist die Ergebnisrelation vollständig evaluiert worden.

Das Konzept des Cursors kann man auch im Zusammenhang mit dem Mediator anwenden. Daß in einer Anfrage eine Konstante auftaucht kann zwei Ursachen haben:

1. Die Konstante steht direkt in der Mediator-Klausel, aus der die konjunktive Anfrage extrahiert wurde. In diesem Fall hat man in der Tat nur einen unveränderlichen Wert vor sich.

2. Die Konstante entstand inmitten der Inferenzarbeit des Mediators, bei der es nun erforderlich ist, die Anfrage an die Wissensquelle zu stellen; es wurde also eine Variable in der Klausel vom Mediator gebunden.

Das Cursorkonzept ist zentral für die Kommunikation zwischen Mediator und Übersetzer. Weitere Einzelheiten stehen in Kapitel 4.

### 3.2.5 Disjunktion

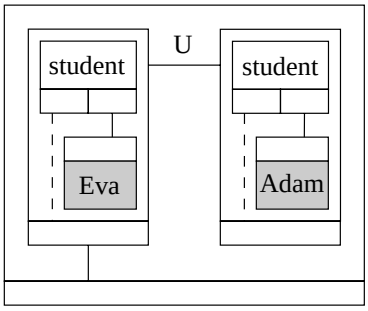
Die Unterstützung der Disjunktion in regelbasierten Sprachen, wie sie dem Mediator in KOMET zugrunde liegt, ist gering. Um z.B. die Matrikelnummern aller Studenten zu erhalten, die „Eva“ oder „Adam“ heißen, fragt man wie folgt an:

$$\begin{aligned} ans(Matrnr) &\leftarrow student(Matrnr,'Eva'). \\ ans(Matrnr) &\leftarrow student(Matrnr,'Adam'). \end{aligned}$$

Die Disjunktion wird also mit Hilfe *zweier* Klauseln formuliert. Der Übersetzer erwartet vom Mediator Disjunktionen von Relationen oder ganzen konjunktiven Anfragen, im Beispiel also die Form

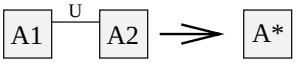
$$ans(Matrnr) \leftarrow student(Matrnr,'Eva') \text{ or } student(Matrnr,'Adam'). \qquad (3.6)$$

Es bietet sich an, die Disjunktion als Vereinigung von Relationen zu realisieren. Dies führt einen neuen Kantentyp zwischen zwei *Kästen* (nicht mehr zwischen Stellen) ein. Anfrage 3.6 sieht in Kastendarstellung wie folgt aus, wobei U für „union“ steht:




Man bemerke, daß die Einkapselung jedes Disjunktionsglieds in einen K-Kasten notwendig ist, da ansonsten die Reihenfolge der Kanten nicht bedacht würde. Dies ist der erste Fall, in dem ein K-Kasten im Anfrage-K-Kasten auftaucht. Bei der Abarbeitung der Kanten innerhalb eines K-Kastens ist dabei so vorzugehen, daß zunächst alle K-Kästen übersetzt (allerdings nicht materialisiert) und erst dann die Kanten abgearbeitet werden.

Mit dieser neuen Kantenart gibt es auch eine neue Regel. In der SQL-Übersetzung kommt etwa folgende Regel hinzu:




Aus der Anfrage 3.6 ergibt sich damit der SQL-Befehl:

```
(select matrnr                  (select matrnr
from student          union     from student
where name = 'Eva')             where name = 'Adam');
```

Das Ergebnis ist korrekt, aber nicht optimal. Wenn man die Aktion der Regel so ändert, daß bei gleichen from- und select-Listen der Anfragen A1 und A2 die beiden where-Listen disjunktiv verknüpft werden, bekommt man die bessere Übersetzung in

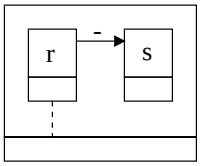
```
select matrnr
from student
where (name = 'Eva') or (name = 'Adam');
```

### 3.2.6 Negation

Im Gegensatz zur Disjunktion ist die Negation in KOMET eine essentielle Operation. Sie wird in der Kastendarstellung mittels der Differenz zweier Relationen dargestellt. Eine Anfrage wie

$$ans(X) \quad \leftarrow \quad r(X) \;\&\; \neg s(X). \tag{3.7}$$

wird zu

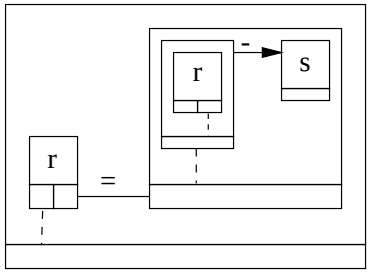



Die Kante muß gerichtet sein. „–“ bezeichnet die Mengendifferenz.

Dies war ein einfacher Fall. Die Übersetzung der Anfrage

$$ans(X) \quad \leftarrow \quad r(X,Y) \ \& \ \neg s(Y). \tag{3.8}$$

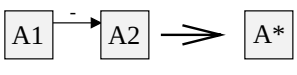
ist nicht so leicht zu bewerkstelligen, da sie keine direkte Differenz zwischen $r$ und $s$ ausdrückt. $\neg s(Y)$ ist die Relation, die alle Tupel enthält, außer denen, die in $s$ sind. Wenn $DOM_s$ die Relation ist, die aus allen Tupeln besteht, die dieselbe Struktur wie die in $s$ haben (also dieselben Attributsorten) wie die in $s$, dann bedeutet $\neg s(Y)$ nichts anderes als $DOM_s - s$. $DOM_s$ selbst ist aber nicht gegeben, oder zumindest sehr groß. Daher wird im folgenden bei Anfragen mit Negationen gefordert, daß sie *sicher* sind. Bei sicheren Anfragen taucht jede Variable, die in einem negierten Ausdruck steht, auch bei einem positiven Ausdruck auf. Diese Eigenschaft sichert, daß man nicht mit $DOM_s$ arbeiten muß, sondern es mit Hilfe eben jener positiven Variablen einschränken kann. In Beispiel kann $Y$ nur die Werte annehmen, die die Relation $r$ in ihrem zweiten Attribut liefert. Die Übersetzung von 3.8 ist also:




Zu beachten ist, daß hier die beiden „r“-N-Kästen die gleichen sind, im Gegensatz etwa zu den N-Kästen „tutor“ in Beispiel 3.3. Die Kastendarstellung erzwingt aber eine getrennte Darstellung.

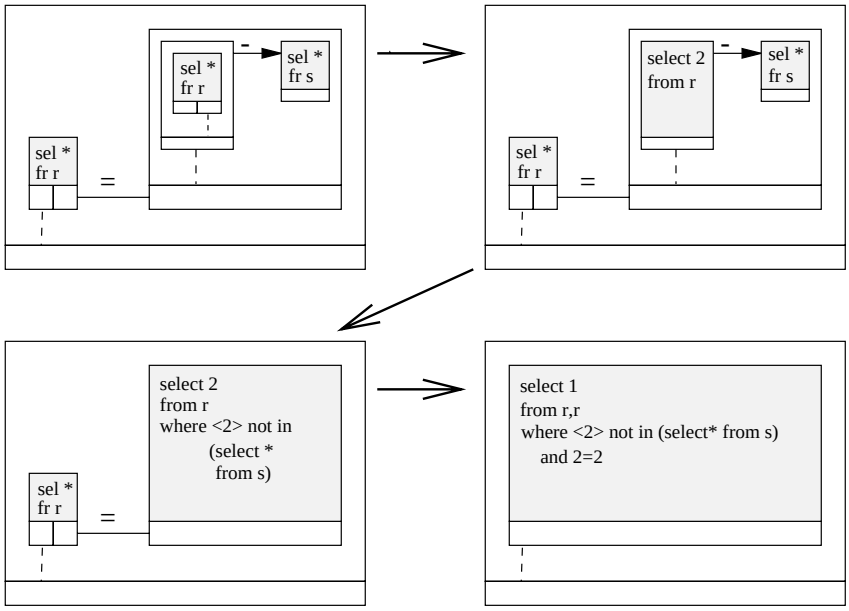
Diese auf dem ersten Blick umständliche Darstellung der Negation hat zwei Vorteile: Sie ist leicht und eindeutig aus der Anfrageklausel erzeugbar und sie spiegelt wieder, wie vorzugehen ist, wenn eine Wissensquelle die Negation *nicht* beherrscht, was häufig der Fall ist. In diesem Fall muß der Übersetzer wieder einspringen. Die so realisierte Darstellung läßt sich leicht implementieren.

In SQL ist die Negation mit dem „not in“-Konstrukt möglich. Die Aktion sieht wie folgt aus:




{ Anfrage A* = A1 mit der um
"<" || A1-selectlist || "> not in (" || A2-Anfrage || ")"
erweiterten where-Liste. }

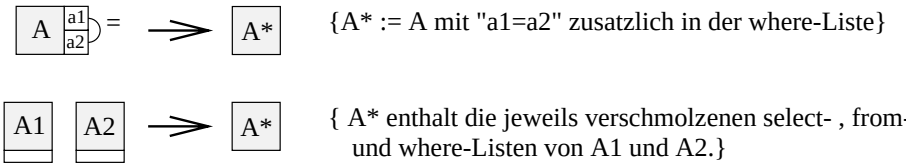
Eine Übersetzung läuft dann folgendermaßen ab:




Die entstandene SQL-Anfrage ist nicht recht befriedigend. Zum einen darf in der from-Liste das 'r' nur einmal auftreten, zum anderen steht in der where-Liste eine Tautologie; ebenfalls eine Folge des doppelten Auftretens des N-Kastens 'r'. Diese Probleme sind aber behebbar.

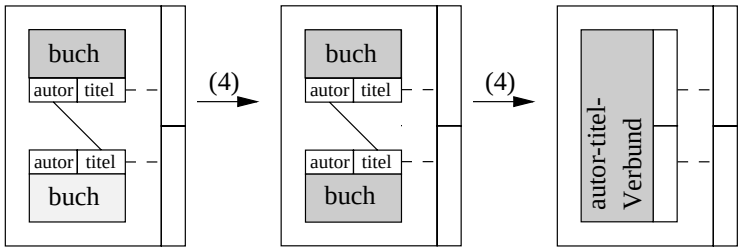
### 3.2.7 Weitere Regeln

Damit alle Anfragen und somit Kastendarstellungen mit den Regeln zu einem einzigen „Ziel"-Kasten (der materialisiert wird) umgeformt werden können, damit also die beschreibende Grammatik vollständig ist, fehlen noch zwei Regeln:

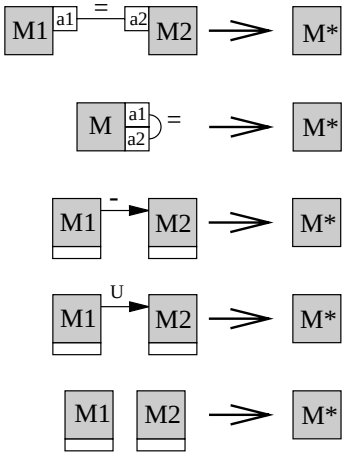



Die erste drückt eine Attributselektion innerhalb ein und desselben Kastens aus, die zweite das Kreuzprodukt zweier Kästen, die keine Kante mehr miteinander verbindet. Das Produkt eines A-Kastens mit einem M-Kasten ist nicht als Regel vorhanden. Die angegebenen Aktionen beziehen sich wieder auf die SQL-Übersetzung, die somit nun vollständig spezifiziert ist.

Die prinzipiellen Regelformen sind nun vorgestellt worden. Es fehlen allerdings noch Regeln, an denen nur M-Kästen beteiligt sind. Man stelle sich vor, in Beispiel 3.5 hätte die Wissensquelle nicht nur Regel (3), sondern auch Regel (2) nicht beherrscht, d.h. die entsprechende Aktion sei nicht vorhanden. Die angefangene Übersetzung würde dann wie folgt weitergehen:




Es gibt daher noch folgende Regeln:




Die semantischen Aktionen dieser Regeln können fest implementiert sein und sind für jede Wissensquelle gleich, da sie nur auf die Cursor der M-Kästen zugreifen. Diese haben aber eine einheitliche Schnittstelle. Bei der Entwicklung eines neuen Übersetzers kann man sich also zunächst auf das Implementieren der Aktionen der beiden Nicht-Kanten-Regeln Instanziierung und Materialisierung beschränken, die den Übergang eines N-Kastens zu einem A-Kasten bzw. eines A-Kastens zu einem M-Kasten realisieren. In der Tat *müssen* diese beiden Regelaktionen immer implementiert werden. Den Rest können aber die eben vorgestellten M-Kästen-Regeln bewerkstelligen. Dies ist im allgemeinen natürlich nicht effizient, da jede einzelne Relation materialisiert werden muß.

Es gibt also grundsätzlich mehrere Wege, wie eine Übersetzung vorangehen kann. Welcher Weg eingeschlagen wird, hängt von der Menge der implementierten Regelaktionen ab, also von der Mächtigkeit der Wissensquellen. Abbildung 3.3 zeigt für eine Kastendarstellung verschiedene Übersetzungspfade. Im allgemeinen sollte jener Weg genommen werden, der optimal bezüglich Antwortzeit und Resourcenverbrauch ist. In der noch zu erläuternden Implementation wurde die Reihenfolge der Regeln fest kodiert. Man könnte die Reihenfolge auch abhängig von der Wissensquelle machen oder sogar für jede Anfrage die Kosten aller Wege berechnen und den besten wählen.

### 3.2.8 Bindungsmuster

Relationale Datenbanken und Bibliotheksysteme haben die Eigenschaft, daß sie für eine Relation alle Bindungsmuster zulassen. Sie können bei der Relation $student(Matrnr, Name)$ zu einer Matrikelnummer den Namen des Studenten geben, zu einem Namen die passenden

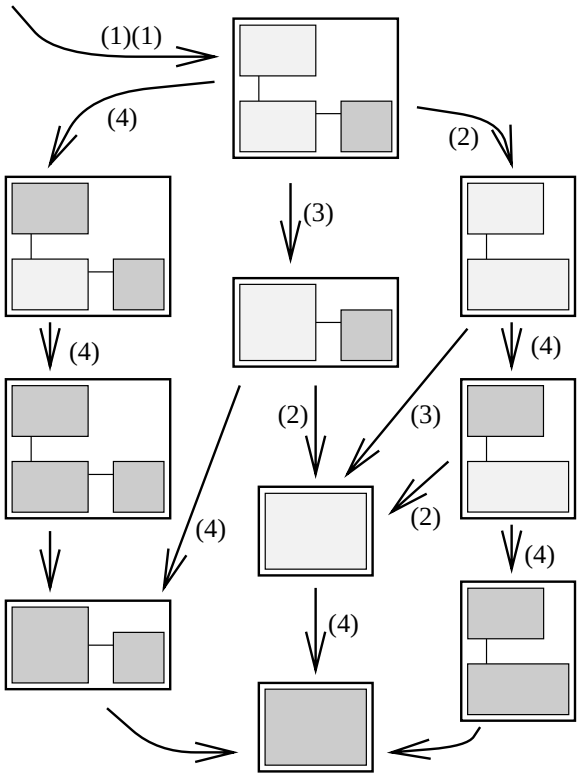



Abbildung 3.3: Verschiedene Übersetzungspfade

Matrikelnummern, auch testen, ob eine Kombination Name-Nummer vorkommt oder sogar alle überhaupt existierenden Paare ausgeben. Dies ist nicht bei allen Wissensquellen der Fall. Ein Taschenrechner z.B. kann nur *Funktionen* auswerten, d.h. alle Werte des Definitionsbereichs müssen bei der Anfrage angegeben werden, die Attribute des Wertebereichs müssen hingegen frei sein. So sind z.B. bei der Relation $hoch(Res, Basis, Exp)$ nur Anfragen der Art $ans(Res) \leftarrow hoch(Res, 2, 3)$ möglich und sinnvoll, $ans(Exp) \leftarrow hoch(8, 2, Exp)$ jedoch nicht, wenn es auch wünschenswert – und mit einer Relation *log* sogar machbar – wäre.

Aus diesem Grund werden den Relationen im Exportschema noch Informationen über die erlaubten Bindungsmuster beigegeben. Das Exportschema einer Datenbank hat für *student* also das Muster $(*, *)$, der Taschenrechner für *hoch* nur das Muster $(f, g, g)$. Eine Relation kann mehrere erlaubte Muster haben. In der Tat ist ja $(*, *)$ eine Abkürzung für alle vier Kombinationsmöglichkeiten.

Die Überprüfung, ob eine vom Mediator kommende Anfrage moduskonform bzw. konfliktfrei ist, d.h. ob im Exportschema für jede Relation in der konjunktiven Anfrage die geforderten Bindungsmuster stehen, entscheidet in KOMET nicht der Mediator, sondern der Übersetzer, denn manche Bindungskonflikte sind lösbar, indem ein Anfrageverwalter Teile der Anfrage geeignet zusammenfügt. Diese Verwaltung ist aber einfacher vom Übersetzer als Wissensquellen-naher Prozeß erledigbar.

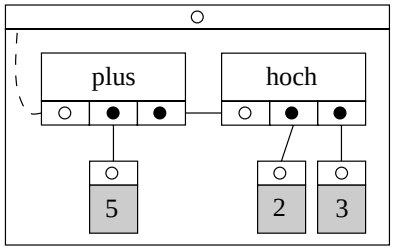
Die Bindungskonfliktlösung wird im nächsten Abschnitt erläutert. Hier wird zunächst darauf eingegangen, wie die Bindungsmuster in der Kastendarstellung repräsentiert sind und wie

eine Anfrageübersetzung z.B. an einen Taschenrechner abläuft.

Als Beispiel wird die Anfrage $5 + 2^3$ gestellt:

$$ans(Res) \quad \leftarrow \quad plus(Res, 5, X) \;\&\; hoch(X, 2, 3). \tag{3.9}$$

Eine Bindung $g$ erscheint in der Kastendarstellung in einer Stelle als $\bullet$ und eine $f$-Bindung als $\circ$. Eine Anfrage ist dann moduskonform, wenn jede $\bullet$-Stelle genau eine Kantenverbindung zu einer $\circ$-Stelle hat und $\circ$-Stellen nur Verbindungen zu $\bullet$-Stellen haben, aber durchaus – im Gegensatz zu den $\bullet$-Stellen – auch gar keine Verbindung haben dürfen. Für eine Kastendarstellung der bisherigen Art wird vor dem Übersetzungsvorgang versucht, eine solche Konstellation herzustellen. Man beachte, daß im allgemeinen, anders als im vorliegenden Beispiel, für eine Relation mehrere Bindungsmuster zugelassen sein können. Im Beispiel wird die moduskonforme Konstellation sofort gefunden:

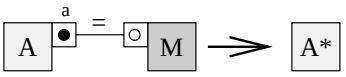



Man bedenke, daß die Bindungen angeben, was die zugrundeliegende Relation erzwingt. Die M-Kästen haben immer $\circ$-Stellen, da sie ja Werte liefern und nicht erwarten. Die Projektionskanten (gestrichelt) geben die Bindung nach außen weiter, schließlich ändert sich durch eine Projektion nicht die Art des Attributs.

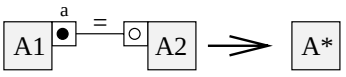
Die Übersetzungsregeln müssen nun erweitert werden. Zusätzlich zu den Arten der an einer Regel beteiligten Kästen, ist es noch wesentlich, welche Bindung eine Stelle hat. Im folgenden werden Übersetzungsregeln für eine Wissensquelle angegeben, die Verkettungen von Funktionen ausführt. Im Abschnitt 3.4 wird hiermit eine Anbindung an ein Computeralgebra-System vorgestellt. Eine Anfrage besteht in diesem Fall aus Funktionen und ihren Argumenten, die ebenfalls Funktionen oder atomare Werte sein können. Man bedenke, daß die Relationen funktionaler Wissensquellen immer nur genau eine $\circ$-Stelle haben können, daher kann die dritte Regel so kurz ausfallen. Dort dürfen A1 und A2 aber nicht vertauscht werden!

(1) 

{ Trage Funktionsname in A* ein. }

(2) 

{ Trage Convert(M) als atomaren Wert an a-ter Stelle in A* := A ein. }

(3) 

{ Trage Funktion A2 an a-ter Stelle in A* := A1 ein. }

(4) 

{ Schicke Anfrage A an Wissensquelle }

An dieser Stelle ist es sinnvoll zusammenzufassen, welche Arten von Kästen es gibt und welche Attribute/Merkmale sie haben. Jeder Kasten hat eine Reihe von Stellen, die leer

sein kann. Eine **Stelle** hat dabei einen Namen (den die Wissensquelle eventuell braucht), einen Bindungstyp, einen Verweis auf sein Auftreten an einem Kasten und seine Herkunft. Mit letzterem wird vermerkt, in welcher Relation das diese Stelle repräsentierende Attribut vorkam. Diese Information würde ansonsten im Übersetzungsprozeß verloren gehen. Ein **N-Kasten** hat einen Namen und eine laufende Nummer (zur Unterscheidung gleicher, aber verschieden genutzter Relationen in einer Anfrage). Ein **A-Kasten** speichert eine Anfrage, die nicht zwingend vollständig sein muß. Sie ist sicherlich dann nicht korrekt bzw. nicht vollständig, wenn eine der äußeren Stellen eine •-Bindung hat. Wie eine Anfrage aussieht, ist vollkommen abhängig von der Wissensquelle. Ein **M-Kasten** schließlich enthält einen Cursor als Attribut. Ein **K-Kasten** schließlich enthält einen Graphen wie in den Abbildungen, also Kästen und Kanten.

### 3.2.9 Konflikte bei den Bindungsmustern

Im Abschnitt 3.2.4 wurde der Fall betrachtet, daß von einer Wissensquelle mehr verlangt wurde, als sie konnte, nämlich konjunktive Anfragen mit mehr als einer Relation auf einmal zu beantworten. Der Übersetzer übernahm in diesem Fall die Organisation der Stellung mehrerer einzelner Anfragen zur Berechnung der gesamten Mediatoranfrage. Er hat somit die Wissensquelle dazu erst befähigt. Man nennt diese Rolle des Übersetzers dementsprechend auch „facilitator“. Die Regel, die anwendbar, aber von der Wissensquelle nicht durchführbar war, mußte der Übersetzer übernehmen.

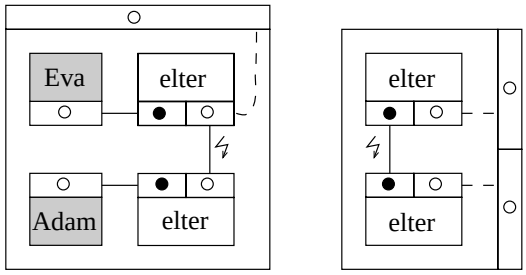
Nun kann es im Zusammenhang mit der Ermittlung der geeigneten Bindungsmuster für eine Anfrage passieren, daß nicht überall eine •-○-Konstellation herstellbar ist. Es können auch die beiden Fälle von ○-○- und •-•-Kanten entstehen. Diese beiden Fälle kann *keine* Wissensquelle behandeln. Auch hier muß der Übersetzer die entsprechende Arbeit leisten.

Die beiden Problemfälle sollen an einem einfachen Beispiel (angelehnt an [RSU95]) erläutert werden. Man nehme dazu eine Wissensquelle an, die folgende zwei Relationen beherrscht: $elter : Kindname \times Elternname$ mit der erlaubten Bindung $(\bullet, \circ)$ und $kind : Kindname$ mit der Bindung $(\circ)$. Sie kann also zu einem Kind die Eltern liefern oder alle Kinder, mehr nicht. Dies wird z.B. aufgrund einer Speicherung erzwungen, bei der die Relation nach den Kindernamen indiziert wurde (Listen). Die beiden folgenden Anfragen verursachen dabei Probleme:

$$ans(X) \leftarrow elter('Eva', X) \;\&\; elter('Adam', X). \tag{3.10}$$

$$ans(X, Y) \leftarrow elter(Kind, X) \;\&\; elter(Kind, Y). \tag{3.11}$$

Anfrage 3.10 fragt nach allen gemeinsamen Eltern von Eva und Adam, und Anfrage 3.11 nach allen Elternpaaren mit gemeinsamen Kindern. Man bedenke, daß alle *Paare* verlangt waren und nicht die Eltern einzeln. Die beiden Anfragen mit ihren einzig möglichen Bindungskombinationen zeigt folgendes Bild:

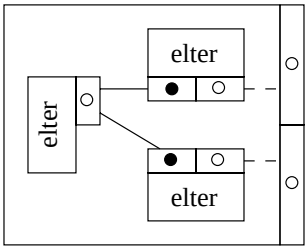



**○-○-Konflikt** Hier muß der Übersetzer einen Vergleich der beiden Stellen selbst durchführen. Da es keine ○-○-Regeln gibt, endet die Übersetzung mit einer ○-○-Kante zwischen zwei M-Kästen. Diese beiden Relationen sind mittels Cursor Tupel für Tupel zu durchlaufen, und nur dort, wo zwei Tupel gleich sind, ist die Kantenbedingung erfüllt. Der Übersetzer kann an dieser Stelle nur =-Kanten behandeln, d.h. er kann zwischen zwei Attributwerten nur ihre Gleichheit testen.

**●-●-Konflikt** Hier muß der Übersetzer dafür sorgen, daß die Stellen wie gefordert gebunden werden. Da die Stellen einen bestimmte Sorte haben, wäre es bei ●-●-Konflikten prinzipiell möglich, ein **Sortenprädikat** einzusetzen, also ein Prädikat, das alle Elemente der entsprechenden Sorte liefert. Dies ist in diesem Fall nicht realistisch, müßten doch alle irgend möglichen Namen durchlaufen werden. Bei Aufzählungstypen jedoch ist dies durchaus möglich; dort ist die Menge der Instanzen einer Sorte gering. Nach Einsatz des Sortenprädikats ändert sich die Anfrage wie folgt:

$$ans(X, Y) \leftarrow elter(Kind, X) \ \& \ elter(Kind, Y) \ \& \ name(Kind).$$

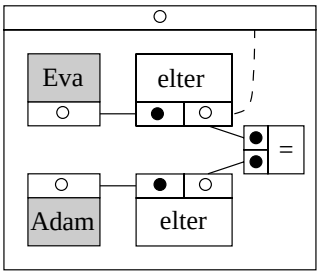
Die Beispiel-Wissensquelle liefert uns aber ein geeignetes Sortenprädikat, nämlich die Relation *kind*. Nur die Kinder in *kind* können Kinder von Elternpaaren sein. Bedenkt man jetzt noch, daß *kind* nichts anderes ist als *elter* mit der Bindung $(\circ, \cdot)$ ist, wobei $\cdot$ meint, daß diese Stelle gar nicht projiziert wird, so erhält man in der Kastendarstellung die Anfrage mit Hilfe des genutzten Sortenprädikats in naheliegender Darstellung[2] wie folgt:




Es gibt demnach zwei Möglichkeiten: Entweder die Wissensquelle liefert das Sortenprädikat, oder der Übersetzer muß es einsetzen. Im letzten Fall hat er wiederum zwei M-Kästen geeignet zu verbinden.

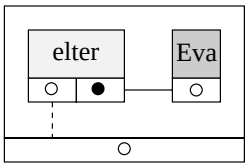
[2] Vielleicht doch nicht so naheliegend: Die erste Stelle einer Relation ist in Abhängigkeit davon, an welcher Seite des Kastens die Stellen gezeichnet sind: Sind sie links oder rechts, so ist die erste Stelle oben; sind sie unten oder oben so ist die erste Stelle links. Das ganze ist also nicht rotationsinvariant dargestellt.

Dieses Vorgehen der Konflikt-Lösung läßt sich ein wenig einheitlicher darstellen: So wie man mit einer Transformationsregel (Abschnitt 3.2.3) aus einem Kasten eine Kante macht, so kann man umgekehrt anstelle jeder =-Kante den =-Kasten einsetzen. Natürlich geht dies unendlich oft, da hierbei ja wieder zwei neue =-Kanten entstehen. Der zweistellige =-Kasten ist in allen vier möglichen Bindungen nutzbar. Da die Gleichheit symmetrisch ist, sind die $(\bullet, \circ)$ und die $(\circ, \bullet)$-Kombinationen austauschbar. Diese beiden einzusetzen macht keinen Sinn. Die anderen beiden sind aber genau dort einsetzbar, wo es Bindungskonflikte gibt: Eine ○-○-Kante wird durch den =-Kasten mit der Bindung $(\bullet, \bullet)$ ersetzt, womit der Konflikt aufgelöst ist, wie das Bild bezüglich Anfrage 3.10 zeigt:




Diese Art von =-Kasten testet für zwei Eingaben, ob sie gleich sind. Das ist genau das, was der Übersetzer an dieser Stelle auch tun muß. Beim anderen Konflikt wird eine •-•-Kante durch einen =-Kasten mit der Bindung $(\circ, \circ)$ ersetzt. Diese Art liefert alle gleichen Werte. Da beide Ausgaben also gleich sind, kann man sie auch zu einer einzigen zusammenfassen, und da alles typisiert ist, ergibt sich diese Art von =-Kasten zu dem oben genannten Sortenprädikat.

Ein Sortenprädikat muß ebenfalls eingesetzt werden, wenn eine •-Stelle ohne Kante bleibt. Dies ist eine unzulässige Konstellation, da die Stelle schließlich belegt werden muß. Außerdem dürfen M- und K-Kästen nur ○-Stellen besitzen. Ferner ist zu beachten, daß das Materialisieren eines A-Kastens nicht möglich ist, wenn mindestens eine der Stellen dieses Kastens eine •-Bindung hat. Dies kann vorkommen, wenn eine Wissensquelle bestimmte Operationen nicht ausführen kann. Man betrachte z.B. folgenden Zustand:




Für den Fall, daß die Wissensquelle nun die vorhandene Stellenkante nicht bearbeiten kann, würde zunächst versucht, den A-Kasten „elter“ zu materialisieren, um die Kante durch den Übersetzer abarbeitbar zu machen. Dies geht nicht wegen der unbesetzten •-Stelle. Ein weniger pathologischer Fall liegt vor, wenn man mittels

$$ans(X) \leftarrow hoch(X, 2, X).$$

einen Fixpunkt der Zweierpotenzfunktion erhalten will. Auch hier ist ein Weiterübersetzen nicht möglich, wenn die Wissensquelle die entstehende Kante nicht abarbeiten kann, was sehr gut denkbar ist.

### 3.2.10 Alternativen zur Kastendarstellung

In Abschnitt 3.1 wurden Techniken des Übersetzerbaus erläutert, z.B. auch Baumtransformationen. Es hat im wesentlichen zwei Gründe, warum diese Technik in der vorliegenden Arbeit nicht hauptsächlich angewendet wurde:

- Der Benutzer müßte die Regeln selbst erstellen, denn je nach Zielsprache gibt es andere Transformationsregeln. Das würde bedeuten, daß die Regeln nicht fest kodierbar wären und somit ein allgemeines Mustersuchen implementiert werden müßte. Dies wäre nicht so effizient wie die vorgeschlagene Realisierung. Außerdem gäbe es Probleme mit der Terminierung einer Übersetzung. Letztlich dürfte es auch nicht einfach sein, solche Regeln zu erstellen und einzugeben.

  Die Baumtransformationsregeln gemäß Abschnitt 3.2.3 haben selbstverständlich auch diese Probleme. In der derzeitigen Implementation ist es daher auch so, daß diese eine, dort vorgestellte Regel fest kodiert wurde (für beliebige Vergleichsrelationen). Dies ist eine Folge obengenannter Probleme und der Tatsache, daß es bislang noch keine Notwendigkeit gab, andere Regeln zu nutzen.

- Bei einem Versuch, die an sich einfache Übersetzung in SQL mittels Baumtransformationsregeln zu bewerkstelligen, waren zehn Regeln notwendig. Dies wurde dadurch verursacht, daß die select-, from- und where-Teile einer SQL-Anfrage in einer konjunktiven Anfrage „verteilt“ kodiert sind. Die Gleichheit zweier Variablen, deren Gegenstück in den where-Teil kommt, ist als solche z.B. nicht an einer Stelle lokalisiert. Dies bedeutet, daß die entsprechenden Teile eingesammelt werden müssen. Dieser Weg erschien daher sehr umständlich. Eine Alternative wäre es, die Kastendarstellung als Quellsprache heranzuziehen. Damit entstehen aber gerade die hier vorgestellten Regeln.

Eine andere Frage, die sich stellt, ist: Warum wurden nicht existierende Compiler-Compiler, etwa Yacc, eingesetzt? Diese haben den Nachteil, für beliebige kontextfreie Grammatiken einsetzbar zu sein. Hier dagegen liegt eine *bestimmte* Grammatik vor. Eine darauf spezialisierte Implementierung kann daher viel besser auf ihre Eigenarten eingehen. Insbesondere können Heuristiken in der Regelreihenfolge-Auswahl realisiert werden. Zudem bieten Compiler-Compiler nicht die Möglichkeit einer Zurückübersetzung, wie in Abschnitt 3.2.4 gefordert. Weiterhin bleibt es fraglich, wie die Bindungsmuster und die daraus resultierenden Einschränkungen realisiert werden sollten.

Die Kastendarstellung hat im Vergleich zu einer konjunktiven Anfrage und zu anderen String-orientierten Darstellungen einer Anfrage die folgenden drei Vorteile:

1. Die Kanten fassen gleiche Variablen zusammen, die in einer konjunktiven Anfrage unangenehm verteilt sind. Außerdem können die Konstanten als M-Kasten auf gleiche Weise angewendet werden wie ganze Ergebnisrelationen (siehe Abschnitt 3.2.4).

2. Der Übersetzungsprozeß wird anschaulich.

3. Die Kastendarstellung liefert geeignete Datenstrukturen für die Implementation.

## 3.3 Semantik und Erzeugung der Kastendarstellung

In diesem Abschnitt wird die Syntax und Semantik der Kasten-Darstellung formal definiert. Es ist dazu notwendig, auf die relationale Algebra zurückzugreifen. Einige Bezeichnungen sind an denen in [AHV95] genutzten angelehnt.

### 3.3.1 Allgemeine Definitionen

Eine **Sorte** (Domäne) ist eine Menge von **Term**en. Sie entspricht einem Datentyp oder Wertebereich. *string* ist z.B. die Menge aller Zeichenketten. Ein **Relationenbezeichner** $R[S]$ besteht aus einem Namen ($R$) und einer Folge $sort(R) = S$ von Sorten. Mit $|R|$ wird abkürzend $|sort(R)|$ bezeichnet, also die Stelligkeit eines Relationenbezeichners. Ein **Wissensquellenschema** (Exportschema) $D$ ist eine Menge von Relationenbezeichnern, z.B. $Uni = \{Student\ [string, integer], Note[integer, string, real]\}$. Die einzelnen Sorten der Relationenbezeichner werden im folgenden als geordnet angesehen und als **Attribute**[3]. Attribute haben im Rahmen dieses Unterkapitels keinen Namen, obschon sie im Exportschema einer Wissensquelle auftauchen. Sie werden statt dessen per Index in der Folge $sort(R)$ referenziert.

Ein **Tupel** $t$ ist ein Element aus einem Kreuzprodukt $D = s_1 \times \cdots \times s_n$ von Sorten. $D$ wird im folgenden selbst wieder als Sorte bezeichnet. Die Stelligkeit ist $|t| = n$, die Domäne ist $sort(t) = D$. $t_i$ bezeichnet das i-te Element von $t$. Tupel werden in spitzen Klammern geschrieben; es ist also $t = \langle t_1, \ldots, t_n \rangle$. Eine **Relation** $I(R)$ oder $I_R$ über einen Relationenbezeichner $R[S]$ ist eine Menge von Tupeln $t$ mit $sort(t) = S$ für alle $t \in I(R)$. Man beachte den Unterschied zwischen einer leeren Relation $\{\}$ und einer nicht-leeren Relation $\{\langle\rangle\}$ über ein leeres Schema. Eine **Wissensquelle** $I$ („Instanz“) über ein Wissensquellenschema $D$ bildet jeden Relationenbezeichner $R \in D$ auf eine Relation $I(R)$ ab. Zum Beispiel kann

$$\begin{aligned} UniDB(Student) &= \{\langle \text{“Olaf Meyer“}, 701234 \rangle\} \qquad \text{und} \\ UniDB(Note) &= \{\langle 701234, \text{“Formale Systeme“}, 2.0 \rangle, \\ &\quad\ \langle 701234, \text{“Rechnerstrukturen“}, 2.7 \rangle\} \end{aligned}$$

sein.

Eine **Variable** $v$ ist ein Bezeichner für ein beliebiges Element aus einer Sorte $sort(v)$. Ein **freies Tupel** besteht aus Termen und Variablen. Ein Tupel, das nur aus Termen besteht, wird in Unterscheidung dazu **nicht-freies Tupel** oder weiterhin Tupel genannt. $var(t)$ ist die Menge der Variablen in Tupel $t$. Ein **Atom** ist ein Ausdruck $R(t_1, \ldots, t_n)$, wobei $R$ der Name einer Relation und $t = \langle t_1, \ldots, t_n \rangle$ ein eventuell freies Tupel ist mit $|R| = n$ und $sort(R) = sort(t)$. Ein Atom heißt auch **Faktum**, wenn $t$ nicht-frei ist. Eine Menge von Fakten der Form $R(t)$ beschreibt eine Wissensquelle $I$ über $D$, wenn für alle $R \in D$ gilt: $t \in I(R)$. Die Wissensquelle $UniDB$ aus dem obigen Beispiel kann also auch geschrieben werden als

$$\{\quad Student(\text{“Olaf Meyer“}, 701234),$$

[3] Man beachte, daß das Wort Attribut auch noch im Sinne der vererbten Attribute – etwa eine Anfrage als Attribut eines A-Kastens – beim Übersetzerbau vorkommt.

$$Note(701234, \text{“Formale Systeme“}, 2.0),$$
$$Note(701234, \text{“Rechnerstrukturen“}, 2.7) \quad \}$$

Ein **Literal** ist ein Atom $R(t)$ oder ein **negiertes Atom** $\neg R(t)$. Es ist $\neg R(t)$ ein Faktum in $I$, wenn $t \notin I(R)$.

### 3.3.2 Relationale Algebra

Die relationale Algebra definiert eine Menge von Operationen auf Relationen. Da Relationen Mengen sind, werden die Vereinigung $\cup$, Differenz $-$ und der Schnitt $\cap$ übernommen. Ebenfalls definiert sind somit die Gleichheit $=$ und Teilmengenbeziehung $\subseteq$ zwischen Relationen. Da Relationen aber Mengen über Tupeln sind, gibt es weitere nützliche Operatoren:

- Kreuzprodukt: $I_R \times I_S := \{\langle r_1, \ldots, r_n, s_1, \ldots, s_m\rangle \mid r \in I_R, s \in I_S\}$ mit $n = |R|$ und $m = |S|$. Das mehrstellige Kreuzprodukt ist entsprechend definiert.
- Attributselektion: $\sigma_{i\theta j}(I_R) := \{r \in I_R \mid r_i \ \theta \ r_j\}$ mit $sort(r_i) = sort(r_j)$ und $1 \leq i, j \leq |R|$. Hierbei ist $\theta$ ein beliebiger Vergleich zwischen zwei Termen gleicher Sorte, z.B. $=$ oder $\leq$.
- Werteselektion: $\sigma_{i\theta t}(I_R) := \{r \in I_R \mid r_i \ \theta \ t\}$ mit Term $t \in sort(r_i)$ und $1 \leq i \leq |R|$.
- Projektion: $\pi_{j_1,\ldots,j_m}(I_R) := \{\langle r_{j_1}, \ldots, r_{j_m}\rangle \mid r \in I_R\}$ mit $0 \leq m$. Für $m = 0$ ist $\pi(I_R) = \{\langle\rangle\}$ wenn $I_R$ nicht leer ist, ansonsten $\{\}$.

Die folgenden nützlichen Operatoren lassen sich aufgrund der vorherigen definieren:

- Verbund: $I_R \bowtie_{i\theta j} I_S := \sigma_{i\theta(j+|R|)}(I_R \times I_S)$. Die Werteselektion läßt sich durch den Verbund ersetzen: $\sigma_{i\theta t}(I_R) = I_R \bowtie_{i\theta 1} \{\langle t\rangle\}$
- Verallgemeinerte Projektion: $\pi_{(j_1^R,\ldots,j_n^R),(j_1^S,\ldots,j_m^S)}(I_R, I_S) := \pi_{j_1^R,\ldots,j_n^R}(I_R) \times \pi_{j_1^S,\ldots,j_m^S}(I_S)$. Die mehrstellige verallgemeinerte Projektion ist entsprechend definiert. Die oben schon definierte Projektion ist die einstellige Variante. Außerdem gilt

$$I_R \times I_S = \pi_{(1,\ldots,|R|),(1,\ldots,|S|)}(I_R, I_S)$$

Aufgrund der Definition der Operatoren bilden {Attributselektion(S), Projektion(P), Kreuzprodukt(C)} (SPC-Algebra) und {Verbund, verallgemeinerte Projektion, Attributselektion} (hier verwendete Algebra) beide jeweils zusammen mit der Vereinigung und der Differenz Basen der gleichen relationalen Algebra. Man beachte, daß alle Operatoren als Parameter und Ergebnis Relationen haben. Im folgenden werden Formeln der relationalen Algebra kurz **relationale Ausdrücke** genannt.

### 3.3.3 Syntax der Kastendarstellung

**Definition 3.1 (Syntax der Kastendarstellung)** *Ein* **Kasten** $K$ *besteht aus einem* **Merkmal**[4] $m(K)$ *und einer Folge* $st(K)$ *von Stellen. Eine* **Stelle** $s$ *hat einen Index* $ord(s, K)$

[4] Dies ist das Attribut in Sinne des Übersetzerbaus.

*in der Folge $st(K)$ und eine zugeordnete Sorte $sort(s)$. Es gibt vier verschiedene Kastentypen:*

*(1) Ein* **N-Kasten** *hat als Merkmal einen Relationenbezeichner.*

*(2) Ein* **M-Kasten** *hat als Merkmal eine Relation = Tupelmenge.*

*(3) Ein* **A-Kasten** *hat als Merkmal einen relationalen Ausdruck.*

*(4) Ein* **K-Kasten** *hat als Merkmal (hier auch Inhalt genannt) eine Menge von Kästen und gerichteten, markierten* **Kanten**. *Es gibt vier Kantenarten:*

*(4.1) Eine* **Kastenkante** *liegt zwischen zwei Kästen.*

*(4.2) Eine* **Verbundkante** *liegt zwischen zwei Stellen verschiedener Kästen.*

*(4.3) Eine* **Attributselektionskante** *liegt zwischen zwei Stellen desselben Kastens. Diese und die Verbundkante heißen auch kurz* **Stellenkanten**.

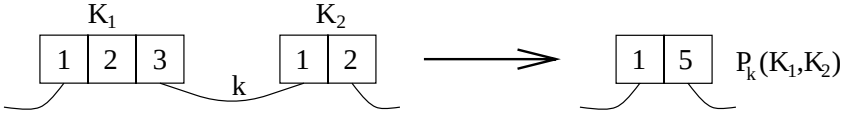
*(4.4) Eine* **Projektionskante** *liegt zwischen einer Stelle eines Kastens und einer Stelle des K-Kastens, der diesen Kasten enthält.*

Alle *Stellen eines K-Kastens sind nach innen mittels einer Projektionskante mit einer Stelle eines Kastens aus dessen Inhalt verbunden.*

Ein Kasten entspricht also einer Relation bzw. einem Relationenbezeichner und eine Stelle dieses Kastens einem Attribut der Relation. Die graphische Darstellung von Kästen wurde schon im Abschnitt 3.2 gezeigt. Es folgt noch die Definition einer weiteren nützlichen Operation, die dem im Abschnitt 3.2.2 vorgeführten Umfassen einer Regelprämisse und dem Schneiden der Kanten entspricht. Das im folgenden definierte $P$ enthält diejenigen Stellen, die nach Abarbeiten einer Kante – aufgrund einer Regelanwendung – noch „gebraucht“ werden.

**Definition 3.2 (Herausprojektion)** *Sei eine Kante $k$ (im folgenden „abzuarbeitende Kante“ genannt) gegeben, die keine Projektionskante ist. $k$ gehe direkt (als Kastenkante) oder indirekt (als Stellenkante) von Kasten $K_1$ nach Kasten $K_2$. Die* **Herausprojektion** *$P_k(K_1, K_2)$ ist die Folge $(s_1^1, \ldots, s_n^1, s_1^2, \ldots, s_m^2)$ aller Stellen aus $K_1$ und $K_2$, die an mindestens einer beliebigen Kante (inklusive Projektionskanten!), die* nicht *$k$ ist, liegen. $P_k(K_1)$ ist die entsprechende Folge mit nur einem Kasten, $P(K_1)$ die Folge ohne abzuarbeitende Kante und $P(K_1, \ldots, K_n)$ die Folge für mehr als zwei Kästen ohne abzuarbeitende Kante. $ord(P)$ ist die entstehende Indexfolge der Herausprojektion $P$, d.h. $ord(P) = (ord(s_1^1, K_1), \ldots, ord(s_n^1, K_1), n + ord(s_1^2, K_2), \ldots, n + ord(s_m^2, K_2))$, entsprechend für das ein- und mehrstellige $P$ mit und ohne abzuarbeitende Kante $k$.*

Folgendes Bild illustriert die Definition :



### 3.3.4 Semantik und Dynamik der Kastendarstellung

Nun läßt sich die Bedeutung der Kastendarstellung mit Hilfe relationaler Operatoren definieren. Es wird eine Abbildung *red* („Reduktion“) definiert, die einen Kasten in einen relationalen Ausdruck wandelt. Diese Übersetzung geschieht analog zu den Übersetzungsregeln, auf die danach eingegangen wird.

**Definition 3.3 (Semantik der Kastendarstellung)** *Sei $I$ eine beliebige Wissensquelle über $D$ und $R \in D$. Die Reduktion $red(K, I)$ oder kurz $red(K)$ eines Kastens $K$ ist strukturell induktiv wie folgt definiert:*

*(1) Ist $K$ ein N-Kasten mit Relationenbezeichner $R$, so ist $red(K) = I(R)$.*

*(2) Ist $K$ ein M-Kasten mit Tupelmenge $M$, so ist $red(K) = M$.*

*(3) Ist $K$ ein A-Kasten mit relationalem Ausdruck $A$, so ist $red(K) = A$.*

*(4) Ist $K$ ein K-Kasten, so ist $red(K) = red(K')$, wobei $K'$ aus $K$ durch Anwendung einer der folgenden Regeln, deren Prämisse[5] erfüllt ist, hervorgeht:*

*(4.1) Wenn $K$ eine Kastenkante $k$ von $K_1$ nach $K_2$ mit der Marke $\theta$ enthält, dann ersetze $k$,$K_1$,$K_2$ durch einen neuen A-Kasten mit dem relationalen Ausdruck $(\pi_{ord(P_k(K_1,K_2))}(red(K_1)\ \theta\ red(K_2)))$ als Merkmal und den Stellen $P_k(K_1, K_2)$.*

*(4.2) Wenn $K$ eine Verbundkante $k$ von Stelle $s_1$ an Kasten $K_1$ nach Stelle $s_2$ an Kasten $K_2$ mit der Marke $\theta$ enthält, dann ersetze $k$,$K_1$,$K_2$ durch einen neuen A-Kasten mit dem relationalen Ausdruck $(\pi_{ord(P_k(K_1,K_2))}(red(K_1) \bowtie_{ord(s_1)\theta ord(s_2)} red(K_2)))$ als Merkmal und den Stellen $P_k(K_1, K_2)$.*

*(4.3) Wenn $K$ eine Attributselektionskante $k$ von Stelle $s_1$ nach Stelle $s_2$, beide an Kasten $K_1$, mit der Marke $\theta$ enthält, dann ersetze $k$,$K_1$ durch einen neuen A-Kasten mit dem relationalen Ausdruck $(\pi_{ord(P_k(K_1))}(\sigma_{ord(s_1)\theta ord(s_2)}(red(K_1))))$ als Merkmal und den Stellen $P_k(K_1)$.*

*(4.4) Besteht $K$ nur noch aus Projektionskanten und den Kästen $K_1, \ldots, K_n$, so ist $K'$ gleich einem A-Kasten mit dem relationalen Ausdruck*

$$(\pi_{ord(P(K_1)),\ldots,ord(P(K_n))}(K_1, \ldots, K_n))$$

*als Merkmal und den Stellen $P(K_1, \ldots, K_n)$. Es wird also* ganz *$K$ ersetzt.*

*Mindestens eine der Regelprämissen ist aufgrund der Syntax der Kastendarstellung stets erfüllt. Die Stellen werden mitsamt ihren Kanten in den neu erzeugten A-Kasten übernommen.*

Die Bedeutung eines K-Kastens K hängt also aufgrund der N-Kästen von einer Wissensquelle $I$ ab. $red(K)$ liefert einen relationalen Ausdruck, dessen Auswertung dementsprechend auch von der Wissensquelle abhängt.

[5] Siehe Fußnote in Abschnitt 3.2.2

Die Regeln entsprechen genau den vier Kantenfällen aus Definition 3.3. Dies waren die **Kantenregeln**, die in ihrer Prämisse verschiedene Ausprägungen aufgrund des Typs der Kästen haben können. Es kommen noch zwei **Kastenregeln** hinzu:

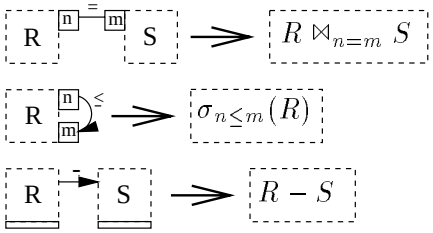
*(4.6) Ein N-Kasten $K$ wird durch einen A-Kasten mit Merkmal $red(K)$ und den Stellen $st(K)$ ersetzt.*

*(4.7) Ein A-Kasten $K$ wird durch einen M-Kasten mit Merkmal $K(I)$ (siehe nachfolgende Definition) und den Stellen $st(K)$ ersetzt. Hier wird also der relationale Ausdruck $m(K)$ bezüglich einer Wissensquelle evaluiert. Dies entspricht im Übersetzungsprozeß dem konkreten Abschicken der Anfrage $m(K)$ an die Wissensquelle.*

Eine Kastendarstellung ist aufgrund eines relationalen Ausdrucks definiert. Instanziiert man diesen mit Hilfe einer Wissensquelle, so kann man von einer Relation einer Kastendarstellung sprechen.

**Definition 3.4 (Relation einer Kastendarstellung)** *Sei $K$ ein Kasten und $I$ eine Wissensquelle mit passendem Exportschema, dann bezeichnet $K(I)$ die Relation = Tupelmenge, die durch Evaluation von $red(K, I)$ entsteht.*

Es wäre interessant, die Korrektheit einer Übersetzung zeigen zu können. Ein Übersetzungsvorgang teilt sich in zwei Schritte: Zunächst wird eine Anfrage vom Mediator in die Kastendarstellung konvertiert. Diese Konvertierung ist Thema des nächsten Abschnittes. Dann wird diese Kastendarstellung per Regeln und Aktionen in die Wissensquellenanfrage übersetzt. Die Aktionen müssen das Merkmal des neuen Kastens füllen, insbesondere also auch das Merkmal eines A-Kastens. In diesem Merkmal muß die Übersetzung der Kastendarstellung einer Anfrage in der Sprache der Wissensquelle stehen. Eine Aktion ist also genau dann **korrekt**, wenn die Semantik des Merkmals des Conclusio[6]-Kastens gleich der Semantik der Reduktion (ausgedrückt mittels eines relationalen Ausdrucks) des Conclusio-Kastens ist. Als Beispiel seien drei Regeln mit der Bedeutung des Conclusio-Kastens angegeben. Dabei sind die Typen der Kästen wohlgemerkt unwichtig.




Ein Übersetzer ist korrekt, wenn alle seine Aktionen korrekt sind. Diese Definition der Korrektheit hat zur Folge, daß alle Anfragen an Wissensquellen mittels der Operatoren der relationalen Algebra äquivalent ausdrückbar sind. Dies ist eine wesentliche Bedingung. Zum Korrektheitsbeweis eines Übersetzers ist es außerdem notwendig, eine Semantik der lokalen Anfragesprache der Wissensquelle – etwa SQL – definiert zu haben. Dies ist sehr selten der Fall.

[6]Siehe Fußnote in Abschnitt 3.2.2

In Abschnitt 3.2 wurde erwähnt, daß in Sonderfällen der Übersetzer die Ausführung von Regeln übernimmt, z.B. bei einer Verbundkante zwischen zwei M-Kästen. Was dort zu tun ist (insbesondere die Semantik der Cursor), bestimmt ebenfalls Definition 3.3.

Die Aktion einer Regel besteht nicht nur aus der programmiersprachlichen Aktion, die in Abschnitt 3.2 in geschweiften Klammern stand, sondern auch aus dem automatischen Teil der Erzeugung eines neuen A-Kastens, insbesondere der Erzeugung von dessen Stellen. Diese ist korrekt, weil aufgrund Definition 3.2 die Stellenmenge $P$ den Projektionsattributen $ord(P)$ entspricht.

An dieser Stelle ist noch zu vermerken, daß $red(K)$ eine beinahe optimale (gemäß [Ull89], Ch. 11.6) Übersetzung eines Kastens in einen relationalen Ausdruck ist, denn es werden stets so früh wie möglich maximale Projektionen vorgenommen (aufgrund der ständigen Herausprojektion) und ein Kreuzprodukt wird mit einer Selektion zu einem Verbund zusammengefaßt, so daß niemals eine Selektion über ein zu großes Produkt gefordert wird. Was jedoch nicht entdeckt wird, sind z.B. gleiche Teilausdrücke. Diese Aussage ist aber nicht sehr bedeutend, denn man kann nicht annehmen, daß eine Optimalität bzgl. der relationalen Algebra (und der naiven Berechnung ihrer Ausdrücke) auch Optimalität bzgl. der einzelnen Wissensquellen bedeutet.

### 3.3.5 Erzeugung der Kastendarstellung einer Anfrage

Zunächst wird die Art der Anfragen, die vom Mediator kommen, definiert:

**Definition 3.5 (Konjunktive Anfrage, Klauselanfrage)** *Sei $D \supseteq \{R_1, \ldots, R_{n+m}\}$ ein Wissensquellenschema. Eine* **konjunktive Anfrage** *$q$ über $D$ hat die Form*

$$ans(x) \leftarrow R_1(x_1)\ \&\cdots\&\ R_n(x_n)\ \&\ \neg R_{n+1}(x_{n+1})\ \&\cdots\&\ \neg R_{n+m}(x_{n+m}).$$

*wobei $n, m \geq 0$, $ans \notin D$, $x$ ein Tupel aus Variablen und $x_1, \ldots, x_{n+m}$ (evtl. freie) Tupel der entsprechenden Sorte und Stelligkeit sind. $sort(ans)$ und $|ans|$ werden durch $x$ definiert. Es muß ferner $var(x) \subseteq var(q) := \bigcup_{i=1}^{n+m} var(x_i)$ gelten.*

Der Teil $ans(x)$ heißt **Kopf**, die Konjunktion rechts vom Pfeil **Rumpf** einer Anfrage. Die Reihenfolge der Literale im Rumpf ist beliebig, die oben dargestellte bietet lediglich eine angenehme Durchnumerierung. Da hier eine neue Relation ($ans$) aufgrund schon bestehender Relationen definiert wird, nennt man diese Form in Anlehnung an die logische Programmierung auch **Regel**, in Anlehnung an die Datenbanktechnik auch **Sicht**.

Eine Substitution ist wie üblich definiert:

**Definition 3.6 (Substitution)** *Sei Var eine ausgezeichnete Menge von Variablen und Term die Menge aller Terme. Eine* **Substitution** *$\nu$ (valuation) ist eine totale Funktion $\nu : Var \cup Term \to Term$, die eine Variable auf einen Term gleicher Sorte und einen Term auf diesen selbst abbildet. Ferner ist $\nu(\langle t_1, \ldots, t_n\rangle) := \langle \nu(t_1), \ldots, \nu(t_n)\rangle$ für $t_i \in Var \cup Term$.*

Auch hier kann man von einer Relation einer konjunktiven Anfrage sprechen:

**Definition 3.7 (Relation einer konjunktiven Anfrage)** *Sei $I$ eine Wissensquelle über $D$ und $q$ wie in Definition 3.5 eine konjunktive Anfrage. Dann ist $q(I) := I_{ans} := \{\nu(x) \mid \nu$ ist Substitution über $var(q)$ mit $\nu(x_i) \in I_{R_i}$ und $\nu(x_{n+j}) \notin I_{R_{n+j}}$ für jedes $i \in [1,n], j \in [1,m]\}$*

Dies ist die vom Übersetzer aus Mediatorsicht zu erwartende Semantik einer Anfrage an eine Wissensquelle. Die Menge ist dabei als Mehrfachmenge zu verstehen, d.h. wenn es mehrere verschiedene erfüllende Substitutionen gibt, deren Bilder über $x$ jedoch gleich sind, so erscheinen diese dennoch alle in der Menge.

Der nun folgende Algorithmus *Konv* konvertiert eine konjunktive Anfrage in eine Kastendarstellung. Dabei werden Negationen und Bindungsmuster zunächst nicht beachtet.

**Algorithmus 3.1 (Erzeugung der Kastendarstellung einer Anfrage ohne Negation)** *Konv bekommt eine konjunktive Anfrage $q$ als Eingabe.*

1. *Erzeuge für jedes Atom $R$ in $q$ einen N-Kasten $K_R$ mit Merkmal $R$ und $|R|$ vielen Stellen.*
2. *Erzeuge für jede Variable $v$ in $var(q)$ eine leere Liste $L_v$ von Stellen.*
3. *Für jedes Atom $R$ in $q$ und jedes seiner Argumente an Position $i$:*
   (a) *Wenn an Position $i$ eine Variable $v$ ist, dann füge die $i$-te Stelle von $K_R$ in $L_v$ ein.*
   (b) *Wenn an Position $i$ eine Konstante $c$ ist, dann erzeuge einen M-Kasten mit Merkmal $\{\langle c\rangle\}$ und eine Stelle $s$. Erzeuge eine neue Variable $v$ mit leerer Stellenliste $L_v$. Füge schließlich $s$ und die $i$-te Stelle von $K_R$ in $L_v$ ein.*
4. *Für jede Variable $v$ nimm eine Stelle $s_v$ aus $L_v$ und erzeuge $=$-Kanten von $s_v$ zu jeder anderen Stelle aus $L_v$, wenn vorhanden.*
5. *Erzeuge K-Kasten $K$ mit $|ans|$ vielen Stellen, der alle bislang erzeugten Kästen und Kanten enthält.*
6. *Für jede Variable $v$ in $var(x)$ an Position $i$ setze eine Projektionskante von der $i$-ten Stelle von $K$ zu $s_v$.*

Abbildung 3.4 zeigt die Schritte des Algorithmus am Beispiel der Anfrage

$$ans(X) \leftarrow Student(X,Y) \;\&\; Semester(Y,1).$$

*Konv* soll im folgenden als korrekt bewiesen werden. Dazu werden zunächst noch Umformulierungen von konjunktiven Anfragen und relationalen Ausdrücken definiert.

**Definition 3.8 (Anfrage mit expliziter Gleichheit)** *In einer konjunktiven Anfrage mit expliziter Gleichheit kommt in den Literalen im Rumpf keine Konstante und jede Variable nur einmal vor. Die Gleichheit zweier Variablen $X$ und $Y$ bzw. einer Variablen $X$ zu einer Konstanten $c$ wird explizit durch Hinzufügen eines Konjunktionsglieds $X = Y$ bzw. $X = c$ ausgedrückt.*

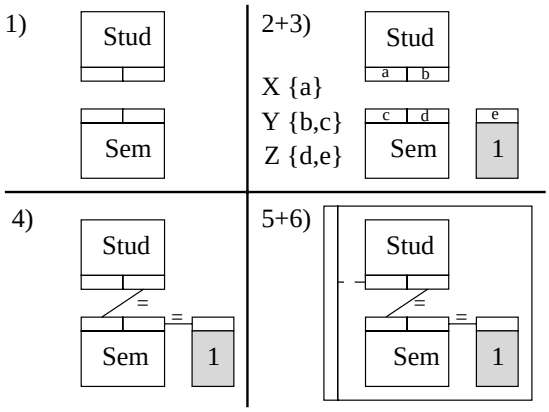



Abbildung 3.4: Der Algorithmus $Konv$

Zum Beispiel wird

$$ans(X) \leftarrow Student(X, Y)\ \&\ Note(Y, Z, 2.0).$$

zu

$$ans(X) \leftarrow Student(X, Y_1)\ \&\ Note(Y_2, Z, C)\ \&\ Y_1 = Y_2\ \&\ C = 2.0.$$

**Definition 3.9 (Äquivalenz zweier Anfragen/relationaler Ausdrücke)** *Zwei Anfragen/relationale Ausdrücke $q_1, q_2$ heißen* **äquivalent**, *wenn für jede Wissensquelle $I$ gilt: $q_1(I) = q_2(I)$.*

**Satz 3.1** *Zu jeder konjunktiven Anfrage gibt es eine äquivalente Anfrage mit expliziter Gleichheit (aber nicht umgekehrt).*

Der **Beweis** ist in [Ull88], Lemma 3.1, zu finden. Dort heißen konjunktive Anfragen mit expliziter Gleichheit „rectified“. Sie unterscheiden sich von den hier benutzten darin, daß auch Variablen im Kopfatom nirgens in einem Rumpfatom vorkommen.

**Definition 3.10 (Anfrage ohne Konstante)** *In einer Anfrage ohne Konstante kommt eine Konstante weder in den Literalen noch in den =-Prädikaten vor. Ein Prädikat $X = c$ in Anfragen mit expliziter Gleichheit wird ausgedrückt als $X = Y\ \&\ \{\langle c\rangle\}(Y)$. $\{\langle c\rangle\}$ bezeichnet dabei eine neue Relation mit genau diesem Inhalt.*

Im obigen Beispiel ergibt sich die Umformung zu

$$ans(X) \leftarrow Student(X, Y_1)\ \&\ Note(Y_2, Z, C_1)\ \&\ \{\langle 2.0\rangle\}(C_2)\ \&\ Y_1 = Y_2\ \&\ C_1 = C_2.$$

Die Relation einer Anfrage mit expliziter Gleichheit und ohne Konstante ist entsprechend Definition 3.7 definiert, wobei zusätzlich für die erfüllende Substitution $\nu$ und jedes Prädikat $X = Y$ gelten muß: $\nu(X) = \nu(Y)$.

**Satz 3.2** *Zu jeder Anfrage mit expliziter Gleichheit $q$ und Kopf $ans(x)$ gibt es eine äquivalente Anfrage $q'$ mit Kopf $ans'(x')$ ohne Konstante.*

Zum **Beweis** nehme man an, daß $\nu$ eine Substitution über $var(q)$ ist, so daß $\nu(x) \in q(I)$. Definiere $\nu' := \nu$ mit zusätzlich $\nu'(Y) = c$ für jede Konstante $c$ in einem Ausdruck $X = c$ aus $q$ und mit $Y$ als neuer Variable. Es gilt $\nu'(X) = \nu'(Y)$ und $\nu'(\langle Y \rangle) = \langle c \rangle \in \{\langle c \rangle\}$, also ist $\nu'$ eine Substitution über $var(q')$, so daß $\nu'(x') \in q'(I)$. □

Man bedenke, daß hiermit neue (built-in) Prädikate der Art $\{\langle c \rangle\}$ erzeugt werden, für welche $\{\langle c \rangle\}(c)$ und nichts anderes gilt. Daher ändert diese Umformung auch nicht die Herbrand-Basis einer konjunktiven Anfrage, also die Menge der Konstanten (uninterpretierte Funktionssymbole tauchen nicht auf). Die Umformung ist eine rein syntaktische.

Auch für relationale Ausdrücke wird nun eine kanonische Form definiert.

**Definition 3.11 (Normalform eines relationalen Ausdrucks)** *Ein relationaler Ausdruck ist in Normalform, wenn er die Form*

$$\pi_{j_1,\ldots,j_m}(\sigma_{a_1=b_1} \circ \cdots \circ \sigma_{a_k=b_k}(I_{R_1} \times \cdots \times I_{R_n}))$$

*hat, wobei $j_i, a_i, b_i$ erlaubte Indizes sind.*

**Satz 3.3** *Zu jedem relationalen Ausdruck mit den Operationen Verbund, Attributselektion und Projektion gibt es einen äquivalenten relationalen Ausdruck in Normalform.*

Der **Beweis** ist in sehr ähnlicher Form in [AHV95], Proposition 4.4.2, zu finden. Man kann also relationale Ausdrücke, die sich durch die Reduktion einer Kastendarstellung ergeben, normalisieren.

Die Korrektheit des Algorithmus $Konv$ zeigt nun folgender Satz:

**Satz 3.4** *Es gilt für alle Wissensquellen $I$: $q(I) = Konv(q)(I)$, d.h. die Relation der eingehenden Mediatoranfrage ist gleich der Relation der Kastendarstellung, wobei sich letztere aus der Reduktion (red) auf einen relationalen Ausdruck ergibt.*

**Beweis**: Sei $I$ beliebig und $Konv(q)$ ein relationaler Ausdruck in der Normalform $P \circ S \circ K$ aus Definition 3.11. $P$ ist der Projektionsteil $\pi_{j_1,\ldots,j_m}$, $S$ der Selektionsteil $\sigma_{a_1=b_1} \circ \cdots \circ \sigma_{a_k=b_k}$ und $K$ der Produktteil $I_{R_1} \times \cdots \times I_{R_n}$. Sei ferner die konjunktive Anfrage $q$ mit expliziter Gleichheit ohne Konstante gegeben als:

$$\begin{aligned} ans(v_{j_1}, \ldots, v_{j_m}) \quad \leftarrow \quad & R_1(v_1, \ldots, v_{|R_1|}) \;\&\; \cdots \;\&\; R_n(v_{p-|R_n|}, \ldots, v_p) \;\& \\ & v_{a_1} = v_{b_1} \;\&\; \cdots \;\&\; v_{a_k} = v_{b_k}. \end{aligned}$$

mit $p = |R_1| + \cdots + |R_n|$. Die hier behauptete Korrespondenz der Bezeichner und Indizes aus $q$ und $Konv(q)$ stellt der Algorithmus $Konv$ her, denn:

- Für jedes Literal $R_i$ wird ein N-Kasten erzeugt, dessen Reduktion $I_{R_i}$ ist.
- Für jedes $v_{a_i} = v_{b_i}$ wird eine Kante zwischen den entsprechenden Stellen erzeugt, deren Reduktion $\sigma_{a_i=b_i}()$ entspricht. In der Tat wird in $Konv$ von einer Anfragedarstellung *ohne* explizite Gleichheit ausgegangen. Der Algorithmus zieht die Kanten immer von *einer* Stelle zu allen anderen innerhalb einer Variablen. Wegen der Transitivität der Gleichheit und der Assoziativität und Kommutativität der Attributselektion $\sigma$ ist dies aber äquivalent zu einem Pfad von Kanten entsprechend der =-Prädikate in $q$.

- Für jede Variable in $var(x)$ wird eine Projektionskante eingesetzt, die dafür sorgt, daß diese Stelle gemäß $red$ projiziert wird.
- Für jede auftretende Konstante $c$ realisiert $Konv$ die Konversion zur konstantenfreien Anfragedarstellung gemäß Definition 3.10. Er fügt einen M-Kasten hinzu, dessen Reduktion $\{\langle c\rangle\}$ ist.

Die Äquivalenz von $Konv(q)$ und $q$ ergibt sich nun wie folgt:

$$
\begin{aligned}
& \langle t_{j_1},\ldots,t_{j_m}\rangle \in Konv(q)(I) = P\circ S\circ K \\
\Leftrightarrow\ & \exists t_1,\ldots,t_p : \langle t_1,\ldots,t_p\rangle \in S\circ K \text{ mit } \{j_1,\ldots,j_m\}\subseteq\{1,\ldots,p\} \\
& \text{(da } P \text{ genau diese Teilmenge herausprojiziert)} \\
\Leftrightarrow\ & \langle t_1,\ldots,t_p\rangle \in K \wedge \bigwedge_{i=1}^{k} t_{a_i} = t_{b_i} \text{ (nach Definition von } \sigma) \\
\Leftrightarrow\ & \langle t_1,\ldots,t_{|R_1|}\rangle \in I_{R_1} \wedge \ldots \wedge \langle t_{p-|R_n|},\ldots,t_p\rangle \in I_{R_n} \wedge \bigwedge_{i=1}^{k} t_{a_i} = t_{b_i} \\
& \text{(nach Definition von } \times) \\
\Leftrightarrow\ & \nu(\langle v_1,\ldots,v_{|R_1|}\rangle) \in I_{R_1} \wedge \ldots \wedge \nu(\langle v_{p-|R_n|},\ldots,v_p\rangle) \in I_{R_n} \wedge \bigwedge_{i=1}^{k} \nu(v_{a_i}) = \nu(v_{b_i}) \\
& \text{(wobei Substitution } \nu(v_i) := t_i \text{ für alle } i\in\{1,\ldots,p\}) \\
\Leftrightarrow\ & \nu(\langle v_{j_1},\ldots,v_{j_m}\rangle) = \langle t_{j_1},\ldots,t_{j_m}\rangle \in q(I) \text{ (nach Definition von } q(I)). \ \square
\end{aligned}
$$

Bislang wurden die Baumtransformationsregeln gemäß Anschnitt 3.2.3 noch nicht erwähnt. Sie werden nach $Konv$ angewendet. Ihre Semantik und wissensquellen-anhängige Korrektheit ergibt sich direkt aus der Semantik der Regelprämisse und -conclusio.

### 3.3.6 Anfragen mit Negation

Sei $R_{n+i}[S_1,\ldots,S_n]$ ein Relationenbezeichner und tauche $R_{n+i}$ negiert in einer Anfrage auf. Um festzustellen, ob $\nu(x_{i+n}) \notin I_{R_{i+n}}$ (Definition 3.7) kann man testen, ob $\nu(x_{i+n}) \in \overline{I_{R_{i+n}}} = DOM - I_{R_{i+n}}$ ist, wobei $DOM := s_1\times\cdots\times s_n = sort(R_{n+i})$ die Obermenge bezüglich $I_{R_{i+n}}$ ist. $DOM$ ist also ein Kreuzprodukt von Sortenprädikaten und läßt sich aufgrund der Größe der Ausprägung vieler Sorten (z.B. integer) nicht sinnvoll anwenden. Daher wird verlangt, daß in einer konjunktiven Anfrage mit negiertem Literal $\neg R_{i+n}(x_{i+n})$ alle $v \in var(x_{i+n})$ auch in $x_j$ eines positiven Literals $R_j(x_j)$ für mindestens ein $j$ vorkommt. Der Wertebereich von $v$ wird somit nicht auf $sort(v)$, sondern auf den Bereich eingeschränkt, den $v$ aufgrund von $R_j$ annehmen kann. Anfragen dieser Art nennt man **sicher**:

**Definition 3.12 (Sicherheit einer Anfrage)** *Eine konjunktive Anfrage gemäß Definition 3.5 heißt sicher, wenn*

$$\bigcup_{i=1}^{n} var(x_i) \supseteq \bigcup_{i=1}^{m} var(x_{n+i})$$

.

Im folgenden werden nur sichere konjunktive Anfragen mit expliziter Gleichheit und ohne Konstante betrachtet. Definitionen 3.8 und 3.10 ergänzen sich um negierte Literale naheliegend. In der Normalform relationaler Ausdrücke (Definition 3.11) können nun anstelle von Relationen $R_i$ im Kreuzproduktteil auch Ausdrücke der Form $(DOM - R_i)$ auftauchen.

Nach [Ull88], Theorem 3.7 und 3.8, sind konjunktive Anfragen und die relationale Algebra für sichere Anfragen gleichmächtig. Im folgenden wird nun auf die Konstruktion von $DOM$ als relationaler Ausdruck und darauf folgend als Kasten eingegangen und somit $Konv$ um negierte Anfragen erweitert.

Angenommen, $DOM$ ließe sich korrekt bilden und ebenfalls angenommen, $red(Konv(q))$ würde einen korrekten relationalen Ausdruck liefern. Dann gilt Satz 3.4 weiterhin und auch der Beweis bleibt gleich, denn nach Definition von $DOM$ ist $DOM - R = \overline{R}$ und $x \in \overline{R} \Leftrightarrow x \notin R$, in Übereinstimmung mit Definition 3.7.

Die Konstruktion von $DOM$ als relationalen Ausdruck vollzieht sich wie folgt: Aufgrund der Sicherheit der Anfrage existiert zu jeder Variablen $v_k \in var(x_{n+j})$ eines negierten Literals $\neg R_{n+j}(x_{n+j})$ eine Gleichheit $v_k = v_{p_k}$, wobei $v_{p_k} \in var(x_{i_k})$ eine Variable eines positiven Literals $R_{i_k}(x_{i_k})$ ist. $DOM_{n+j}$ ist konstruierbar als $\prod_k \pi_{p_k}(I_{R_{i_k}})$ ($\prod$ als $\times$-Reihe). Begründung: Weil $v_k = v_{p_k}$ auch Sortengleichheit impliziert, ist $DOM_{n+j}$ eine Untermenge des maximalen $DOM$ als Kreuzprodukt der Sortenmengen. Andererseits ist diese Einschränkung nicht zu groß, da alle weggelassenen Elemente aufgrund der weiterhin bestehenden Einschränkung $v_k = v_{p_k}$ nachher (siehe Normalform eines relationalen Ausdrucks) ohnehin herausselektiert werden.

Zur Illustration der Konstruktion soll folgendes kleines Beispiel dienen:

$$
\begin{aligned}
q &= ans(X) \leftarrow r(X,Y)\ \&\ \neg s(Y). \\
&= ans(X) \leftarrow r(X,Y_1)\ \&\ \neg s(Y_2)\ \&\ Y_1 = Y_2. \quad \text{(q mit expliziter Gleichheit)} \\
&\rightsquigarrow I_{ans} = \pi_1(I_r \bowtie_{2=1} \overline{I_s}) \quad \text{(Konv(q)(I))} \\
&= \pi_1(\sigma_{2=3}(I_r \times \overline{I_s})) \quad \text{(Normalisierung)} \\
&= \pi_1(\sigma_{2=3}(I_r \times (DOM - I_s))) \\
&= \pi_1(\sigma_{2=3}(I_r \times (\pi_2(I_r) - I_s))) \quad \text{(Konstruktion von } DOM_r\text{)}
\end{aligned}
$$

Nun soll $Konv$ auf Anfragen mit negierten Literalen so erweitert werden, daß der entstehende Kasten reduziert den oben angegebenen relationalen Ausdruck erzeugt. Für die Differenz wird ein neuer K-Kasten erzeugt, der einen weiteren K-Kasten $DOM$ enthält. Ein K-Kasten erfüllt somit die Rolle einer Klammerung im relationalen Ausdruck. Siehe Abschnitt 3.2.6 für die Illustrierung des obigen Beispiels.

**Algorithmus 3.2 ($Konv$-Erweiterung für Negation)** *Im Schritt 1 wird jeder N-Kasten, der aus einem negierten Literal entsteht, als negiert markiert. Zwischen Schritt 4 und Schritt 5 wird folgendes eingefügt:*
*Ersetze jeden negierten N-Kasten $N$ über $\neg R_{n+j}$ durch einen K-Kasten $K$, der die Stellen $v_k$ von $N$ mit ihren Kanten übernimmt. $K$ enthält einen weiteren K-Kasten $DOM$ von dem aus eine gerichtete Kastenkante mit Markierung „-“ zu einem neu erzeugten N-Kasten $N$ in $K$ führt. $DOM$ hat dieselbe Stelligkeit wie $N$. Alle Stellen von $DOM$ sind mit den Stellen von $K$ mittels Projektionskanten verbunden. $DOM$ wird nun wie folgt gefüllt: Für jede Kante von einer Stelle $v_k$ an $K$ zu einer Stelle $v_{p_k}$ (Gleichheit) an Kasten $N'$ über $R_{i_k}$ erzeuge in $DOM$ einen einstelligen K-Kasten $S_{i_k}$ mit einer Kopie von $N'$ und einer Projektionskante von dessen Stelle $v_{p_k}$ nach außen zur Stelle von $S_{i_k}$.*

Diese kompliziert erscheinende Erweiterung operationalisiert nur jenes, was oben bei der Konstruktion von *DOM* gefordert wurde. Man beachte die Korrespondenz der Bezeichner, um sich von der Korrektheit des Algorithmus – daß also *Konv* den gewünschten relationalen Ausdruck erzeugt – zu überzeugen. Da der Kasten *DOM* mehrere Kästen nebeneinander enthält, wird auf diese Weise das Kreuzprodukt ausgedrückt (Fall *(4.4)* aus Definition 3.3).

Diese Kodierung negativer Anfragen ist also zwar im allgemeinen korrekt, in vielen Spezialfällen aber umständlich. Siehe dazu auch den Ausblick-Abschnitt 5.3. Insbesondere könnte man Konstanten, die an $N$ liegen, dort belassen, da sie niemals woanders gebraucht werden.

### 3.3.7 Zuteilung von Bindungsmustern

Jede Stelle bekommt zusätzlich eine erzwungene Bindung $\circ$ oder $\bullet$. Dies drückt eine Bedingung aus, nämlich daß *Stellenkanten nur eine* $\bullet$*-Stelle mit einer* $\circ$*-Stelle verbinden dürfen* und daß *jede* $\bullet$*-Stelle genau eine Kante haben muß*. Dies beschreibt die Semantik der Bindungen. Zu welchem Zweck sie eingesetzt werden, bleibt offen. Gedacht sind sie dafür, bei Relationenbezeichnern Eingabe-($\bullet$) von Ausgabe-($\circ$) Positionen zu unterscheiden. Dazu werden die erlaubten Bindungen immer insgesamt bezüglich eines Kastens betrachtet. Die erlaubten Bindungsmuster eines N-Kastens werden durch sein Relationenbezeichner-Merkmal aufgrund des Wissensquellenschemas (Exportschemas) festgelegt. Eine Wissensquelle bietet pro Relation nur bestimmte Bindungsmuster.

Der *Konv*-Algorithmus 3.1 muß erweitert werden, um die oben genannte Bedingungen herzustellen. Zunächst muß zwischen Schritt 3 und 4 eine Zuordnung von Bindungsmustern zu den N-Kästen und somit zu deren Stellen stattfinden. M-Kästen haben stets eine $\circ$-Stelle. Es gibt optimale und weniger optimale Zuordnungen. Eine vollständige Suche gewährleistet das Finden der optimalen Zuordnung. Mehr dazu in Kapitel 4.1. Der Schritt 4 in Algorithmus 3.1 muß geändert werden zu:

**Algorithmus 3.3 (Erweiterung: Konv mit Bindungsmustern)**

*4. Für jede Variable $v$ suche Stelle in $L_v$ mit $\circ$-Bindung*

*(a) Wenn es keine solche Stelle gibt, dann füge entsprechenden Sorten-M-Kasten ein mit einer $\circ$-Stelle $s$ und setze $s_v = s$.*

*(b) Wenn es genau eine solche Stelle gibt, setze $s_v$ gleich diese Stelle*

*(c) Wenn es mehrere solche Stellen gibt, setze $s_v$ gleich derjenigen von diesen, deren zugrunde liegender N-Kasten die kleinste Relation als Merkmal hat.*

*Erzeuge =-Kanten von $s_v$ zu jeder anderen Stelle aus $L_v$.*

Ein Sorten-M-Kasten liefert alle möglichen Terme einer Sorte. Eventuell ist dies aufgrund der theoretisch unendlichen vielen Elemente unpraktikabel. In einem solchen Fall ist die Konvertierung nicht möglich. Das Hinzufügen eines Sorten-M-Kastens ändert an der Korrektheit von *Konv* nichts, da zu der konjunktiven Anfrage nur explizit ein Konjunktionsglied hinzugefügt wird (Sortenprädikat), das aufgrund der Sortenbedingungen in Definition 3.5 zwangsweise erfüllt ist.

Man beachte, daß eventuell ○-○-Kanten erzeugt werden, im Gegensatz zur oben genannten Bedingung. Dies wird dadurch aufgefangen, daß ○-○-Kanten nur zwischen M-Kästen ausgeführt werden, die ohnehin ausschließlich ○-Stellen zulassen. Wohlgemerkt ist eine Materialisierung eines A-Kastens nicht möglich, wenn dieser ●-Stellen enthält.

Mehr zur Theorie der Bindungsmuster (Modussysteme) ist in [RSU95] und [CGH94] zu finden.

## 3.4 Anwendungen

In diesem Abschnitt werden Anfrageübersetzer für vier verschiedene Wissensquellen vorgestellt: Eine Oracle-Datenbank, Mathematica, eine HTML-Seite mit tabellarischem Aufbau und eine (objektorientierte) ObjectStore-Datenbank. Es wird jeweils erwähnt, *welche* semantischen Aktionen *wie* realisiert werden müssen, wie das Exportschema aussieht und welche Probleme auftreten. Diese vier Übersetzer wurden im Rahmen dieser Arbeit implementiert.

### 3.4.1 Oracle-Anbindung

Oracle ist ein relationales DBMS das mittels SQL ansprechbar ist. SQL ist normiert, so daß man die schon in Unterkapitel 3.2 entwickelten Aktionen verwenden kann. Es können hier alle Aktionen realisiert werden. Die speziellen Oracle-Erweiterungen (PL/SQL) sind dabei allerdings nicht nutzbar. Das Materialisieren eines A-Kastens muß noch speziell für Oracle implementiert werden. Details dazu werden in Kapitel 4 behandelt, so daß an dieser Stelle darauf verzichtet werden kann.

Eine relationale Datenbank ist eine „angenehme“ Wissensquelle. Zum einen ist das zugrundeliegende Modell konform mit dem einheitlichen Datenmodell in KOMET, so daß die Erstellung des Exportschemas einfach ist, zum anderen beherrscht sie sämtliche Aktionen der Übersetzungsregeln. Eine Datenbank läuft in der Regel als Server-Prozeß, so daß die Kommunikation und deren Aktivitäten losgekoppelt sind vom Übersetzer-Prozeß.

### 3.4.2 Mathematica-Anbindung

Mathematica ist ein Computer-Algebrasystem und als solches eine funktionale Wissensquelle. Hier sind also die Bindungsmuster wichtig. Es gilt das im Abschnitt 3.2.8 erwähnte. Kasten-Kanten-Regeln für die Negation beherrscht Mathematica allerdings nicht, ebenso wie die in Abschnitt 3.2.7 zusätzlich eingeführten Regeln, die das Kreuzprodukt und die Attributselektion beschreiben. Die Implementierung dieser drei semantischen Aktionen entfällt also.

Mathematica bietet mit dem API „MathLink“ eine Schnittstelle, um Anfragen von außen an das Programm heranzutragen. Es wird zu einem Mathematica-Prozeß ein sogenannter Link geöffnet. Über diesen Link werden Pakete geschickt. Ein Paket besteht aus einer Funktion mit ihren Argumenten, die ebenfalls Funktionen sein können. Dies erfüllt exakt die Anforderungen an eine funktionale Wissensquelle.

Das Exportschema einer funktionalen Wissensquelle beinhaltet als Relationen die einzelnen Funktionen. Das Ergebnis einer solchen Funktion steht standardmäßig an erster Position, de-

ren Bindung immer ∘ sein muß, die Funktionsargumente an den folgenden Positionen, deren Bindungen immer • sein müssen. Es gibt pro Relation also immer nur eine mögliche Bindung. Dies schränkt die erlaubten Anfragen natürlich stark ein. Die Definitionsbereiche sind Zahlen oder Zeichenketten. Beide Sorten sind nicht geeignet für ein Sortenprädikat, so daß grundsätzlich alle •-Stellen gebunden sein müssen. ∘-∘-Konflikte können jedoch behandelt werden, was interessante Anfragen ermöglicht, die Mathematica in einer einzigen Anfrage und ohne Hilfe des Übersetzers nicht bearbeiten könnte. So kann man z.B. anfragen, ob 1+2 = 2+1 ist:

$$ans() \leftarrow plus(X, 1, 2) \ \& \ plus(X, 2, 1).$$

oder ob 1+2 = 3 ist:

$$ans() \leftarrow plus(3, 1, 2).$$

Ein Problem ist, daß es für Mathematica keinen Unterschied zwischen der Funktion „Plus“ gibt und dem Konstruktor „Plus“, mit dem man z.B. Polynome zusammensetzt. Wenn man ein Polynom faktorisieren will, übergibt man in dem Paket nach der „Factor“-Funktion ein Polynom, d.i. eine Reihe von „Plus“ und “Power“-Ausdrücken. Mathematica benutzt Symbole, so daß etwa „Factor[1+x]“ anfragbar ist. Die Repräsentation einer solchen Anfrage, die als Ergebnis 1+x, also „Plus[1,x]“ hat, fällt im Mediatormodell schwer. „Plus“ muß auch als (von Mathematica interpretiertes, unausgewertetes) Funktionssymbol in komplexen Termen vorliegen, außerdem müssen die Symbole repräsentiert werden. Man könnte ein implizites Symbol definieren, indem man Polynome als Koeffizientenliste repräsentiert, was Mathematica mit einer kleinen Ergänzung auch interpretieren kann. Aber dann fallen Polynome mit mehr als einem Symbol weg.

So oder so ist es notwendig, Objekte (z.B. Polynome) als komplexe Terme darzustellen. In [Trc96] wird ein entsprechendes Mediatormodell vorgestellt. Im Rahmen dieser Arbeit konnte darauf jedoch nicht eingegangen werden, so daß nur elementare Typen einsetzbar sind, was die Anwendungen von Mathematica wesentlich einschränkt.

### 3.4.3 WWW-Anbindung

Eine HTML-Seite als Wissensquelle zu nutzen bereitet zwei Probleme: Zum einen haben diese Seiten, mit Ausnahme eher uninteressanter Verwaltungsinformationen, keinerlei einheitliche Struktur, also kein Schema und ein geradezu primitives Modell (es gibt nur eine Zeichenkette), zum anderen kann man sagen: Oracle konnte alles, Mathematica nur die Hälfte, eine HTML-Seite kann gar nichts.

Versuchsweise wurde ein Übersetzer geschrieben, der folgende Funktionalität bietet: Es wird angenommen, daß in einer Datei etwas Tabellenartiges vorhanden ist, wie etwa:

```
Aach-Muen.Bet N    840000 15.05.1996 Mi     990.00      1.02%
AEG                503800 15.05.1996 Mi     167.30      0.30%
AGIV               502820 15.05.1996 Mi      29.20      3.36%
Allianz Hold N     840400 15.05.1996 Mi    2650.00      0.30%
Altana             760080 15.05.1996 Mi     952.00      0.32%
        ...
```

Die einzelnen Spalten dieser Tabellen liegen an bestimmten Positionen. In jeder Zeile ist genau ein Tupel einer solchen Tabelle. Eine Tabelle fängt in der Datei ab einer bestimmten Zeile an und hört an einer bestimmten Zeile auf. Das Exportschema besteht aus einer Relation pro so strukturierter Datei, mit entsprechender Anzahl von Attributen, deren Sorte ausnahmlos *string* ist. Dies deckt natürlich nur einen kleinen Teil von Seiten im WWW ab, ist aber brauchbar auch für allgemeine Text-Dateien. Dieses Vorgehen ist sehr empfindlich gegenüber Änderungen in der Seite. Es kann somit nur als Demonstration verstanden werden. Insbesondere ist es selbstverständlich nicht möglich, einen einheitlichen Übersetzer für alle WWW-Seiten zu entwickeln. Es ließe sich aber überlegen, ob man eine Bibliothek zur Verfügung stellt, die Mustersuche in Text-Dateien unterstützt, oder ob man Programme wie „awk“ oder „grep“ einsetzt.

Der Übersetzer ist einfach zu erstellen. Nur die beiden elementaren Regeln, die Instantiierung und die Materialisierung, können realisiert werden. Die ganze Arbeit steckt in der Materialisierung. Hier muß man einen Cursor bereitstellen, der einzelne Werte eines Tupels (= Textzeile im Tabellenbereich) lesen und von Tupel zu Tupel springen kann. Die restliche Arbeit besorgt der Übersetzer selbst, indem er diesen Cursor steuert und die Kanten selbst abarbeitet, wie im Abschnitt 3.2.4 beschrieben wurde.

Interessanterweise läßt sich das Verhalten, keine der A-Kasten-Regeln anzuwenden, auch mit den Bindungsmustern realisieren. Man muß dazu einfach überall $\circ$-Bindungen verlangen, was hier ja auch der Wissensquelle entspricht: Eine Datei oder eine einfache WWW-Seite kann nur etwas darstellen, also ausgeben, und nichts einlesen und selektieren.

Man kann solche Dateien, auf die der Übersetzer zugreift, auch selbst erstellen. Auf diese Weise hat man eine minimale relationale Datenbank vor sich, mit der ganzen Flexibilität der Mediator-Anfragesprache. Am Beispiel der oben abgedruckten Datei, die aus einer real existierenden HTML-Seite extrahiert wurde, läßt sich z.B. anfragen:

$$ans(X,Y) \leftarrow Stock(X,\_,\_,Close,\_) \ \&\ Stock(Y,\_,\_,Close,\_).$$

Dies liefert die Namenpaare aller Aktien mit gleichem Schlußkurs. Jedes _ bezeichnet eine beliebige neue Varibale, die nirgendwo an anderer Stelle auftaucht. Sie wird auch anonyme Variable genannt.

### 3.4.4 ObjectStore-Anbindung

ObjectStore ist eine objektorientierte Datenbank, und als solche nicht mehr konform mit dem einheitlichen Datenmodell in KOMET. Daher folgt eine etwas ausführlichere Beschreibung des entsprechenden Übersetzers.

Ein häufiges Problem beim Entwickeln eines Anfrageübersetzers ist es, zu klären, wie man Anfragen an die Wissensquelle im Mediatormodell (einheitliches Datenmodell) repräsentieren kann. Dies wurde schon bei der Mathematica-Anbindung deutlich und ist bei der Anbindung einer objektorientierten Wissensquelle ein noch größeres Problem. Oracle bietet ein zum einheitlichen Datenmodell in KOMET etwa gleichmächtiges Modell. Eine Datei bietet ein deutlich schwächeres Modell. Zusammen mit Mathematica, das Terme verlangte, ist das objektorientierte Modell mächtiger als das KOMET-Modell. Der Unterschied liegt in der Integration von Methoden, der orthogonalen Nutzbarkeit mengenwertiger Attribute und

der Existenz von Objektidentitäten, welche mehrere Objektgleichheiten definierbar macht (siehe [LL95]). Im folgenden wird der Versuch unternommen, unter Verzicht der Abbildung von Objekt-Methoden, einen Anfrageübersetzer für objektorientierte Wissensquellen zu beschreiben, ohne Termausdrücke im Mediatormodell vorauszusetzen. Im Anschluß daran wird ObjectStore beschrieben und die Möglichkeit, es als Wissensquelle zu benutzen, dargelegt.

Folgendes kleine Beispiel soll benutzt werden, um die Technik zu illustrieren. Es gibt die beiden Klassen `Vorlesung` und `Note` mit den Strukturen (in C++ ähnlicher Notation):

```
class Vorlesung
    string name;
    Set_of<Note> noten;

class Note
    int wert;
    Vorlesung vorlesung;
```

Es gibt atomare Attribute (`Vorlesung::name` und `Note::wert`) und komplexe Attribute, bei letzteren einelementige (`Note::vorlesung`) oder mengenwertige (`Vorlesung::noten`). Natürlich sind auch Mengen über atomare Objekte möglich.

Anfragen an ein solches objektorientiertes Schema sollen in einer SQL-ähnlichen Sprache gestellt werden. Ohne die genaue Semantik dieser Sprache zu definieren, sollten die folgenden Beispiele verständlich sein. Dabei sind `Noten` und `Vorlesungen` sogenannte *Wurzeln*. Wurzeln sind Mengen von Objekten. Von ihnen ausgehend kann man navigierend per Pfadausdrücken (z.B. `n.vorlesung.name`) auf die aggregierten Werte eines Objektes in diesen Wurzeln zugreifen. `Noten` enthält alle Objekte der Klasse `Note` und `Vorlesungen` alle Objekte der Klasse `Vorlesung`. Bei der linken Anfrage läuft `n` über alle Noten. Von denjenigen, deren Wert 5 ist, sollen die Namen ihrer Vorlesungen projiziert werden. Bei der rechten Anfrage wird von der `Vorlesungen`-Wurzel ausgegangen (Bezeichner `v`). `n` läuft über alle Noten einer Vorlesung `v`. Es werden die Namen der Vorlesungen ausgegeben, bei denen eine Note mit Wert 5 existiert. Beide Anfragen sind in ihrem Ergebnis also identisch, denn sie liefern die Namen derjenigen Vorlesungen, in denen schon mal eine 5 geschrieben wurde.

```
select n.vorlesung.name          select v.name
from n in Noten                  from v in Vorlesungen, n in v.noten
where n.wert = 5                 where n.wert = 5
```

Bei manchen objektorientierten Datenbanken kann man Klassen als Wurzeln benutzen, bei anderen muß man selbst Wurzel-Container kreieren und sie füllen. Die Objekte in Wurzeln sind die persistent gemachten Objekte der Datenbasis. Man beachte, daß mengenwertige Attribute auch wie Wurzeln nutzbar sind.

Es soll ein Übersetzer gebaut werden, der ObjectStore anbindet. Es ist sinnvoll, dies in zwei Schritten zu vollziehen. Zunächst wird in eine allgemeine Anfrageform gemäß der obigen Beispiele übersetzt und dann von dieser in ObjectStore-Anfragen. Auf diese Weise kann man den ersten Schritt wiederverwenden, wenn ein anderes objektorientiertes DBMS angebunden werden soll. Ähnlich ist auch bei SQL und Oracle vorgegangen worden. In diesem Fall besteht jedoch das Problem, daß es *keinen* Standard gibt.

Die Übersetzung eines objektorientierten Schemas in ein Exportschema läuft wie folgt:

1. Jede Klasse K ist eine Relation K. Jedes Attribut der Klasse K ist ein Attribut der Relation K. Ist das Attribut atomar, so wird dieses Attribut direkt übernommen. Ist das Attribut komplex mit Klasse T, so ist das entsprechende Attribut in der Relation ein Identifikator (kurz ID) auf T (Fremdschlüssel). Zusätzlich erhält Relation K als erstes Attribut (Surrogat) eine ID auf K. Die Attributnamen werden übernommen.

2. Jede Wurzel W als Menge über Objekte der Klasse K ist eine Relation mit genau einem Attribut, einer ID auf K.

3. Für jeden genutzten mengenwertigen Typ T über Klasse K gibt es eine Relation T mit zwei Attributen: Das erste ist eine ID über T, das zweite eine ID über K. Im Gegensatz zu den beiden Fällen oben ist hier das erste Argument *kein* Schlüssel der Relation, denn für jedes tatsächlich vorkommende mengenwertige Attribut mit Typ T gibt es eine T-ID, für jedes Element in dieser Menge gibt es eine K-ID. Eine T-ID kommt also mehrmals vor.

Es sind alle Bindungsmuster erlaubt, allerdings macht es wenig Sinn, z.B. einen Objektidentifikator vorzugeben. Im obigen Beispiel gibt es demnach folgende Relationen:

```
Vorlesung(id: Vorlesung-ID, name: string, noten: SetofNote-ID)
Note(id: Note-ID, wert: int, vorlesung: Vorlesung-ID)
SetofNote(id: SetofNote-ID, nid: Note-ID)
Vorlesungen(vid: Vorlesung-ID)
Noten(nid: Noten-ID)
```

Methoden werden nicht betrachtet. Man könnte in Spezialfällen eine Methode als Attribut ansehen, im allgemeinen ist dies nicht möglich. Man könnte Methoden auch als Relationen mit funktionalen Bindungsmustern wie bei Mathematica realisieren.

Vererbungen werden schon ausgeführt repräsentiert. Ist also ein Student mit einer Matrikelnummer `matr` eine Person (Oberklasse), welche das Attribut `name` hat, so sind `name` *und* `matr` Attribute von `Student`.

Diese Art der Kodierung (siehe auch [MYK+93] und [LL95]) wurde aus zwei Gründen gewählt: Zum einen sollen möglichst alle Anfragen formulierbar sein. In der Tat wurde sogar über dieses Ziel hinausgegangen, wie sich bald zeigen wird. Zum anderen wurde versucht, Redundanzen zu verhindern. Es hätte die Alternative gegeben, die Objektidentifikatoren wegzulassen und mengenwertige Typen nicht als neue Relation zu kodieren. Dies hätte bedeutet, daß z.B. die Noten einer Vorlesung und die Noten aus der Noten-Wurzel zwar gleich, aber nicht identisch wären. Würde man beide Relationen im Mediator materialisieren, so wäre jede Note zweimal gespeichert, wobei eine (Teilweise-) Materialisierung ist durchaus sinnvoll zur Optimierung des Mediator-Programms wäre.

Die beiden Beispiel-Anfragen können im Mediator nun wie folgt gestellt werden:

$$\begin{aligned} ans(Name) &\leftarrow Noten(N)\ \&\ Note(N,5,V)\ \&\ Vorlesung(V,Name,\_). \\ ans(Name) &\leftarrow Vorlesungen(V)\ \&\ Vorlesung(V,Name,SN)\ \& \end{aligned}$$

$$SetofNote(SN, N)\ \&\ Note(N, 5, \_).$$

Hier fällt zunächst einmal die Kompliziertheit dieser einfachen Anfragen auf. Die Kastendarstellung ergibt sich aus den Mediator-Anfragen wie üblich. Bei der Herstellung der äquivalenten OO-Anfrage muß darauf geachtet werden, daß die Pfadausdrücke richtig konstruiert werden. Dabei sind nicht alle Kanten Teil eines Pfadausdrucks, sie können auch einen üblichen Verbund zwischen zwei Relationen ausdrücken. Die Konstruktion der Pfadausdrücke wird vorgenommen, indem an jeder Stelle vermerkt wird, wie sie erreichbar ist. Der Pfad wird dabei von hinten nach vorne erweitert, bis eine Wurzelvariable am Anfang steht.

Die Aktion zu der Instantiierungs-Regel, die einen N-Kasten in einen A-Kasten wandelt, sieht fallunterscheidend wie folgt aus:

1. Ist der N-Kasten eine Wurzel mit Namen N, so füge „n in N“ zu dem from-Teil der Anfrage. Gib der (einzigen) Stelle den Namen „n“.

2. Ist der N-Kasten eine Klasse, so gib den Stellen die Namen der entsprechenden Attribute der Klasse.

3. Ist der N-Kasten ein Mengentyp, so mache nichts.

Die Aktion zu der (zweiten) Regel, die zwei A-Kästen A1 und A2, die mit einer Kante jeweils an den Stellen a1 und a2 verbunden sind, zu einem neuen A-Kasten A umwandelt, sieht wie folgt aus:

1. Wenn A1 eine Wurzel und A2 eine Klasse oder ein Mengentyp sind, dann erweitere jeden Namen der Stellen aus A2 durch Voranstellen eines „a1.“, wobei „a1“ für den Namen der Stelle a1 steht. a2 muß hier die erste Stelle von A2 sein, ansonsten ist eine Übersetzung nicht möglich.

2. Wenn A1 und A2 Wurzeln sind, dann füge „a1 = a2“ zum where-Teil der Anfrage.

3. Wenn A1 und A2 beides Klassenrelationen sind, dann:

   (a) Wenn a1 erste Stelle von A1 ist, a2 aber nicht von A2, dann erweitere jede Stelle aus A1 um vorangestelltes „a2.“. Dieser Fall ist symmetrisch bzgl. A1 und A2.

   (b) Wenn a1 und a2 beides nicht-erste Stellen sind, dann füge „a1 = a2“ zum where-Teil. Hier findet also ein Verbund statt und keine Pfaderweiterung.

   Der Fall, daß sowohl a1 als auch a2 erste Stellen sind, darf nicht auftreten, da Klassen ohne Vererbungshierarchie disjunkt sind. Die Antwort wäre immer leer.

4. Wenn A1 eine Klasse ist und A2 ein Mengentyp, dann:

   (a) Wenn a1 erste Stelle von A1 ist, a2 aber nicht von A2, dann erweitere jede Stelle aus A1 um voransgestelltes „a2.“. Dies ist identisch zum entsprechenden Teil im vorhergehenden Fall.

(b) Wenn a2 erste Stelle von A2 ist, a1 aber nicht von A1, dann füge „a in a1“ zur from-Liste, wobei a ein neuer Name ist, und erweitere die zweite Stelle in A2 um vorangestelltes „a.“.

(c) Wenn a1 und a2 beides nicht-erste Stellen sind, dann füge „a1 = a2“ zum where-Teil der Anfrage.

Treffen zwei Mengentyp-Kästen aufeinander, so wird entsprechend dem letzten Fall vorgegangen. Weitere Fälle dürfen nicht auftreten. Die Aktionen zu den anderen Regeln werden nun naheliegend definiert und sollen hier nicht weiter ausgeführt werden. Es muß noch geklärt werden, welchen Status (Wurzel, Klasse, Menge) der Ergebniskasten einer Regel hat. In der Tat hängt der Status nicht global vom Kasten, sondern von jeder einzelnen Stelle ab. Ist eine Stelle anfangs eine Stelle an einer Klassenrelation gewesen, so bleibt sie dies bis zum Ende. Also nicht die Kästen haben einen Status, sondern die Stellen.

Die Abbildung 3.5 zeigt die Übersetzung des ersten der beiden Beispielanfragen. „Entsprechend Fall 3(b)“ soll heißen, daß dort eine Kante zwischen einem A- und einem M-Kasten erscheint, der Fall 3(b) aber zwei A-Kästen erwartet. Die Aktion ist aber identisch.

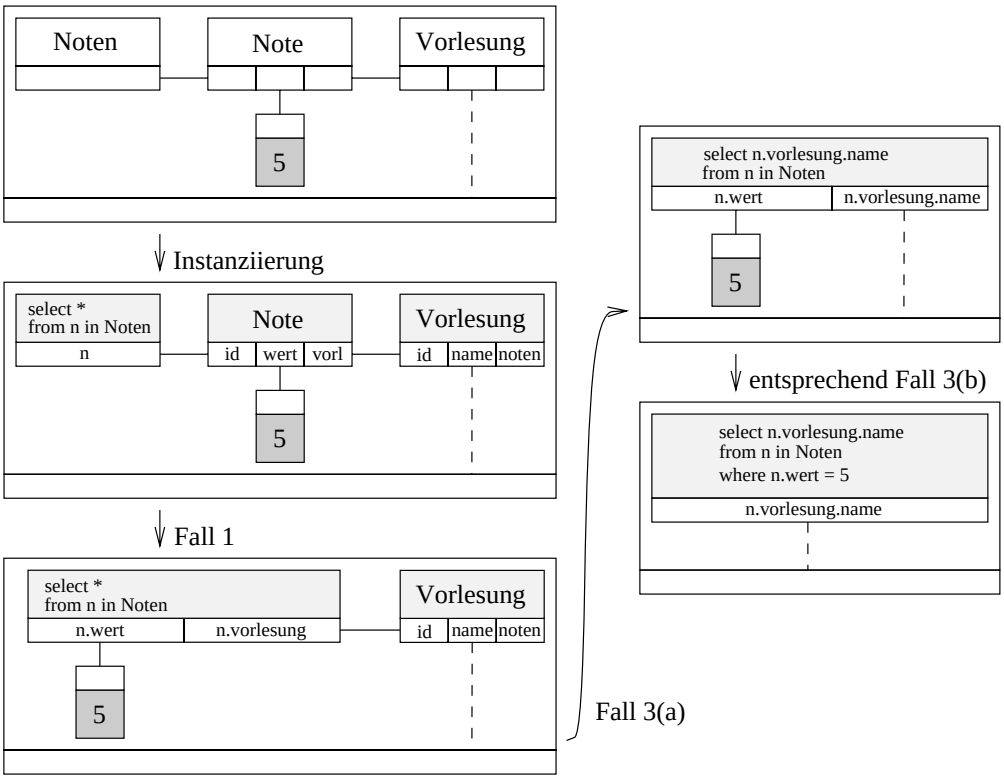



Abbildung 3.5: Übersetzung des ersten Beispiels

Die im Algorithmus erwähnten Ausnahmen führen dazu, daß Anfragen formulierbar sind, die nicht beantwortet werden können, z.B. wenn in der Anfrage keine Wurzelrelation steckt, oder wenn von einer solchen nicht alle genutzten Stellen erreichbar sind. Ein weiteres Problem ist, daß eventuell eine Stelle (also ein Attribut) von zwei oder gar mehreren Wurzeln mittels Pfadausdrücken erreichbar ist. Nur einer dieser Wege darf genutzt werden.

Im folgenden wird schließlich die konkrete Anbindung an das OO-DBMS ObjectStore betrachtet. ObjectStore ist eine C++-Erweiterung, die es erlaubt, Collections, also Container von C++-Objekten, persistent zu machen. Diese persistent gespeicherten Wurzeln können mit einem eindeutigen Namen wieder angesprochen werden. Nur über solche Collections können Anfragen gestellt werden. Eine Anfrage wird direkt in den Programmcode geschrieben. Sei `col` eine Collection aus der Datenbank `db` über den Elementtyp `elem*`, dann kann man mit

```
res = col.query("elem*", "querystring", db)
```

in `res` (ebenfalls Collection über `elem*`) alle Elemente aus `col` erhalten, die den Ausdruck `querystring` erfüllen. Dieser Ausdruck muß einen Wahrheitswert liefern. Er besteht aus Junktionen von Pfadausdrücken, beginnend bei Attributen von `elem`. Beispiel:

```
fuenfer = Noten.query("Note*", "wert == 5", db);
```

Mengenwertige Attribute können über eingebettete Ausdrücke untersucht werden (navigierender Zugriff):

```
fuenfer = Vorlesungen.query("Vorlesung*", "noten [: note == 5 :]", db);
```

Diese Anfragenotation hat zwei entscheidende Nachteile[7]: Zum einen können keine Projektionen ausgedrückt werden. Um die *Namen* dieser Vorlesungen zu erhalten, muß man nachträglich projizieren. Zum anderen ist der Name der (immer nur einzigen) Wurzel fest im Programm kodiert; er kann nicht als String übergeben werden. Diese Eigenschaft macht ObjectStore nahezu ungeeignet als Wissensquelle, an die man dynamisch Anfragen stellen will. Man kann dieses Problem nur umgehen, indem man für jede einzelne Relation im Exportschema eine eigene Befehlszeile vorsieht und mittels einer case-Anweisung zur Laufzeit zu der richtigen Zeile springt und nachträglich noch die Projektionen durchführt. Für eine andere ObjectStore-Datenbank müßte das gleiche neu implementiert werden. Dies ist ausgesprochen umständlich. Der Grund ist, daß die Anfragemechanismen nur *embedded* sind, nicht *dynamisch* (im Sinne von SQL).

Die Übersetzung der erzeugten SQL-ähnlichen Anfragerepräsentation in eine ObjectStore-Anfrage gestaltet sich also schwierig. Das fängt schon dort an, wo in der Anfrage mehr als eine Wurzel steckt. ObjectStore muß solche Verbunde zwischen zwei Wurzeln in zwei Anfragen teilen und den Vergleich explizit in einer Schleife ausführen. Im Prinzip ist dies das übliche Zurückweisen einer Regelaktion.

[7] Betrachtet wurde das nackte ObjectStore-Produkt, Release 3.0, Dezember 93

# Kapitel 4

# Implementierung eines Anfrageübersetzers

In diesem Kapitel wird zum einen die Programmbibliothek beschrieben, die als Rahmenwerk zur Realisierung eines Anfrageübersetzers benutzt werden kann, und zum anderen eine Anleitung gegeben, wie man beim Entwurf eines Übersetzers vorgehen kann. Dies wird am Beispiel des Oracle-Übersetzers Schritt für Schritt demonstriert.

## 4.1 Aufbau der Bibliothek

Die Bibliothek ist in C++ geschrieben und hat den Namen *Ikarus*[1]. Die Implementierung der Konstrukte weicht an wenigen Stellen von dem bislang beschriebenen Konzept ab:

- Die Reihenfolge der Regeln ist festgelegt und vom Entwickler nicht bestimmbar. In allen Versuchen hat sich gezeigt, daß dies sinnvoll ist. Außerdem kann man auf diese Weise noch diverse Heuristiken in der Regelauswahl realisieren, die mit einer reinen Aneinanderreihung nicht auszudrücken wären.
- Es ist möglich, eine Regelaktion nicht grundsätzlich zurückzuweisen bzw. nicht implementiert zu haben, sondern zur Laufzeit in Abhängigkeit von der vorliegenden Eingabe eine Regelbearbeitung zu unterbinden. Allerdings wurde in den beschriebenen Anwendungen von dieser Möglichkeit nie Gebrauch gemacht.
- Es wurde keine allgemeine Umgebung zur Graphentransformation implementiert, um die in Abschnitt 3.2.3 beschriebenen Vorweg-Transformationen zu ermöglich. Einzig unterstützt wird zur Zeit die dort beschriebene Transformation eines Kastens in eine Kante.

Die Kastendarstellung einer Anfrage bietet nicht nur eine angenehme graphische Illustration einer Übersetzung, sondern deutet auch direkt auf die verwendbaren Datenstrukturen hin.

[1] Dies ist keine Abkürzung. Ikarus war der Sohn Daedalus', und Daedalus ist der Name der Mediatorshell aus [Kul95].

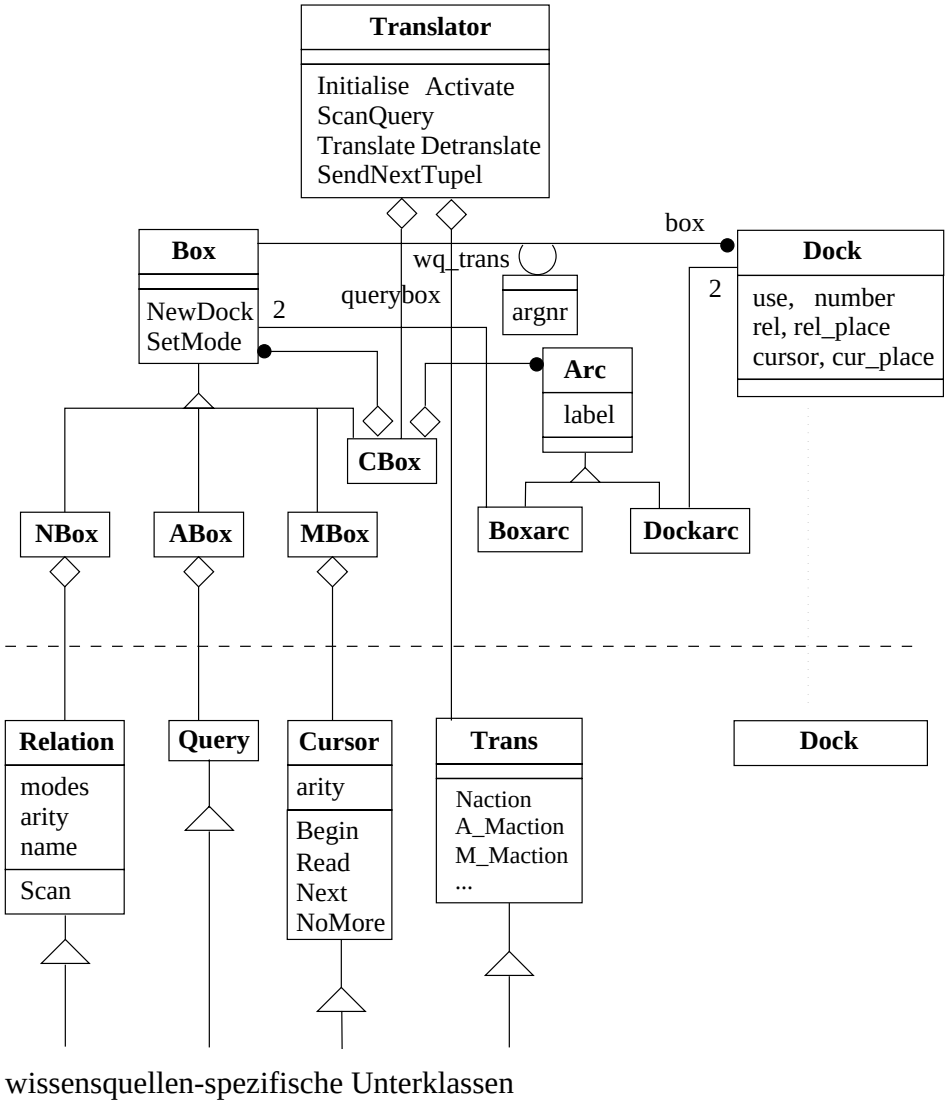



Abbildung 4.1: Objektmodell von Ikarus

Daher ist es nicht überraschend, die schon bekannten Begriffe auch in der Implementierung wiederzufinden. Das Objektmodell in OMT-Notation [RBP+91] zeigt Abbildung 4.1. Ein Kasten heißt nun **Box**, die Kante **Arc** und eine Stelle **Dock**. Die Klassen sind in Abbildung 4.1 zweigeteilt: Oben sind die internen Klassen dargestellt, die diejenigen Objekte beschreiben, die für die Übersetzungsorganisation notwendig sind. Zentral ist dort der *Translator*, der die steuernden Methoden *Translate, Detranslate* und *Activate* beinhaltet. Unten sind die Klassen gezeichnet, mit denen der Nutzer der Bibliothek zu tun hat. Diese Klassen *Relation, Query, Cursor* und *Trans* sind Basisklassen, von denen der Nutzer abgeleitete Klassen implementiert, die die wissensquellenspezifischen Aktionen und Datenstrukturen enthalten. *Trans* enthält den Code für die Aktionen, *Cursor* (Attribut eines M-Kastens) implementiert den einheitlichen Zugriff auf die Antwortdaten, *Query* beschreibt das Anfrageattribut eines A-Kastens, dessen Gestaltung völlig von der Wissensquelle abhängt. *Relation* schließlich

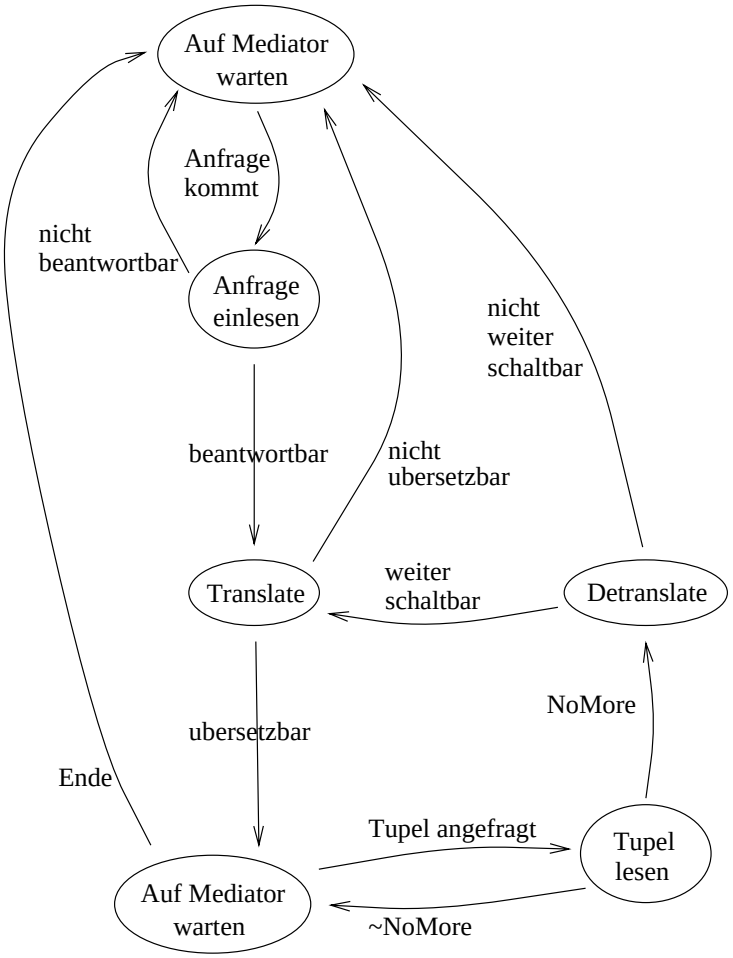



Abbildung 4.2: Dynamisches Modell der Hauptroutine in Ikarus

beinhaltet die Exportschema-Repräsentation einer Relation. Bislang gab es in den Anwendungen noch keine wissensquellenspezifische Ableitung von dieser Klasse. Eine solche wird dann sinnvoll, wenn spezielle Informationen nötig sind. Zur Zeit ist das Exportschema aber noch einheitlich und unabhängig definiert.

Zusätzlich gibt es noch die Klasse *Dock*, mit der der Nutzer zu tun hat. Im Verlauf des Übersetzungsprozesses bleibt die Menge der Stellen konstant, imGegensatz zu den in Kapitel 3.2 gezeigten Verbildlichungen. Jede Stelle gehört zu einem Zeitpunkt genau zu einem Kasten (Attribut *box*). Mit Ausnahme der Stellen an den Konstanten (M-Kästen) in einer Anfrage gehört jede Stelle anfangs zu einem N-Kasten und somit zu einer Relation (*rel* der Nummer *number* am Indexort *rel_place*). Diese Werte werden auch beim Übergang zu einem A-Kasten beibehalten. Man braucht sie, um eine Anfrage zu bilden. Erst beim Materialisieren eines A-Kastens bekommen die beteiligten Stellen einen Cursor (*cursor* am Indexort *cur_place*) zugeordnet, den sie nicht mehr verlieren, auch wenn sie – etwa durch Verbinden zweier M-Kästen – ihren Kasten wechseln. Um den Wert an einer Stelle auszulesen, muß man also dieses Cursor-Attribut nutzen.

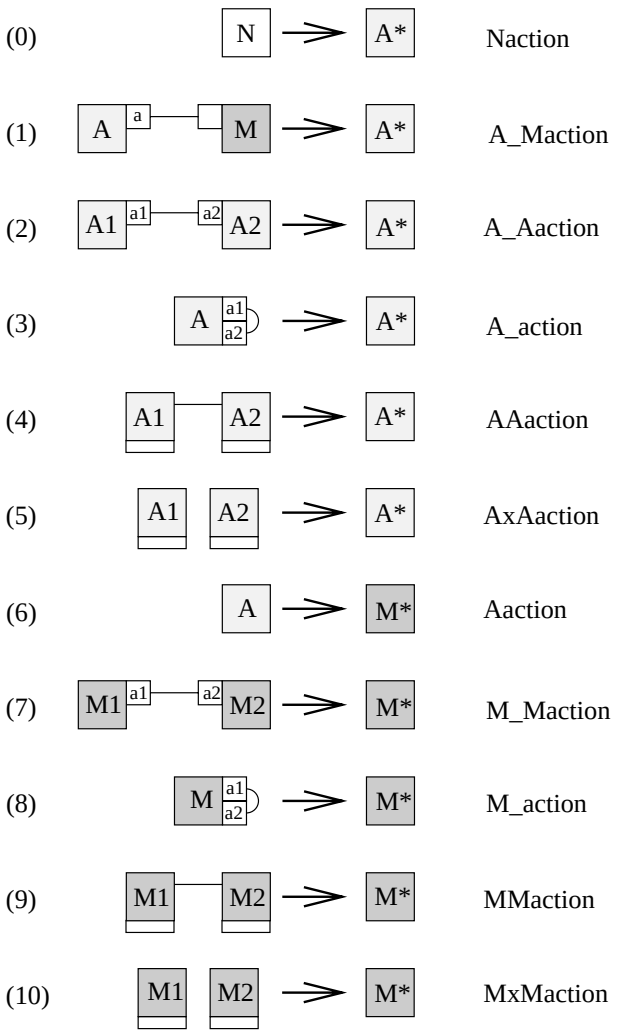


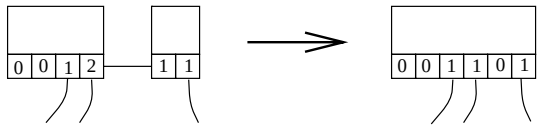

Abbildung 4.3: Die Regeln mit den Namen der Aktionen

Abbildung 4.2 zeigt den groben Ablauf der Interaktion mit dem Mediator, die in der Methode *Activate* in *Translator* implementiert ist. *Translator::Translate* ist die Methode, die die Regelaktionen in einer festgelegten Reihenfolge, in Abhängigkeit von der Anfrage-Eingabe, aufruft. Ihr Ergebnis ist ein Cursor auf die Antwort der Wissensquelle. Das Lesen der Tupel erfolgt per Cursor-Durchlauf. Liefert der Cursor keinen Wert mehr, wird *Detranslate* aufgerufen. *Detranslate* macht die Übersetzungsschritte wieder rückgängig, solange bis ein M-Kasten „wiederaufgedeckt“ wird, dessen Cursor fortschaltbar ist. Dies wird getan und wieder von neuem übersetzt. Das ist der Prozeß, der in Abschnitt 3.2.4 schon beschrieben wurde.

In der Methode *Translate* werden die Aktionen in Abhängigkeit von den vorliegenden Kanten und Kästen in dem *querybox*-Attribut aufgerufen. *querybox* ist der Anfrage-K-Kasten. Man bedenke, daß dieser ebenfalls K-Kästen enthalten kann (bei Negationen), die vorher mittels *Translate* bearbeitet werden müssen. Die Aktionen-Methodennamen aus *Trans* sind mit ihren Parametern in Abbildung 4.3 zusammengefaßt. Die Routine *Translate* läuft wie folgt ab:

```
Wandle alle N-Boxen in A-Boxen mit Naction
aktion = 1
kante = Erste Stellenkante in der Anfrage
Solange aktion <= 11
     Wenn aktion = 4, dann
          Für alle Kastenkanten zwischen zwei A-Kästen
               führe AAaction damit aus, wenn implementiert
          aktion = aktion + 1
     Wenn aktion = 5, dann
          Suche zwei A-Kästen ohne Stellenkanten
          Wenn es diese gibt, führe AxAaction damit aus
          ansonsten oder wenn AxAaction nicht implementiert
               aktion = aktion + 1
          Gehe an den Anfang der Schleife
     Wenn aktion = 6, dann
          Suche A-Kasten ohne •-Stelle
          Wenn es diesen gibt, führe Aaction damit aus
          aktion = 1
          kante = Erste Stellen-Kante in der Anfrage
          Gehe an den Anfang der Schleife
     Wenn aktion = 9, dann
          Für alle Kastenkanten zwischen zwei M-Kästen
               führe MMaction damit aus
          aktion = aktion + 1
     Wenn aktion = 10, dann
          Suche zwei M-Kästen ohne Stellenkanten
          Wenn es diese gibt, führe MxMaction damit aus
          ansonsten aktion = aktion + 1
          Gehe an den Anfang der Schleife
     Wenn aktion = 11, dann
          Wenn noch ein A-Kasten vorhanden ist, dann brich erfolglos ab
          Liefere Cursor des nun alleinig gebliebenen M-Kastens als Ergebnis
     Wenn kante zur Prämisse von aktion paßt, dann
          Führe aktion mit kante aus, falls implementiert
          kante = nächste Stellenkante
          Wenn keine nächste Kante vorhanden, dann
               aktion = aktion + 1
               Gehe an den Anfang der Schleife
          ansonsten
               aktion = 1
               Gehe an den Anfang der Schleife
     kante = nächste Stellenkante
     Wenn keine nächste Kante vorhanden, dann
          aktion = aktion + 1
          kante = Erste Stellen-Kante in der Anfrage
```

Jede Aktion löscht die Kante und/bzw. Kästen ihrer Regel-Prämisse aus der Anfragedarstellung und fügt die Conclusio mitsamt des neuen Attributs an die passende Stelle ein. Dieses Verschmelzen zweier Kästen zu einem neuen Kasten geschieht schematisch wie folgt:




Für den entstehenden Kasten werden einfach alle Stellen übernommen. An jeder Stelle wird vermerkt (Attribut *use*), wieviele Kanten noch an ihr sind. Ist dieser Zähler auf Null, so bedeutet dies, daß die Stelle zu ignorieren ist. Die Stellen werden nicht gelöscht, weil sie beim Zurückübersetzen (*Detranslate*) wieder gebraucht werden. Aus diesem Grund werden auch die Ursprungskästen nicht gelöscht, sondern bei der erzeugten Box vermerkt. *Detranslate* hat damit die notwendigen Informationen zur Verfügung.

Die Aktionen über M-Kästen (*M_Maction, M_action, MxMaction* und *MMaction* aus *Trans*) sind schon fertig programmiert, d.h. der Benutzer der Bibliothek braucht sie nicht zu implementieren. Es macht auch keinen Sinn, sie zu überschreiben, da sie allein auf den allgemeinen Methoden von *Cursor* basieren. Die Semantik der Cursor-Methoden ist die folgende:

| | | |
|---|---|---|
| { } | Konstruktor | { *Begin()* } |
| { } | *Begin()* | { *NoMore()*, schaltet auf ersten Wert } |
| { i $\leq$ Stelligkeit } | *Read(i)* | { liefert Wert an i-ter Stelle, schaltet nicht weiter } |
| { $\neg$*NoMore()* } | *Next()* | { *NoMore()*; schaltet auf nächsten Wert } |
| { } | *NoMore()* | { liefert True wenn *Read()* nicht definiert ist } |

Exemplarisch wird Methode *M_Maction* vorgeführt. Sie erhält zwei Cursor und liefert einen neuen Cursor für den Ziel-M-Kasten. Dieser neue Cursor ist aus der Klasse *Join_Cursor*. *Join_Cursor* erhält zwei Cursor und startet diese. Ein *Read()* liefert dann ein definiertes Ergebnis, wenn beim sukzessiven Iterieren beider Quellcursor ein Wert gefunden wurde, der bei beiden gleich ist. Ansonsten gilt *NoMore()*. Dies realisiert die geforderte Join-Semantik aus Definition 3.3. Siehe dazu auch Abschnitt 4.2.3. Entsprechend haben die anderen M-Aktionen auch ihren speziellen Cursor, der die geforderte Semantik realisiert.

Bevor der Übersetzungsvorgang gestartet werden kann, müssen den einzelnen Stellen noch Bindungen zugewiesen werden. Dies geschieht auf einfachste Weise: Es werden für die N-Kästen alle (aufgrund der Ihnen zugrunde gelegten Relationen) möglichen Bindungsmuster ausgetestet, indem die Kosten für ihre Übersetzung berechnet werden. Die Stellen an den M-Kästen bekommen eine $\circ$-Bindung. Die Bindungskonstellation mit den geringsten Kosten wird genommen. Die Kosten einer Konstellation werden dabei wie folgt berechnet: Die Kosten sind am Anfang gleich 1. Wird eine Konstellation gefunden, deren Endkosten 1 bleiben, so kann man diese nehmen, denn eine bessere ist nicht zu finden. Die Stellen werden partitioniert nach ihrer Variablenzugehörigkeit. Die Stellen, die zu einer Variablen in einer Anfrage gehören, müssen untereinander transitiv mit =-Kanten verbunden werden. Dabei ist es das Ziel, daß genau eine dieser Stellen eine $\circ$-Bindung und alle anderen $\bullet$-Bindungen haben. In diesem Fall gibt es Kanten von dieser $\circ$-Stelle zu jeder der $\bullet$-Stellen. Es können zwei abweichende Fälle auftreten: a) Es gibt keine $\circ$-Stelle. In diesem Fall muß ein Sortenprädikat eingefügt werden, dessen Stelle als $\circ$-Stelle dienen kann. Die Kosten werden dabei mit

der Größe der Ausprägungsmenge dieser Sorte multipliziert. Ist sie zu groß oder unendlich, wird diese Bindungskonstellation zurückgewiesen. b) Es kann mehr als eine o-Stelle auftreten. In diesem Fall werden für jede so entstehende o-o-Kante die Kosten multipliziert mit dem Produkt der Größen der beiden zugrundeliegenden Kästen[2]. Dies geschieht, weil eine solche Kante nicht von der Wissensquelle behandelt werden kann. Der Übersetzer muß sie bearbeiten mit einer *M_Maction*. Wohlgemerkt: Eine *A_Aaction* mit einer Kante, die einen o-o-Konflikt hat, wird vom Übersetzer erst gar nicht aufgerufen!

Dieser Algorithmus ist natürlich nicht sehr effizient; er ist exponentiell in der Anzahl der erlaubten Bindungsmuster. Jedoch sind sowohl die Anzahl der Prädikate (also Kästen) als auch die Anzahl der Variablen in einer Anfrage gering. Eine Heuristik bei der Suche nach der besten Bindungskonstellation wäre es, von der oder den optimalen Bindungen für eine gegebene Anfrage auszugehen und von dort aus die nächst beste *mögliche* zu suchen. Um die Suche potentiell erschöpfend zu gestalten, muß man aber nicht nur von den besten Wunsch-Konstellationen ausgehend suchen, sondern auch von den zweitbesten, drittbesten, etc. Erst wenn eine mögliche Konstellation gefunden wurde, deren Kosten geringer sind als die der aktuellen x-besten, kann mit der Suche abgebrochen werden.

Bei Anfragen mit Negationen wird außerdem vorher noch geprüft, ob die Anfrage sicher ist, ob also jede Variable in negierten Literalen auch in positiven Literalen vorkommt. Die Konstruktion der Kastendarstellung geschieht dann mit K-Kästen wie in Abschnitt 3.2.6 beschrieben.

## 4.2 Nutzung der Bibliothek

Um einen neuen Anfrageübersetzer zu entwickeln, kann man nach folgenden Schritten vorgehen:

1. Entwirf das Exportschema. Dies ist eventuell kein leichtes Unterfangen, wie schon am Beispiel der objektorientierten Wissensquellen klar geworden ist. Man muß sich überlegen, wie man im Mediator eine Anfrage an die Wissensquelle stellen kann. In vielen Fällen ist dies aber eindeutig, wie z.B. für alle relationalen Datenbanken.

2. Schaue, ob es schon einen Übersetzer für diese Wissensquelle gibt. Für alle Oracle-Datenbanken reicht es z.B., nur das Exportschema neu zu entwerfen. Man lädt dies in den Oracle-Übersetzer und ist schon fertig. Wenn noch kein Oracle-Übersetzer vorhanden ist, aber ein anderer SQL-Übersetzer, dann kann man einiges widerverwenden. Im folgenden wird aber angenommen, daß nichts wiederverwendet werden kann.

3. Leite von den Klassen *Query, Cursor* und *Trans* neue Klassen ab. Überlege dabei, welche Attribute eine Anfrage an diese Wissensquelle hat und welche Datenstrukturen Cursor dieser Wissensquelle benötigen. Implementiere jede semantische Aktion (in *Trans*), die die Wissensquelle behandeln kann, zumindest aber die Aktionen *Naction* und *Aaction*. Die Methoden in *Cursor* müssen unter Beachtung ihrer Semantik alle implementiert werden. *Query* hingegen hat keine Methoden, und deren Objekte dienen als reine Datenhalter.

[2]Die Größe einer Relation muß also im Exportschema stehen.

4. Erstelle die *main*-Routine. In ihr müssen eventuell spezielle Daten gelesen oder übergeben werden.

Die folgenden Unterabschnitte behandeln diese Schritte detailierter. Ist für eine Wissensquelle *WQ* eine Klasse *Klasse* abgeleitet worden, so wird sie im folgenden *WQ_Klasse* genannt. Als Beispiel wird der Oracle-Übersetzer herangezogen.

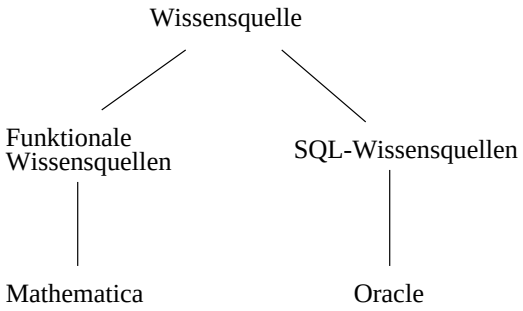
### 4.2.1 Erstellung eines Exportschemas

Der Entwurf des Exportschemas ist unter Umständen schwierig, wie schon in Abschnitt 3.4 deutlich wurde. Er wird hier nicht mehr behandelt, zumal es keine allgemeine Anleitung dafür gibt. Im Fall eines Oracle-Übersetzers ist die Erstellung jedoch sehr einfach. Jede Relation in der Datenbank ist eine Relation in dieser Domäne. Die elementaren Datentypen werden naheliegend aufeinander abgebildet. Der Oracle-Datentyp *Date* wird als String übersetzt. Angenehmerweise bietet schon Oracle diese Konvertierung an. Andernfalls müßte man sie in der Methode *Read()* der Klasse *Oracle_Cursor* (s.u.) realisieren. Diese Datendarstellungs-Konvertierung wurde in dieser Arbeit wie schon erwähnt nicht behandelt.

Es existiert eine lesbare Darstellung (`.dom`-Dateien) des Exportschemas für den Mediator in KOMET, siehe [Trc96]. Um den Codeumfang für den Übersetzer zu begrenzen, wurde eine andere, einfachere Darstellung (`.exp`-Dateien) entwickelt. Das Exportschema in dieser Darstellung wird von der Methode *Initialize()* in *Translator* geladen. Die Methode *Scan()* in *Relation* liest eine Relation ein. Ein Zusatzprogramm *convert* konvertiert `.dom`- in `.exp`-Dateien.

### 4.2.2 Wiederverwendung von Übersetzerteilen

Es war ein Ziel, Übersetzer möglichst einfach und schnell erstellen zu können. Dies wird insbesondere erreicht, wenn man schon einmal erstellte Teile wiederverwenden kann. Dazu wird eine neue Wissensquelle mit anderen Wissensquellen verglichen. Je standardisierter eine Wissensquelle ist, desto eher läßt sie sich in eine zweistufige Hierarchie einordnen, wie etwa die folgende:




Schon das „Können“ einer beliebigen Wissensquelle (Wurzel) kann teilweise wiederverwendet werden, nämlich die Aktionen über M-Kästen mit ihren speziellen Cursorn, wie im vorherigen Unterkapitel beschrieben. Der Nutzer muß also die Aktionen *M_Maction, M_action, MxMaction* und *MMaction* nicht mehr implementieren. Die erste Ebene der Hierarchie faßt

Wissensquellen mit gleichartiger Anfragestruktur zusammen. Funktionale Wissensquellen z.B. haben als Anfragen Ausdrücke wie $f(a, g(h(b), c))$, SQL-basierte Wissensquellen haben SQL-Anfragen. Kann man eine Wissensquelle hier einordnen, dann dürfte es in den meisten Fällen möglich sein, die dort schon programmierte *WQ_Query*-Klasse wiederzuverwenden, als auch alle semantischen Aktionen in der Klasse *WQ_Trans*, mit Ausnahme der Aktion *Aaction*, die einen Cursor erzeugt. In diese erste Ebene der Hierarchie fügen sich also Typen von Wissensquellen ein.

Die zweite Ebene bestimmt die genaue Software-Ausprägung eines Wissensquellentyps. Kann man eine Wissensquelle in der zweiten Ebene einordnen, so läßt sich sämtlicher Code wiederverwenden, häufig einschließlich der *main*-Routine. Hier muß nur noch das Exportschema erstellt und geladen werden. Dies kann sogar ebenfalls entfallen, wenn es ein solches schon gibt oder es fest mit der Wissensquellenart verbunden ist, wie etwa bei Mathematica. Dann bedeutet dies aber nichts anderes, als das man einen solchen Übersetzer schon mal geschrieben hat.

Die Verzeichnisstruktur des Quellcodes sowie der Aufbau des `makefile`s spiegeln diese Hierarchie wieder.

### 4.2.3 Abgeleitete Klassen

Am Beispiel des Oracle-Übersetzers sollen die Klassen *Query, Cursor* und *Trans* und deren wissensquellenspezifischen Erweiterungen vorgestellt werden. Da Oracle eine SQL-Datenbank ist, ist es sinnvoll, zwei Stufen der Hierarchie zu implementieren: Zuerst einen SQL-Übersetzer und darauf aufbauend einen Oracle-Übersetzer.

**Klasse** *Query*

```
class Query
{
public:
                Query();
virtual         ~Query();
        // Dump: Darstellung einer Anfrage auf den Output-Stream s
virtual void    Dump(ostream& s);
};
```

*Query* dient als Attribut für A-Kästen und soll somit eine Anfrage repräsentieren. Da jede Wissensquelle ihr eigenes Format hat, läßt sich in der Basisklasse nichts spezifizieren. *Dump()* dient zu Testzwecken und zur Verlaufsanzeige.

```
class SQL_Query : public Query
{
        StringListe selectlist, fromlist, wherelist;

public:
```

```
        // Konstruktor fuer Naction
                SQL_Query(Relation* r, int c);
        // Konstruktor fuer AxAaction
                SQL_Query(Query* q1, Query* q2);
        // Konstruktor fuer A_Maction
                SQL_Query(Query* q, Dock* dock,
                          Cursor* c, Dock* cdock, String label, int ad);
        // Konstruktor fuer A_Aaction und A_action
                SQL_Query(Query* q1, Dock* sdock,
                          Query* q2, Dock* tdock, String label, int ad);
        // Konstruktor fuer AAaction
                SQL_Query(Query *q1, Dock** docks1, int d1,
                          Query *q2, Dock** docks2, int d2, String label);
virtual         ~SQL_Query();
        // GetFullQuery liefert die SQL-Anfrage als String
        String  GetFullQuery();
        void    Dump(ostream& s);
        // RebuildSelectList baut die selectlist aufgrund
        // der Belegung der Docks auf
        void    RebuildSelectList(Dock** docks, int arity);
        // GetDockName liefert den Namen, der hinter einer Stelle steht
        String  GetDockName(Dock* dock);
};
```

*SQL_Query* enthält drei Stringlisten als Attribute, je eine für den select-, from- und where-Teil einer SQL-Anfrage. Dies wurde schon in Abschnitt 3.2.2 erwähnt. Die verschiedenen Konstruktoren resultieren aus den verschiedenen Arten, wie eine neue Anfrage entstehen kann. Man siehe *SQL_Trans* für die Parameterliste. Exemplarisch folgt die Implementierung eines solchen Konstruktors. Der Parameter *ad* gibt für eine gerichtete Kante an, ob sie von d1 nach d2 (Wert 1), oder umgekehrt (Wert 0) verläuft. `++` ist die String-Konkatenation, *Concat()* die String-Listen-Konkatenation. Die *selectlist* wird hier zunächst nicht aufgebaut. Dies wird mittels *RebuildSelectList()* erst ganz am Schluß erledigt, wenn die Anfrage abgeschickt werden soll.

```
// Konstruktor fuer A_Aaction und A_action
SQL_Query::SQL_Query(Query* q1, Dock* d1, Query* q2, Dock* d2,
                     String label, int ad) : Query()
{
        String newstr;
        // Die Listen sind anfangs leer, da Konstruktor
        fromlist.Concat((SQL_Query*)q1)->fromlist);
        wherelist.Concat((SQL_Query*)q1)->wherelist);
        fromlist.Concat((SQL_Query*)q2)->fromlist);
        wherelist.Concat((SQL_Query*)q2)->wherelist);
        if (ad == 1) newstr = GetDockName(d1) ++ label ++ GetDockName(d2);
        else newstr = GetDockName(d2) ++ label ++ GetDockName(d1);
```

```
        wherelist.Insert(newstr);
}
```

*Oracle_Query* erweitert wie schon erwähnt *SQL_Query* nicht.

**Klasse** *Cursor*

```
class Cursor
{
        int arity;

public:
                Cursor(int arity);
virtual         ~Cursor();
virtual Bool    Begin() = 0;
virtual Sort    Read(int i) = 0;
virtual Bool    Next() = 0;
virtual Bool    NoMore() = 0;
};
```

Die allgemeine *Cursor*-Schnittstelle wurde schon in Abschnitt 4.1 vorgestellt. Am Cursor wird stets vermerkt, welche Breite ein Ergebnistupel hat, welchen Definitionsbereich also das Argument für *Read()* haben darf. Die Methoden sind abstrakt, so daß nur Objekte von abgeleiteten Klassen erzeugt werden können, wenn alle vier Methoden implementiert wurden. *Sort* ist eine Oberklasse aller Ergebniswerte-Klassen. Im Rahmen dieser Implementierung sind dies nur *Integer, Float, String.*

*SQL_Cursor* implementiert die Methoden nicht, da es nicht *den* SQL-Cursor an sich gibt. Dieser läßt sich nur für jede spezielle Wissensquelle realisieren.

```
#include "oracode.h"

class Oracle_Cursor : public SQL_Cursor {

        sword ncols;
        describe *desc;
        define   *def;
        Cda_Def *cda;
        Lda_Def *lda;
        Bool  end_reached;

public:
                Oracle_Cursor(Lda_Def *lda, Query* q, Dock** d, int arity);
                ~Oracle_Cursor();
        Bool    Begin();
        Sort    Read(int r);
```

```
        Bool    Next();
        Bool    NoMore();
};
```

*Oracle_Cursor* beinhaltet nun speziell für das Oracle Call Interface (OCI) notwendige Datenstrukturen, deren Zwecke hier nicht erläutert werden sollen. Das Flag *end_reached* sorgt dafür, daß die Semantik der Methoden eingehalten wird. Es wird auf *True* gesetzt, wenn ein *Next()*-Aufruf nicht erfolgreich war. *NoMore()* liest diesen Indikator aus. In `oracode.h` sind Oracle-spezifische Routinen, die auf dem OCI basieren. Hier entsteht viel Entwicklungsarbeit. Man muß sich mit dem Produkt gut auskennen. Es ist wohl kaum möglich, daß die Bibliothek hier weitere Unterstützung bietet.

An dieser Stelle sollen auch die speziellen Cursor für die schon implementierten Aktionen über M-Kästen erläutert werden. Dies sind *Join_Cursor* für die *M_Maction*, *Sel_Cursor* für die *M_action*, *X_Cursor* für die *MxMaction* und schließlich *Neg_Cursor* für die *MMaction* mit Kantenlabel „-“ für die Differenz. So ist z.B. die Methode *Next()* in *Join_Cursor* wie folgt implementiert:

```
Bool
Join_Cursor::Next()
{
        if (!c1->Next()) {
                if (!c2->Next()) return False;
                if (!c1->Begin()) return False;
        }
        if (c1->Read(p1) != c2->Read(p2))
                if (!Next()) return False;
        return True;
}
```

*Join_Cursor* hat die beiden zugrunde liegenden Cursor in *c1, c2* und die auszulesenden Positionen in *p1, p2* gespeichert. Es werden hier die beiden Cursor solange sukzessive weitergeschaltet (rekursive Prozedur), bis beide Cursorwerte gleich sind – dann wird `True` geliefert – oder bis beide Cursor nicht mehr fortschaltbar sind – dann wird `False` geliefert. Beim Erzeugen eines *Join_Cursor* wird darauf geachtet, daß dieser nur Kanten mit einem „=“ bearbeiten kann. Siehe dazu auch den nun folgenden Paragraphen.

**Klasse** *Trans*

```
class Trans
{

public:
                Trans();
virtual         ~Trans();
```

```
virtual Query*  Naction(Relation*, int number, Dock** docks, int arity) =0;
virtual Query*  A_Maction(Query*, Dock**, int arity1, int index1,
                          Cursor*, Dock**, int arity2, int index2,
                          String label, int direction) { return NULL; }
virtual Query*  A_Aaction(Query*, Dock**, int arity1, int index1,
                          Query*, Dock**, int arity2, int index2,
                          String label, int direction) { return NULL; }
virtual Query*  A_action(Query*, Dock**, int arity, int index1,
                                                    int index2,
                         String label, int direction) { return NULL; }
virtual Query*  AxAaction(Query*, Dock**, int arity1,
                          Query*, Dock**, int arity2) { return NULL; }
virtual Query*  AAaction(Query*, Dock**, int arity1,
                         Query*, Dock**, int arity2,
                         String label) { return NULL; }
virtual Cursor* Aaction(Query*, Dock**, int arity) =0;
        Cursor* M_action(Cursor*, Dock**, int arity, int index1,
                                                     int index2,
                         String label);
        Cursor* M_Maction(Cursor*, Dock**, int arity1, int index1,
                          Cursor*, Dock**, int arity2, int index2,
                          String label);
        Cursor* MxMaction(Cursor*, Dock**, int arity1,
                          Cursor*, Dock**, int arity2);
        Cursor* MMaction(Cursor*, Dock**, int arity1,
                         Cursor*, Dock**, int arity2,
                         String label);
};
```

In *Trans* sind die semantischen Aktionen spezifiziert. Die Signaturen der einzelnen Aktionen sind die folgenden: Wird eine Aktion aufgrund einer Stellenkante aufgerufen – also *A_Maction, A_Aaction, A_action, M_action* und *M_Maction* – so werden die Daten der beiden assoziierten Kästen (das sind das Attribut *Query* oder *Cursor*, die Stellenliste, deren Anzahl (*arity*) und die Nummer *index* der Stelle, die an der Kante liegt), die Marke der Kante (*label*) und die Richtung der Kante (*direction*) übergeben. Die Richtung ist 1, wenn die Kante vom zuerst angegebenen Kasten zu dem zweiten geht, ansonsten 0. *A_action* und *M_action* sind Sonderfälle, weil beide Kästen identisch sind, daher gibt es nur einmal die Kastendaten, aber dennoch zwei Indizes für die beiden mit der Kante assoziierten Stellen. *Naction* und *Aaction* erhalten die Daten des einen Kastens. Jeder N-Kasten hat zusätzlich eine eindeutige Nummer (*number*). Kastenkantenaktionen (*AAaction* und *MMaction*) bekommen die Daten der beiden Kästen (ein Index ist dabei nicht nötig) und die Marke der Kante. Nicht-Kantenaktionen (*AxAaction* und *MxMaction*) erhalten ebenfalls die Daten beider Kästen, jedoch natürlich ohne irgendwelche Kanteninformationen. In vielen Fällen werden nicht alle Parameter gebraucht, um eine Aktion auszuführen, so z.B. die Liste der Stellen eines Kastens.

*Trans* implementiert schon die vier reinen M-Kasten-Aktionen. Als Beispiel sei *M_Maction* gezeigt:

```
Cursor*
Trans::M_Maction(Cursor* c1, Dock** sdocks, int, int sd,
                 Cursor* c2, Dock** tdocks, int, int td, String label)
{
        if (label != "=") throw BadLabel(label);
        Join_Cursor* res = new Join_Cursor(c1, c2, sdocks[sd]->argnr,
                                           tdocks[td]->argnr, label);
        return (Cursor*) res;
}
```

Der *Join_Cursor*-Konstruktor sieht wie folgt aus:

```
Join_Cursor::Join_Cursor(Cursor* crs1, Cursor* crs2, int p1, int p2, String):
        Cursor(crs1->arity+crs2->arity), c1(crs1), c2(crs2), p1(p1), p2(p2)
{
        if (c1->NoMore() || c2->NoMore())
                return; // mindestens ein Cursor ist leer
        if (c1->Read(p1) != c2->Read(p2))
                Next(); // auf ersten Wert schalten
}
```

Dieser wurde nur zur vollständigen Dokumentation erwähnt. Der Benutzer muß sich um *M_Maction, MxMaction, M_action* und *MMaction* nicht mehr kümmern. In der Tat reicht es für jeden Übersetzer, nur die Aktionen *Naction* und *Aaction* zu implementieren. Die erste wandelt einen N-Kasten in einen A-Kasten, d.h. es muß eine Anfrage nach einer ganzen Relation generiert werden, die zweite wandelt einen A-Kasten in einen M-Kasten, d.h. eine Anfrage muß an die Wissensquelle abgeschickt und aus der Antwort ein Cursor generiert werden. Auf diese Weise läßt sich sehr schnell ein Übersetzer erstellen. In vielen Fällen ist es aber unbedingt sinnvoll, möglichst viele Aktionen zu implementieren, denn die alleinige Nutzung nur dieser beiden Aktionen führt natürlicherweise zu sehr langsamen und umfangreichen Anfragebearbeitungen. Eine Aktion wird als nicht implementiert erkannt, wenn sie ein NULL-Objekt zurück liefert.

In der Tat beherrschen SQL-Datenbanken alle Aktionen. Daher lassen sich in *SQL_Trans* alle Aktionen implementieren, nur die *Aaction* nicht, da sie wissensquellenspezifisch ist. Mit Hilfe der *SQL_Query*-Konstruktoren lassen sich diese Aktionen leicht implementieren, z.B.:

```
Query*
SQL_Trans::A_Aaction(Query* q1, Dock** ds, int, int s,
                     Query* q2, Dock** dt, int, int t, String l, int ad)
{
        SQL_Query* res = new SQL_Query(q1, ds[s], q2, dt[t], l, ad);
        return (Query*)res;
}
```

*Oracle_Trans*, abgeleitet von *SQL_Trans*, erbt all diese Aktion-Implementierungen und fügt noch die der *Aaction* hinzu:

```
Cursor*
Oracle_Trans::Aaction(Query* q, Dock** docks, int arity)
{
        Oracle_Cursor* res = new Oracle_Cursor(lda, q, docks, arity);
        return (Cursor*)res;
}
```

*lda*, genauso wie *hda*, ist eine Datenstruktur, die das OCI verwendet. An dieser Stelle muß auch der *Oracle_Trans*-Konstruktor erwähnt werden:

```
Oracle_Trans::Oracle_Trans(String user, String pass) : SQL_Trans()
{
        lda = new Lda_Def();
        hda = new ub4[HDA_SIZE / 4];
        cout << "Trying to connect to Oracle as user " << user << endl;
        if (!connect_user(lda, hda, username, password))
                throw Error("Oracle-Connection failed");
        cout << "Connection established" << endl;
}
```

*connect_user* ist wiederum eine C-Routine aus `oracode.h`, die die entsprechenden OCI-Aufrufe startet. Verlaufsanzeigen werden auf die Standardausgabe gelenkt.

### 4.2.4 Erstellung von *main*

*main* muß im wesentlichen das *Translator*-Objekt erzeugen und die zentralen Routinen aufrufen. Dazu braucht es ebenfalls ein Objekt der *Trans*-Implementierung der Wissensquelle.

```
#include <fstream.h>
#include "oracle_trans.h"
#include "translator.h"
#include "ikarus.h"

int main(int argc, char* argv[])
{
    Init();
    if (argc != 3) {
            cerr << "Zwei Argumente: Exportschema Anfrage" << endl;
            return -1;
    }
    ifstream exp(argv[1]);
    if (!exp) { cerr << "Cannot open " << argv[1] << endl; return -1; }
    ifstream que(argv[2]);
```

```
    if (!que) { cerr << "Cannot open " << argv[2] << endl; return -1; }

    try {
    String user("jekutsch@daedtcp"), pass("passwort");
    Translator* translator  = new Translator(new Oracle_Trans(user,pass));
    translator->Initialize(exp,"= >= > <= < !=");
    translator->Activate(que,cout);
    Clear();
    delete translator;
    return 0;
    }
    catch (Error err) {
            cout << endl << "Internal Error: ";
            cout << err.message << endl;
            Clear();
            delete translator;
            return -1;
    }
}
```

*Init()* und *Clear()* sind interne Verwaltungsroutinen aus `ikarus.h`, die am Anfang resp. am Ende aufgerufen werden müssen. *Oracle_Trans* bekommt den Oracle-Benutzernamen (mit der Kennung der Datenbank) und das Paßwort. *Translator::Initialize()* wird mit der Exportschema-Datei und einem String mit den Namen der Relationen aufgerufen, für die eine Transformation gemäß Abschnitt 3.2.3 durchgeführt werden soll. Schließlich wird der Übersetzer aktiviert mit einem Eingabe- und einem Ausgabe-Stream für die Kommunikation mit dem Mediator. In diesem Fall sind es eine Datei und die Standardausgabe.

# Kapitel 5

# Abschluß

In diesem abschließenden Kapitel werden im ersten Abschnitt verwandte Arbeiten vorgestellt. Dabei werden nur Arbeiten speziell zur Anfrageübersetzung betrachtet. Danach folgt eine Zusammenfassung der Arbeit, die die Schwerpunkte, Neuansätze aber auch Schwächen des vorgestellten Ansatzes zusammenträgt. Darauf folgt im abschließenden Abschnitt ein Ausblick auf noch zu leistende Arbeiten und ein wenig „Zukunftsmusik“.

## 5.1 Vergleichbare Arbeiten

Die Aufgabe der Modell- oder Anfrageübersetzung taucht in allen Systemen auf, die der Mediatorarchitektur von KOMET entsprechen und heterogen sind, und davon gibt es einige. Einen State-of-the-Art-Bericht vom Ende der achtziger Jahre geben [Gup89, SL90]. Um so erstaunlicher ist, daß in den wenigsten Fällen die Anfrageübersetzung durch eine Entwicklungsumgebung unterstützt wird. Meistens wird nur erwähnt, daß eine solche nötig sei. Häufig liegt der Schwerpunkt der Übersetzung auch mehr bei der *Schema*übersetzung, der Rest scheint keiner Erwähnung wert. Dazu muß allerdings gesagt werden, daß die Trennung von Schema- und Modellübersetzung selten so explizit gemacht wird wie in dieser Arbeit.

Im folgenden werden einige relevante, ähnliche Arbeiten vorgestellt, die die Modellübersetzung vertieft behandeln. Nicht betrachtet wird die Schemaübersetzung oder die Mediatorarchitektur. Die Vorstellung ist so geordnet, daß die Ähnlichkeit zu KOMET und dem hier vorgestellten Ansatz abnimmt.

HERMES [AE95] ist für einen Vergleich mit KOMET wichtig, weil ihm dasselbe Mediatormodell zugrunde liegt, nämlich die annotierten logischen Klauseln mit Constraintteil zum Ansprechen der externen Wissensquellen. Unterschiedlich ist aber schon die Art der Constraintausdrücke. Eine Anfrage an eine Wissensquelle sieht in HERMES z.B. wie folgt aus:

```
DB:project(select>('parts',"cost",50),"partid")
```

Dieses Beispiel fragt die Wissensquelle DB nach der 'partid' aller 'parts', deren 'cost' größer als 50 ist. Diese Art der Kodierung unterscheidet sich deutlich von der in KOMET verwendeten. Während bei KOMET Constraintrelationen wie üblich als eine zusammengefaßte Menge

von Tupeln aufgefaßt und Anfragen über diese erlaubt werden, werden Anfragen in HERMES mittels Funktionsaufrufe ausgedrückt. Der eigentliche externe Datenspeicher, also die Relation, ist *Argument* dieser Funktionen. Eine Wissensquelle muß in obiger Notation sehr viele verschiedene Funktionen kennen, z.B. wird es außer `select>` viele andere Selektionsfunktionen geben. Außerdem bleibt zu fragen, welche Signaturen diese Funktionen haben, da in dem obigen Beispiel an dritter Stelle speziellerweise eine ganze Zahl steht. In HERMES ist dies allerdings kein Problem, da gar nicht typisiert wird. Da HERMES externe Funktionen aufruft, gibt es keine Bindungsmuster. In sofern ist der hier vorgestellte Ansatz allgemeiner.

Eine Variablenbindung kann man mit zwei speziellen Relationen 'in' und '=' realisieren, z.B. ist

```
in(P,DB:project('parts',"color")) & =(P.color, ''green'')
```

wahr, wenn es ein Objekt in 'parts' mit einer grünen Farbe gibt. Die Übersetzung einer so gestellten Anfrage in eine Wissensquellenanfrage ist in der Literatur leider nicht weiter ausgeführt. Fraglich ist z.B., ob die `project(select>())`-Anfrage wirklich in *eine einzige* Wissensquellen-Anfrage übersetzt wird. Werden auch konjunktive Anfragen unterstützt? Negation wird nirgends erwähnt. Neuere Arbeiten, die sich mit Bindungsmustern und konjunktiven Anfragen beschäftigen, sind in [AQ95] zu finden. Ein Cursorkonzept ist in HERMES nicht vorgesehen. Statt dessen wird das Ergebnis einer Anfrage vollständig in einer Datei materialisiert. Diese wird dann zwar mittels eines Cursors ausgelesen, aber er wird nicht zur Realisierung eines Verbunds über zwei Relationen der gleichen Wissensquelle herangezogen.

HERMES unterstützt Pfadausdrücke und hat somit eine direkte Unterstützung für Anfragen an objektorientierte Datenbanken. Um die Namen aller Universitäten zu erhalten, die eine Informatik-Fakultät haben, kann man schreiben:

```
in(Univ, OODB:equal(university_db, university.dept.name, "Informatik")
  & =(Univ.university.name, Universityname)
```

Zunächst werden dabei alle Universitätsobjekte an `Univ` gebunden und dann daraus der Name extrahiert. Die beiden Schritte der Selektion und Projektion sind getrennt. Dies hat den Nachteil, daß ganze Universitätsobjekte zum Mediator geschickt werden müssen. Angenehm ist jedoch, daß Pfadausdrücke direkt eingebbar sind.

Die Beschreibung der Anfrageübersetzung im Projekt Tsimmis [PGGU95] ist die einzige detailierte, die im Zusammenhang mit Mediatoren veröffentlicht wurde. Das Mediatormodell und somit auch das einheitliche Datenmodell ist in Tsimmis ein anderes als in KOMET. Es heißt Object Exchange Model (OEM) und hat die Eigenschaft, recht minimal zu sein. Ein Studentenobjekt `ob1` sieht etwa wie folgt aus:

```
<ob1, student, {sub1,sub2}>
<sub1, name, 'Sebastian Jekutsch'>
<sub2, matrnr, 699952>
```

In diesem Beispiel hat das Objekt `ob1` (Objektidentifikator) mit der natürlichsprachlichen Beschreibung `student` zwei Unterobjekte aggregiert: Einen Namen und eine Matrikelnummer. Man beachte, daß die Reihenfolge der Unterobjekte unerheblich ist.

Zu bemerken ist, daß es *kein* explizites Schema, und somit auch kein Exportschema gibt. Die Schemainformation steckt in jedem einzelnen Objekt. Es findet keine Typisierung statt. Jedes Objekt hat eine Objektidentität (z.B. `sub2`), einen Namen (`matrnr`) und einen Wert (`699952`), der auch eine Menge von Objekten (z.B. {`sub1`,`sub2`}) sein kann. Eine Anfrage (Sprache: MSL) kann wie folgt gestellt werden:

```
*O :- <O student {<M matrnr 699952>}>}
```

Sie ist eine Schablone für ein Objekt. Gesucht ist hier der Objektidentifikator eines Objekts mit dem Namen `student`, das ein Unterobjekt `<M matrnr 699952>` mit beliebigen Identifikator `M` hat. Dies liefert in `O` den Identifikator `ob1` von oben. In den geschweiften Klammern steht eine Menge von Selektionsausdrücken. Jeder Ausdruck ist ein Muster für die gesuchten Objekte. Ganz links steht die Menge der zu projizierenden Daten. Die Übersetzung einer MSL-Anfrage in eine Anfrage an eine Wissensquelle geschieht mittels *templates*, das sind parametrisierte MSL-Anfragen. Paßt ein solches Template auf eine gestellte Anfrage, so wird eine daran geknüpfte Aktion ausgeführt. Diese erhält die eingesetzten Parameter und kann normaler Programmiersprachencode sein. Durch den Einsatz von Nichtterminalen werden die Templates zu nichts anderem als einer Grammatik der zu übersetzenden MSL-Anfragen. Der Ansatz ist somit mit der traditionellen Compilerbautechnik der attributierten Grammatiken und also auch dem hier vorgestellten Ansatz eng verwandt. Der Übersetzer muß leider einige Probleme, die das OEM erzeugt, lösen, womit der Tsimmis-Ansatz viel komplizierter wird als eigentlich nötig. So ist z.B. die Reihenfolge der Unterobjekte eines Objekts beliebig – anders als bei dem relationalen Modell – so daß ein und dasselbe Objekt auf verschiedenste Weise dargestellt (die Reihenfolge der in ihm enthaltenen Objekte ist nicht festgelegt) und somit auch angefragt werden kann. Bei Objekten mit vielen Unterobjekten vereinfacht das OEM aber auf diese Weise die Anfrageformulierung für den Benutzer. In [PGGU95] werden konjunktive, disjunktive und negierte Anfragen nicht behandelt. Schwerpunkt ist hingegen die Unterscheidung zwischen direkt, indirekt und nur logisch unterstützten Anfragen. *Direkt* unterstützte Anfragen sind solche, die eins-zu-eins auf ein Template passen, *indirekt* unterstützte Anfragen können aufgeteilt werden in eine direkt unterstützte Anfrage und einen Filter, der das Ergebnis der Anfrage noch nachbessert (z.B. indem nachträglich eine Selektion gemacht wird). Dies korrespondiert mit dem in der vorliegenden Arbeit benutzten Konzept der Bindungsmuster. In der Tat liegt die besondere Leistung des Tsimmis-Übersetzers auf diesem Gebiet. Die *logisch* unterstützten Anfragen schließlich noch sind Anfragen, die äquivalent auf eine direkt unterstützte Anfrage abbildbar sind. Ein solcher Aspekt, abgesehen von den Uneindeutigkeiten, die allein das OEM erzeugt, wurde in dieser Arbeit nicht betrachtet. So sind z.B. die folgenden beiden Anfragen äquivalent:

```
*O :- <O student {<name 'Sebastian Jekutsch'> <matrnr 699952>}>
*O :- <O student {<name 'Sebastian Jekutsch'>}>
      AND <O student {<matrnr 699952>}>
```

In dieser Arbeit wird davon ausgegangen, daß logisch äquivalente konjunktive Klauselanfragen auch auf logisch äquivalente Anfragen an die Wissensquelle abgebildet werden. Es bleibt allenfalls der Aspekt des Findens der optimalsten der äquivalenten Anfragen. Das Problem der Umwandlung einer Ordnungsrelation in eine Ordnungsselektion aus Beispiel 3.4 ist mit dem Tsimmis-Ansatz nicht auf einfache Weise lösbar.

Das Konzept der Bindungsmuster in dieser Arbeit basiert wesentlich auf [PGGU95, RSU95].

Nach der kurzen Vorstellung der Projekte HERMES und Tsimmis, die direkt mit KOMET vergleichbar sind, folgen nun einige Arbeiten die in einzelnen Punkten mit der vorliegenden in Beziehung stehen.

In [CRE87] wird eine Anfragerepräsentation verwendet, die eine gewisse Ähnlichkeit mit der hier vorgestellten Kastendarstellung hat. Die Verbundkanten werden dort allgemein *Connectors* genannt. Mittels Operatoren können innerhalb ein und desselben (relationalen) Modells Anfragen in Anfragen transformiert werden. Eine Folge von Operatoren ist also eine Schemaübersetzung. Da angenommen wird, daß die Wissensquellen ebenfalls relational sind, macht eine Anfrageübersetzung keine großen Probleme. Diese findet aber *nicht* mit Hilfe der Connectors statt, sondern mit Hilfe einer Übersetzung in Ausdrücke der relationalen Algebra [RC85]. Interessanterweise treten Verbundkanten auch in der graphischen Darstellung einer Anfrage im Microsoft Access DBMS auf.

In [PSJ87] werden zwei Anfragesprachen ineinander übersetzt, indem jeweils Übersetzer von und in eine mächtige Zwischensprache geschrieben werden. Die Übersetzung wird dabei mit Hilfe von condition-action-Paaren beschrieben. Diese Vorgehensweise ist aufgrund vieler nützlicher Operatoren sehr mächtig. Die Zwischensprache hat den Zweck der Aufwandminimierung. Hat man einen Übersetzter von den Sprachen A und B in die Zwischensprache und zurück geschrieben, so sind die Übersetzungen A $\leftrightarrow$ B realisiert. Kommt nun eine weitere Sprache hinzu, so braucht man wiederum nur von und zur Zwischensprache zu übersetzen, um Übersetzer von und zu A und B zu erhalten. Die Anzahl der Übersetzer steigt also mit jeder neuen Sprache nur linear und nicht quadratisch. Das Problem ist, daß a) die Zwischensprache (auch Interlingua genannt) ausdrucksstark sein muß und b) eine Übersetzung $A \rightarrow ZwSp \rightarrow B$ sicherlich weniger spezifisch ist, als $A \rightarrow B$. Die Notwendigkeit einer Zwischensprache ist bei KOMET jedoch nicht gegeben, da stets nur vom Mediatormodell in ein Modell einer Wissensquelle übersetzt wird. Das Mediatormodell definiert sozusagen schon die Zwischensprache. Eine Zwischensprache wird z.B. auch im Projekt Mermaid benutzt [Tem87].

So gesehen ist dies ein Schwachpunkt von KOMET: Das Mediatormodell ist nicht mächtig genug, die vielen verschiedenen Modelle der Wissensquellen adäquat abzubilden und somit auch „tiefstrukturierte“ Anfragen zu stellen. Schon die Darstellung eines objektorientierten Modells bereitete Schwierigkeiten (Abschnitt 3.4). Aus diesem Grund sind viele Autoren dazu übergegangen, ein objektorientiertes Modell als allgemeines Datenmodell zu benutzen. Aber auch dies kann nur eine Verlegenheitstat sein. Und: Je mächtiger ein Modell ist, desto schwieriger und ineffizienter sind Inferenzen mit diesem Modell zu realisieren. In KOMET wurde daher ein Kompromiß eingegangen: Die Verantwortung zur korrekten Interpretation der Daten in den „mächtigen“ Wissensquellen liegt beim Entwickler, da viele der gelieferten semantischen Informationen im Mediator gar nicht mehr sichtbar sind. Dies ist eine zwar unschöne, aber offensichtlich notwendige Folgerung, die auch z.B. beim Tsimmis-Projekt mit dem sehr einfachen OEModell gezogen wurde.

In [HFG87] wird ebenfalls eine Entwicklungsumgebung zur Anfrageübersetzung vorgestellt. Dies ist die einzige der hier vorgestellten Quellen, die explizit Compilerbautechniken erwähnt und die Notwendigkeit, den Entwickler mit einem Werkzeug zu unterstützen. Es wird auch hier eine eigenständige Anfragerepräsentation eingesetzt, und zwar die Baumdarstellung ei-

nes relationalen Ausdrucks. Die Übersetzung findet statt, indem Teilbäume dieser Darstellung direkt übersetzt werden (ohne Attribute). Dieser Ansatz dürfte daher nur für Wissensquellen geeignet sein, deren Konzept stark an der relationalen Algebra angelehnt ist. Bindungsmuster werden nicht betrachtet.

In [TL94] werden nur Baumtransformationen zur Übersetzung eingesetzt. Allerdings werden Daten*repräsentationen*, also *Modelle*, ineinander übersetzt, nicht Anfragen an sich. Obschon Anfragen mittels parametrisierter Datenrepräsentation formuliert werden können (indem die angefragten Stellen durch Variablen ersetzt werden), wird dies nicht erwähnt. In KOMET werden eben *nicht* die Datenbasen innerhalb der Modelle übersetzt, sondern „nur“ die Anfragen und die Anfrageergebnisse. Dies ist aber nur ein kleiner Unterschied zur Übersetzung von Modellen.

Die Idee, eine Zwischensprache zu verwenden, und der Ansatz, die Daten*modelle* zu übersetzen, sind auch zentral für den *Knowledge Sharing Effort* (KSE) [NFF+91]. Hier soll das Wissen aus verschiedenen Quellen interoperabel gemacht werden, das heißt, die eine Wissensquelle soll die andere „verstehen“ können, oder zwei Wissensbasen sollen wiederverwendbar und verbindbar sein. Die Interlingua ist hier das Knowledge Interchange Format (KIF). Der Wissensbereich (Ontologie) einer Wissensquelle wird in KIF übersetzt. KIF ist sehr mächtig; es lehnt sich an die Prädikatenlogik erster Stufe mit Ergänzungen an. KIF selbst ist keine Inferenzsprache mehr. Es gibt außer in [BF95, Gru93] keine Beschreibung der Übersetzer. Eine Kritik an der Herangehensweise ist in [vHPS94] zu finden. Der Schwerpunkt im KSE liegt stark bei der semantischen Integration von Wissensquellen, d.h. es wird möglichst viel der semantischen Informationen, die in den Daten der Wissensquellen stecken, in der KIF-Ontologie ausgedrückt. Entsprechend umfangreich wird eine Übersetzung. Wegen der deutlichen Datenbank-Orientierung in KOMET wurde ein solches Vorgehen verworfen.

Auf der Seite der reinen Datentypkonvertierung bietet [WWRT91] einen entsprechenden „semantischen“ Ansatz. Dort werden nicht Bits auf Bits oder Arrays auf Listen, sondern abstrakte Datentypen auf ebensolche abgebildet. Ein Array wird also z.B. als Keller konvertiert, nicht als eine lückenlose Folge von Elementen.

Es gibt noch eine Fülle von Arbeiten, die Übersetzungen zwischen zwei speziellen Datenmodellen behandeln, z.B. relational $\leftrightarrow$ objektorientiert oder Netzwerk $\rightarrow$ relational oder auch Übersetzungen zwischen einem abstrakten Entwurfsmodell und einem konkreten Datenbankmodell [LL95]. Interessant ist noch der Artikel [BNPS89]. Dort wird zum ersten Mal der Unterschied zwischen dem Übersetzen von Modellen, d.h. einer generischen Abbildung der Wissensbasis, und dem Übersetzen von Anfragen bzw. Funktionen als ein structural versus operational mapping beschrieben.

## 5.2 Zusammenfassung

In dieser Arbeit wurde eine Vorgehensweise zur Entwicklung von Programmen vorgestellt, die Anfragen in Klauseldarstellung in Anfragen verschiedener anderer Anfragesprachen übersetzen. Die Herangehensweise basiert wesentlich auf Techniken des Compilerbaus. Mit Hilfe der Kastendarstellung von Anfragen im relationalen Modell wurde eine geeignete Basis geschaffen, die Übersetzer zu spezifizieren und zu implementieren. Die Beachtung von Bindungsmustern, die Verwaltung von Cursorn und das selektive Implementieren von Regelaktionen

ermöglichen es, Wissensquellen anzufragen, die die geforderte Anfragemächtigkeit gar nicht besitzen. Der Übersetzer kompiliert aus einer Quell-Anfrage eigenständig die nötige Anzahl von Ziel-Anfragen und sorgt für eine eventuell nachträglich noch notwendige Nachbearbeitung der Antworten einer Wissensquelle.

Die Kastendarstellung wurde in dieser Ausarbeitung formal definiert und notwendige Voraussetzungen zur Korrektheit einer Übersetzung aufgezeigt. Der Ansatz wurde anhand einiger häufiger Beispiele evaluiert.

Zusammenfassend kann im Hinblick auf die vergleichbare Literatur gesagt werden, daß die vorliegende Arbeit mit Ausnahme von [PGGU95] die erste uns bekannte ist, die an das Problem der Anfrageübersetzung mit Hilfe einer generischen Entwurfsumgebung herangeht. [PGGU95] wird in wesentlichen Teilen subsumiert und in einigen Punkten konkretisiert, so z.B. in der Behandlung unpassender Bindungen, der Behandlung der Disjunktion und Negation und der einheitlichen Behandlung konjunktiver Anfragen.

## 5.3 Ausblick

In diesem letzten Abschnitt folgt noch ein Ausblick auf „liegengebliebene“, absehbare und richtungsgebende Arbeiten.

**Anfrage-Optimierungen** Es genügt in der Regel nicht, einfach nur eine korrekte lokale Anfrage zu erzeugen, es muß auch das Ziel sein, aus den vielen möglichen korrekten Anfragen die optimale herauszufinden. Optimal heißt in diesem Zusammenhang: schnellst möglich beantwortbar. Da die Zielsprachen sehr verschieden sind, ist eine Optimierung aber nicht einfach. Die Kastendarstellung basiert zwar auf der relationalen Algebra und dort gibt es Heuristiken, wie eine Anfrage optimal zu stellen ist, aber es gibt keinen Hinweis darauf, daß eine algebra-optimale Anfrage auch wissensquellen-optimal ist. Schließlich nutzen die meisten Wissensquellen eben *nicht* die Operatoren der relationalen Algebra. Dennoch sollte man sich nicht auf die wissensquellen-eigene Optimierung verlassen, denn die existiert selten. Zudem rechnet der Übersetzer ja auch, und zwar wenn er die Cursor einsetzt. Also ist es dennoch sinnvoll anzunehmen, daß eine Algebra-optimale Anfrage „ganz gut“ sein muß.

Nur dort wo noch Spielraum ist, d.h. dort wo in den beschriebenen Algorithmen Indeterminismen auftreten, lassen sich Überlegungen zur Optimierung ansetzen. Es gibt zwei Stellen:

- Beim Zuordnen der erlaubten Bindungstypen an die Stellen der Kästen kann es im Falle von Bindungskonflikten die Auswahl geben, *wo* der Konflikt auftreten soll. ○-○-Konflikte sollten an möglichst kleinen Kästen auftreten. Klein bedeutet hier, daß die Anzahl der Tupel in der dazugehörigen Relation gering sein sollte. Man braucht also Informationen über die Größe der den N-Kästen zugrundeliegenden Relationen und müßte die Größe von K-Kästen a-priori abschätzen. Die Größe der M-Kästen ist dem Übersetzer ohnehin bekannt, zumindest zu Beginn einer Übersetzung. ●-●-Konflikte sollten nur zwischen Stellen auftreten, deren Typausprägung, also die Menge der zu einem Typ gehörenden Objekte, klein ist. Bei

großer Ausprägung sollte eine Zurückweisung der Anfrage geschehen, bei unendlicher Ausprägung muß sie sogar geschehen. Diese Art der Optimierung ist bislang in der Implementierung nur ansatzweise realisiert worden.

- Beim Anwenden von Regeln kann es vorkommen, daß eine Regel zu einem Zeitpunkt mehrmals anwendbar ist. Dann sollte man als nächstes eine Regel nehmen, deren Ergebniskasten möglichst klein ist. Hier braucht man also zusätzlich noch Informationen über die Größe der in der Prämisse stehenden A-Kästen, die ja im Laufe einer Übersetzung auftreten und die aus den hervorgehenden Kästen berechenbar sein muß. Es kann auch sein, daß die Reihenfolge zweier Regeln gar nicht wichtig ist. Auch hier können Informationen über die Kastengrößen den Indeterminismus sinnvoll auflösen helfen.

**Softwaretechnik** Die Programmierumgebung, die im Laufe dieser Arbeit erstellt wurde, ist sicherlich noch zu verbessern. Man kann sich z.B. eine graphische Unterstützung vorstellen, nicht nur in Form einer graphischen Oberfläche, sondern auch in Form von Debugging-Hilfen. Die Kastendarstellung legt dies nahe.

**Unterstützende Operatoren** In [Vie89] und [PSJ87] werden einige über den hier genannten Rahmen hinausgehende Techniken und Hilfestellungen erwähnt. Man muß prüfen, welche auch in diesem Ansatz, der eher einfach geblieben ist, nutzbar sind.

**Aggregatfunktionen und andere Besonderheiten** Eine der wichtigsten Übersetzungen ist wohl die in SQL. SQL bietet mehr als das reine select-from-where-Konstrukt. Wichtig sind z.B. die Aggregatfunktionen. Es wäre angenehm, wenn man dieses Können auch nutzen kann, es fehlt jedoch das Konzept von Aggregatfunktionen im Mediatormodell. Dies gilt auch für Konstrukte wie:

```
select item
from supplies
where price >= all (select price from supplies)
```

welches das teuerste `item` findet, oder:

```
select job, sal+comm
from emp
where deptno = 20
```

welches in der Projektion eine Operation enthält. Viele solcher spezieller Möglichkeiten bleiben ungenutzt, weil das allgemeine Datenmodell keine Repräsentation dafür hat.

**Anwendungen** Natürlich ist es immer interessant, die Übersetzer für immer mehr und immer andere Wissensquellen einzusetzen; je mehr, desto besser. Dabei sollte man möglichst viele Standards der Informationstechnik ausnutzen. Ein Mediator lebt davon, überall einsetzbar zu sein. Besonders interessant sind dabei Quellen, die ihre Daten deutlich strukturierter repräsentieren, als es mit dem relationalen Modell möglich ist. Dies würde per trial-and-error die Frage beantworten: Ist das vorliegende Mediatormodell mächtig genug? Interessant wäre es auch, die Bindungsmuster ausgiebiger zu nutzen. Bislang gab es nur beliebige Muster und das „funktionale“ Muster. Bei der

WWW-Anbindung mit ihrer speziellen Struktur hätten man auch reine ∘-Bindungen angeben können.

**Erweiterung der Bindungsinformationen** Die Bindungsmuster beziehen sich auf die Selektionsmöglichkeiten einer Relation in einer Wissensquelle, d.h. es wird spezifiziert, welche Attribute belegt werden können/müssen und welche Attribute Ergebnisse liefern. Man könnte dies auf die Projektionen erweitern. Pro Attribut wird dabei angegeben, ob es projiziert wird/werden kann oder nicht. Manche Wissensquellen projizieren grundsätzlich alle Attribute, z.B. erhält man bei der Literaturrecherche meistens immer alle Informationen zu einem Buch, auch die, die man vorgegeben hatte, etwa den Autorennamen. Ein Taschenrechner hingegen projiziert nur das Ergebnis, also jenes Attribut, welches als einziges das Bindungsmuster *frei* hat. Aber selbst solche freien Attribute, die von der Wissensquelle berechnet wurden, werden nicht immer projiziert. Ein solches Beispiel kam schon in Abschnitt 3.2.9 vor. Dort war die Relation *kind* das gleiche wie die Relation *elter*, nur daß das zweite Attribut nicht projiziert wird. Diese Information kann man zur Bildung möglichst kleiner Sortenprädikate ausnutzen, wie dort auch vorgeführt wurde. Für die Arbeit des Cursors wäre eine solche Information ebenfalls hilfreich, schließlich macht es einen Unterschied, ob eine Wissensquelle auf die Frage „Wer ist der Vater von Olaf?“ mit „Alfons“ oder mit „Alfons ist der Vater von Olaf“ antwortet. Bislang muß der Entwickler selbst dafür sorgen, daß die richtige Information ausgewählt wird.

All diese Überlegungen deuten auf die Notwendigkeit einer umfassenden Klassifizierung aller Wissensquellen bezüglich ihres Anfrage- und Antwortverhaltens hin. Erst eine solche theoretische Arbeit würde die Vollständigkeit eines Anfrageübersetzer-Rahmenwerkes, wie in dieser Arbeit vorgestellt, belegbar machen. Die Einführung von Bindungen und Regeln sind schon ein wichtiger Schritt dorthin.

**Sortenprädikate** Der Einsatz von Sortenprädikaten ist hier nur stiefmütterlich behandelt. Sie erweitern die Anfragemöglichkeit einer nur auf wenige Bindungsmuster eingeschränkten Wissensquelle wesentlich. Jedoch scheitert dieser Ansatz meist daran, daß die Menge der Ausprägungen einer Sorte theoretisch unendlich und praktisch sehr groß ist. Man könnte versuchen, Relationen detailierter zu spezifizieren und die Reihenfolge der Elemente, die ein Sortenprädikat liefert, nach gewissen Heuristiken zu bestimmen[1]. Wichtig ist es auf jeden Fall, die Sorte eines Attributs schon im Exportschema so weit wie möglich einzuschränken.

**Zuteilung der Bindungsmuster** Implementiert wurde bislang ein Algorithmus, der vollständig nach der optimalen Bindungskonstellation sucht. In Abschnitt 4.1 wurde ein besserer, heuristischer Algorithmus angegeben. Hier lohnt es sich, weiter zu forschen. Auch das Zusammenspiel zwischen Bindungskonflikten und dem Weglassen von Regelaktionen ist noch nicht geklärt, insbesondere ist die in Abschnitt 3.2.9 erwähnte Idee, =-Kästen bei Konflikten einzufügen, noch nicht ausgeführt.

**Datenkonvertierung** Wie in der Einleitung erwähnt, wird ein Konflikt von Datenrepräsentationen zu den semantischen Konflikten gezählt. Daher ist dies kein Aufgabengebiet des Modellübersetzers. Hier wurde die Philosophie verfolgt, daß das Exportschema (im

[1] Vielen Dank an Jim Lu für diese Idee.

Mediator- = allgemeinen Datenmodell) möglichst ähnlich dem lokalen Schema (im lokalen Modell) sein sollte. Wenn Mathematica z.B. Polynome als eine Verkettung von Additionen und Potenzierungen darstellt, so wird der entsprechende Anfrageübersetzer Polynome auch genauso darstellen, und nicht etwa als Koeffizientenliste. Dies hat zweierlei Gründe:

1. Warum sollte der Übersetzer etwas anderes exportieren als die Darstellung der Wissensquelle, bzw. *welche* Darstellung sollte er exportieren? Es gibt keinen Grund, eine einzelne zu bevorzugen. Bei Polynomen wäre z.B. auch eine Liste von Koeffizienten-Exponenten-Paaren sinnvoll. Es gibt nicht *die* kanonische Darstellung. Noch deutlicher wird dies, wenn man etwa Personenobjekt-Darstellungen exportiert. Hier wird jede Wissensquelle ihre eigene Darstellung haben, und der Mediator vermutlich noch eine andere bevorzugen. Das Extrem existiert allerdings auch noch in der anderen Richtung: Bei atomaren Objekttypen, etwa den ganzen Zahlen oder Zeichenketten, sollte eine kanonische Darstellung gewählt werden. Diese atomaren Typen gehören schließlich auch zum *Modell* und können nicht im Schema definiert werden.
2. Ein Anfrageübersetzer für eine Wissensquelle sollte, zusammen mit dem Exportschema, unabhängig vom Einsatzort, d.h. vom föderativen Schema, wiederverwendbar sein. Idealerweise kann ein solcher Übersetzer isoliert erstellt werden. Welche Datendarstellung sollte in einem solchen Fall aber gewählt werden, wenn nicht diejenige, die der Wissensquelle am nächsten kommt? Jede Konvertierung erscheint aus diesem Gesichtspunkt unnötig oder zumindest unbegründet.

Dennoch ist das Problem der Datenkonvertierung in KOMET noch nicht zufriedenstellend gelöst. Der Vorschlag, Schemakonflikte mittels Mediatorcode (also GAP-Klauseln) zu lösen, hat hier seine Grenzen, denn dies wäre nicht nur umständlich zu formulieren, sondern auch ineffizient. So müßte z.B. bei zwei verschiedenen Polynomdarstellungen für jede Klausel, in der beide Darstellungen vereint werden, Konvertierungsroutinen explizit aufgerufen werden. Dies würde man gerne verstecken und dem Übersetzer überantworten. Es scheint also notwendig zu sein, zwischen dem Mediator und dem Anfrageübersetzer einen Schemaübersetzer zu einzusetzen. Dieser kann einen Großteil der semantischen Konflikte lösen, und zwar mit eventuell besseren Techniken, als sie der Mediator bietet. Solche zu finden ist eine der wichtigen im Rahmen von KOMET zu erledigenden Arbeiten.

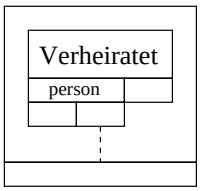
**Typen von Wissensquellen** Im Rahmen dieser Arbeit wurden zwei Typen von Wissensquellen erwähnt: Funktionale und SQL-Wissensquellen. Das Entdecken solcher Typen bietet eine große Hilfe bei der Wiederverwendung von Code. Daher wäre es interessant, weitere Typen auszumachen. Dies erinnert sehr an Design Patterns in der Softwaretechnik oder Problemlösungstypen im Knowledge Engineering.

**Neues Mediatormodell** Sehr wichtig ist die Nutzung des neuen Mediatormodells aus [Trc96]. Das Modell erlaubt komplexe Terme in Anfragen, z.B.

$$ans(Y) \leftarrow Verheiratet(person(X, Y), Z).$$

was alle Nachnamen von verheirateten Personen liefert. Die Kastendarstellung müßte entsprechend um komplexe Stellen erweitert werden, also um Stellen, die andere Stellen

beinhalten. Dabei muß auch das Problem der rekursiv definierten Term-Typen gelöst werden. Die Kastendarstellung des Beispiels könnte wie folgt aussehen:




Spätestens hier wäre es wichtig, auf die Projektions-Fähigkeiten einer Wissensquelle einzugehen. So kann es sein, wie etwa bei ObjectStore, daß die Wissensquelle ihre Ergebnisobjekte zwar vollstrukturiert ausgibt, aber nicht Teile von Objekten, sondern immer nur die ganzen Objekte ausgeben kann, wie auch in [PGGU95] erwähnt.

Komplexe Terme werden zum Beispiel für Mathematica gebraucht. Aber ebenso sinnvoll ist es, objektorientierte Schemata mittels komplexer Typen im Exportschema auszudrücken. So ist es eigentlich umständlich, wie in Abschnitt 3.4.4 alle Klassen als Relation zu repräsentieren, zum einen, weil sie fälschlicherweise als Wurzel interpretierbar sind, zum anderen, weil auf diese Weise die Anfragen im Mediator sehr lang und übersichtlich werden. Es muß aber darauf geachtet werden, daß weiterhin keine unnötigen Redundanzen auftreten und das Prinzip der Objektidentitäten beibehalten wird. Pfadausdrücke lassen sich per komplexen Termen beinahe direkt ausdrücken und müssen nicht mittels besonderer Verbunde simuliert werden.

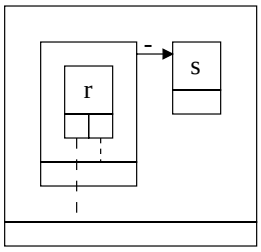
**Darstellung der Negation** Der Algorithmus in Abschnitt 3.3.6 erzeugt aus der Anfrage

$$ans(X) \leftarrow r(X)\ \&\ q(Y)\ \&\ \neg s(X,Y).$$

den relationalen Ausdruck

$$\pi_1(\sigma_{1=3} \circ \sigma_{2=4}(r \times q \times ((\pi_1(r) \times \pi_1(q)) - s)))$$

was viel einfacher als $\pi_1((r \times q) - s)$ ausdrückbar ist. Hier könnte eine nachträgliche Vereinfachung auf Ebene der Kastendarstellung Abhilfe schaffen. Auch die Tatsache, daß die Semantik der Kastendarstellung auf der Basis einer Übersetzung in relationale Ausdrücke definiert ist, verhindert einfachere Kodierungen. So ließe sich das in Abschnitt 3.2.6 vorgestellte Beispiel auch wie folgt darstellen:




Hierbei gibt es die Möglichkeit, von einem Kasten mehrere Projektionen zu definieren, was in einem relationalen Ausdruck nicht möglich ist. Dies verursacht auch implementationstechnische Probleme.

Es wird hier deutlich, daß die Stellen eines K-Kastens „virtuell“ sein müssen. Er hat keine eigenen, denn sie setzen sich aus Stellen der in ihm vorhandenen Kästen zusammen. Die Projektionskanten sind also nicht wirklich Kanten, sondern deuten nur eine Identität von Stellen an. Es ist aber aus theoretischen Gründen – man siehe die Definition (3.2) der Herausproduktion – dennoch sinnvoll, sie als Kanten anzusehen.

**Fremdcursor** In Abschnitt 3.2.4 wurde erwähnt, daß es zwei Ursachen geben kann, wenn eine Konstante in einer Anfrage an eine Wissensquelle vorkommt. Die zweite entsteht, wenn eine Variable im Laufe des Inferenzprozesses des Mediators gebunden wurde. In diesem Fall ist es wahrscheinlich, daß es im weiteren Inferenzprozeß eine weitere andere Bindung an diese Variable geben wird und dieselbe Anfrage an die Wissensquelle gestellt wird, nur mit einer anderen „Konstanten“. Hier ist es sinnvoll, die Konstante ebenfalls als einen Cursor anzusehen, den weiterzuschalten der *Übersetzer* bewerkstelligen kann. Es ist sogar möglich, daß der Wert der entsprechenden zu bindenden Variable direkt von einer Anfrage an eine andere Wissensquelle abhängt, der Cursor also von *dieser* bereitgestellt wird. In diesem Fall ruft der Übersetzer einen anderen an, seinen Cursor weiterzuschalten, d.h. die Übersetzer kommunizieren miteinander. Dieses Szenario soll Bild 5.1 verdeutlichen. Dort steht im Mediatorcode eine Klausel, bei der zwei

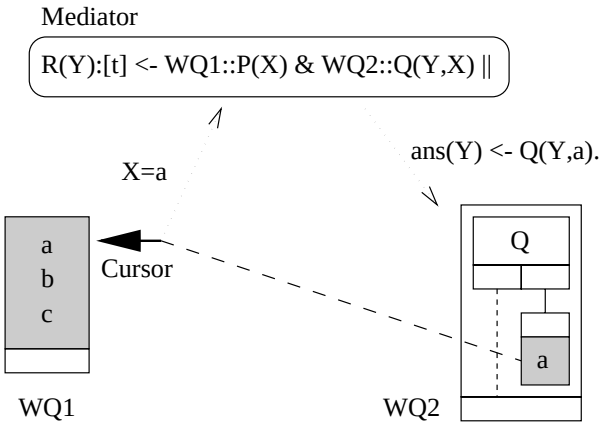



Abbildung 5.1: Fremdcursor

Constraint-Relationen angesprochen werden, beide bezüglich einer anderen Wissensquelle aber mit einer gemeinsamen Variable. Die gepunkteten Pfeile geben an, wie es bislang läuft: Der Mediator liest die einzelnen Werte zur Bindung von $X$ von WQ1 und gibt sie an WQ2 weiter (gezeigt ist der erste Wert ‘a‘). Der große gestrichelte Pfeil verdeutlicht die Idee der Fremdcursor. Hier ist die „Konstante“ in der Anfrage an WQ2 der Cursor auf die Antwort von WQ1. Dies würde den Mediator nicht weiter belasten und unnötige Netzbelastung verhindern.

**Front-end zur vereinfachten Anfrageformulierung** Aufgrund des relationalen Modells sind Anfragen an Mathematica oder ObjectStore nur umständlich formulierbar. Schön wäre es, wenn man statt $plus(Y, Z, 3)$ & $mult(Z, X, 4)$ schreiben könnte $Y = 3 + X * 4$, oder wenn man, wie in HERMES, bei einer Anfrage an eine objektorientierte Quelle einen Pfadausdruck direkt eintippen könnte, statt ihn mittels Verbunde zu simulieren. Ein Vorübersetzer müßte dabei die angenehme Darstellung in die vom Übersetzer

erwartete konvertieren.[2]

**Unsichere Anfragen** Indem der Übersetzer dem Mediator Ausnahmetupel übermittelt, ist es möglich, auch unsichere Anfragen mit Negationen zu bearbeiten. Die unsichere Anfrage $ans() \leftarrow \neg r(1, X)$ würde z.B. für den Fall, daß $I_r = \{\langle 1,2\rangle, \langle 1,3\rangle\}$ ist, zur Antwort „Ja, wenn $X \neq 2$ und $X \neq 3$“ führen. Dazu müßte aber der KOMET-Mediator erweitert werden. Der Übersetzungsvorgang wäre in diesem Fall einfach, da es genügt, die Relation $I_r$ zu materialisieren.

**Zusammenhang Bindungsmuster und Regelaktionen** Am Beispiel der WWW-Anbindung wurde deutlich, daß es zwei Wege gibt, die Unfähigkeit einer Datei/WWW-Seite, irgendwelche Verknüpfungs-Operationen selbst auszuführen, auszudrücken: Zum einen kann man sämtliche Regelaktionen (außer Initialisierung und Materialisierung) unimplementiert lassen, zum anderen kann man nur Bindungsmuster mit os angeben; beides führt zum selben Ziel. Es wäre interessant, die so entstehenden Alternativen zwischen dem Weglassen von Aktionen und der Einschränkung mit Bindungen genauer zu untersuchen und evtl. Äquivalenzen zeigen zu können.

[2] Dank an Joachim Schü für diese und die folgende Idee

# Anhang A

# Glossar

**N-Kasten** Ein N-Kasten wird nicht schattiert dargestellt. Er repräsentiert einen Relationenbezeichner, also einen Namen für ein Prädikat.

**A-Kasten** Ein A-Kasten wird hell schattiert dargestellt. Er repräsentiert eine Wissensquellen-Anfrage.

**M-Kasten** Ein M-Kasten wird dunkel schattiert dargestellt. Er repräsentiert eine Relation = Menge von Tupeln. Darunter fallen auch die Konstanten einer Anfrage.

**K-Kasten** Ein K-Kasten wird (nicht schattiert) mit in ihm gezeichneten Kästen und Kanten dargestellt. Er repräsentiert eine Klammerung, z.B. eine Anfrage oder eine Grundmenge $DOM$ für die Negation.

**Stelle** Eine Stelle wird als kleines Rechteck an einem Kasten gezeichnet. Sie repräsentiert ein Attribut (Spalte) einer Relation bzw. eines Relationenbezeichners.

**Bindungsmuster** Einer Stelle wird eine Bindung $\circ$ oder $\bullet$ zugeordnet. $\circ$ gibt an, daß an dieser Stelle etwas ausgegeben wird, $\bullet$ gibt an, daß an dieser Stelle eine Eingabe erwartet wird.

**Kante** Eine Kante repräsentiert einen Operator der relationalen Algebra bzw. eine in einer Anfrage ausgedrückte Bedingung, z.B. die Gleichheit zweier Variablen. Die Art der Bedingung gibt seine Markierung an. Fehlt sie, so ist die Markierung „=“ gemeint. Es gibt Stellen- und Kastenkanten.

**Stellenkante** Eine Stellenkante verbindet zwei Stellen miteinander. Sie repräsentiert die Gleichheit zweier Variablen in einer konjunktiven Anfrage. Es gibt die Verbund- und die Attributselektionskante.

**Verbundkante** Die Verbundkante ist eine Stellenkante zwischen zwei verschiedenen Kästen. Sie repräsentiert die Verbund-Operation $\bowtie$.

**Attributselektionskante** Die Attributselektionskante ist eine Stellenkante zwischen Stellen, die an dem gleichen Kasten liegen. Sie repräsentiert die Attributselektions-Operation $\sigma$.

**Kastenkante** Eine Kastenkante verbindet direkt zwei Kästen miteinander. Sie repräsentiert einen Operator auf zwei Relationen, also z.B. die Differenz oder Vereinigung.

**Sortenprädikat** Ein Sortenprädikat ist eine Relation, die die Menge der einer Sorte zugehörigen Elemente enthält.

**Relationale Algebra** Zu der relationalen Algebra und deren Operationen siehe die Seiten 7 und 35.

**Relation,Relationenbezeichner,Sorte...** Eine Relation ist eine Menge von Tupeln und ein Relationenbezeichner beschreibt eine Klasse von Relationen, aufgrund deren Name und der Sorten der Attribute. Siehe auch Seite 34.

**DB,DBMS,...** Zu diesen Begriffen siehe Seite 7.

**Schema,Modell,...** Zu diesen Begriffen und den verschiedenen konkreten Modellen siehe Seite 7

**Allgemeines Datenmodell,Exportschema,...** Zu diesen Begriffen, die die verschiedenen Schemata und Modelle in einer Mediatorumgebung beschreiben, siehe Abschnitt 2.1.

# Literaturverzeichnis


[AE95] Sibel Adalı und Ross Emery. A uniform framework for integrating knowledge in heterogenous knowledge systems. In *11th IEEE International Conference on Data Engineering*, pages 513–521, Taipei, Taiwan, March 1995.

[AHV95] Serge Abiteboul, Richard Hull und Victor Vianu. *Foundations of Databases.* Addision-Wesley, 1995.

[AQ95] Sibel Adalı und Xiaolei Qian. Query transformation in heterogeneous information systems. `http://www.cs.umd.edu/projects/hermes`, October 1995.

[ASU86] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman. *Compilers.* Addison-Wesley, Reading, Massachusetts, 1986.

[BF95] Saša Buvač und Richard Fikes. A declarative formalization of knowledge translation. In *The 4th International Conference on Information and Knowledge Management*, 1995. Auch als technical report KSL-94-59, Stanford University.

[Bib93] Wolfgang Bibel. *Wissensrepräsentation und Inferenz: eine grundlegende Einführung.* Vieweg, Braunschweig; Wiesbaden, 1993.

[BNPS89] E. Bertino, M. Negri, G. Pelagatti und L. Sbattella. Integration of heterogeneous database applications through an object-oriented interface. *Information Systems*, 14(5):407–420, 1989.

[CDJS96] J. Calmet, D. Debertin, S. Jekutsch und J. Schü. An executable graphical representation of mediatory information systems. In *12th IEEE International Conference on Data Engineering*, pages 124–131, New Orleans, March 1996.

[CGH94] Armin B. Cremers, Ulrike Griefhahn und Ralf Hinze. *Deduktive Datenbanken.* Vieweg, Braunschweig/Wiesbaden, 1994.

[CJK+96] J. Calmet, S. Jekutsch, P. Kullmann, J. Schü, J. Svec, M. Taneda, S. Trcek und J. Weißkopf. *KOMET.* Interner Bericht, 1996.

[CRE87] B. Czejdo, M. Rusinkiewicz und D. W. Embley. An approach to schema integration and query formulation in federated database systems. In *Proc. 3rd IEEE International Conference on Data Engineering, Los Angeles*, February 1987.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, June 1993.

[Gup89] Amar Gupta (Ed.). *Integration of Information Systems: Bridging Heterogenous Databases*. IEEE Press, 1989.

[HFG87] D. I. Howells, N. J. Fiddian und W. A. Gray. A source-to-source meta-translation system for relational query languages. In *Proceedings 13th VLDB Conference*, pages 227–234, Brighton, 1987.

[Inm96] W. H. Inmon. *Building the data warehouse*. Wiley, New York, 1996.

[JL87] J. Jaffar und J.-L. Lassez. Constraint logic programming. In *Proc. 14th Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[Kul95] Peter Kullmann. Entwurf und Implementierung einer Expertensystem-Shell für annotierte Logik. Diplomarbeit, Institut für Algorithmen und Kognitive Systeme, Universität Karlsruhe, 1995.

[LL95] Stefan M. Lang und Peter C. Lockemann. *Datenbankeinsatz*. Springer, Berlin; Heidelberg, 1995.

[LNS96] Jim Lu, Anil Nerode und V.S. Subrahmanian. Towards a theory of hybrid knowledge bases. *To appear in IEEE Transactions on Knowledge and Data Engineering*, 1996.

[Mye94] K. L. Myers. Hybrid reasoning using universal attachment. *Artificial Intelligence*, 67:329–375, 1994.

[MYK+93] W. Meng, C. Yu, W. Kim, G. Wang, T. Pham und S. Dao. Construction of a relational front-end for object-oriented database systems. In *9th IEEE International Conference on Data Engineering*, pages 476–483, Vienna, Austria, April 1993.

[NFF+91] Robert Neches, Richard Fikes, Tim Finin, Thomas Gruber, Rameshi Patil, Ted Senator und William R. Swarton. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36–51, Fall 1991.

[PGGU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina und J. D. Ullman. A query translation scheme for rapid implementation of wrappers. In *International Conference on Deductive and Object-Oriented Databases*, 1995.

[PL91] Chiang-Choon Danny Poo und Hongjun Lu. Multi-domain expert systems. *Expert Systems*, 8(2):67–73, May 1991.

[PSJ87] Gregory Piatetsky-Shapiro und Gabriel Jakobson. An intermediate transformation to different database languages. *Data & Knowledge Engineering*, 2:1–29, 1987.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, Fr. Eddy und W. Lorensen. *Object-oriented modelling and design*. Prentice-Hall, 1991.

[RC85] Marek Rusinkiewicz und Bogdan Czejdo. Query transformation in heterogeneous distributed database systems. In *Proceedings 5th Conference on Distributed Computing Systems*, Colorado, 1985.

[RSU95] Anand Rajaraman, Yehoshua Sagiv und Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *14th ACM Symposium on Principles of Database Systems*, pages 105–112, May 1995.

[SL90] A. Sheth und J. Larson. Federated database systems for managing heterogeneous and autonomous databases. *ACM Computing Surveys*, 22:183–235, September 1990.

[Tem87] M. Templeton. Mermaid – a front-end to distributed heterogeneous databases. *Proceedings of the IEEE*, 75(5):695–708, May 1987. Auch in [Gup89].

[TL94] Henry Tirri und Greger Lindén. Alchemist - an object-oriented tool to build transformations between heterogeneous data representations. In H. El-Rewini und B.D. Shriver (Eds.), *Proceedings of the 27th Hawaii Int. Conference on System Sciences*, volume II, pages 226–235, January 1994.

[Trc96] Silvia Trcek. Entwurf und Implementierung eines Datenmodells zur Integration heterogener Informationsquellen. Diplomarbeit, Institut für Algorithmen und Kognitive Systeme, Universität Karlsruhe, 1996.

[Ull88] Jeffrey D. Ullman. *Priciples of database and knowledge-base systems*, volume I. Computer Science Press, 1988.

[Ull89] Jeffrey D. Ullman. *Priciples of database and knowledge-base systems*, volume II. Computer Science Press, 1989.

[vHPS94] G. van Heijst, W. Post und A. Th. Schreiber. Knowledge based integration of representation formalisms. In A. Cohn (Ed.), *11th European Conference on Artificial Intelligence*, pages 319–323, Amsterdam, the Netherlands, August 1994.

[Vie89] Bertram Vielsack. Spezifikation und Implementierung der Transformation attributierter Bäume. Diplomarbeit, Fakultät Informatik, Universität Karlsruhe, 1989.

[vM92] Frank von Martial. Einführung in die verteilte Künstliche Intelligenz. *KI-Künstliche Intelligenz*, (1):6–11, März 1992.

[Wei96] Jörg Weißkopf. Entwurf und Implementierung einer portablen graphischen Oberfläche zur Integration von Objektsichten. Studienarbeit, Institut für Algorithmen und Kognitive Systeme, Universität Karlsruhe, 1996.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

[WWRT91] Jack C. Wileden, Alexander L. Wolf, William R. Rosenblatt und Peri L. Tarr. Specification-level interoperability. *Communications of the ACM*, 34(5):72–87, May 1991.