Änderungskontrolle in deduktiven Objektbanken

Manfred Jeusfeld

geboren am 16. August 1960 in
Mesum, jetzt Rheine (Westfalen),
wohnhaft in Eynatten (Belgien)

Vollständiger Abdruck der am 24. Juni 1992 von der Fakultät für Mathematik und Informatik der Universität Passau zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

angenommenen Dissertation.

Erster Gutachter: Prof. Dr. Matthias Jarke
Zweiter Gutachter: Prof. Dr. Hugo Volger

Kurzfassung

Die Rolle von Datenbanken wird traditionell so gesehen, daß sie große Datenmengen über lange Zeiträume für viele Anwender zu verwalten haben. Mittlerweile hat sich aber herausgestellt, daß dieses Verständnis für eine große Zahl von Problemstellungen, z.B. aus dem Bereich der Programmentwicklung, unzureichend ist. Heutzutage erwartet man „intelligente“ Systeme, die an den jeweiligen Anwendungsbereich angepaßt werden können.

In der Forschung der letzten zehn Jahre haben sich zwei Denkschulen herauskristallisiert, wie man diesem Anspruch gerecht werden kann. Die von ihnen propagierten Systeme werden als deduktive bzw. objektorientierte Datenbanken bezeichnet. Dieses Buch erläutert die Eigenschaften beider Systeme und entwickelt ein Konzept, wie sie kombiniert werden können. Das Resultat ist eine deduktive Objektbank. Es stellt sich heraus, daß ein solches System ein breites Spektrum von Anwendungen abdeckt. Beispiele sind die (Meta-)Modellierung von Datenmodellen, die Unterstützung der Softwareentwicklung und die Integration heterogener verteilter Datenbanken.

Zum Geleit

Spätestens seitdem die Künstliche Intelligenz Anfang der 80er Jahre von der Spielzeugebene kleiner Demonstrationsprogramme in praktische Industrieanwendungen hineingewachsen ist, stellt sich die Frage der Integration von KI-Techniken mit denen anderer Softwaresysteme, insbesondere der Datenbanken. Frühe Versuche zur Kopplung erreichten nicht die notwendige Effizienz. Neuere Ansätze erforschen daher integrierte Wissensbank-Managementsysteme (KBMS), in denen die Funktionalität existierender Datenbankmodelle durch aus der KI entlehnte Konzepte erweitert wird, ohne daß die bekannten Vorteile von Datenbanken – Effizienz, Robustheit und Mehrbenutzerunterstützung – verloren gehen.

Zwei Richtungen dominieren die derzeitige Diskussion. Deduktive Datenbanken erweitern das kommerziell weitverbreitete relationale Datenmodell um abgeleitete Informationen, die durch Deduktionsregeln und Integritätsbedingungen in einer Teilmenge der Logik erster Stufe (wegen der Effizienzanforderung) berechnet werden können. Diese deklarative Organisation ermöglicht die sehr weitgehende Optimierung und informative Überbeantwortung von Anfragen, gezielte Änderungskontrolle und aktives Verhalten der Datenbank bei vordefinierten Ereignissen durch Trigger.

Objektorientierte Datenbanken geben den Vorteil der Deklarativität zugunsten einer noch weitergehenden Ableitungsmöglichkeit durch beliebig programmierte Methoden auf. Darüberhinaus beheben sie einen entscheidenden Mangel deduktiver und relationaler Datenbanken, nämlich das Fehlen von Abstraktionsprinzipien zur verbesserten Organisation großer Wissensbanken durch komplexe Objekte, Vererbung, Metaklassenhierarchien und Arbeitskontexte.

Die Dissertation von Manfred Jeusfeld hat sich die Aufgabe gestellt, die Vorteile beider Ansätze in einer deduktiven Objektbanksprache zu kombinieren. Im Vergleich zu einigen konkurrierenden Ansätzen wird dabei ein konservativer Ansatz gewählt, bei der jedes der Sprache hinzugefügte objektorientierte Konstrukt eine präzise definierte logische Semantik haben muß. Herr Jeusfeld kann sogar zeigen, daß eine in seiner Sprache O-Telos definierte Wissensbasis exakt einer speziellen deduktiven Datenbank entspricht, so daß alle für deduktive Datenbanken und ihre Erweiterungen erfundenen Optimierungsverfahren ohne weiteres auf deduktive Objektbanken übertragbar sind. In dieser Hinsicht ist seine Sprachversion ein Fortschritt gegenüber früheren Telos-Versionen, die wir gemeinsam mit der Gruppe von John Mylopoulos (University of Toronto) entwickelt hatten. Gerade auch deshalb konnte mit dem System ConceptBase eine umfassende Implementierung entwickelt werden, die mittlerweile an über zwanzig Forschungsinstitutionen in vier Kontinenten in Gebrauch ist. Dies hebt seinen Ansatz von fast allen anderen Vorschlägen zur Integration deduktiver und objektorientierter Datenbanken ab, deren experimentelle Implementierungen sich in einem sehr viel früheren Stadium befinden.

Die Arbeit geht aber über die bloße Übernahme deduktiv-relationaler Optimierungstechniken deutlich hinaus. Sie zeigt, daß die durch Objektorientierung gewonnene Struktur nicht nur vom Benutzer in der Modellentwicklung genutzt werden kann, sondern auch vom System in der Anfrage- und Integritätsoptimierung, sowie in der Wartung komplexer Sichten auf die Wissensbank. Interessant erscheinen vor allem die Ergebnisse zur statischen Typprüfung und zur

Optimierung von Meta-Sprachkonstrukten, die es gestatten, die Sprache beliebig zu erweitern. Diese Ideen sollen im ESPRIT-Projekt COMPULOG 2 noch um Aspekte des nichtmonotonen Schließens und der Klassensubsumtion erweitert werden.

Mit dem von Manfred Jeusfeld entwickelten ConceptBase-System auf der Basis von O-Telos wurden eine Reihe von Anwendungserfahrungen in den Bereichen Softwareumgebungen, Hypertext-Autorensysteme und Requirements Engineering gesammelt. Die Dissertation beschreibt Arbeiten im ESPRIT-Projekt DAIDA als Beispiel des erstgenannten Anwendungsbereichs und stellt ein Verfahren vor, mit dem nicht nur Datenbanken, sondern auch andere CASE-Tools (insbesondere wissensbasierte Systeme) in die Wissensbank integriert werden können, ohne daß die semantische Kontrolle völlig verlorengeht. In diesem Bereich der *semantischen Interoperabilität* – einem der „heißen Themen“ in der Informatik der 90er Jahre – ist allerdings mit den vorliegenden Ergebnissen nur ein erster Schritt getan. Ein von Manfred Jeusfeld geleitetes Projekt zur Systemintegration im Bereich der Qualitätssicherung in der computergestützten Fertigung soll hier weitere Fortschritte bringen.

Insgesamt stellt die hier vorgelegte Arbeit eine schöne Kombination theoretischer und praktischer Forschungsergebnisse dar, die für alle diejenigen von Interesse sein dürfte, die sich auf Datenbank- oder KI-Seite mit der Verwaltung umfangreicher, strukturierter Wissensbanken beschäftigen.

Aachen, im September 1992 Matthias Jarke

Vorwort

Diese Arbeit entstand während meiner Tätigkeit als Assistent an der Universität Passau in den Jahren 1987 bis 1991. Der ursprüngliche Anlaß war die Aufgabe der Entwicklung einer globalen Wissensbank für den Software-Entwurf innerhalb des Esprit-Projekts DAIDA. Wurde diese Wissensbank auch für einen speziellen Zweck konzipiert, so stellte sich doch bald heraus, daß die ausgewählte Sprache auch für den allgemeineren Begriff einer deduktiven Objektbank tragfähig war.

Ein weiterer Anstoß war ein Vortrag von David Maier auf einer Konferenz in Venedig im Jahre 1990. Dort vertrat er die These, daß Daten und Schema in einer Datenbank zu trennen seien. Vertreter der gegenteiligen Auffassung forderte er auf zu zeigen, daß die zusätzliche Information über das Schema auch zur Effizienzsteigerung im Datenbanksystem nutzbar ist. Und tatsächlich: sie ist nutzbar.

Mein besonderer Dank gilt Prof. Matthias Jarke für seine Unterstützung bei der Definition des Themas und seine wertvollen Hinweise auf noch offene Probleme. Mit Vergnügen denke ich an die vielen Diskussionen mit ihm und meinem Kollegen Thomas Rose zurück. Sie drehten sich um so tiefgreifende Fragen wie „Was ist eine Metaklasse?“ und „Ist ein Werkzeug auch ein Objekt?“ Gut zu wissen, daß wir auf einige der Fragen Antworten fanden.

An dieser Stelle möchte ich auch an die vielen Teilnehmer des Projektes DAIDA erinnern. Es war ein großes Glück für mich, in einem so guten Team mitarbeiten zu dürfen. Alle Namen zu erwähnen, würde den Rahmen sprengen. Besonders danke ich Michael Mertikas, Alain Rouge, Ingrid Wetzel und Ariane Ziegler.

Die Umsetzung der Ergebnisse in ein lauffähiges System wäre nie so weit gediehen, wenn sich nicht so engagierte Studenten wie André Klemann, Eva Krüger, Martin Staudt und Thomas Wenig gefunden hätten. Ich danke Euch sehr!

Prof. Hugo Volger danke ich für die vielen Stunden der Erörterung von Grundlagen der deduktiven Datenbanken. Sie gaben mir ein sicheres Gefühl dafür, welche Rolle Deduktion zu spielen vermag und welche nicht.

Ich denke gern an die Zeit in Passau zurück. Ich danke Shicheng Chen, Stefan Eherer, Ralf Felter, Burkhard Freitag, Udo Hahn, Andreas Miethsam, Ursula von Pilgrim-Zizlsperger, Robert Stabl, Gerhard Steinke und Rainer Unland für ihre Aufgeschlossenheit und dafür, daß ich ein Stück weit mit ihnen gehen konnte.

Immer wieder erstaunt es mich, wieviele Rechtschreibfehler man auf einer einzigen Seite Text produzieren kann. Ein Gruß und ein Dankeschön an Klaus Schüßler.

Das letzte Wort gilt Susan: *Ton sourire est dans mon cœur.*

Eynatten, im September 1992 Manfred Jeusfeld

Inhalt

„Für ihn war – nach eigenem Bekenntnis – die Orgelkunst Johann Sebastian Bachs seit seiner frühesten Jugend Richtschnur. Die Große Orgel in Alkmaar repräsentiert trotz mancher späterer Veränderungen den Orgeltyp des 18. Jahrhunderts. Walcha, der die 'Orchesterorgel' der Romantik ablehnt, findet in einer solchen Orgel das adäquate Instrument für die Darstellung und Herausarbeitung der Bachschen Polyphonie."

Wilfried Hannich im Jahre 1964 über Helmut Walcha

Kapitel 1: Das Ziel der deduktiven Objektbanken

Die Forschung der 70-er Jahre auf dem Gebiet der Datenbanken war stark geprägt vom relationalen Datenmodell [CODD70]. Kritik an seiner Ausdrucksschwäche und anspruchsvolle Anwendungsgebiete wie CAD ließen den Wunsch nach einem geeigneten Nachfolger entstehen. Eine Reihe von Vorschlägen hierzu wurde in den 80-er Jahren gemacht. Im wesentlichen kann man zwei Richtungen unterscheiden: die post-relationalen Datenbanken, die sich als eine Weiterentwicklung der relationalen Datenbanken verstehen, und die objektorientierten Datenbanken, die scheinbar radikal mit dem relationalen Datenmodell brechen. Zwei Zitate sollen die unterschiedlichen Standpunkte verdeutlichen. Ullman beschreibt in seinem Standardwerk [ULLM89] die historische Entwicklung der Datenbanken und stellt folgende Thesen (S. 21-29) auf:

> *"There is a fundamental difference between the object-oriented and logical approaches to design of an integrated DML/host language; the latter is inherently declarative and the former is not. [...] It appears that true object-orientedness and declarativeness are incompatible. [...] In favor of value-orientation, it appears that object-identity preservation does not mesh well with declarativeness. Furthermore, encapsulation, which is another characteristic of object-oriented systems, appears antithetical to declarativeness, as well. [...] Our prediction is that in the 1990's, true KBMS's will supplant the OO-DBMS's just as the relational systems have to a large extent supplanted the earliest DBMS's. [...] We predict that they will be inherently value-oriented and logic-based."*

Er argumentiert, daß deklarative Anfragesprachen die gewünschte Antwort spezifizieren. Dem gegenüber werde in den prozeduralen Anfragesprachen objektorientierter Datenbanken der Algorithmus zur Berechnung der Antwort angegeben. Seiner Meinung nach sollen zukünftige Datenbankmodelle den starken Bezug zur Logik, den das relationale Modell zweifellos hat, erweitern, indem zusätzlich zu den variablenfreien Tupeln nun auch logische Formeln für deduktive Regeln Teil der Datenbank sein können. Im Gegensatz dazu begründet Maier [MAIE86], weshalb objektorientierte Datenbanken für anspruchsvolle Anwendungen wie CAD besser geeignet sind als relationale:

"Why are database systems an infrequent component of CAD systems? [...] Commercial database systems aren't fast enough to support simulators and interactive design tools. [...] Each fetch or store incurs the cost of a procedure call from the application program to the database. [...] A procedure call can't compete with simple offset addressing for accessing a field of program memory. [...] Connections between entities in a relational system are logical, through keys. At least one address translation is required to get from a key value to the location of a tuple. In program memory, records can point to other records directly. [...] An object-oriented database supports complex objects with identity, allows arbitrary connectivity among objects and permits modeling of behavior. [...] Global object identifiers can be swizzled to local memory addresses when an object resides in main memory [...].

Maier sieht als wichtigstes Argument für objektorientierte Datenbanken ihre größere Effizienz bei der Manipulation bestimmter Datenkomponenten. Dies soll durch eine starke Anlehnung an imperative Programmiersprachen erreicht werden. Während also Ullmann eher den Benutzer ins Zentrum stellt, betont Maier die Unterstützung des Programmierers.

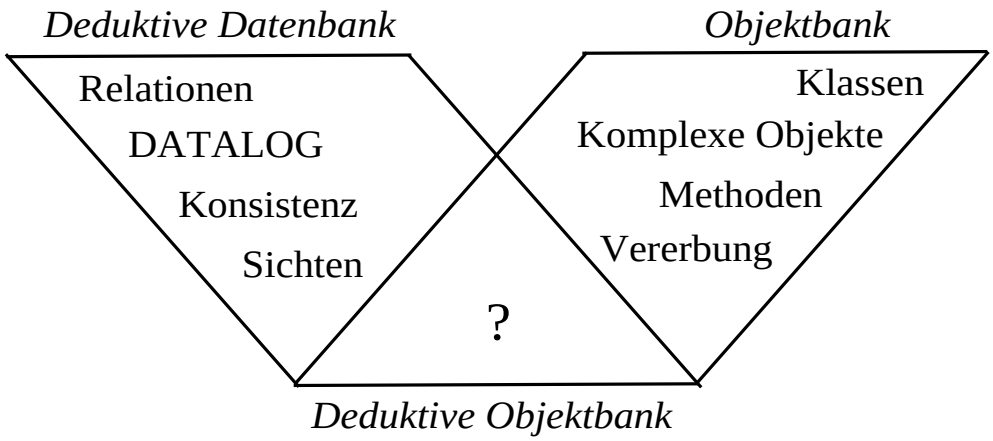

Abb. 1-1: Deduktive Objektbanken

Das Anliegen der vorliegenden Arbeit ist es, die Lücke zwischen den beiden oben aufgeführten Standpunkten schließen zu helfen. Der größere Zusammenhang wird in Abbildung 1-1 veranschaulicht. Deduktive Datenbanken stellen als wesentliche Konzepte deduktive Regeln (DATALOG) und Integritätsbedingungen zur Verfügung. Die kleinste Informationseinheit ist ein Tupel aus einer Relation. Objektbanken hingegen manipulieren sogenannte Objekte, die Klassen zugeordnet werden. Statt der festen Tupel-Datenstruktur

können Objekte beliebig komplex aufgebaut sein. Auf Objekte eines bestimmten Typs sind Operationen (Methoden) anwendbar. Das Konzept der Spezialisierung erlaubt es, Begriffstaxonomien aufzubauen. Mittels Vererbung fällt die Wiederholung bereits definierter Eigenschaften für Unterbegriffe weg.

Gibt es nun sogenannte deduktive Objektbanken, die alle positiven Eigenschaften sowohl der deduktive Datenbanken als auch der Objektbanken vereinigen? Diese Frage wird vielfach als offen angesehen, manchmal wird sie verneint [ULLM91]. Die vorliegende Arbeit widmet sich Teilen der Frage, die konstruktiv beantwortet werden können:

Frage 1) Können die Verfahren zur Integritätsprüfung in deduktiven Datenbanken auf deduktive Objektbanken übertragen werden?

Frage 2) Sind die übertragenen Verfahren mehr oder weniger effizient?

Frage 3) Wie interagieren logische Formeln mit Klassen und komplexen Objekten?

Der Autor geht von folgenden Thesen aus, die zur Rechtfertigung von Entwurfsentscheidungen herangezogen werden:

These 1) Gegenstand von Datenbanken – mithin auch von Objektbanken – ist hauptsächlich die Verwaltung sehr großer Datenmengen. Der operationelle Teil besteht zum großen Teil aus Prozeduren unterhalb der Schwelle der Turing-Vollständigkeit. Hauptbeispiele sind die Ableitung von Information aus der Datenbank (Anfragen, deduktive Regeln) und der Test auf konsistente Zustände (Integritätsbedingungen).

These 2) Die Datenstrukturen einer Datenbank sollen unabhängig von bestimmten Anwendungen sein. Der Wert der Daten sowie der Beschreibung ihrer Eigenschaften übersteigt im allgemeinen den Wert der Anwendungsprogramme.

These 3) Datenbanksysteme sollen in erster Linie benutzerfreundlich und in zweiter Linie programmiererfreundlich sein. Benutzerfreundlichkeit kann durch Verwendung benutzernaher Konzepte wie Logik und Begriffstaxonomien erreicht werden.

These 4) Sowohl Daten als auch die Beschreibung ihrer Eigenschaften (Typsystem, Schema, Datenmodell, Verhalten) sind Aussagen über den modellierten Weltausschnitt und gehören daher zusammen. Die Beschreibung der Eigenschaften sollte in gleicher Weise revidierbar sein wie die Daten selbst.

Besonders These 4 stellt hohe Ansprüche an die Erweiterbarkeit eines Objektbanksystems. Wenn z.B. das Schema oder das Datenmodell in der gleichen einfachen Weise wie die Daten modifizierbar sein sollen, so müssen sie auch einfach (d.h. deklarativ) spezifizierbar

sein. Die Konsequenz einer Änderung am Schema ist aber i.a. viel komplexer als bei der Änderung eines Datums. Um von diesem Unterschied zu abstrahieren, gehen wir von einer *zweistufigem Aufbau* eines Objektbanksystems aus. In der oberen Ebene sind die Spezifikationen der Objekte und Aktionen angesiedelt. Diese Spezifikationen werden möglichst automatisch auf die darunterliegende Implementierungsebene abgebildet.

Der erste Schwerpunkt dieser Arbeit ist die Verfügbarmachung von Methoden zur deduktiven Integritätsprüfung. Sowohl deduktive Regeln als auch Integritätsbedingungen sind anerkannte Konzepte für Datenbanken, die in ihrer Kombination bisher kaum für Objektbanken untersucht wurden [BB*90]. Dieses Ziel muß aufgrund der Komplexität der Aufgabe und der Vielzahl verschiedener Formalisierungen des Objektbankbegriffs näher eingegrenzt werden.

Einschränkung 1) Es wird hauptsächlich die effiziente Auswertung logischer Ausdrücke im Zusammenhang mit Änderungen auf der Objektbank untersucht. Implementierungsaspekte wie die Abbildung auf geeignete Datenstrukturen mit den zugehörigen Operationen sind ausdrücklich nicht Gegenstand dieser Arbeit.

Einschränkung 2) Der Begriff Objektbank wird anhand eines noch auszuwählenden Objektmodells formalisiert. Dieses Modell ist für solche Objektbanken repräsentativ, in denen die Konzepte der Spezialisierung, der Klassifikation und der Aggregation auf Prädikate abbildet werden können.

Die Grundlagen der Logik und der relationalen Datenbanken werden in Kapitel 2 referiert. Eine Erweiterung stellen die *deduktiven Datenbanken* dar. Sie erklären neben Fakten auch sogenannte deduktive Regeln als Teil der Datenbank. Die Regeln werden zum Ableiten neuer Fakten aus bestehenden Fakten herangezogen. Für deduktive Datenbanken können geschlossene logische Formeln als Integritätsbedingungen formuliert werden, die von der Datenbank einzuhalten sind. Integritätsbedingungen sind das Standardbeispiel für logische Formeln, die in starkem Zusammenhang mit Änderungen auf der Datenbank stehen. Anders als in Theorembeweisern werden in Datenbanken traditionell nur einfache logische Formeln zugelassen. Hinzu kommt die Beschränkung auf ganz bestimmte Aussagen, die überhaupt gefolgert werden können. Der Grund ist die extreme Größe der Datenbanken, die den Gigabytebereich vielfach überschreitet. Effizienz ist dann nur noch möglich, wenn Formeln und Kalküle einfach sind. Das für diese Arbeit wichtigste Verfahren ist die Vereinfachungsmethode. Am Ende des Kapitels wird begründet, weshalb relationale und deduktive Datenbanken als erweiterungsbedürftig angesehen werden.

Die am häufigsten zitierten Nachfolger der relationalen Datenbanken sind die objektorientierten Datenbanken, kurz *Objektbanken*. Kapitel 3 gibt einen Überblick über diese Art von Datenbanken. Es werden zwei Aspekte unterschieden. Aus Sicht der Programmiersprachen sind Objektbanken im wesentlichen die Erweiterung einer objektorientierten Programmiersprache um den datenbankspezifischen Aspekt der Persistenz: „Daten können die Lebensdauer eines Prozesses, der sie manipuliert, überdauern". Die zweite Wurzel der Objektbanken sind die Wissensrepräsentationssprachen. Ihr Ziel ist es, Aussagen über einen Weltausschnitt so darzustellen, daß aus ihnen automatisch Schlüsse gezogen werden können. Am Schluß werden die Vorschläge hinsichtlich ihrer Eignung für das Thema dieser Arbeit diskutiert. Die Wahl fällt auf eine einfache Wissensrepräsentationssprache namens Telos [MBJK90]. Sie weist neben Objektidentität und Klassifikation bereits die Konzepte für deduktive Regeln und Integritätsbedingungen auf. Jedoch fehlen typische Eigenschaften wie Objektkomplexität und Programmierbarkeit.

Kapitel 4 entwickelt aus der Wissensrepräsentationssprache Telos ein formales Modell für eine *deduktive Objektbank*. Als Ergebnis kommt eine deduktive Datenbank mit nur einer einzigen Basisrelation bei einer vergleichsweise großen Anzahl von Axiomen heraus. Etwa ein Drittel der Axiome wird aus der Literatur übernommen. Die restlichen Axiome sind neu. Das vorgestellte Objektmodell O-Telos unterscheidet sich von seinem Vorgänger durch

a) die durchgängige Benutzung von Identifikatoren für alle Objekte,

b) das Weglassen der Zeitkomponente,

c) die Beschränkung auf nur fünf Systemklassen und

d) eine strikte Variante der multiplen Generalisierung, die für eine effiziente Behandlung von Spezialisierung im Zusammenspiel mit deduktiven Regeln konzipiert wird.

Prädikatenlogische Formeln werden in einen engen Bezug zur Objektbank gestellt. Danach ist die Übertragung der relationalen Algorithmen einfach. Das Resultat ist die Definition einer deduktiven Objektbank mit Integritätsbedingungen. Die Definition unterstützt Änderungsoperationen bis auf Attributebene.

Die nachfolgenden drei Kapitel sind dem zweiten Schwerpunkt der Arbeit, nämlich dem Zusammenspiel der logischen Formeln mit den speziellen Eigenschaften einer Objektbank gewidmet. Daten in einer Objektbank sind von vornherein einer größeren Zahl von Bedingungen unterworfen als in relationalen Datenbanken. Kapitel 5 zeigt einen Weg auf, wie dieses Wissen für die *Optimierung* von Formeln im Sinne der semantischen Anfrageoptimierung [CGM90] nutzbar zu machen ist. Als Beispiele werden die Objektidentität und die Klassifikation von Attributen vorgeführt. In Kapitel 6 wird das

in den beiden vorangehenden Kapiteln ausgeklammerte Konzept der *Objektkomplexität* behandelt. Komplexe Objekte werden als Sichten auf die Objektbank definiert. Dazu können die in Kapitel 4 erklärten deduktiven Regeln verwendet werden. Unter gewissen Zusatzannahmen kann für komplexe Objekte eine weitere Optimierung der Auswertung von Formeln erreicht werden.

Kapitel 7 behandelt das Zusammenspiel von logischen Formeln und *Metaklassen*. Metaklassen sind solche Klassen, deren Instanzen wiederum Klassen sind. Regeln und Integritätsbedingungen auf dieser Stufe sind besonders ausdrucksstark, jedoch problematisch hinsichtlich ihrer effizienten Auswertung. Eine zweistufige Vereinfachungstechnik weist hier einen Ausweg für eine große Teilmenge von Formeln. Als Anwendung wird die Spezifikation von Datenmodelleigenschaften anhand eines erweiterten Entity-Relationship-Modells vorgestellt.

Das zugrunde gelegte Objektmodell O-Telos klammert die „aktive Komponente“ von Objektbanken, nämlich die *Operationen* (Methoden) auf Objekten aus. Ausnahme sind Methoden, die deduktive Regeln und Integritätsbedingungen auswerten, da diese automatisch aus den Formeln abgeleitet werden können. Eine Implementierung von O-Telos braucht jedoch nicht auf benutzerdefinierte Operationen zu verzichten. Kapitel 8 beschreibt die Funktionalität von Operationen mit einem Prozeßmodell, das das Klassenkonzept und das aus den relationalen Datenbanken bekannte Triggerkonzept kombiniert. Damit kann der Aufruf einer Operation und die Verwaltung der Ergebnisse beschrieben werden. Letzteres macht die Abstraktion zwischen Spezifikations- und Implementierungsebene eines Datenbanksystems explizit.

Die Ergebnisse dieser Arbeit wurden durch die Entwicklung der deduktiven Objektbank *ConceptBase* [JARK91] validiert. Kapitel 9 stellt die Implementierung vor und berichtet über Erfahrungen, die im Projekt DAIDA [JMSV91] gesammelt wurden. Insbesondere stellten sich die von Telos unterstützten Metaklassen als für die Modellierung der Anwendungen in diesem Projekt als vorteilhaft heraus. Anhang 1 enthält ein Protokoll einer Sitzung mit ConceptBase, in dem der Effekt der Formeloptimierung aus Kapitel 5 an einem Beispiel vorgeführt wird. Anhang 2 ist eine Ergänzung von Kapitel 9. Er gibt die vollständige Modellierung der Fallstudie wieder.

Ein entscheidender Grund für den Erfolg der relationalen Datenbanken war ihre Deklarativität, ausgedrückt in einem einfachen Datenmodell und einer benutzernahen Anfragesprache. Durch die Beschränkung auf Sprachen unterhalb der Ausdrucksfähigkeit von Programmiersprachen gelang die Entwicklung effizienter Verfahren zur Anfrageauswertung. Zur Erweiterung der Ausdrucksfähigkeit kamen allgemeine Integritätsbedingungen und deduktive Regeln hinzu. Trotzdem bleiben eingebaute Schwächen dieses Datenmodells bestehen.

Kapitel 2: Logische Formeln in relationalen Datenbanken

Relationale Datenbanken haben einen starken Bezug zur Logik, da die Relationen als Interpretationen von Literalen angesehen werden können. Dies hat die Einführung logikbasierter Anfragesprachen, allgemeiner Integritätsbedingungen und deduktiver Regeln erleichtert, wenn nicht überhaupt ermöglicht. Die letztgenannten Rollen für logische Formeln sind ein Mittel, um die Bedeutung der Daten *innerhalb* der Datenbank zu beschreiben [GEBH87]. Integritätsbedingungen sind Aussagen über die Datenbank, die immer erfüllt sein müssen. Deduktive Regeln sind Formeln, die zur Herleitung neuer Information aus den abgespeicherten Daten dienen.

In letzter Zeit [MGN89] wird auch stärker der Bezug zur Logikprogrammierung gesehen und für die Entwicklung neuer Techniken genutzt [LLOY90]. Dieses Kapitel ist wie folgt aufgebaut. Zunächst werden die Grundbegriffe der Prädikatenlogik nach [RICH78,CGT90,REIT84] wiederholt. Für diese Arbeit wird grundsätzlich nur Logik erster Stufe mit klassischer Interpretation gebraucht. Dann werden relationale Datenbanken [CODD70,ULLM89,GV89,DATE90,VOSS91] gemäß der beweistheoretischen Semantik von [REIT84] definiert. Die Formeln werden so eingeschränkt, daß ihre Wahrheit nur vom Datenbankinhalt und nicht von der Interpretation abhängt. Des weiteren wird die *closed world assumption* als gültig angesehen: alles, was nicht aus der Datenbank folgt, ist als logisch falsch anzusehen.

2.1. Grundbegriffe der Logik erster Stufe

Abzählbare Mengen $VAR=\{x_1, x_2, ...\}$ von sogenannten **Variablen**, $FUN=\{f_1, f_2, ...\}$ von **Funktionssymbolen**, und $\text{PRED}=\{P_1, P_2, ...\}$, $\text{PRED} \cap FUN=\emptyset$, von **Prädikatssymbolen** seien vorgegeben. Auf $\text{PRED} \cup FUN$ sei eine totale, natürlichzahlwertige Funktion *arity* erklärt, die die **Stelligkeit** eines Symbols angibt. Funktionssymbole f mit $arity(f)=0$ heißen auch **Konstanten**. Die Menge T der **Terme** ist als kleinste Menge definiert, die folgende zwei Bedingungen erfüllt: $VAR \subseteq T$ und, falls $t_1, ..., t_n \in T$

und $f \in FUN$, $arity(f)=n$, so ist auch $f(t_1,...,t_n) \in T$. Bei Konstanten wird f mit $f()$ identifiziert. Für ein Prädikatensymbol P mit Stelligkeit n und Terme $t_1,...,t_n$ heißt $P(t_1,...,t_n)$ **Atomformel** oder kurz **Atom**. Ein Atom ohne Variablen in seinen Argumenten heißt **Grundatom** oder **Fakt**. Die Menge WFF der **wohlgeformten Formeln** ist die kleinste Menge mit

1) Jede Atomformel ist in WFF.
2) Falls φ_1,$\varphi_2 \in$ WFF, so auch $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \Rightarrow \varphi_2)$, $(\varphi_1 \Leftrightarrow \varphi_2)$, $\neg\varphi_1$.
3) Falls $\varphi \in$ WFF und $x \in VAR$, so sind auch $\exists x\varphi$ und $\forall x\varphi$ in WFF.

Eine Atomformel L heißt auch **positives Literal**, ihre Negation $\neg L$ **negatives Literal**. Ohne Beschränkung der Allgemeinheit nehmen wir an, daß eine Variable in einer wohlgeformten Formel höchstens einmal quantifiziert vorkommt. Dies kann durch Umbenennung immer erreicht werden. Eine Formel mit mindestens einer frei vorkommenden Variable heißt **offen**, umgekehrt heißt sie **geschlossen**, falls alle Variablen gebunden sind. Klammern werden durch Ausnutzung der üblichen Präzedenzregeln unter logischen Operatoren aus Gründen der Lesbarkeit ggf. unterdrückt und anstatt $\forall(\exists)x\forall(\exists)y\varphi$ schreiben wir auch kurz $\forall(\exists)x,y\,\varphi$. Eine geschlossene Formel der Gestalt

$$Q_1x_1Q_2x_2...Q_mx_m\ \varphi$$

mit $Q_i \in \{\exists,\forall\}, 1 \leq i \leq m$ heißt in **Pränexform**, falls φ keinen Quantor enthält. Die Zeichenkette $Q_1x_1Q_2x_2...Q_mx_m$ heißt **Präfix** der Formel. Die Rest φ heißt auch **Matrix** der Formel.

Die klassische **Interpretation** wohlgeformter Formeln wird durch eine Abbildung auf die Boolesche Algebra angegeben. Variablen werden durch eine Belegungsfunktion u den Elementen eines **Universums** A zugeordnet, die dann zu einem Homomorphismus $\hat{u}$ auf T fortgesetzt wird. Funktionssymbole werden durch Funktionen geeigneter Signatur und Prädikate durch entsprechende Relationen über A interpretiert. Die Auswertungsfunktion v_u ordnet dann jeder Formel φ einen Wahrheitswert aus $\{0,1\}$ zu, indem Atomformeln mit 1 bewertet werden, falls das Tupel in seiner Interpretation existiert, ansonsten mit 0. Auf den logischen Operatoren wird v_u gemäß den üblichen Regeln der Booleschen Algebra homomorph fortgesetzt. Der Wert quantifizierter Formeln $\exists x\varphi$ bzw. $\forall x\varphi$ wird davon abhängig gemacht, ob die Formel φ für alle bzw. eine Belegung den Wert 1 hat.

Man sagt u **erfüllt** φ (bzgl. einer Interpretation I der Funktions- und Prädikatssymbole), falls $v_u(\varphi)$=1. Die Formel φ heißt wahr (falsch), wenn sie von jeder (keiner) Belegung erfüllt wird. Zwei Formeln φ_1, φ_2 heißen (semantisch) **äquivalent**, falls für jede Belegung u gilt: $v_u(\varphi_1) = v_u(\varphi_2)$. Ein **Modell** für eine Formelmenge Σ ist eine

Struktur von Relationen und Funktionen, für die alle Formeln aus Σ in jeder Belegung wahr sind. Der semantische Folgerungsoperator $\Sigma \models \varphi$ besagt, daß jedes Modell für Σ auch Modell für φ ist. Σ heißt **konsistent**, wenn für kein φ gilt: $\Sigma \models \varphi$ und $\Sigma \models \neg\varphi$. Ein syntaktischer Folgerungsoperator (Kalkül) $\Sigma \vdash \varphi$ wird durch eine Menge von Grundformeln (den Axiomen) und eine Menge von Zeichenersetzungsmustern (den Regeln) angegeben. Nach dem Gödel'schen Vollständigkeitssatz für die Logik erster Stufe gibt es vollständige und korrekte Kalküle, in denen also „$\models$“ und „$\vdash$“ übereinstimmen, z.B. der Hilberttypkalkül (siehe [RICH78]). Wir gehen im folgenden von Kalkülen mit vordefiniertem reflexiven, transitiven, kommutativen und substitutiven Gleichheitsprädikat „=“ [RICH78] aus. Statt $\neg(x = y)$ schreiben wir auch kurz $x \neq y$. Ferner seien zwei Atome *true* und *false* vorgegeben. Das erste wird immer als logische 1, das zweite als logische 0 interpretiert.

In der Informatik interessiert vorwiegend der syntaktische Folgerungsoperator, da er die Implementierung von automatischen Theorembeweisern erlaubt. Allerdings gilt hier die Unentscheidbarkeit von $\Sigma \vdash \varphi$. Für syntaktisch eingeschränkte Formelmengen wurden besonders elegante und effizient implementierbare Kalküle entwickelt. Das wohl wichtigste Beispiel sind die Hornklausen mit dem auf ihnen definierten speziellen Resolutionskalkül. Sie führten zur Entwicklung logikorientierter Programmiersprachen, etwa Prolog [CKPR73,KOWA79]. Eine **Hornklause** ist eine geschlossene Formel der Gestalt

$$\forall x_1, \ldots, x_m \; P_1 \wedge \ldots \wedge P_n \Rightarrow P$$

wobei $P_1, \ldots, P_n, P$ Atomformeln sind, und $x_1, \ldots, x_m$ genau die in der Formel vorkommenden Variablen sind. Ein Kalkül besitzt mehrere Freiheitsgrade, welche Regeln auf welche Formeln anzuwenden sind. In einer Implementierung wird eine Strategie angewandt, um diesen Nichdeterminismus zu eliminieren. Im Falle von Prolog wird meist die SLD-Strategie benutzt: man geht von einer Disjunktion negativer Literale aus, dem sogenannten *Ziel*. Es wird immer das erste Literal des Ziels ausgewählt und mit dem Folgerungsprädikat einer passenden Regel resolviert, d.h. es findet eine Variablensubstitution und ein Ersetzen des Zielliterals durch die Bedingungsliterale der Klause statt. Falls am Ende eine leere Zielliste erreicht wird, so ist der Widerspruch der Negation des Ziel bewiesen, also gilt das Ziel. Terminierung ist grundsätzlich nicht garantiert. Insbesondere Linksrekursion kann zu Endlosberechnungen führen. Eine Variante ist die sogenannte SLDNF-Strategie (*negation as failure*). Sie gestattet den Gebrauch negierter Literale in Klausen: $\neg P$ ist wahr, falls P nicht ableitbar ist.

2.2. Relationale Datenbanken als Theorien

Seien $D_1, ..., D_k$ Mengen (die sogenanten Domänen). Dann heißt eine endliche Menge $R \subseteq D_1 \times ... \times D_k$ **Relation**. Ein Element einer Relation heißt auch **Tupel**. Eine Familie $\{R_1, ..., R_s\}$ von Relationen heißt **relationale Datenbank** . Ihr **Schema** wird mittels der Relationennamen und deren Teilmengenbeziehungen zu den kartesischen Produkten der Domänen angegeben. Die Komponenten eines Tupel einer Relationen heißen auch **Attribute**. Aus mnemotechnischen Gründen werden für die Attribute Namen vergeben.

Die Prädikatenlogik wird im Zusammenhang mit relationalen Datenbanken zur Formulierung von Anfragen, deduktiven Regeln und Integritätsbedingungen benutzt. Die vorwiegende Sicht ist dabei die modelltheoretische: die relationale Datenbank ist ein Modell für eine Formel (oder eben auch nicht). Eine gleichwertige Alternative dazu ist die beweistheoretische Sicht. In ihr werden die Tupel einer Relation als Grundatome $P_R(x_1, ..., x_k)$ zu Axiomen einer Theorie. Bei relationalen Datenbanken steht die Verwaltung großer Informationsmengen im Vordergrund und die Ausdrucksstärke eher im Hintergrund. Dies drückt sich in der nun folgenden Zusammenfassung in Form von Einschränkungen aus, die insbesondere die Entscheidbarkeit des Kalküls implizieren. Die erste Einschränkung resultiert aus der 1. Normalform für relationale Datenbanken, die „skalare“ Tupelkomponenten erzwingt [SS83a]. Für die Logik bedeutet dies, daß nur Konstanten als Funktionssymbole zugelassen sind.

Eine **relationale Theorie** ist ein Tripel $(\mathrm{EDB}, \mathrm{AX}, \mathrm{IC})$, wobei EDB die (endlich vielen) Atome für die Tupel der Relationen enthält, AX die Axiome der relationalen Datenbanken, und IC eine Menge von geschlossenen Formeln, die sogenannten Integritätsbedingungen. Folgende Axiome [REIT84] werden zugrunde gelegt:

- ▷ **Bereichsabschluß.** Variablen stehen nur für Konstanten der EDB. Sei $\{c_1, ..., c_t\}$ die endliche Menge solcher Konstanten.

$$\forall\, x\ (x = c_1) \vee ... \vee (x = c_t) \qquad (R_1)$$

- ▷ **Eindeutigkeit der Konstanten.** $\{c_1, ..., c_t\}$ sei wiederum die Menge der Konstanten in EDB. Dann wird für jedes Paar (c_i, c_j) mit $i < j, 1 \leq i, j \leq t$ ein Axiom in AX eingefügt.

$$c_i \neq c_j \qquad (R_{ij})$$

- ▷ **Vollständigkeitsaxiome** Die EDB-Prädikate sind genau aus EDB ableitbar. Sei P ein EDB-Prädikat, und $\{P(c_1^1, ..., c_k^1), ..., P(c_1^m, ..., c_k^m)\} \subseteq$ EDB alle Atome mit Prädikatssymbol P in EDB. Dann ist folgendes Axiom in AX.

$$\forall\, x_1, ..., x_k\ \neg P(x_1, ..., x_k) \vee (x_1 = c_1^1 \wedge ... \wedge x_k = c_k^1) \vee \quad (R_P)$$
$$...\vee(x_1 = c_1^m \wedge ... \wedge x_k = c_k^m)$$

Sonst sind keine Axiome in AX. Das letztere Axiom umschreibt die *closed world assumption*: Tupel, die nicht in EDB sind, sind auch nicht wahr. [REIT84] zeigt damit, daß EDB $\cup$ AX ein eindeutig bestimmtes Modell I hat, und daß jede Formel, die in I wahr ist auch aus EDB $\cup$ AX gefolgert werden kann. Für eine Integritätsbedingung $\varphi \in$ IC fordern wir: EDB $\cup$ AX $\vdash \varphi$. Alternative Definitionen finden sich in [REIT90]. Sie werden aber hier nicht weiter verfolgt. Die Bereiche $D_1, ..., D_k$ einer Relation R können durch einstellige Typprädikate $P_{D_1}, ..., P_{D_k}$ in Integritätsbedingungen der Form

$$\forall\, x_1, ..., x_k\ P(x_1, ..., x_k) \Rightarrow P_{D_1}(x_1) \wedge ... \wedge P_{D_k}(x_k)$$

ausgedrückt werden.

Eine direkte Erweiterung der relationalen Datenbanken sind die **deduktiv-relationalen** Datenbanken [GM78]. Hier wird zwischen **Basisrelationen**, die in der EDB (extensionale Datenbank) dargestellt werden, und **abgeleiteten Relationen** unterschieden, die durch die Formeln der intensionalen Datenbank IDB definiert werden. Für die formale Definition deduktiver Datenbanken sei D eine abzählbare Menge von Konstanten, PRED = PRED$_{EDB}$ $\cup$ PRED$_{IDB}$ $\cup$ $\{=, \mathit{true}, \mathit{false}\}$ eine endliche Menge von Prädikatssymbolen mit PRED$_{EDB}$ $\cap$ PRED$_{IDB}$ $= \emptyset$ und VAR eine Menge von Variablen.

DEFINITION 2-1

Sei EDB eine Menge von Grundatomen über PRED$_{EDB}$ und D, IDB eine Menge geschlossener Formeln der Form

$$\forall x_1, ..., x_n\ \varphi \Rightarrow P(y_1, ..., y_k) \quad (1)$$

mit $y_i \in D \cup \{x_1, ..., x_n\}$. Ferner sei IC eine Menge geschlossener Formeln. Dann heißt das Tripel (EDB, IDB, IC) **deduktive Datenbank**. Die Formeln in IDB heißen auch **deduktive Regeln**.

Die Definition ist in ähnlicher Form auch z.B. in [ML90] zu finden. Auf die explizite Angabe der Axiome aus AX kann mit den folgenden Festlegungen verzichtet werden.

A1) Die *closed world assumption* wird durch eine Inferenzregel $\Sigma \vdash \neg L$ gdw. $\Sigma \nvdash L$ für Grundatome L beschrieben.

A2) Als Interpretationen werden nur *Herbrand*-Interpretationen [CGT90] betrachtet. Darin werden Konstanten durch sich selbst interpretiert. Folglich sind verschiedene Konstanten immer ungleich.

A3) Als Formeln der deduktiven Datenbank werden nur die sogenannten bereichsbeschränkten (engl: *range-restricted*) Formeln [NICO82,DECK86] zugelassen, die den Bereichsabschluß sicherstellen. Bereichsbeschränkte Formeln können syntaktisch beschrieben werden.

Eine Verallgemeinerung der deduktiven Datenbanken sind die disjunktiven deduktiven Datenbanken (siehe z.B. [VOLG89,DECK91]). In solchen Datenbanken sind Klausen als Regeln zugelassen, die eine Disjunktion von positiven Literalen als Folgerung haben. Die Schwierigkeiten dieser Art von Deduktion ist die Nichteindeutigkeit der Semantik [VOLG89], wenn in einer Formel Negation vorkommt, und das Fehlen effizienter Auswertungsalgorithmen. Die Theorie der disjunktiven Datenbanken ist noch in der Entwicklung, während für herkömmliche deduktive Datenbanken bereits effiziente Implementierungen existieren. Wir beschränken uns daher für diese Arbeit auf deduktive Regeln mit genau einem Folgerungsprädikat.

Definition 2-2 (nach [BDM88]) gibt eine einfache Version der Charakterisierung für Bereichsbeschränkung an. Intuitiv ausgedrückt werden quantifizierte Variablen an Werte aus der (endlichen) Datenbank gebunden. Da dem Gleichheitsprädikat „=“ keine Relation der Datenbank entspricht und es eine überabzählbare Extension hat, wird es nicht zur Bindung von Variablen an Konstanten zugelassen.

DEFINITION 2-2

Eine geschlossene Formel heißt **Bereichsformel** genau dann, wenn sie in eins der folgenden Formate äquivalent transformierbar ist:

$$\exists x_1, \ldots, x_n \; L_1 \wedge \ldots \wedge L_m \wedge \varphi \tag{2}$$

$$\forall x_1, \ldots, x_n \; \neg L_1 \vee \ldots \vee \neg L_m \vee \varphi \tag{3}$$

$L_1, \ldots, L_m$ sind positive Literale. Die Variablen $x_1, \ldots, x_n$ $(n \geq 0)$ kommen in mindestens einem der Literale $L_1, \ldots, L_m$ ohne dem Gleichheitsliteral vor. Die Formel φ ist selbst in einem der beiden Formate, oder sie ist von der Form $(\varphi_1 \wedge \varphi_1)$ bzw. $(\varphi_1 \vee \varphi_2)$, wobei sowohl φ_1 als auch φ_2 in einem der beiden Formate sind, oder φ ist *true* bzw. *false*. Die Literale $L_1, \ldots, L_m$ heißen auch **Bereichsprädikate**.

Als Transformationsregeln werden die üblichen Äquivalenzen der Boole'schen Algebra angenommen. Häufig werden wir den Implikationsoperator $\varphi_1 \Rightarrow \varphi_2$ anstatt $\neg\varphi_1 \vee \varphi_2$ verwenden.

In einer deduktiven Datenbank (EDB, IDB, IC) sind die Atome der EDB trivialerweise Bereichsformeln. Damit eine deduktive Regel (1) Bereichsformel ist, muß sie als

$$\forall x_1, \ldots, x_n \; \neg L_1 \vee \ldots \vee \neg L_m \vee (\varphi \vee P(y_1, ..., y_k)) \tag{4}$$

geschrieben werden können. Die dazugehörige Formel

$$\forall x_1, \ldots, x_n \; \neg L_1 \vee \ldots \vee \neg L_m \vee \varphi \tag{5}$$

heißt **Bedingungsformel** der deduktiven Regel. Die hier benutzte Definition für deduktive Regeln weicht von der üblichen Notation, nämlich DATALOG (siehe [CGT90]), ab. In DATALOG sind nur Hornklausen über Konstanten und Variablen zugelassen. Eine Variante, DATALOG$^\neg$, erlaubt auch Negationen der Bedingungsliterale in der Klausel. Nach [LT84] kann eine deduktive Regel der Form (1) immer in eine endliche Zahl von DATALOG$^\neg$-Klauseln transformiert werden. Diese sind wegen Form (4) auch immer bereichsbeschränkt [NICO82,DECK87,SK88]. Integritätsbedingungen sind allgemeine Bereichsformeln. Manche Autoren verwenden auch hier die Hornklauselform als sogenannte *denials* [SK88,OLIV91]. Hier gilt wieder grundsätzlich die Gleichwertigkeit durch Transformation (siehe z.B. [SK88]).

Bereichsbeschränkte Formeln garantieren nicht nur das Bereichsabschlußaxiom, sondern eine weitere wichtige Eigenschaft: der Wahrheitswert der Formel hängt nur von den Prädikaten in der Formel ab. Da Prädikate mit Relationen in enger Beziehung stehen, kann somit bei Änderungen an Relationen, die in einer Formel nicht auftauchen, ausgeschlossen werden, daß sie diese Formel (direkt) betreffen. Eine ausführliche Diskussion dieser Eigenschaft findet sich in [BRY88] für den Fall quantifizierter Anfragen.

In deduktiven Datenbanken ist man aus Effizienzgründen nicht an der Ableitung beliebiger Formeln interessiert, sondern im Grunde nur an den abgeleiteten Relationen. Diese Ableitung ist für reines DATALOG, wo die intensionale Datenbank nur aus funktionsfreien Hornklausen besteht, problemlos. Hier stimmen „$\models$“ und der Resolutionskalkül „$\vdash$“ überein [CGT90]. Falls IDB endlich ist, so ist auch die Menge der Konsequenzen endlich:

$$cons(\mathrm{EDB} \cup \mathrm{IDB}) := \{L |\; L \text{ Grundatom}, \mathrm{EDB} \cup \mathrm{IDB} \vdash L\} \tag{6}$$

Für ein k-stelliges Prädikat $P \in \mathrm{PRED}_{EDB} \cup \mathrm{PRED}_{IDB}$ heißt

$$ext(P) := \{P(x_1, \ldots, x_k) |\; x_1, ..., x_k \in D\} \cap cons(\mathrm{EDB} \cup \mathrm{IDB})$$

die Extension dieses Prädikats. Für ein Prädikatsvorkommen mit Variablen und Konstanten wird gefordert, daß die Extension an den Stellen der Konstanten mit dem

Prädikatsvorkommen übereinstimmt:

$$ext(P(v_1, ..., v_k)) := ext(P) \cap \{P(x_1, \ldots, x_k) |\ v_i \in D \Rightarrow x_i = v_i\}$$

Anfragen werden wie bei der Logikprogrammierung [KOWA79] durch ein negatives Literal $\neg P(v_1, ..., v_k)$ sowie durch deduktive Regeln mit k-stelligem Folgerungsprädikat P formuliert. Die Menge $ext(P(v_1, ..., v_k))$ heißt dann auch **Antwort** auf die Anfrage. Wenn eine Antwort über längere Zeit aufbewahrt und mit der Datenbank aktuell gehalten werden soll, so nennt man sie auch **Sicht**.

Für die Berechnung von *cons* wird ein Fixpunktoperator definiert. Er garantiert neben Korrektheit und Vollständigkeit insbesondere die Terminierung. Das Resultat ist das eindeutig bestimmte minimale Herbrand-Modell der IDB. Falls jedoch die IDB auch deduktive Regeln mit negativen Bedingungsliteralen enthält, so ist das minimale Herbrand-Modell nicht mehr eindeutig. Für diese unerwünschte Situation wurde der Begriff der Stratifikation entwickelt, mit dem die Regeln der intensionalen Datenbank so eingeschränkt werden, daß es wieder eindeutige Modelle gibt, die sogenannten **perfekten Modelle**. Eine **Stratifikation** ist eine Partition $\{S_1, ..., S_r\}$ der zugelassenen Prädikatssymbole PRED mit den folgenden zwei Eigenschaften:

Seien L_1, L_2 zwei Literale, die in einer Regel der IDB vorkommen. P_1 sei der zu L_1 gehörige Prädikatsname mit $P_1 \in S_i$, und P_2 der zu L_2 gehörige Prädikatsname mit $P_2 \in S_j$. Die Elemente der Partition heißen auch **Stratifikationsebenen**.

1) Wenn L_1 Folgerungsprädikat einer Regel ist, in der L_2 positiv in der Bedingung vorkommt, dann gilt $i \geq j$.

2) Wenn L_1 Folgerungsprädikat einer Regel ist, in der L_2 negativ in der Bedingung vorkommt, dann gilt $i > j$.

Nicht alle deduktiven Datenbanken sind stratifizierbar. Ein einfaches Beispiel ist die IDB, die nur aus der folgenden Regel besteht (aus [CGT90]):

$$\forall\ x, y\ Q(x, y) \wedge \neg P(x, y) \Rightarrow P(x, y) \tag{7}$$

Der Test auf Stratifizierbarkeit wird durch eine einfache Suche nach Zykeln in einem Graphen realisiert. Falls eine deduktive Datenbank stratifiziert ist, so existiert mindestens ein minimales Herbrand-Modell, das durch schrittweise Fixpunktberechnung auf den Stratifikationsebenen berechnet werden kann. Das so erhaltene eindeutig bestimmte Modell heißt auch *perfektes* Modell, da es die minimale Anzahl von Grundatomen aus

höheren Stratifikationsebenen enthält. Stratifizierbarkeit ist ein hinreichendes Kriterium für die Existenz von eindeutigen Modellen für DATALOG$^\neg$- Regelmengen. Es gibt mit dem Begriff der *lokalen Stratifizierbarkeit* noch ein schwächeres hinreichendes Kriterium [CGT90]. Es kann jedoch erst zur Laufzeit getestet werden und hat wegen dem daraus folgenden Effizienzproblem noch keine große Bedeutung erlangt. Wie [AK89] zeigen, kann eine Anfrage an eine stratifizierbare, bereichsbeschränkte deduktive Datenbank in polynomieller Zeit in der Größe der Datenbank beantwortet werden.

DEFINITION 2-3

Eine deduktive Datenbank $(\mathrm{EDB}, \mathrm{IDB}, \mathrm{IC})$ heißt **integer**, wenn sie stratifizierbar ist und für alle $\varphi \in \mathrm{IC}$ gilt:

$$cons(\mathrm{EDB} \cup \mathrm{IDB}) \vdash \varphi$$

Wenn ein $\varphi \in \mathrm{IC}$ diese Bedingung nicht erfüllt, so sagt man, daß φ **verletzt** sei.

Der Integritätsbegriff in deduktiven Datenbanken ist in mehrfacher Hinsicht schwächer als der Konsistenzbegriff der Prädikatenlogik. So werden nicht beliebige Formeln auf Konsistenz hin untersucht sondern nur die Integritätsbedingungen. Außerdem beschränkt sich der Kalkül auf die Ableitung von Grundatomen. Die Rechtfertigung der Einschränkungen sind die enormen Größen, die Datenbanken annehmen können. De facto sind nur solche Operationen handhabbar, die schneller als in linearer Zeit ausgeführt werden können.

2.3. Deduktive Integritätstester

Datenbanken werden traditionell für zwei Aufgaben eingesetzt: Information eingeben (die *Änderungen*) und Information ausgeben (die *Anfragen*). Das Problem ist nun, über die ganze Lebensdauer der Datenbank ihre Integrität sicherzustellen. Dazu muß erstens eine Verletzung einer Integritätsbedingung festgestellt werden, und zweitens muß die Verletzung behoben werden. Die letztere Aufgabe kann z.B. durch einfache Rückgängigmachung der Änderung bewirkt werden. Weitergehende Ansätze, die zur Einhaltung der Integritätsbedingungen auch die bestehende extensionale Datenbank „reparieren“ sind in [CW90,ML90] beschrieben.

Für das Feststellen einer Integritätsverletzung genügt es, nach jeder Änderung die endlich vielen Formeln in IC auszuwerten. Diese Methode verbietet sich jedoch für große Datenbanken aus Effizienzgründen. Daher wurden Verfahren entwickelt, den Suchraum der zu testenden Integritätsbedingungen in Abhängigkeit von der Änderung zu begrenzen. Das wohl erfolgreichste Verfahren ist die Vereinfachungsmethode [NICO79,NICO82]. Sie

wurde von [LST86,DECK87, BDM88,SK88] auf deduktive Datenbanken verallgemeinert[1]. Dazu wird der Begriff der Transaktion wie folgt eingeführt [ML90]:

DEFINITION 2-4

Sei (EDB,IDB,IC) eine deduktive Datenbank. Dann heißt eine Sequenz

$$T =< op_1(p_1), op_2(p_2), ..., op_m(p_m) >,$$

wobei $op_i \in \{\text{Insert}, \text{Delete}\}$ und p_i Grundatome von EDB-Prädikaten, eine **Transaktion**.

Eine Transaktion wird entweder ganz oder gar nicht ausgeführt (*Atomizität*). Zur effizienten Auswertung von Integritätsbedingungen geht man davon aus, daß vor einer Transaktion T die Datenbank integer ist. Anstatt aller Integritätsbedingungen müssen dann nur die *vereinfachten Formen simp*(φ) mit $\varphi \in$ IC, in die jeweils die Information eines *op* aus T eingebunden ist, auf der neuen Datenbank getestet werden. Die Funktion *simp* ist total und berechenbar (siehe Abb. 2-1) und liefert eine endliche Menge sogenannter **Trigger** . Ein Trigger hat nach [DATE90] allgemein die Form

`ON` *Event* `CHECK` ψ `THEN` *Action* `ELSE` *Action*

Für den Zweck des Integritätstests steht *Event* für ein Element einer Transaktion wie oben beschrieben. Dabei dürfen die Grundatome auch Platzhalter enthalten. Diese werden im folgenden mit gestrichenen Symbolen x' gekennzeichnet. Die Formel ψ ist eine zu testende vereinfachte Form einer Integritätsbedingung. In ihr können an den Stellen, an denen Konstantensymbole erlaubt sind, auch die Platzhalter aus dem *Event*-Teil vorkommen. Die THEN- und ELSE-Teile sind im folgenden weggelassen. Sie bestünden je nach Ausgang des Tests auf ψ aus dem Akzeptieren der Transaktion bzw. ihrer Rückgängigmachung oder der Reparatur der Datenbank (siehe unten).

Eingabe des Algorithmus in Abbildung 2-1 ist eine Integritätsbedingung φ. Es wird die folgende Hilfsdefinition benötigt. Sei φ_p eine (effektiv berechenbare) äquivalente Pränexformel zu φ. Eine Variable x_i von φ heißt **total allquantifiziert**, falls das Präfix von φ_p die Form

$$\forall x_1 \forall x_2 \ldots \forall x_j \; Q_{j+1} x_{j+1} \ldots Q_m x_m$$

hat und $1 \leq i \leq j$ gilt. Vor dem Allquantor von x_i dürfen mithin keine Existenzquantoren stehen.

[1] Bei Leistungsmessungen [DW89] haben sich die Varianten von [LST86] und [BDM88] als die effizientesten erwiesen.

Für jedes Literalvorkommen wird genau ein Trigger erstellt. Durch die Unifikation fallen i.a. Allquantoren, die nicht unter Existenzquantoren stehen, weg. Die Methode ist immer dann effizient, wenn das unifizierbare Literal nicht hinter Existenzquantoren steht. Letztere können durch das Vereinfachungsverfahren nicht eliminiert werden (siehe z.B. [SV87]). Algorithmus 2-1 wird zur Zeit der Einfügung einer neuen Integritätsbedingung ausgeführt. Die berechneten Trigger werden zum Testen einer Transaktion T auf mögliche Integritätsverletzungen eingesetzt. Falls eine Operation *op* aus T mit dem ON-Teil eines Triggers unifizierbar ist, so wird der CHECK-Teil auf dem neuen Datenbankzustand getestet. Die in Schritt 2.1 eingeführten Konstantensymbole werden dabei als Platzhalter benutzt.

1) Forme φ in seine Bereichsform φ' um. Setze $Trig = \emptyset$.
2) Führe für jedes Literalvorkommen L in φ' aus:
   - 2.1 Bilde L', indem in L alle Variablenvorkommen x durch neue Konstantensymbole x' ersetzt werden.
   - 2.2) Ersetze in φ' die total allquantifizierten Variablen von L durch die entsprechenden Komponenten von L'. Streiche alle Quantifizierungen der ersetzten Variablen. Falls L negatives Literal, dessen Variablen alle ersetzt wurden, dann ersetze L durch *false*. Sei ψ das Ergebnis der Ersetzung.
   - 2.3) Wende Neutralitätsgesetz und Idempotenz der Boole'schen Algebra an, um Vorkommen von *true* bzw. *false* in ψ zu eliminieren.
   - 2.4) Falls L positiv, so füge „`ON` Delete(L') `CHECK` ψ“ der Menge $Trig$ hinzu.
   - 2.5) Falls L negativ, so füge „`ON` Insert(L') `CHECK` ψ“ der Menge $Trig$ hinzu.
3) Gebe $Trig$ aus.

Abb. 2-1: Algorithmus zur Berechnung von Triggern nach [BDM88]

Die Vereinfachungstechnik kann auch auf deduktive Regeln angewandt werden, um die Konsequenzen einer Änderungsoperation effizient zu berechnen. In [JK90] wird die Bedingungsformel einer Regel vereinfacht. Je nach Vorzeichen des Literals in der Bedingungsformel können durch die so vereinfachte Regel die Änderungen an der Extension des Folgerungsprädikats effizient berechnet werden. Ein weitergehender Ansatz wird in [OLIV91] vorgeschlagen. Dort werden für jedes Prädikat P zwei neue Prädikate τP und δP eingeführt. Das erste ist wahr, wenn ein entsprechendes Prädikat P in einer Transaktion neu eingefügt wird, das zweite, wenn es gelöscht wird. Für positive Literalvorkommen von

P gilt dann

$$\forall\, x_1, ..., x_k\ P'(x_1, ..., x_k) \Leftrightarrow (P(x_1, ..., x_k) \wedge \neg \delta P(x_1, ..., x_k)) \vee \tau P(x_1, ..., x_k)$$

und für negative Literalvorkommen

$$\forall\, x_1, ..., x_k\ \neg P'(x_1, ..., x_k) \Leftrightarrow (\neg P(x_1, ..., x_k) \wedge \neg \tau P(x_1, ..., x_k)) \vee \delta P(x_1, ..., x_k)$$

Dabei steht P' für das Prädikat *nach* der Transaktion. Mit dieser Äquivalenz werden Literale der deduktiven Regeln umgeschrieben. Das Ergebnis sind deduktive Regeln (sogenannte **interne Ereignisregeln**) für $\tau P(x_1, ..., x_k)$ und $\delta P(x_1, ..., x_k)$, die die den Effekt einer Transaktion auf die Extension des (abgeleiteten) Prädikats P beschreiben. Wie in [BDM88] wird zum Zeitpunkt der Eintragung einer deduktiven Regel ein Abhängigkeitsnetz aufgebaut, das beschreibt, in welche Trigger von Integritätsbedingungen und welche internen Ereignisregeln die Extensionen von τP bzw. δP eingehen. Durch Umformung in Hornklauselform entstehen bis zu 2^h interne Ereignisregeln für jede deduktive Regel, wobei h die Anzahl der Prädikate im Bedingungsteil der Regel ist. Dieser Nachteil wird in [KÜCH91] teilweise behoben.

Die meisten Integritätstester gehen davon aus, daß eine Transaktion zunächst ganz ausgeführt wird, und dann erst die Integrität der Datenbank getestet wird. Ist sie verletzt, so ist die Rückgängigmachung der Transaktion die (programmtechnisch) einfachste Art der Sicherstellung der Integrität. Sie hat allerdings den Nachteil, daß die Information der Transaktion verloren geht. Häufig liegt die „wahre“ Ursache der Integritätsverletzung in Fakten, die in früheren Transaktionen eingetragen bzw. gelöscht wurden. Vor diesem Hintergrund schlägt [ML90] eine Methode der **Reparatur** vor, die die extensionale Datenbank so ändert, daß keine Verletzung mehr vorliegt. Die Methode leitet aus einem fehlgeschlagenen Beweis für eine Integritätsbedingung die sogenannten *Sypmptome* ab, also Fakten, die notwendig für das Fehlschlagen des Beweises sind. Diese Fakten sind möglicherweise aus deduktiven Regeln abgeleitet. In einem zweiten Schritt werden die Symptome auf die sogenannten *Ursachen*, d.h. Fakten der EDB bzw. ihre Negationen, die notwendig für die Integritätsverletzung sind, abgebildet. Eine Reparatur ist dann eine Ausführung der Ursache auf der Datenbank, d.h. negative Fakten werden eingefügt und positive gelöscht. Ein iterativer Algorithmus zählt solche Reparaturen auf, die keine Integritätsbedingungen mehr verletzen. Im allgemeinen gibt es mehrere mögliche Reparaturen. Sie werden nach Kriterien wie Länge, Anzahl der Löschungen usw. sortiert, so daß die kostengünstigste Reparatur als erstes vorgeschlagen wird.

2.4. Beispiel

Das folgende Beispiel (siehe auch [JJ91]) zeigt eine Integritätsbedingung und eine deduktive Regel für eine Patientendatenbank. Patienten sind durch Name, Alter und das Medikament, das sie nehmen, charakterisiert. Eine weitere Relation beschreibt die Symptome, an denen die Patienten leiden. Medikamente bestehen aus Wirkstoffen, die wiederum auf Symptome wirken. Eine deduktive Regel besagt, daß ein Medikament ein Symptom bekämpft, falls es einen Wirkstoff gegen dieses Symptom enthält. Die letzte Relation über Allergien dient zur Formulierung einer Integritätsbedingung: Patienten dürfen nur dann ein Medikament nehmen, wenn dieses Medikament eines ihrer Symptome bekämpft, und wenn sie nicht gegen einen der Wirkstoffe allergisch sind.

Schema der EDB:

$$Patient(pname, age, takesDrug),\ Suffers(patient, symptom)$$
$$Drug(dname, component),\ Agent(aname, effectedSymptom)$$
$$Allergy(pname, aname, since)$$

IDB:

$$\forall\ d, s, a\ Drug(d, a) \wedge Agent(a, s) \Rightarrow Against(d, s) \tag{8}$$

IC:

$$\begin{aligned} \forall\ p, x, d\ Patient(p, x, d) \Rightarrow & (\exists s\ Suffers(p, s) \wedge Against(d, s)) \wedge \\ & (\forall\ a, t\ Drug(d, a) \Rightarrow \neg Allergy(p, a, t)) \end{aligned} \tag{9}$$

In der Integritätsbedingung (9) kommen fünf Literale vor: $Patient(p, x, d)$ und $Allergy(p, a, t)$ sind in der zugehörigen Bereichsformel negativ, die restlichen Literale sind positiv. Mit Algorithmus 2-1 ergeben sich die Trigger von Abbildung 2-2.

Das Beipiel zeigt eine Schwäche zumindest dieser Darstellung logischer Formeln für relationale Datenbanken. In der Integritätsbedingung wird über das Alter eines Patienten quantifiziert, obwohl das Alter von der Intention her keine Rolle spielen sollte. Durch die Aufteilung der Daten auf Relationen steht das Alter aber immer zusammen mit Patientenname und dem Medikament. Ohne weitere Vorkehrungen werden also Änderungen an der Komponente für das Patientenalter zu unnützen Auswertungen der vereinfachten Form führen.

Ein weiterer Nachteil (siehe auch [JACK90]) ist die unnatürliche Aufteilung der Information. So realisiert die Relation *Suffers* eine Eigenschaft von Patienten. Dieses Wissen ist allerdings bei der Abbildung auf das relationale Datenmodell verlorengegangen. Zudem verlangt das relationale Datenmodell in seiner ursprünglichen Form keine referentielle Integrität: ein Tupel $Suffers(p, s)$ kann existieren, ohne daß es einen Eintrag in der

ON Insert($Patient(p', x', d')$) CHECK
$(\exists\ s\ Suffers(p', s) \wedge Against(d', s))\ \wedge (\forall\ a, t\ Drug(d', a) \Rightarrow \neg Allergy(p', a, t))$

ON Delete($Suffers(p', s')$) CHECK $\forall\ x, d\ Patient(p', x, d) \Rightarrow$
$(\exists\ s\ Suffers(p', s) \wedge Against(d, s))\ \wedge (\forall\ a, t\ Drug(d, a) \Rightarrow \neg Allergy(p', a, t))$

ON Delete($Against(d', s')$) CHECK $\forall\ p, x\ Patient(p, x, d') \Rightarrow$
$(\exists\ s\ Suffers(p, s) \wedge Against(d', s))\ \wedge (\forall\ a, t\ Drug(d', a) \Rightarrow \neg Allergy(p, a, t))$

ON Insert($Drug(d', a')$) CHECK $\forall\ p, x\ Patient(p, x, d') \Rightarrow$
$(\exists\ s\ Suffers(p, s) \wedge Against(d', s))\ \wedge (\forall\ t\ \neg Allergy(p, a', t))$

ON Insert($Allergy(p', a', t')$) CHECK $\forall\ x, d\ Patient(p', x, d) \Rightarrow$
$(\exists\ s\ Suffers(p', s) \wedge Against(d, s))\ \wedge \neg Drug(d, a')$

Abb. 2-2: Trigger für Integritätsbedingung (9)

Patientenrelation mit Schlüssel p gibt. Diese wünschenswerte Eigenschaft wurde erst in [CODD79] dem Relationenmodell hinzugefügt.

Das Beispiel zeigt, daß die Vereinfachungsmethode besonders effizient für Einfügeoperationen auf Relationen mit vielen Attributen ist. Hier fallen besonders viele Quantoren weg (siehe erster Trigger in Abb. 2-2). Das Dilemma ist jedoch, daß eine hohe Anzahl von Attributen auch die Anzahl unerwünschter Quantoren (wie oben das Attribut für das Patientenalter) potentiell erhöht. Die *Normalisierung* [ULLM89] von Relationenschemata zur Vermeidung von sogenannten Änderungsanomalien führt allerdings in der Tendenz eher zu Relationen mit wenigen Attributen.

2.5. Diskussion

Relationale Datenbanken haben wichtige Vorteile. Als erstes sei ihre klare, einfache Theorie genannt und die daraus resultierende Entscheidbarkeit. Die Einfachheit der Theorie wird durch Beschränkung auf „flache“ Relationen erreicht. In der Logik fallen somit komplexe Terme mit Funktionen weg. Desweiteren werden nur endliche Strukturen betrachtet. Variablen werden nur mit den endlich vielen Konstanten der Datenbank belegt. Diese zwei Einschränkungen implizieren die Entscheidbarkeit von (logischen) Anfragen. Der zweite Vorteil ist die Möglichkeit, deklarativ Zusammenhänge in Form von Anfragen, deduktiven Regeln und Integritätsbedingungen aufzuschreiben. Direkt verbunden damit sind vielfältige Methoden der Optimierung [JARK84, JK84, BMSU86, BR86, CGM90, BRY90] auf der Sprachebene der logischen Formeln. Auf der physischen

Ebene sind hauptsächlich Verfahren zur Umformung von Ausdrücken der relationalen Algebra [FREY87,GD87] und der Einsatz von Hilfsdatenstrukturen (siehe z.B. [BM90]) entwickelt worden.

Auf der anderen Seite müssen gewichtige Nachteile gesehen werden. Sie können unter den Punkten inadäquate Informationsdarstellung und mangelnde Unterstützung für die Entwicklung von Anwendungen eingeordnet werden. Zum ersten Punkt kritisiert [JACK90], daß es im relationalen Modell keinen Begriff der Identität eines darzustellenden Gegenstandes gibt. Aus diesem Grund wird für den konzeptionellen Entwurf häufig ein anderes Datenmodell, z.B. das Entity-Relationship-Modell [CHEN76], benutzt. Das Schlüsselkonzept für Relationen ist kein vollwertiger Ersatz, da die Werte der Schlüsselattribute änderbar sind. Ein weiterer Kritikpunkt ist das Verbot zusammengesetzter Attribute, etwa der Name einer Person, die aus Vor- und Nachname besteht. In der Literatur ist dies als erste Normalform bekannt. Sie hat unter anderem den Vorteil, daß die Logik frei von Funktionen ist. In vielen Anwendungen sind allerdings komplexe Werte als Attribute wünschenswert, z.B. in der Dokumentenverwaltung und in Entwurfsumgebungen.

Der nächste Nachteil ist die unnatürliche Aufteilung der Information auf die Relationen. Diese Aufteilung entsteht durch Umformungen des Relationenschemas mit dem Ziel der Redundanzelimination. Sie kann durch entsprechende Anfragen zwar wieder reproduziert werden, jedoch ist der ursprüngliche Zusammenhang der Attribute verloren. Als letzter Kritikpunkt soll der große Abstand der relationalen Datenbanken von den gängigen Programmiersprachen genannt werden. Letztere verarbeiten komplexe Datenstrukturen, z.B. Felder, Bäume und Listen. Sie werden von imperativen Programmiersprachen sequentiell, d.h. ein Wert nach dem anderen, abgearbeitet. Relationale Datenbanken bieten jedoch einzig Mengen von Tupeln als Schnittstelle zum Anwendungsprogramm an. Während im Datenbanksystem die Antwort auf eine Anfrage als Ganzes berechnet wird, werden diese Antworten im Anwendungsprogramm „Tupel nach Tupel“ verarbeitet. Durch umständliche Umformatierung muß aus der Antwort des Datenbanksystems die vom Anwendungsprogramm weiter benutzte Datenstruktur erstellt werden.

Die Schwächen des relationalen Datenmodells waren Anlaß für die Entwicklung einer Reihe von Nachfolgern dieser Art von Datenbanken. Das Konzept der Identität von Entitäten wurde von [CODD79] vorgeschlagen. Die NF^2-Datenbanken [SS83] erlauben anstatt atomarer Werte auch komplexe Attributwerte in Tupeln. Diese beiden Ansätze lehnen sich noch recht nah an das relationale Modell an und erben viele der positiven Eigenschaften. Eine andere Gruppe von Nachfolgern, die *objektorientierten* Datenbanken, können nicht mehr als Erweiterungen angesehen werden. Sie stellen die adäquate Darstellung von Information und die Nähe zu Programmiersprachen in den Vordergrund. Eine offene Frage ist, inwieweit relationale (deduktive) und objektorientierte Datenbanken wieder

zusammengeführt werden können (siehe [KNN90,DKM91]). Die Algorithmen für logische Formeln in relationalen Datenbanken entlasten den Anwendungsprogrammierer von der Implementierung einfacher Konsistenztests und Ableitungsregeln, da die Formelmengen so eingeschränkt wurden, daß die automatisch auf ausführbare Programme abgebildet werden können. Für den verbleibenden Rest der Anwendungsentwicklung gibt das relationale Datenmodell allerdings wegen seiner Ferne zu den Datenstrukturen gängiger Programmiersprachen wenig Unterstützung. Einen Ausweg bieten die Datenbankprogrammiersprachen [SCHM77,ATKI91], die eine herkömmliche Programmiersprache um die datenbankspezifischen Konzepte Persistenz, Massendaten und Mehrbenutzerbetrieb ergänzen. Sie bieten insofern also eine einheitliche Entwicklungsumgebung an. Dieser Vorteil wird aber damit erkauft, daß Dinge, die im Prinzip mit Integritätsbedingungen oder deduktiven Regeln ausgedrückt werden können, per Hand zu implementieren sind.

Der Rest der Arbeit soll einen Weg zeigen, beide Ziele zu verbinden. Kapitel 3 gibt einen Überblick über das Gebiet der objektorientierten Datenbanken, die sich durch Nähe zum Anwendungsprogramm und durch ein reicheres Datenmodell auszeichnen. Kapitel 4 baut auf einer Wissensrepräsentationsprache ein axiomatisches Objektmodell auf. Dieses Modell ist gleichzeitig ein sehr spezielles relationales Datenmodell und ein objektorientiertes Datenmodell, indem es Klassifikation, Attributierung und Spezialisierung von Objekten enthält. Es wird sich herausstellen, daß die Verfahren aus diesem Kapitel nicht nur effizienter werden, sondern daß sich im Zusammenspiel von deduktiven mit objektorientierten Eigenschaften neue Anwendungen der Verfahren wie Meta-Modellierung und die deduktive Beschreibung komplexer Objekte ergeben.

Der Begriff der objektorientierten Datenbank wird abgegrenzt. Als Hauptrichtungen werden der programmiersprachliche und der darstellende Ansatz erkannt. Ziel ist die Auswahl eines Datenmodells, das für die Übertragung der in Kapitel 2 besprochenen Verfahren zur Behandlung logischer Formeln geeignet erscheint.

Kapitel 3: Objektmodelle und objektorientierte Datenbanken

Das relationale Datenmodell verwirklicht weitgehend das Prinzip der Unabhängigkeit der Daten von den Anwendungsprogrammen. Einerseits wird dadurch eine universelle Einsetzbarkeit der Datenbank erreicht, andererseits ist die Lücke zu den Anwendungen sehr groß. Insbesondere sind die Datentypen der Anwendungsprogrammiersprachen viel reichhaltiger als sie das relationale Datenmodell anbietet. Der Zwang zur Transformation auf das „schwächere“ Datenmodell wird als Hindernis (*impedance mismatch* [MS87,BBB*88]) empfunden. Des weiteren wird die strikte Trennung der Daten von den Programmen kritisiert. Tatsächlich haben sich ja in Programmiersprachen Datentypen und Operationen aufeinander zubewegt, was schließlich im Konzept der abstrakten Datentypen seinen stärksten Ausdruck fand [GUTT75,GTW78,EM85].

Historisch hat diese Situationsbeschreibung dazu geführt, daß ab Ende der 70er Jahre eine ganze Reihe von Nachfolgesystemen entworfen und implementiert wurden. Unter ihnen haben die sogenannten objektorientierten Datenbanksysteme die größte Beachtung erlangt. Für diese Datenbanken gibt es im Gegensatz zu ihren relationalen Vorgängern kein einheitliches Objektmodell. Es ist daher nur eine ungefähre Eingrenzung durch Angabe vager Eigenschaften möglich. Nach dem „Object-Oriented Database System manifesto“ [ABD*90] soll ein Datenbanksystem folgende 13 Kriterien erfüllen, um sich objektorientiert nennen zu dürfen:

1. *Objektkomplexität:* Aus bestehenden Objekten können mittels Konstruktoren komplexere Objekte erstellt werden. Beispiele für solchermaßen konstruierte Objekte sind Mengen, Listen und Tupel. Die Konstruktoren müssen beliebig kombinierbar sein.

2. *Objektidentität:* Ein Objekt hat eine eindeutige Bezeichnung, die von seinem inneren Zustand unabhängig ist. Objektidentität muß vom Schlüsselbegriff in relationalen Datenbanken unterschieden werden. Dort können Schlüsselattribute eines Tupels verändert werden, der Objektidentifikator (OID) ist nicht veränderbar.

3. *Einkapselung:* Die Manipulation eines Objektes, also Erfragen und Ändern seiner Eigenschaften, erfolgt nur über bestimmte Operationen. Diese Operationen werden bei dem Typ bzw. der Klasse des Objektes definiert. Die Implementierung der Operationen des Objektes ist nach außen hin unsichtbar. Sie kann problemlos geändert werden, solange die Schnittstelle unberührt bleibt. Ob der innere Zustand des Objektes nach außen sichtbar sein soll, ist umstritten und hängt davon ab, ob eine assoziative Anfragesprache neben den Operationen auf Objekte zugreifen darf.
4. *Klassifizierung oder Typisierung:* Typen geben die Struktur eines Objektes (genauer: des Wertes eines Objektes) und die darauf anwendbaren Operationen an. Typinformation wird nur zur Schemadefinitionszeit benötigt. Sie wird wie bei Programmiersprachen zur Typprüfung eingesetzt. Im Gegensatz dazu werden Klassen als endliche Mengen von Objekten (auch Instanzen der Klasse genannt) angesehen. Wie bei Typen können für Klassen Operationen definiert werden, die jedoch auf Objekten (besser: Objektidentifikatoren) statt auf Werten arbeiten.
5. *Hierachien über Typen oder Klassen:* Eng verbunden mit Klassen und Typen ist die Einführung einer Hierarchie auf ihnen. Die Hierachie ist ein gerichteter azyklischer Graph, bei dem die Klassen bzw. Typen als Knoten dargestellt werden und die Verbindungen als Spezialisierungsbeziehungen interpretiert werden. Entlang der Hierarchie werden die Eigenschaften, insbesondere die Methoden, der Oberklassen an ihre Unterklassen vererbt.
6. *Dynamisches Binden von Methoden:* Einer Unterklasse ist es erlaubt, die Methoden der Oberklasse zu „überschreiben“, d.h. unter Verwendung der gleichen Signatur neu zu implementieren. Dies bedingt, daß erst bei der Ausführung entschieden werden kann, ob die Methode der Klasse oder einer ihrer Oberklassen anzuwenden ist.
7. *Turing-Vollständigkeit:* In der Datenmanipulationssprache sollte jede berechenbare Funktion ausdrückbar sein. Diese Eigenschaft kann ggf. auch durch die Kopplung mit einer bestehenden Programmiersprache erreicht werden.
8. *Erweiterbarkeit:* Das Typ- bzw. Klassensystem muß für Anwendungsprogrammierer erweiterbar sein. Vordefinierte und neue Typen (Klassen) müssen hinsichtlich ihrer Nutzbarkeit für den Programmierer ununterscheidbar sein.
9. *Persistenz:* Die Daten sollen länger existieren können, als ein Programm, das sie benutzt. Diese Eigenschaft müssen alle Objekte unabhängig von ihrem Typ haben. Persistenz ist eine implizite Eigenschaft. Sie wird nicht durch explizite Speicherungsbefehle im Anwendungsprogramm erreicht.
10. *Sekundärspeicherverwaltung:* Unter diesem Punkt versteht man die klassischen Mechanismen der physischen Ebene, die einen effizienten Umgang mit den Objekten ermöglichen: Indizes, Packen von Daten, Pufferverwaltung usw.

11. *Mehrbenutzerbetrieb:* Mehrere Benutzer oder Anwendungsprogramme dürfen gleichzeitig auf eine Datenbank zugreifen, ohne sich gegenseitig zu stören.
12. *Wiederherstellung:* Nach Programm- oder Gerätefehlern muß das System wieder zu einen konsistenten Zustand zurückfinden.
13. *Ad-Hoc-Anfragen:* Das Datenbanksystem soll eine deklarative Anfragemöglichkeit bereitstellen, die unabhängig von bestimmten Anwendungen und für jede Datenbank einsetzbar ist.

Die Punkte 1 bis 8 umschreiben das Adjektiv *objektorientiert*, während die letzten fünf Eigenschaften die klassische Datenbankfunktionalität einfordern. Komplexe Objekte tauchen im Zusammenhang mit Datenbanken zuerst bei den sogenannten NF^2-Modellen [SS83b] auf. Im Unterschied zum relationalen Modell darf hier die 1. Normalform [SS83a] durch relationenwertige Attribute verletzt sein. Der Anstoß für ein solches Datenmodell kam aus dem Bereich der Dokumentenverwaltung. Es bleibt anzumerken, daß der letzte Punkt über die Anfragesprache in einem gewissen Konflikt zu der Einkapselung steht: mittels Anfragen soll möglichst der gesamte Datenbestand erfaßt werden, während die Idee der Einkapselung gerade das Verstecken von Information (sogenannte Implementierungsdetails) propagiert. Nicht einigen konnten sich die Autoren des Manifestes über die Bedeutung von Integritätsbedingungen und Regeln in Objektbanken. So sagt etwa Bancilhon [BB*90], es sei zu früh, deduktive und objektorientierte Paradigmen zu mischen. Die Technologie der deduktiven Datenbanken sei – im Gegensatz zu Objektbanken – noch nicht reif. Außerdem mangele es an Anwendungen für diese Technik. Entgegengesetzte Einschätzungen werden in [MANT90,SSU91,ULLM91] geäußert: eine deklarative Regelsprache sei ein wesentlicher Bestandteil zukünftiger Datenbanksysteme. Offen sei allerdings noch, welches der beste Weg ist, diese Sprache zur Turing-Vollständigkeit zu erweitern.

Das Manifest übernimmt im wesentlichen das Objektkonzept von den objektorientierten Programmiersprachen. Diese Art von Datenbanken soll als *programmiersprachlicher* Ansatz bezeichnet werden und ist Thema des nachfolgenden Unterkapitels. Im Anschluß daran wird eine zweite Linie, die *darstellenden* Objektbanken beschrieben. Sie stellt die Aspekte der Weltmodellierung und Klassifikation von Information in den Mittelpunkt. Das letzte Unterkapitel ist den Formalisierungen von Objektmodellen gewidmet. Interessanterweise dominieren hier Modelle mit deduktiven Regeln.

3.1. Der programmiersprachliche Aspekt

Besonders die frühen Objektbanken sind durch die enge Anlehnung an eine objektorientierte Programmiersprache gekennzeichnet. So übernimmt GemStone [MS87] das

Klassensystem von SmallTalk [GR83], Orion [KBC*88] setzt auf eine objektorientierte Variante von Lisp auf, und Exodus [GD87] erbt das Typsystem von C. Die Entscheidung, von einer Programmiersprache auszugehen, ist eine Antwort auf den schon erwähnten *impedance mismatch*. Wenn die Objektbank die gleichen bzw. ähnliche Datenstrukturen wie das Anwendungsprogramm benutzt, so fällt die umständliche Ankopplung weg bzw. sie wird stark erleichert.

Das Schema der Objektbank ist durch eine Menge von *Klassen* gegeben. Diese Klassen sind mit den Datentypen in Programmiersprachen vergleichbar. Aus der Theorie der Programmiersprachen wird auch das Konzept der abstrakten Datentypen übernommen: einer Klasse kann eine Anzahl von Operationen zugeordnet werden, die hier *Methoden* genannt werden. Methodenaufrufe heißen *Nachrichten*. Das Format einer Nachricht entspricht der Signatur der Operation eines abstrakten Datentyps und ist durch Emfänger-, Argument- und Ergebnisklassen gegeben.

Die Ebene der *Objekte* wird vom Schema streng getrennt. Objekte werden als *Instanz* einer Klasse definiert, was der Zuordnung einer Programmvariablen zu einem Datentyp entspricht. Ein Objekt wird durch seinen Identifikator (OID) referenziert, ganz ähnlich einer Adresse, die auf einen Speicherplatz zeigt. Der Inhalt oder Wert des Speicherplatzes gibt die momentanen Eigenschaften des Objektes an und wird nach außen verborgen (*encapsulation*). Die Methoden einer Klasse werden in einer vollständigen Programmiersprache implementiert. Da die Datenbank nur über die im Schema festgelegten Methoden manipulierbar ist, kann auf diese Weise der erlaubte Umgang mit der Datenbank kontrolliert werden.

Am Beispiel von GemStone sollen die Auswirkungen dieses Ansatzes auf die Funktionalität der Objektbank beschrieben werden. Die Ausgangssprache SmallTalk ist hinsichtlich ihrer Programmierkonstrukte als *imperativ* anzusehen. Folglich wird ein Objekt als ein „Stück privaten Speicherplatzes mit einer öffentlichen Schnittstelle“ [MS87,S.360] angesehen. Der Effekt von Methodenaufrufen ist letztendlich eine Veränderung des Zustandes dieses Speicherplatzes. Bild 3-1 zeigt graphisch ein Objekt einer Klasse *Employee*. Diese Klasse hat drei Attribute, die wiederum auf andere Objekte zeigen. Das Attribut *name* ist wiederum aus zwei Objekten, in diesem Fall Zeichenketten, zusammengesetzt. Die beiden anderen Attribute weisen auf Zahlobjekte. Die Attributwerte können beliebige Objekte sein. Die Klasse *Employee* fordert lediglich das Vorhandensein dreier Attribute in Form von Zeigern zu Objekten. Ein zweites Objekt derselben Klasse dürfte also etwa als Attribut *ssNo* ein zusammengesetztes Objekt haben. Der Zugriff auf Komponenten wird mit der bekannten Punktnotation, beispielsweise $e.name.first$ für den Vornamen, ausgedrückt.

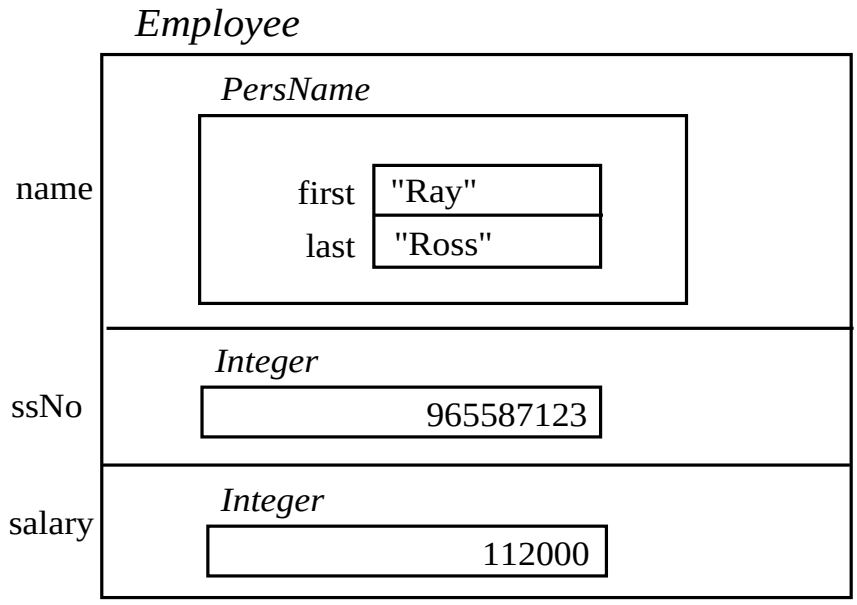

Abb. 3-1: Beispiel eines Objektes in GemStone nach [MS87]

Als Datenbank zeichnet sich Gemstone dadurch aus, daß sowohl der Adreßraum der OID's als auch die erlaubte Größe eines einzelnen Objektes weit ausgedehnt wurden. Objekte sind jetzt grundsätzlich persistent. Zur Manipulation werden sie – für den Anwender unsichtbar – in einen Puffer kopiert. GemStone erlaubt den Mehrbenutzerbetrieb mit *optimistischer Nebenläufigkeitskontrolle*: erst nach Transaktionsende (*commit*) wird auf mögliche Konflikte getestet. Von jeher ist die Effizienz eines Datenbanksystems von zentraler Bedeutung. GemStone wurde auch deshalb entwickelt, weil die traditionellen Systeme für Anwendungen wie CAD nicht mehr ausreichten [MAIE86][2]. Die meisten Verfahren hierzu fallen unter die Überschrift Anfrageoptimierung. GemStone stellt Indizes [MS86] für *Pfadausdrücke* der Form $x.l_1.l_2 \cdots .l_n = y$ bereit. Ein neuerer Ansatz dazu findet sich in [KM90a], der sowohl die Vorwärtsrichtung (x bekannt, y gesucht) als auch die Rückwärtsrichtung (y bekannt, x gesucht) unterstützt. SmallTalk legt keine Restriktionen auf die Komponenten eines Objektes, sondern nur auf die Argumente der Methoden. GemStone erbt diese Eigenschaft, was auch von den Autoren [MS87] als gewisser Nachteil angesehen wird. Hinzu kommt ganz allgemein das Unentscheidbarkeitsproblem. Da SmallTalk alle berechenbaren Funktionen ausdrücken kann, ist es z.B. prinzipiell unmöglich, die Terminierung eines Methodenaufrufs unter allen Umständen zu garantieren. Dieses Problem war für relationale Datenbanken nicht vorhanden, da die Anfragesprachen

[2] In [DD88] wird gezeigt, daß programmiersprachliche Objektbanken durchaus diesem Anspruch gerecht werden können. Besonders bei der Verfolgung von Referenzen sind deutliche Vorteile im Vergleich zu relationalen Datenbanken gegeben.

[ULLM89] bewußt auf Entscheidbarkeit hin eingeschränkt wurden. Der nicht entscheidbare Anteil wurde sozusagen in die Zuständigkeit der Anwendungsprogramme verschoben.

Als zweites Beispiel dient das System O_2 [BBB*88]. Wie GemStone wird als Ziel die Überwindung der Sprachlücke zu den Anwendungsprogrammen genannt. Allerdings ist das Datenmodell von O_2 [LRV88] unabhängig von einer speziellen Programmiersprache definiert. Ausgehend von einer Menge von Basistypen (Integer, String, Nil, Menge der OID's usw.) werden mit Tupel-, Listen- und Mengenkonstruktoren *strukturierte Typen* definiert. Ihre Elemente heißen auch Werte. Ein *Objekt* ist dann ein Paar (i, v) aus einem OID und einem Wert. Auf den Typen wird eine partielle Ordnung „$\leq$“ etabliert, die als Mengeninklusion interpretiert wird. Für jeden Typ dürfen *Methoden* mit fester Signatur und Funktionssemantik definiert werden. Entlang der „$\leq$“-Hierarchie sind Methoden mit gleicher Signatur und unterschiedlicher Interpretation erlaubt (*overriding*). Welche Interpretation für einen Methodenaufruf gültig ist, wird erst zur Laufzeit in Abhängigkeit der Argumente entschieden. Das System O_2 hält an der klassischen Trennung von Schemainformation (Typ- und Methodenbeschreibungen) und der Menge der Objekte fest. So haben Typen und Methoden keinen OID, und somit sind auf sie selbst keine Methoden anwendbar. Eine ausführliche Diskussion dieses Typverständnisses findet sich in [CW85]. Dort werden Typen als Mengen von Werten, auf die die gleichen Operationen anwendbar sind, angesehen. Als Folgerung müssen die Werte auch die gleiche Struktur haben. Objektorientierte Typsysteme haben als zusätzliche Eigenschaft das *Overriding* von Operationen, eine spezielle Form der *Polymorphie:* Werte können mehr als einem Typ, hier allen Obertypen eines bestimmten Typs, angehören.

Anfragesprachen sind an sich redundant, wenn die Informationssuche auch über Methoden realisiert werden kann. Jedoch ist diese Art der Interaktion nicht nur für Laien sehr ineffektiv. Daher wird das Vorhandensein einer *deklarativen* Anfragekomponente auch für Objektbanken als zwingend angesehen. Will man die Deklarativität der Sprache mit effizienter Auswertung (nach Optimierungsschritten) kombinieren, so ist eine Einschränkung auf (Turing-)unvollständige Teilmengen unumgänglich [BL86,AG91] und wünschenswert. Auch O_2 folgt dieser Argumentation und bietet eine SQL-ähnliche Anfragesprache an [BCD89]. Als Anpassung an das Objektmodell sind zwei Punkte herauszustellen. Zum einen stehen Variablen jetzt nicht mehr nur für Tupel, sondern für beliebig strukturierte Werte und Objekte gemäß obiger Definition. Dazu wurden die Konstruktoren für Listen, Tupel und Mengen in die Anfragesprache mit aufgenommen. Außerdem wird ein Operator zur Umwandlung von Mengen von Listen in Mengen von Listenelementen erklärt, um Listenelemente abfragen zu können. Als Ergebnis einer Anfrage sind beliebig strukturierte Werte möglich. Zum zweiten muß der Konflikt mit der Einkapselung des Objektwertes gelöst werden. Dieses Prinzip gestattet nur den Zugriff über Methoden. Somit kann

Information völlig unzugreifbar für die Anfragesprache sein. O_2 legt hierzu fest, daß in der interaktiven Version der Wert eines Objektes sichtbar ist. Hier wird also die Einkapselung aufgegeben. Allerdings darf bei der Verwendung der Anfragesprache innerhalb eines Anwendungsprogrammes wieder nur über die Methoden zugegriffen werden. Dies schützt die Programme vor Änderungen des internen Aufbaus eines Objektes – der Zweck der Einkapselung.

Zur Anfrageoptimierung wendet O_2 eine Reihe von Verfahren an [CD91]. So werden Indizes auf die Attribute der Objekte angelegt, was durchaus mit Indizierungsverfahren in relationalen Datenbanken vergleichbar ist. Neu hinzu kommen Strategien zum *Clustern* der Objektwerte. Die potentiell unbeschränkte Größe von O_2-Objekten macht Überlegungen erforderlich, welche Teile zusammen auf dem Hintergrundspeicher abgelegt werden sollen, um die Anzahl der Zugriffe auf diesen relativ langsamen Speicher zu verringern. Schließlich werden noch einige Verfahren basierend auf der algebraischen Darstellung einer Anfrage angewandt: Anfrageterme werden unter Ausnutzung von Äquivalenzregeln in effizienter auswertbare Terme umgeformt. Speziell wird auf mögliche Faktorisierung von Teilausdrücken untersucht. Es sei noch darauf hingewiesen, daß der Optimierer auch die Bindung der Variablen einer Anfrage an einen Typ ausnutzt.

Neben O_2 und GemStone gibt es einige weitere programmiersprachliche Objektbanken, die auch schon zum Teil vermarktet werden. ORION [KBC*88,KBG*91] basiert auf einer objektorientierten Variante der funktionalen Programmiersprache LISP. Es unterstützt sowohl die Programmierung mittels Methoden als auch den Zugriff mit einer deklarativen Anfragesprache. Spezielles Augenmerk wird auf die Verteilung der Objektbank in einem Rechnernetz und auf die Versionierung von Objekten gelegt. Der letztgenannte Aspekt ist vor allem in Entwurfsumgebungen von Bedeutung und wurde in der Objektbank DAMOKLES [RRR*88] in ein Transaktionsmodell integriert.

Die Objektbank IRIS [LWH90] drückt sowohl Attribute als auch Methoden einheitlich als Funktionen aus (funktionales Datenmodell [SHIP81]). Funktionen haben Objekteigenschaft und können in ihrem Implementierungsteil beliebige Seiteneffekte haben. Beliebige Programmiersprachen können über eine Schnittstelle als sogenannte externe Funktionen eingebaut werden. Die Anfragesprache orientiert sich ebenfalls an SQL, wobei verschachtelte Funktionsaufrufe vorkommen können. Neue Funktionen können mit Ausdrücken über bereits definierten Funktionen erklärt werden. Besonders anzumerken ist, daß IRIS die Änderung des Schemas (also die Typen und die Funktionen darauf) zur Laufzeit erlaubt.

Das Datenbanksystem POSTGRES [SRH90,SJGP90,SK91] wurde als Weiterentwicklung relationaler Datenbanksysteme [ADV90] entworfen. Es bietet daher alle Eigenschaften dieser Systeme einschließlich Relationen und eine Obermenge einer relationalen

Anfragesprache. Zusätzliche Ausdrucksmittel sind Vererbung, Versionierung, abstrakte Datentypen und Regeln. Durch abstrakte Datentypen kann das Typsystem, das die zulässigen Attributwerte beschreibt, erweitert werden. Die Regeln in POSTGRES sind von der Form „Wenn Ereignis und Bedingung, dann Aktion". Ereignissse sind Änderungen an den Objekten der Datenbank, die Bedingung ist eine Anfrage, und die Aktion ist eine neue Änderung auf der Datenbank bzw. eine weitere Anfrage.

```
ON NEW EMP.salary WHERE EMP.name="Fred"
THEN DO REPLACE E(salary=new.salary) FROM E in EMP WHERE E.name="Joe
```

Das obige Beispiel [SK91] zeigt eine POSTGRES-Regel, die jede Änderung des Gehaltes von Fred auf das Gehalt von Joe weiterreicht. Für Regeln dürfen auch Ausnahmen (Schlüsselwort `instead`) formuliert werden. Die Auswertung der Regeln kann auch verzögert und in einer späteren Transaktion erfolgen (Regelauswertungspolitik). Zwei Regelauswerter stehen zur Verfügung, die jeweils eine andere Semantik haben. Dieses Semantikproblem ist typisch für Produktionsregelsysteme, die in ihrem Aktionsteil einen Zustandsraum ändern. In einem solchen System spielt die Reihenfolge der Ausführung der Regeln eine Rolle (siehe z.B. [NILS82]).

Im Projekt HiPAC [DBB*88,DBM88] wurde die Idee der *aktiven Datenbank* ausführlich untersucht. In Form sogenannter ECA-Regeln (für *event-condition-action*) werden Prozeduren als Teil der Objektbank abgelegt. Wenn ein bestimmtes Ereignis eintritt, also z.B. die Löschung eines Objektes oder das Erreichen eines bestimmten Zeitpunktes, so wird die dafür programmierte ECA-Regel „gefeuert". Die Auswertung besteht aus einem Test auf die Objektbank und dem Ausführen einer Operation. Ähnlich wie bei Mehrbenutzerbetriebssystemen kann den Regeln ein Gewicht zugeordnet werden, das die Priorität bei der zeitlich verzahnten Auswertung der ECA-Regeln steuert. Dadurch soll der Einsatz in Echtzeitumgebungen, in denen innerhalb kurzer Zeitintervalle auf Ereignisse reagiert werden muß, ermöglicht werden. Das System der ECA-Regeln bildet eine Programmierumgebung. Der Operationsteil verändert den Zustandsraum. Die ECA-Regeln sind eng mit den Triggern in relationalen Datenbanksystemen (Kap. 2) verwandt. Im Kontext relationaler Datenbanken werden sie mangels einer Turing-vollständigen Programmiersprache hauptsächlich zur Integritätskontrolle [CW90] und zur Sichtenwartung [CW91] eingesetzt.

3.2. Der darstellende Aspekt

Eine Datenbank enthält Wissen über die Welt. Somit sind Bezüge sowohl zu den traditionellen relationalen Datenbanken als auch zu Wissensrepräsentationssprachen gegeben.

Ein früher Vorschlag [ABRI74] basiert auf einem Datenmodell von binären Relationen[3]. Die Objekte der Datenbank werden als Knoten angesehen, die über gerichtete Kanten in Beziehung stehen (siehe Abb. 3-2). Eine Abstraktionsstufe höher bedeuten die Knoten die Kategorien (Klassen) und die Kanten die zwischen ihnen *möglichen* Beziehungen. Die oberste Ebene repräsentiert das Datenmodell selbst.

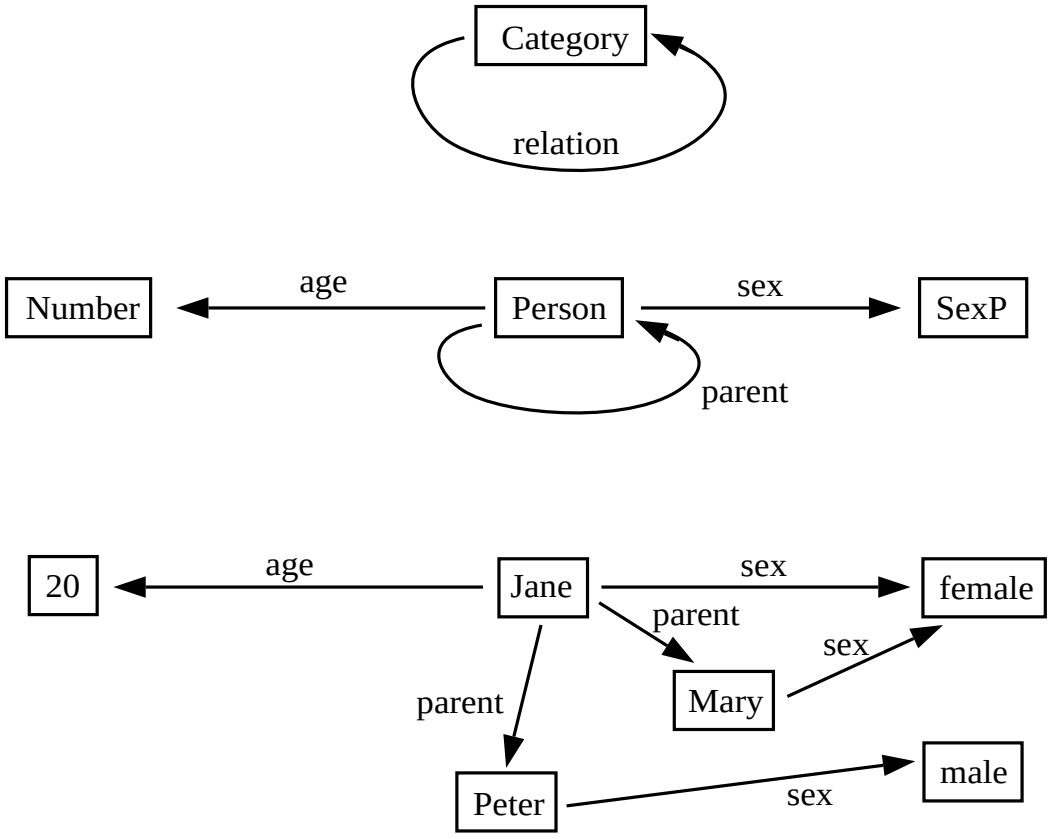

Abb. 3-2: Beispiel einer Datenbank mit Abstraktionsstufen nach [ABRI74]

Den binären Relationen werden als Integritätsbedingungen Schranken für die Kardinalitäten zugeordnet. Die Knoten haben eindeutige Identifikatoren. Die Ausdrucksfähigkeit wird durch deduktive Regeln und eine imperative Programmiersprache angereichert. Im

[3] Binäre Relationen wurden schon früh von Tarski [TARS41] untersucht. Dort ist auch bereits eine Algebra zu finden, die als Vorläufer der Relationenalgebra für Datenbanken (siehe [GV89]) anzusehen ist. Während sich Tarski aus Komplexitätsgründen auf binäre Relationen beschränkt, ist das relationale Modell für mehrstellige Relationen erklärt. Interessanterweise kehren einige Nachfolger des relationalen Datenmodells wieder zu binären Relationen zurück.

Sinne des Manifestes fehlen eine Reihe von Eigenschaften, insbesondere die Klassenhierachie. Es ist jedoch ein wichtiger Vorläufer für spätere darstellende Ansätze, da hier sowohl Objekt- als auch Klasseninformation in der gleichen Weise dargestellt werden. Beide Ebenen stellen (revidierbare) Aussagen über den modellierten Weltausschnitt dar. In [ABRI74] finden sich auch schon Ansätze, das Verhalten der Datenbank mit dem Datenmodell zu beschreiben.

Das *Entity-Relationship*-Modell [CHEN76] ist ein Datenmodell, das gewöhnlich zum Entwurf des Schemas einer relationalen Datenbank herangezogen wird. Es unterscheidet zwischen Entity-Typen, Beziehungstypen und Attributen, die zu Wertebereichen führen. Zu beachten ist, daß Entitäten ein Identifikator (*primary key*) zugestanden wird, womit das Teilen gemeinsamer Unterobjekte (*sharing*) ermöglicht wird. Wie bei [ABRI74] können Integritätsbedingungen in Form von Kardinalitätsschranken für die Relationen angegeben werden. Das Entity-Relationship-Modell ist bis auf die Objektidentität recht weit vom Objektbankbegriff entfernt. Es spielt eine Rolle als Vorgänger für die später entwickelten semantischen Datenmodelle und Datenbankentwurfssprachen, die nun ausführlicher vorgestellt werden.

Semantische Datenmodelle [HK87b] kombinieren Konzepte aus Wissensrepräsentationssprachen und objektorientierten Programmiersprachen. Ihren mannigfaltigen Ausprägungen ist gemeinsam, daß sie den Weltausschnitt durch Objekttypen (Knoten) und Beziehungen zwischen ihnen (Kanten) modellieren. In dem generischen semantischen Datenmodell GSM [HK87b] wird genauer zwischen druckbaren, benutzerdefinierten, konstruierten und abgeleiteteten Typen unterschieden. Zwischen den Typen können durch Attribute ein- oder mengenwertige Beziehungen aufgestellt werden. Die erlaubten Typkonstruktoren sind der Tupelkonstruktor und der Mengenkonstruktur.

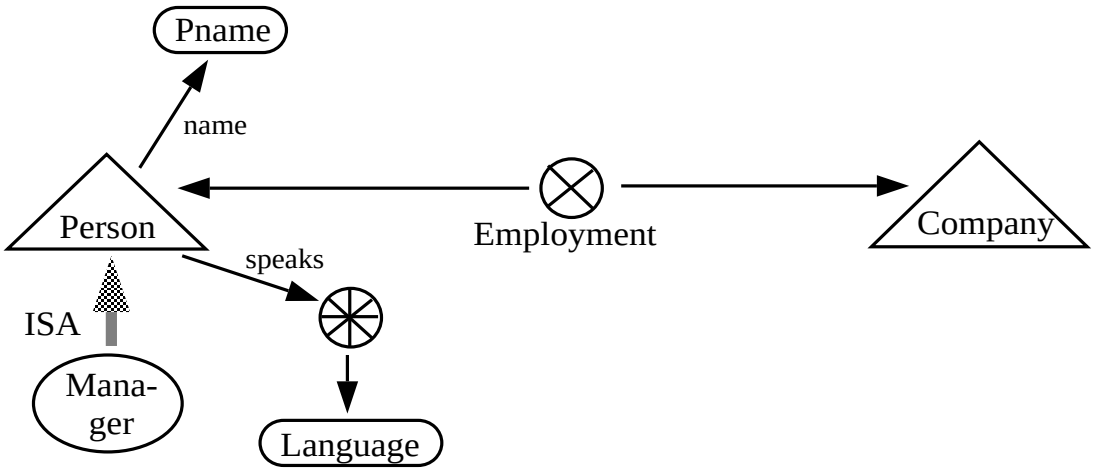

Abb. 3-3: Ein Objektbankschema in GSM nach [HK87b]

Abbildung 3-3 zeigt die verschieden Elemente in GSM. Der benutzerdefinierte Typ *Person* hat zwei Attribute von denen das eine auf den Typ der Personennamen (also druckbar) und das andere auf den Typ der Potenzmenge von Sprachen zeigt. Aus *Company* und *Person* wird mittels des Tupelkonstruktors der Typ *Employment* definiert. Der Untertyp *Manager* wird aus *Person* abgeleitet.

Eine formale Interpretation wird in [AH87] für das semantische Datenmodell IFO angegeben, das mit dem GSM fast identisch ist. Darin werden für die nichtkonstruierten Typen abzählbar unendliche Mengen zugrunde gelegt. Tupelkonstruierte Typen werden als kartesisches Produkt und mengenkonstruierte Typen als Menge aller möglichen Teilmengen der Grundmenge interpretiert. Die Attribute zwischen den Typen haben partielle Funktionen zwischen den Interpretationen der beteiligten Typen als Semantik. Die Spezialisierungsbeziehung (ISA) wird als Mengeninklusion interpretiert. An die Spezialisierungsbeziehungen werden in der Definition der Semantik gewisse Axiome (wie z.B. Nichtzirkularität) gebunden. Integritätsbedingungen, deduktive Regeln und Methoden werden im IFO-Modell nicht behandelt.

Ein weiterer Ansatz, der aus dem Bereich der *Artificial Intelligence* stammt, sind die sogenannten **Konzeptsprachen** [BBH*90,SIGA91]. Diese Sprachen sind sämtlich Varianten der Wissensrepräsentationssprache KL-ONE [BS85,vL90]. Gemäß einer modelltheoretischen Semantik, in der Konzepte als Teilmengen einer Grundmenge T interpretiert werden, sollen dann folgende Arten von Fragen beantwortet werden:

1. Subsumiert das Konzept C_1 das Konzept C_2, d.h. ist jedes mögliche Element von C_2 auch Element von C_1?
2. Sind die Konzepte C_1 und C_2 definitorisch disjunkt, d.h. kann es kein gemeinsames Element geben?
3. Kann das Konzept C überhaupt ein Element haben?

Es ist typisch, daß die Fragen von konkreten Interpretationen der Konzepte, die Datenbankinstanzen entsprechen, abstrahieren. Gesucht werden Aussagen, die für alle möglichen Interpretationen gelten. Neben den Konzepten gibt es noch die *Rollen*, die Beziehungen zwischen Konzepten ausdrücken. Sie werden als binäre Relationen aus $T \times T$ interpretiert. An die Kardinalitäten dieser Rollen können Restriktionen gebunden werden. Die genaue Semantik der Konzeptbeschreibungen kann [BBH*90,vL90] entnommen werden.

Die Abbildung 3-4 erklärt das Konzept *Human* als Teilmenge ($\sqsubseteq$) des Schnittes ($\sqcap$) von *Creature* und der Menge der Objekte, deren Rollen *hasChild* sämtlich zu *Human* zeigen. Das Konzept *Man* wird gleichgesetzt ($=$) mit dem Schnitt von *Human* und *MaleCreature*.

$$
\begin{aligned}
Human &\sqsubseteq Creature \sqcap (\forall\ hasChild.Human) \\
MaleCreature &\sqsubseteq Creature \\
FemaleCreature &\sqsubseteq Creature \\
Man &= Human \sqcap MaleCreature \\
Woman &= Human \sqcap FemaleCreature \\
Parent &= Human \sqcap (\exists\ hasChild) \\
NoFather &= Man \sqcap (\leq 0\ hasChild) \\
Nobody &= Parent \sqcap NoFather
\end{aligned}
$$

Abb. 3-4: Beispiel einer Konzepthierarchie in KL-ONE nach [vL90,BBH*90]

Alle Objekte aus *Human*, die in mindestens einem Tupel aus *hasChild* als erste Komponente vorkommen, bilden das Konzept *Parent*. Elemente aus *Man* die in keiner Rolle *hasChild* auftauchen, ergeben das Konzept *NoFather*. Eine interessante Fragestellung ist die Prüfung, ob das Konzept *Nobody* auch in allen Interpretationen durch die leere Menge dargestellt wird. In [BBH*90] wird ein modifizierter Resolutionsalgorithmus angegeben, der solche Fragen löst. Eine wichtige Anwendung des Algorithmus ist die automatische *Klassifikation* von Konzeptbeschreibungen, d.h. seiner schärfsten Unter- und Oberkonzepte gemäß der Teilmengeninterpretation von „$\sqsubseteq$“.

In der vollen Form von KL-ONE ist das Subsumtionsproblem unentscheidbar [SCHM89] und in manchen abgeschwächten Varianten immer noch coNP-schwierig [BL84,SS91a]. Aus diesem Grund wurde eine Reihe eingeschränkter KL-ONE-artiger Sprachen vorgeschlagen. Die Sprache CANDIDE [BGN89] sieht die Datenbeschreibungssprache (DDL) und die Anfrage-/Manipulationsprache (DML) als eine Einheit. Anfragen sind demnach Konzeptbeschreibungen, deren Lösungen zu bestimmen sind. Konkrete Objekte (Instanzen der Konzepte) können in KL-ONE als Konzepte mit einelementigen Mengen als Interpretation modelliert werden. CANDIDE führt hierzu das neue Konstrukt INSTANCE ein. Das System leistet dann zwei Hauptfunktionen mit seinem Klassifikationsalgorithmus: die Einordnung von Konzeptbeschreibungen und Anfragen in die Hierarchie, und die Bestimmung der Instanzenbeschreibungen, die unter ein Konzept klassifiziert werden können.

Eine weitere Variante ist die Sprache CLASSIC [BBMR89,PGB91]. Sie verbietet bestimmte Restriktionen auf Rollen und erreicht dadurch, daß der Subsumtionstest zwischen zwei Konzepten linear in der Größe der Konzeptbeschreibungen ist. Die Konzepte bilden wiederum das Schema der Datenbank. Die konkrete Datenbank besteht aus unären

Relationen, die für ein Objekt die Konzeptzugehörigkeit darstellen, und einer Ansammlung binärer Relationen für die Attribute, den *Rollenfüllern*. Die Konzeptzugehörigkeit kann vom Benutzer eingetragen werden, wobei das System die Korrektheit testet, oder wird automatisch abgeleitet. Änderungen an einem Objekt stoßen die automatische Reklassifikation an. Anfragen werden zunächst in die Konzepthierarchie eingeordnet. Dann werden die Instanzen der direkten Oberkonzepte dahingehend überprüft, ob sie die Anfrage erfüllen. Während die Semantik von Konzeptsprachen sonst meist denotationell angegeben wird [vL90], gibt es für CLASSIC auch eine beweistheoretische Semantik. Der Kalkül [BORG92] arbeitet auf den Ausdrücken, in denen Variablen für Konzepte (also Mengen von Objekten) stehen. Eine Eigenschaft $P(c)$, die für ein Konzept c bewiesen wird, charakterisiert alle möglichen Elemente dieser Menge. Die Aussage $P(c)$ kann also bei Anfragen nach Elementen von c zur Optimierung herangezogen werden. Der Mechanismus ist ähnlich zur semantischen Anfrageoptimierung [CGM90], allerdings werden die Eigenschaften $P(c)$ *abgeleitet* und nicht vom Datenbankentwerfer als Integritätsbedingungen *vorgegeben*.

Als letztes darstellendes Datenmodell sollen **Spezifikationssprachen** für Informationssysteme vorgestellt werden. Ihr Ziel ist es, ein *konzeptionelles Modell* der in der Datenbank enthaltenen Objekte möglichst nah an der Vorstellungswelt des Entwerfers darzustellen. [BORG85] stellt dazu einige Anforderungen auf:

1. Es gibt eine 1:1-Beziehung zwischen Objekten in der Welt und Objekten in dem Modell.
2. Objekte sind verschieden zu ihren externen Namen. Beispiel: ein Buch ist nicht dasselbe wie seine Nummer in der Bibliothek.
3. Objekte werden in Klassen gruppiert. Diese Klassen sind durch eine Unterklassenbeziehung geordnet.
4. Klassen sind auch Objekte, die ihre eigenen Attribute haben.
5. Attribute dürfen mehrwertig sein.
6. Information darf redundant angegeben werden. Es sollte auch möglich sein, gewisse Eigenschaften aus anderen abzuleiten.
7. Spezielle und allgemeine Integritätsbedingungen werden durch geeignete Klausen ausgedrückt.
8. Ereignisse und Transaktionen werden als Objekte modelliert.
9. Auch die Schnittstelle zwischen System und Benutzer muß modelliert werden.

Die Sprache Taxis [MBW80,NCL*87] kommt diesen Anforderungen nah, indem sie Klassen für Entities, Transaktionen, aufgezählte Klassen (z.B. die Monate eines Jahres) und Grundklassen (z.B. Integer) unterscheidet. Die Transaktionen werden durch Parameter, lokale Variablen, Vorbedingungen und Anweisungen beschrieben. Für längerfristige

Transaktionen werden Petrinetze angegeben. Taxis ist für den Entwurf von Informationssystemen entwickelt worden. Der Entwurf ist ein Prozeß, in dem die zu verhandelnden Daten als Objektklassen und die auszuführenden Aktivitäten als Transaktionen aufgeschrieben werden. Das Ergebnis ist eine Ansammlung von Klassen, die zu implementieren sind.

Ein Nachfolger von Taxis ist die Sprache Telos [MBJK90,MYLO91]. Ihre ersten Versionen RML [GREE84] und CML [STAN86] wurden als Spezifikationssprachen für Datenbanken entwickelt. Objekte in Telos werden alleinig in Form sogenannter Propositionen der Form $P(id, x, l, y, t)$ mit folgender beabsichtigter Bedeutung dargestellt:

> Die Objekte mit den Identifikatoren x und y stehen in der Beziehung l. Diese Aussage ist gültig für das Zeitintervall t. Die erste Komponente ist der Identifikator der Proposition.

Die Beziehung eines Objektes x zu seiner Klasse c wird durch ein Objekt $P(id, x, in, c, t)$ dargestellt. Eine Besonderheit von Telos ist die potentiell unendlich hohe Klassifizierungshierarchie, die durch die uniforme Darstellung von Instanzen und Klassen ermöglicht wird: für das Objekt c kann es eine Klasse mc geben und so weiter. Auch Attributbeziehungen werden Objektidentifikatoren zugestanden. Sie können also selbst Attribute haben oder als Attributwerte anderer Objekte auftauchen. Das Datenmodell wird durch eine mehrsortige Variante der Prädikatenlogik erster Stufe [KMSB89] angereichert. Damit sind deduktive Regeln und Integritätsbedingungen auf Klassenebene ausdrückbar. Während die semantischen Datenmodelle und die Konzeptsprachen die Datenbankinstanz als Interpretation der Klassen bzw. Konzepte ansehen, hebt Telos diese Trennung auf. Die Klassenbeschreibungen und ihre Instanzen sind gleich dargestellt und Teil derselben Objektbank.

3.3. Logik und Objektbanken

Die Logik taucht in zwei Aspekten im Zusammenhang mit Objektbanken auf. Einmal wird sie zur exakten Definition der Semantik von Objektbankschemata herangezogen. Zum anderen treten logische Formeln in manchen Objektbanken als deduktive Regeln auf. In [ABIT90] wird ein Vorschlag zur Kombination des semantischen Datenmodells aus [AK89] mit deduktiven Regeln gemacht. Die Regeln haben das Format

$$head \leftarrow body$$

wobei *head* ein positives Literal und *body* eine Konjunktion positiver und negativer Literale darstellt. Der Ansatz wird dadurch komplex, daß in dem Datenmodell auch Methoden in Form von Funktionen definiert sind. Sowohl Funktionen als auch Prädikate sind getypt. Die Funktionen dürfen in Termen als Argumente von Prädikaten auftauchen. Diese

Situation erzwingt eine Verallgemeinerung der Stratifikation, die in [AG91] beschrieben wird. Dort wird auch eine Fixpunktsemantik und eine bereichsbeschränkte Variante der Regeln eingeführt, die in polynomieller Zeit ausgewertet werden kann.

In [BEER90] wird Logik höherer Ordnung vorgeschlagen, um die Semantik objektorientierter Datenmodelle zu beschreiben. Die Formalisierung ist von der klaren Trennung zwischen Schema und konkreten Daten sowie zwischen Objekten und Werten gekennzeichnet. Werte sind dadurch gekennzeichnet, daß ihre Bedeutung über alle Anwendungen gleich ist und somit im System in einer Standarddarstellung fest vorgegeben werden können. Die Bedeutung wird durch die Bezeichnung selbst ausgedrückt. Objekte hingegen sind *abstrakt*. Ihre Bedeutung ergibt sich aus ihren Beziehungen zu anderen Objekten und Werten. [BEER90] führt ausdrücklich den Begriff des Zustandes $state(o)$ eines Objektes mit dem Identifikator o ein. Der Zustand ist ein komplexer oder atomarer Wert, oder der Identifikator eines anderen Objektes (vgl. auch die semantischen Datenmodelle und O_2). Eine Datenbank ist ein gerichteter Graph, dessen Knoten von den Objekten und Werten gebildet werden, und dessen Kanten die Beziehungen zwischen ihnen ausdrücken. Ein Objekt oder Wert taucht nur höchstens einmal im Graphen auf. Die Schemaebene dient dazu, die Wertetypen zu definieren, sowie Spezialisierungsbeziehungen zwischen ihnen aufzustellen. Es wird zwischen Typ (Menge aller möglichen Objektwerte) und Klasse (Menge der aktuell in der Datenbank vorhandenen Objekte) differenziert. Diese Dichotomie setzt sich fort, indem zwischen Typen eine Beziehung *subtype* bestehen kann, wenn der Untertyp alle Attribute des Obertypen hat. Auf der anderen Seite kann zwischen Klassen eine *isa*-Beziehung bestehen: die Unterklasse ist eine Teilmenge der Oberklasse.

Logische Ausdrücke können Literale der Form $x \in y$ enthalten, wobei y der Name einer Klasse ist und für die Menge seiner Instanzen steht. Dies ist eine Logik zweiter Stufe. Allerdings kann sie durch Einführung eines Literals $mem(x, y)$ auf eine Logik erster Stufe reduziert werden, die axiomatisierbar ist. Diese Logik wird üblicherweise auf eine bereichsbeschränkte Variante eingeschränkt, in der Variablen nur über Werte der Datenbank laufen dürfen.

Ein anderer Ansatz zur Formalisierung von Objektbanken ist die *F-Logik* [KL89,KLW90]. Diese Logik wird gleichzeitig dazu benutzt, die Semantik von Objektbeschreibungen zu definieren, und um Anfragen an die Datenbank zu stellen. Die Syntax wird durch die sogenannten **F-Terme** gebildet, die nachfolgend grob wiedergegeben wird.

a) Spezialisierungsterme $Q : P$. Sie werden durch eine partielle Ordnung auf den Bedeutungen von Q und P interpretiert.

b) Datenterme (Objekte) $P[f_1@Q_{11}, ..., Q_{1k_1} \rightarrow T_1; ...]$. Das Objekt P ist ein Tupel, dessen Komponenten durch Funktionsterme gegeben sind. Falls $k_1 = 0$ ist, so ist f_1 ein einfaches Attribut von P. Die Funktionen dürfen auch mengenwertig sein.

c) Signaturterme (Klassen) $C[f_1@D_{11},...,D_{1k_1} \Rightarrow A_1;...]$. Sie geben die Signatur der oben verwendeten Funktionssymbole an, sind also Typinformation für die Datenobjekte. Funktionssymbole werden durch Funktionen entsprechender Stelligkeit interpretiert. Mit ihnen werden Methoden (Operationen) formalisiert.

Auf den F-Termen wird eine prädikatenlogische Sprache, die **P-Terme**, definiert. Falls $T_1,...,T_n$ F-Terme sind, so ist $P(T_1,...,T_n)$ ein P-Term. Eine Objektbank ist dann eine Menge von Regeln *head* $\leftarrow$ *body* über den P-Termen mit positiven Literal *head*. Hinzu kommen die Signaturterme für die Typdeklarationen. Die Semantik der so gegebenen Theorie wird axiomatisch über einem Herbrand-Modell definiert. Ein System von Inferenzregeln beschreibt die Semantik von Spezialisierungstermen und die Ableitung von Typinformation aus Funktionstermen.

In den letzten Jahren wurde der Erweiterung von DATALOG um komplexe Terme sowie in Richtung der Logik 2. Stufe Beachtung geschenkt, um eine logiknahe Anfragesprache auch für nicht-relationale Datenbanken bereitstellen zu können. HiLog [CKW89] erweitert DATALOG um eingeschränkte Elemente der Logik 2. Stufe. Damit sind auf generische Weise Konstrukte wie die transitive Hülle eines binären Prädikats R definierbar:

```
closure(R)(x,y) :- R(x,y).
closure(R)(x,y) :- R(x,y),closure(R)(z,y).
```

Obwohl diese Syntax in der Logik 2. Stufe liegt, wird durch Einschränkung der erlaubten Regelmengen eine Einbettung der Semantik in die Logik 1. Stufe erreicht. Für diese wird ein angepaßter Resolutionskalkül angegeben.

3.4. Diskussion

Relationale Datenbanken wurden unter anderem wegen ihrer zu großen Entfernung zu den Anwendungsprogrammen kritisiert. Während diese Schwäche bei den programmiersprachlichen Objektbanken behoben ist, droht eine wichtige Stärke der relationalen Datenbanken verlorenzugehen: die Nähe zur Logik. Die Hindernisse ergeben sich aus der erhöhten Anzahl von Konzepten:

1. *Objektkomplexität:* Objektbanken wie O_2 und semantische Datenmodelle wie GSM oder IFO erlauben es, aus gewissen Grundobjekten bzw. Grundwerten komplexere Objekte mittels Tupel-, Mengen- und Listenkonstruktoren zu bilden. Dadurch wird die Anfragesprache komplizierter. So muß in O_2 ein komplexer Wert, etwa eine Liste von Listen, erst durch eine spezielle Anweisung *unnest* in eine flache Datenstruktur überführt werden, ehe eine Abfrage über das Enthaltensein eines Elements ausgeführt werden kann.

2. *Objektidentität:* In Objektbanken haben die Daten Objektidentität. Es stellt sich nun die Frage, ob abgeleitete Daten – seien es Ergebnisse von Anfragen, Folgerungen aus deduktiven Regeln, oder auch Ergebnisse von Methodenaufrufen – auch diese Eigenschaft haben sollen. Selbst wenn den Daten ein OID zugewiesen werden kann, so stellt sich immer noch die Frage, welcher Klasse sie dann zuzuordnen sind [CD91]. Für deduktive Regeln wurden einige Vorschläge gemacht, den abgeleiteten Fakten Identifikatoren zu geben (siehe [AK89,HY90]). Wenn Objekte und ihr (komplexer) Wert unterschieden werden, so gibt es zumindest zwei Arten von Gleichheit: gleiche Identifikatoren und gleiche Werte. [SZ90] führen in ihrer Anfragealgebra sogar für jede Schachtelungsebene i eines komplexen Objektes einen eigenen Gleichheitsoperator „$=_i$“ ein.

3. *Einkapselung:* Einkapselung ist ein Konzept aus dem Bereich der Programmiersprachen, genauer der abstrakten Datentypen. Dort dient sie dazu, die Abhängigkeiten zwischen einzelnen Modulen eines Programms auf eine möglichst kleine Schnittstelle zu beschränken. Dieses Ziel steht im Konflikt zu dem Anspruch einer anwendungsunabhängigen Anfragesprache, von der man erwartet, daß sie den Zugriff auf alle Informationen der Objektbank erlaubt.

4. *Klassifizierung:* Die programmiersprachlichen Objektbanken stellen das Konzept Typisierung in den Vordergrund. Ein Typ beschreibt eine Menge *möglicher* Werte, während eine Klasse eher als Menge der zu einer bestimmten Zeit vorhandenden Objekte verstanden wird. Grundsätzlich sind Klassen- oder Typbeschreibungen ebenso wie die Objekte Aussagen über einen Weltausschnitt. Eine Trennung beschneidet daher die Flexibilität im Umgang mit einer Objektbank. Diese Beobachtung wurde auch schon bei den objektorientierten Programmiersprachen gemacht [COIN87,FERB89].

5. *Hierarchien:* Die Möglichkeit der Spezialisierung ist ein wesentlicher Vorteil von Objektbanken, da durch sie die Redundanz in der Objektbank vermindert wird. Allerdings gibt es eine Überzahl an verschiedenen Ausformungen [BRAC83]. Aus Gründen der einfacheren Implementierung wird die Spezialisierung häufig auf Baumstrukturen eingeschränkt. Dort darf eine Klasse höchstens eine Oberklasse haben. Bei Objektbanken ohne diese Beschränkung spricht man von *multipler* Generalisierung.

6. *Dynamisches Binden:* Diese Möglichkeit ist wiederum typisch für programmiersprachliche Objektbanken. Die Motivation hierzu ist die erhöhte Benutzer- und Programmiererfreundlichkeit, da die gleiche Signatur für unterschiedliche Funktionen erlaubt ist. Vom Standpunkt der Logik bedeutet dies Nichtmonotonie, die einen zusätzlichen Grad an Komplexität in die Theorie einbringt [KLW90].

7. *Turing-Vollständigkeit:* Diese Eigenschaft macht eine Objektbank zu einer Umgebung, in der komplette Anwendungen implementiert werden können. Dieser Vorteil wird vom unausweichlichen Nachteil der Unentscheidbarkeit (z.B. der Terminierung eines Methodenaufrufs) begleitet. Wenn, wie etwa bei O_2, Methoden auch in Anfragen aufgerufen werden können, so ist auch die Terminierung von Anfragen ein unentscheidbares Problem. Relationale und die klassischen deduktiven Datenbanken haben dieses Problem nicht.

In der Summe haben diese Punkte bewirkt, daß für programmiersprachliche Objektbanken zwar deklarative Anfragesprachen, aber kaum Komponenten für deduktive Regeln und/oder Integritätsbedingungen entwickelt worden sind (siehe [BB*90]). Die Ansätze von [ABIT90,AG91,KLW90] erklären deduktive Regeln im Zusammenspiel mit komplexen Objekten und Methoden. Für das Thema dieser Arbeit sind sie allerdings nicht direkt anwendbar. Es ist unklar, wie die von einer Änderung an einer Komponente eines komplexen Objektes betroffenen Regeln bestimmt werden können. Dieses Suchproblem ist typisch für deduktive Datenbanken mit Integritätsbedingungen. Konsequenterweise werden in jenen Ansätzen Integritätsbedingungen und ihre Auswertung nicht behandelt.

Die Wissensrepräsentationssprachen der KL-ONE-Familie wurden für die Untersuchung von Konzepten entwickelt. Solche Fragestellungen sind typisch für die *Entwurfsphase* einer Datenbank, in der ein geeignetes Schema gesucht wird. Interaktion mit einer tatsächlichen Datenbank, also z.B. das Einfügen von Daten, wird hier weniger behandelt. Kennzeichnenderweise sind die behandelten Probleme meist schwierig, und die Menge der Konzeptbeschreibungen eher klein im Vergleich zu realistischen Datenbankgrößen. Dennoch haben die Ergebnisse auch eine Relevanz für den Bereich der Objektbanken. Die automatische Klassifikation kann z.B. zur Einordnung einer Anfrage in eine Klassenhierarchie herangezogen werden. Dadurch kann der Suchraum der zu testenden Objekte eingeschränkt werden.

Ein Blick zurück auf die vorgestellten Formalisierungen des Objektbankbegriffs zeigt, daß für den „passiven“Aspekt [DD86] durchgehend graphische Darstellung bevorzugt werden. Diese Beobachtung wird auch von [BEER90] geteilt. Der Grund ist die starke Korrespondenz zwischen Objekten und ihren Beziehungen untereinander auf der Seite und den graphischen Konzepten von Knoten und Kanten auf der anderen Seite. In dieser Hinsicht unterscheiden sich die Objektmodelle im Grunde nur durch die Bereitstellung spezieller Knoten- und Kantensymbole. Beispielsweise werden im semantischen Modell GSM Knoten mit einem Sternsymbol als Mengenkonstruktor interpretiert.

Weitaus komplizierter ist die graphische Darstellung des „aktiven“ Aspektes der Objektbanken, beispielsweise die Methoden von GemStone [MS86] und die Skripte von

Taxis [NCL*87], um nur zwei Beispiele zu nennen. Interessanterweise ist gerade hier auch die größte Divergenz der Vorschläge in der Literatur festzustellen.

Was ist nun das geeignete Modell, um deduktive Elemente in Objektbanken zu verpflanzen? Im Prinzip kann man zwei Standpunkte vertreten:

A) Eine deduktive Objektbank ist eine objektorientierte Datenbank, in die einige deduktive Elemente eingebaut sind.

B) Eine deduktive Objektbank ist eine deduktive Datenbank, in die einige objektorientierte Elemente eingebaut sind.

Bei der Entscheidung für eine der zwei Varianten folgt der Autor den Argumenten in [ULLM91], der eine komplette Übernahme aller objektorientierten Paradigmen als unvereinbar mit dem deklarativen Charakter deduktiver Datenbanken ansieht. Wir werden also eine deduktive Objektbank in erster Linie als eine deduktive Datenbank betrachten und bei der Übernahme objektorientierter Konzepte selektiv vorgehen. Als unverzichtbar sehen wir Objektidentität, Klassifikation, Spezialisierung und Attributierung an [BORG85,ABD*90]. Dagegen wird auf die Integration einer Turing-vollständigen Sprache in das Objektmodell verzichtet. Mit diesen Argumenten ist die Sprache **Telos** der adäquateste Vorschlag als Objektmodell einer deduktiven Datenbank:

- ▷ Die Informationsdarstellung in Telos ist dem relationalen Datenmodell ähnlich. In der Tat kann man Telos-Objekte als Tupel einer einzigen Relation ansehen. Dies erleichtert die Übertragung der relationalen Algorithmen.
- ▷ In der Sprachdefinition von Telos gibt es bereits eine Teilsprache für logische Formeln in ihren Rollen als deduktive Regeln und Integritätsbedingungen [KMSB89].
- ▷ Objekte, Klassen, Attribute, Klassenzugehörigkeit und Klassenspezialisierung werden alle einheitlich als Knoten und Kanten mit Objektidentität dargestellt. Der passive Aspekt der Objektbank ist mithin optimal abgedeckt, da sowohl Schema (Datenstrukturen) und Objekte (Daten) in jeder der vorgestellten Objektbanken graphisch darstellbar sind. In dieser Hinsicht ist Telos also allgemeingültig für alle Daten- und Objektmodelle.

Nach dieser Wahl ist eine Strategie der Umsetzung zu entwickeln. Als erstes wird in Kapitel 4 der Begriff *Objektbank* definiert. Die Bedeutung von Klassifikation, Spezialisierung und Aggregation werden als Axiome aufgeschrieben. In Kapitel 5 werden die Axiome von Telos dazu herangezogen, die logischen Formeln äquivalent in einfachere Formeln umzuformen. Die Möglichkeit hierzu ergibt sich aus der im Vergleich zum relational-deduktiven Fall stärkeren Struktur einer Objektbank. Telos vernachlässigt die Idee der komplexen Objekte. Kapitel 6 weist einen Weg, wie komplexe Objekte als Schar deduktiver

Regeln angesehen werden können. Die Idee hierzu stammt aus dem Bereich der Software-Konfiguration [RJG*91]. Das gewählte Datenmodell erlaubt Klassen, die selbst Klassen als Instanzen haben. Kapitel 7 untersucht hierzu die Konsequenzen für logische Formeln über solche Objekte.

In der Sprache Telos fehlen Operationen bzw. Methoden. Die Beschreibung kompletter Anwendungen ist daher nicht möglich. Kapitel 8 führt ein bekanntes Software-Prozeßmodell und die änderungsgesteuerten Auswertungsverfahren zusammen. Abgesehen vom Implementierungsteil können damit sowohl die Signatur einer Operation als auch ihr Aufruf beschrieben werden. Kapitel 9 ist der Implementierung der Verfahren in der Objektbank ConceptBase und ihrer Anwendung in Projekten gewidmet.

Das gewählte Objektmodell Telos stellt eine einzige Basisrelation zur Verfügung. Mit ihr werden sowohl Instanzen als auch Klassen und ihre jeweiligen Attribute einheitlich dargestellt. Folglich ist die Übertragung der relationalen Algorithmen sehr einfach. Jedoch muß noch eine Methode entwickelt werden, wie der Suchraum der betroffenen Integritätsbedingungen einzuschränken ist. Dazu wird eine striktere Variante der Sprache unter dem Namen O-Telos axiomatisch eingeführt. Es stellt sich heraus, daß damit die übertragene Methode eine Granularität der Änderungsoperationen unterstützt, die bis auf die Informationseinheit eines einzelnen Attributes heruntergeht.

Kapitel 4: Ein deduktives Objektbankmodell

Die Sprache Telos [MBJK90] ist ursprünglich zur Anforderungsanalyse von Informationssystemen sowie zur Realisierung natürlichsprachlicher Anfrageschnittstellen entwickelt worden. Diese wissensdarstellenden Aspekte werden von den Sprachimplementierungen [KMSB89] bzw. [GALL85] betont. Es stellt sich aber heraus, daß diese Sprache als Datenmodell für deduktive Objektbanken besonders geeignet ist, wenn sie entsprechend dieser Rolle erklärt wird. Nach der formalen Definition einer Objektbank als Menge von Quadrupeln wird die Bedeutung von Objektidentität, Aggregation, Klassifizierung und Spezialisierung mit Axiomen definiert. Alle Axiome basieren auf einem einzigen Prädikat, in dem sowohl Klassen- als auch Instanzeneigenschaften ausgedrückt werden. Hinzu kommen lediglich vordefinierte Prädikate für die Gleichheit, Ungleichheit und Ordnungsbeziehung zwischen Objektidentifikatoren. Die Einfachheit des Datenmodells impliziert Probleme bei der Stratifikation von deduktiven Regeln und bei der Größe des Suchraums für Formelauswertungen. Hierzu wird ein Konzept entwickelt, Literale einer Formel eindeutig den Klassen der Objektbank zuzuordnen.

4.1. Das grundlegende Objektmodell

In O-Telos werden alle Informationen der Objektbank mit einer einzigen Datenstruktur, einem Quadrupel, dargestellt. Damit werden sowohl Objekte und Klassen, als auch deren Attribute, Instanzen- und Spezialisierungsbeziehungen dargestellt. Diese Uniformität hat als Konsequenz, daß all diese Informationseinheiten *Objekte* sind: Klassen sind Objekte, sie können sowohl Instanzen haben als auch selbst Instanz von Klassen sein. Ebenso sind Attribute vollwertige Objekte. Definition 4-1 erklärt den Begriff Objektbank als eine Menge von Quadrupeln.

DEFINITION 4-1

Seien ID, LAB abzählbar unendliche Mengen von Identifikatoren bzw. Namen. Dann heißt eine endliche Teilmenge

$$\mathrm{OB} \subseteq \{P(o,x,l,y) |\ o,x,y \in \mathrm{ID}, l \in \mathrm{LAB}\}$$

extensionale Objektbank (in O-Telos). Die Elemente von OB heißen **Objekte**. Für ein Objekt $P(o,x,l,y)$ heißt o der **Identifikator**, x die **Quelle**, l der **Name** und y das **Ziel** des Objektes. Die Menge

$$\mathrm{OID(OB)} = \{o \in \mathrm{ID} |\ \text{es gibt } x,l,y \in \mathrm{ID} \text{ mit } P(o,x,l,y) \in \mathrm{OB}\}$$

heißt Menge der **Objektidentifikatoren** der Objektbank OB.

Die Definition ist ähnlich zu dem binären Datenmodell [ABRI74] und zu Formalisierungen von semantischen Datenbanken, z.B. [ER91]. Der Unterschied zu ihnen ist der durchgängige Gebrauch von Objektidentifikatoren und die Darstellung des Beziehungsnamens als (dritte) Komponente anstatt als Name der Relation. Zwei Objektnamen werden ausgezeichnet. $P(o,x,in,c)$ drückt aus, daß das Objekt mit Identifikator x eine **Instanz** des Objektes mit Identifikator c ist. Spezialisierungen zwischen Objekten werden durch Objekte der Form $P(o,c,isa,d)$ dargestellt.

Es ist nützlich, Objektbanken als gerichtete Graphen darzustellen. Objekte der Form $P(o,o,l,o)$ werden dabei als Knoten mit der Marke l gezeichnet. Sie heißen auch **Individualobjekte**. Ein **Klassifikationsobjekt** $P(o,x,in,c)$ wird als gestrichelter Pfeil zwischen den graphischen Darstellungen für x und c dargestellt, **Spezialisierungsobjekte** $P(o,c,isa,d)$ tauchen als graue Pfeile auf. Alle anderen Objekte $P(o,x,l,y)$, die sogenannten **Attribute**, werden als einfache Pfeile zwischen x und y gezeichnet und mit l markiert. Individualobjekte der Form $P(v,v,v,v)$ heißen auch **Wertobjekte**. Bei ihnen stimmen Name und Identifikator überein. Wertobjekte werden zur Darstellung von Konstanten aus den natürlichen Zahlen (0,1,2,...), für Zeichenketten ("abc",...) etc. benutzt. Für den Rahmen dieser Arbeit beschränken wir uns auf diese beiden Wertemengen. Graphisch werden Wertobjekte durch Knoten ohne Rahmen dargestellt.

Ein Beispiel zeigt Abbildung 4-1. Das Objekt mit Namen *Jack* hat ein Attribut *drug1* mit Ziel *QuasiForte*. *Jack* ist Instanz des Objektes *Patient*, das wiederum Unterklasse von *Person* ist. Das Attribut *takes* von *Patient* ist eine Klasse des Attributes *drug1* von *Jack*. In dem Bild tauchen nur Objektnamen auf. Dies ist kein Zufall, wenn man bedenkt, daß die Identifikatoren außerhalb der Objektbank (also im Anwendungsprogramm) an sich keine Bedeutung tragen. Die Objektidentifikatoren werden in dem Beispiel aus Gründen der Lesbarkeit entsprechend den Namen gebildet. In einer Implementierung werden hier sinnvollerweise persistente Speicherplatzadressen eingesetzt, die einen schnellen Zugriff auf die Komponenten eines Objektes erlauben.

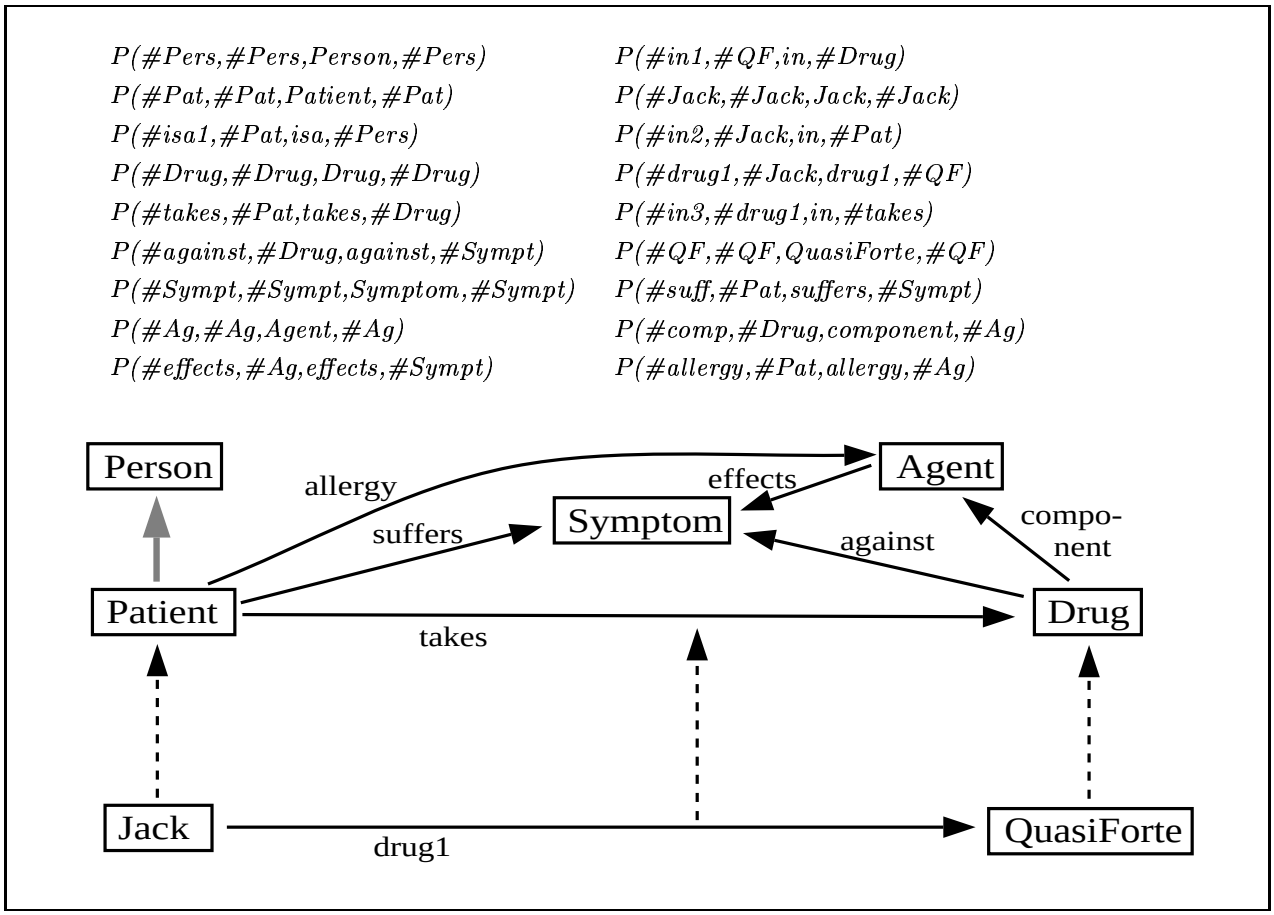

Abb. 4-1: Objektbank in O-Telos und ihre graphische Darstellung

4.2. Axiome von O-Telos

Die Definition der extensionalen Objektbank läßt die Bedeutung der Begriffe Instanz, Objektidentität, Unterklasse usw. offen. In Fortsetzung der beweistheoretischen Definition von relationalen Datenbanken [REIT84] wird die Bedeutung anhand zusätzlicher Axiome definiert. Ausgangspunkt ist die extensionale Objektbank, deren Elemente alle gleichgestaltig sind. Neben den Axiomen für Bereichsabschluß, Konstanteneindeutigkeit und Vollständigkeit (*closed world assumption*) wird als erstes verlangt, daß Objektidentifikatoren eindeutig die Objekte bestimmen.

$$\begin{aligned} &\forall\ o, x_1, l_1, y_1, x_2, l_2, y_2\ \ P(o, x_1, l_1, y_1) \wedge P(o, x_2, l_2, y_2) \Rightarrow \\ &\quad (x_1 = x_2) \wedge (l_1 = l_2) \wedge (y_1 = y_2) \end{aligned} \qquad (A_1)$$

Bei objektorientierten Datenbanken ist es üblich, den Identifikator als Objektadresse anzusehen. In der gesamten Lebenszeit einer Objektbank wird ein Identifikator höchstens einmal an ein Objekt vergeben. Dies unterscheidet ihn von den Schlüsseln einer relationalen Datenbank. Der Objektname soll für die konzeptionelle Ebene, in der Anfragen und

Änderungen formuliert werden, benutzt werden. Wir verlangen, daß dieser Name für Individualobjekte eindeutig ist. Für Attribute muß er in Verbindung mit der Quelle des Attributs eindeutig sein. Bei Instanzen- und Spezialisierungsobjekten ist der Name zusammen mit Quelle und Ziel ein Schlüssel[4]:

$$\forall\ o_1, o_2, l\ P(o_1, o_1, l, o_1) \wedge P(o_2, o_2, l, o_2) \Rightarrow (o_1 = o_2) \qquad (A_2)$$

$$\forall\ o_1, x, l, y_1, o_2, y_2\ P(o_1, x, l, y_1) \wedge P(o_2, x, l, y_2)\ \Rightarrow \qquad (A_3)$$
$$(o_1 = o_2) \vee (l = in) \vee (l = isa)$$

$$\forall\ o_1, x, l, y, o_2\ P(o_1, x, l, y) \wedge P(o_2, x, l, y)\ \wedge \qquad (A_4)$$
$$((l = in) \vee (l = isa)) \Rightarrow (o_1 = o_2)$$

Klassifikation und Spezialisierung sind Grundbegriffe für objektorientierte Datenbanken. In vielen Situationen ist es man nur daran interessiert, daß ein Objekt Instanz bzw. Spezialisierung eines anderen Objektes ist. Daher werden Prädikate definiert, die das Aufschreiben von logischen Aussagen erleichtern:

$$\forall\ o, x, c\ P(o, x, in, c) \Rightarrow In(x, c) \qquad (A_5)$$

$$\forall\ o, c, d\ P(o, c, isa, d) \Rightarrow Isa(c, d) \qquad (A_6)$$

$$\forall\ o, x, l, y, p, c, m, d\ P(o, x, l, y) \wedge P(p, c, m, d) \wedge In(o, p) \Rightarrow A(x, m, y) \qquad (A_7)$$

Falls $In(x, c)$ gilt, so sagen wir, x sei **Instanz** von c. Das Objekt c heißt dann **Klasse** von x. Analog heißt c **Spezialisierung** (**Unterklasse**) von d, wenn $Isa(c, d)$ wahr ist. In diesem Fall nennt man d auch **Generalisierung** oder **Oberklasse** von c. Das Prädikat $A(x, m, y)$ drückt aus, daß das Objekt x ein Attribut mit Ziel y hat, und dieses Attribut Instanz eines Objektes mit Namen m ist. Das Attribut p der Klasse c heißt auch **Attributkategorie**, da es zur Klassifikation der Attribute der Instanzen von c benutzt wird. Mit der extensionalen Objektbank von Abb. 4-1 ist zum Beispiel $A(\#Jack, takes, \#QF)$ mit Axiom A_7 ableitbar. Man beachte, daß es bei festem x und m null, ein oder mehrere Objekte y geben kann, so daß $A(x, m, y)$ gilt. Auf diese Weise werden mengenwertige Attribute simuliert. Zudem kann ein Attribut o (siehe A_7) Instanz beliebig vieler Attributkategorien p sein. Diese Eigenschaft der *multiplen Klassifikation* gilt für alle Objekte in O-Telos.

[4] Zusammengenommen bewirken die Axiome A_2 bis A_4, daß jedes Objekt mit einem Ausdruck über den Objektnamen bezeichnet werden kann (vgl. unten Lemma 4-2). Soweit nicht ausdrücklich angezeigt, verwenden wir im folgenden wegen der kürzeren Schreibweise Objektidentifikatoren.

Das folgende Axiom A_8 besagt, daß ein Objekt nur in einer Klasse mit Attribut m sein kann, wenn es dieses instanziiert, oder aber $A(x,m,y)$ nicht gilt.

$$\begin{aligned} \forall\ x,y,p,c,m,d\ \ & In(x,c) \wedge A(x,m,y) \wedge P(p,c,m,d) \Rightarrow \\ & \exists\ o,l\ P(o,x,l,y) \wedge In(o,p) \end{aligned} \qquad (A_8)$$

Aufbauend auf den Spezialisierungsobjekten wird *Isa* wie üblich [KLW90] als partielle Ordnung (reflexiv, transitiv und antisymmetrisch) erklärt (Axiome A_9, A_{10}, A_{11}). Da der Begriff Klasse in O-Telos nur bezüglich von Instanzenbeziehungen auftaucht, wird die Spezialisierung für alle Objekte (d.h. Instanzen des vordefinierten Objektes *#Obj*, siehe auch Axiome A_{17}, A_{18}, A_{28}) erklärt. Zu beachten ist, daß in O-Telos *multiple Generalisierung* zugelassen ist: zu einem c darf es also beliebig viele d mit $Isa(c,d)$ geben.

$$\forall\ c\ In(c, \#Obj) \Rightarrow Isa(c,c) \qquad (A_9)$$

$$\forall\ c,d,e\ Isa(c,d) \wedge Isa(d,e) \Rightarrow Isa(c,e) \qquad (A_{10})$$

$$\forall\ c,d\ Isa(c,d) \wedge Isa(d,c) \Rightarrow (c = d) \qquad (A_{11})$$

Spezialisierung in O-Telos hat zwei Aspekte. Der erste ist die Vererbung der Klassenzugehörigkeit (Axiom A_{12}). Jede Instanz der Unterklasse ist auch Instanz der Oberklasse. Der zweite Aspekt ist die (implizite) Vererbung der Attribute der Oberklasse auf die Unterklasse, indem die Bedingung $In(x,c)$ für die Attributbenutzung in Axiom A_{13} erfüllt wird.

$$\forall\ p,x,c,d\ In(x,d) \wedge P(p,d,isa,c) \Rightarrow In(x,c) \qquad (A_{12})$$

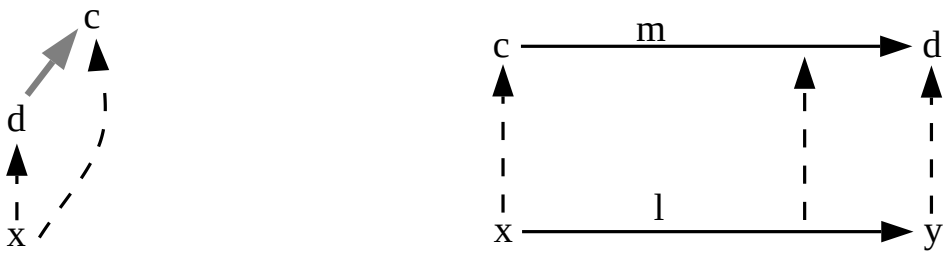

Abb. 4-2: Axiome A_{12} und A_{13} von O-Telos

Graphisch wird A_{12} durch die linke Hälfte von Abb. 4-2 symbolisiert. Die rechte Hälfte stellt das Klassifikationsaxiom A_{13} von O-Telos dar. Wenn ein Objekt mit Quelle x und Ziel y Instanz eines anderen Objektes mit Quelle c und Ziel d ist, so muß x auch Instanz

von c, und y muß Instanz von d sein. Ein Beispiel ist durch die Objekte *#drug1* und *#takes* in Abb. 4-1 gegeben. Für Instanzenbeziehungen zwischen Individualobjekten (z.B. *#Jack* und *#Pat*) ist das Axiom trivialerweise erfüllt, da dort Identifikator, Quelle und Ziel identisch sind [CCI87].

$$\begin{aligned} &\forall\ o,x,l,y,p\ P(o,x,l,y) \wedge In(o,p) \Rightarrow \\ &\quad \exists\ c,m,d\ P(p,c,m,d) \wedge In(x,c) \wedge In(y,d) \end{aligned} \qquad (A_{13})$$

Während die Klassifikation von Objekten an die Elementrelation der Mengenlehre erinnert (ohne aber die dort möglichen Paradoxien aufzuweisen), kann die Spezialisierung mit der Untermengenrelation verglichen werden (siehe Axiom A_{12}). Die folgenden drei Axiome (vgl. Abb. 4-3) bestimmen das Zusammenspiel von Spezialisierung und Attributierung.

$$\forall\ c,d,a_1,a_2,m,e,f \qquad (A_{14})$$
$$Isa(d,c) \wedge P(a_1,c,m,e) \wedge P(a_2,d,m,f) \Rightarrow Isa(f,e) \wedge Isa(a_2,a_1)$$

$$\forall\ c,d,a_1,a_2,m_1,m_2,e,f \qquad (A_{15})$$
$$Isa(a_2,a_1) \wedge P(a_1,c,m_1,e) \wedge P(a_2,d,m_2,f) \Rightarrow Isa(d,c) \wedge Isa(f,e)$$

$$\forall\ x,m,y,c,d,a_1,a_2,e,f\ (In(x,c) \wedge In(x,d) \wedge P(a_1,c,m,e) \wedge P(a_2,d,m,f) \qquad (A_{16})$$
$$\Rightarrow \exists\ g,a_3,h\ In(x,g) \wedge P(a_3,g,m,h) \wedge Isa(g,c) \wedge Isa(g,d))$$

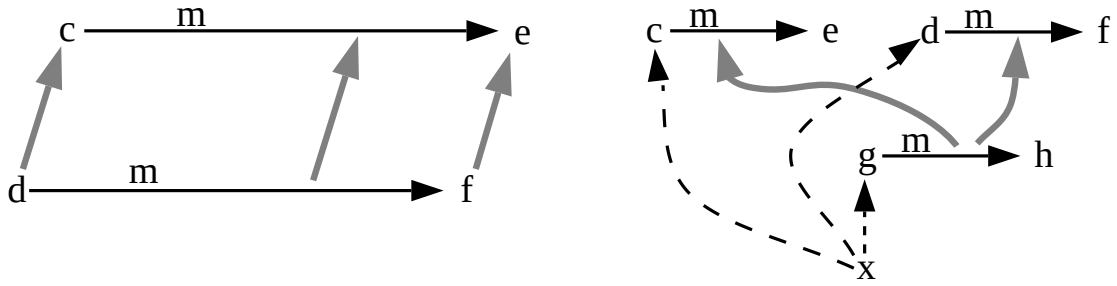

Abb. 4-3: Axiome A_{14} bis A_{16} von O-Telos

Das Axiom A_{14} beschreibt die *Verschärfung* von Attributen bei Unterklassen: wenn eine Unterklasse ein Attribut mit gleichen Namen wie bei einer seiner Oberklassen hat, so müssen sowohl das Attribut vom Oberklassenattribut als auch das Ziel dieses Attributs vom Ziel des Oberklassenattributs spezialisiert sein. Dadurch wird erreicht, daß die Verwendung von Attributen gleichen Namens in Unterklassen mit der Definition der

Oberklasse kompatibel ist. Axiom A_{15} fordert zusätzlich, daß bei Spezialisierungen zwischen Objekten die Quell- und Zielobjekte spezialisiert sein müssen (vgl. Axiom A_{13}). Mit Axiom A_{12} folgt die wichtige Eigenschaft, daß jede Instanz des Unterklassenattributs auch Instanz des Oberklassenattributs sein muß.

In O-Telos darf ein Objekt zu mehreren Klassen Instanz sein. Diese Flexibilität hat als Konsequenz eine Mehrdeutigkeit der Attributzuordnung für den Fall, daß mehrere Klassen eines Objektes x ein Attribut gleichen Namens haben. Dieses Problem tritt insbesondere bei multipler Generalisierung (eine Klasse hat mehr als eine Oberklasse) auf. Häufig wird die Semantik dann mit nicht-monotoner Logik [BRAC83,KLW90] beschrieben, in anderen Objektmodellen [BBB*88] wird Spezialisierung gar auf höchstens eine Oberklasse pro Klasse eingeschränkt. Axiom A_{16} definiert eine strikte Art der Attributzuordnung. Falls ein Objekt x in zwei Klassen c, d ist, die beide ein Attribut gleichen Namens haben, so muß x auch in einer Klasse g sein, die ebenfalls ein solches Attribut hat, welches zu den Attributen von c, d Unterklasse ist. Man beachte, daß c, d, g nicht notwendig verschieden sein müssen. Im nächsten Kapitel wird dieses Axiom dazu benutzt, um Literalvorkommen $A(x, m, y)$ in einer Formel eindeutig einem Attribut mit Namen m zuzuordnen.

Die Axiome A_{17} bis A_{27} regeln die Zugehörigkeit zu den 5 *Systemklassen*, d.h. die durch die Axiome A_{28} bis A_{32} vordefinierten Objekte *#Obj*, *#Indiv*, *#Attr*, *#Inst* und *#Spec* (vgl. Abb. 4-4).

$$\forall\, o,x,l,y\ P(o,x,l,y) \Rightarrow In(o,\#Obj) \tag{A_{17}}$$

$$\forall\, o\ In(o,\#Obj) \Rightarrow \exists\, x,l,y\ P(o,x,l,y) \tag{A_{18}}$$

$$\forall\, o,l\ P(o,o,l,o) \Rightarrow In(o,\#Indiv) \tag{A_{19}}$$

$$\forall\, o\ In(o,\#Indiv) \Rightarrow \exists\, l\ P(o,o,l,o) \tag{A_{20}}$$

$$\forall\, o,x,c\ P(o,x,in,c) \Rightarrow In(o,\#Inst) \tag{A_{21}}$$

$$\forall\, o\ In(o,\#Inst) \Rightarrow \exists\, x,c\ P(o,x,in,c) \tag{A_{22}}$$

$$\forall\, o,c,d\ P(o,c,isa,d) \Rightarrow In(o,\#Spec) \tag{A_{23}}$$

$$\forall\, o\ In(o,\#Spec) \Rightarrow \exists\, c,d\ P(o,c,isa,d) \tag{A_{24}}$$

$$\forall\, o,x,l,y\ P(o,x,l,y) \wedge (o \neq x) \wedge (o \neq y) \wedge (l \neq in) \wedge (l \neq isa) \Rightarrow In(o,\#Attr) \tag{A_{25}}$$

$$\forall\, o\ In(o,\#Attr) \Rightarrow \exists\, x,l,y\ P(o,x,l,y) \wedge (o \neq x) \wedge (o \neq y) \wedge (l \neq in) \wedge (l \neq isa) \tag{A_{26}}$$

$$\forall\, o\ In(o,\#Obj) \Rightarrow In(o,\#Indiv) \vee In(o,\#Inst) \vee In(o,\#Spec) \vee In(o,\#Attr) \tag{A_{27}}$$

Die Axiome A_{17}, A_{18} besagen, daß die Objekte einer Objektbank genau die Instanzen der Klasse mit Namen *Object* (siehe auch Axiom A_{28}) sind. Durch ihre Form werden die Instanzen der Klassen *Individual*, *InstanceOf*, *IsA* und *Attribute* bestimmt. Die drei letzteren sind gemäß $A_{29}, ..., A_{32}$ Attribute von *Object*. Dadurch werden die Begriffe Individualobjekt, Attribut, Spezialisierungs- und Klassifikationsobjekt genau definiert und sind expliziter Teil des Objektmodells. Axiom A_{27} verbietet solche Objekte, die nicht in eine der obigen vier Klassen fallen. So ist etwa $P(\#1, \#1, l, \#2)$ kein zulässiges Objekt.

$$P(\#Obj, \#Obj, Object, \#Obj) \qquad (A_{28})$$

$$P(\#Indiv, \#Indiv, Individual, \#Indiv) \qquad (A_{29})$$

$$P(\#Attr, \#Obj, attribute, \#Obj) \qquad (A_{30})$$

$$P(\#Inst, \#Obj, InstanceOf, \#Obj) \qquad (A_{31})$$

$$P(\#Spec, \#Obj, IsA, \#Obj) \qquad (A_{32})$$

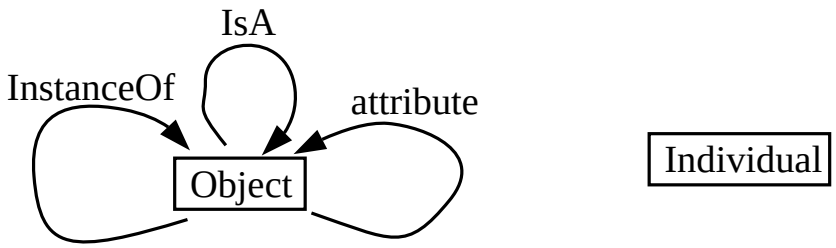

Abb. 4-4: Axiome A_{28} bis A_{32} von O-Telos

Die Axiome A_{10},A_{12} und A_{13} sind [MBJK90] entnommen, wo sie für die Version von Telos mit Zeitkomponente definiert sind. Das Axiom A_{14} über die Spezialisierung von Attributen bei Unterklassen findet sich in [KMSB89] für Individualobjekte. In O-Telos wird die Spezialisierung auf Attribute fortgesetzt (Axiome A_{15}, A_{16}). Diese Erweiterung ist wesentlich für die Interaktion von Spezialisierung und logischen Formeln. Sie wird im Unterkapitel 4.4 benutzt. Das Objektidentitätsaxiom A_1 findet sich zwar nicht in den bisherigen Telos-Formalisierungen, ist aber für eine objektorientierte Sprache quasi selbstverständlich. Die Axiome A_2, A_3, A_4 sind eine Verfeinerung eines Postulats aus [CCI87]. Wie unten noch zu zeigen ist, kann damit unter einer kleinen Zusatzannahme die eindeutige Benennung jedes Objektes einer Objektbank durch einen Ausdruck aus Objektnamen erreicht werden. Die Axiome A_{17} bis A_{32} sind ebenfalls neu. Sie legen eindeutig die Bedeutung der Begriffe Objekt, Individualobjekt, Attribut, Klassifikationsobjekt und

Spezialisierungsobjekt fest. Ihnen entsprechen die fünf Systemklassen. Im Gegensatz zu [CCI87] fehlt ein Axiom, das Attribute von Oberklassen automatisch an Unterklassen vererbt. Ein solches Axiom ist laut [BRAC83] ein reiner Implementierungsaspekt.

4.3. Eigenschaften der Theorie auf O-Telos

Im folgenden gehen wir unter Wiederverwendung der Begriffe aus Kapitel 2 von einer **Objektbanktheorie** (OB,AX,IC) als Spezialfall einer relationalen Datenbanktheorie aus, wobei OB eine extensionale Objektbank – mithin ein Spezialfall einer extensionalen Datenbank – ist und AX die Menge der Axiome R_1, R_{ij}, R_P (Kap. 2) sowie der Axiome A_1 bis A_{32} ist. Die dritte Komponente IC, die für die Integritätsbedingungen reserviert ist, bleibt zunächst leer. Integritätsbedingungen und deduktive Regeln werden im nächsten Unterkapitel behandelt. Eine Reihe wichtiger Eigenschaften kann für Objektbanktheorien bewiesen werden. So ist jedes Objekt automatisch Instanz von mindestens zwei Klassen, und es gilt die referentielle Integrität:

LEMMA 4-1

Sei (OB,AX,IC) eine Objektbanktheorie und $P(o,x,l,y)$ ein beliebiges Objekt aus OB. Dann gilt:

a) $In(o,\#Obj)$ und $In(o,c)$ für genau ein $c \in \{\#Indiv, \#Attr, \#Inst, \#Spec\}$ sind wahr.

b) In OB gibt es Objekte mit Identifikatoren x und y (**referentielle Integrität**).

Beweis. Teil a) folgt aus A_{17} und A_{27}. Mit A_{18} bis A_{26} zeigt man leicht, daß die Disjunktion von A_{27} zu einem exklusivem „Oder“ verschärft werden kann. Teil b) ist wie folgt zu sehen: Wegen A_{17} muß $In(o,\#Obj)$ gelten. Damit ist die Prämisse von A_{13} mit $p{=}\#Obj$ erfüllt. $P(\#Obj,\#Obj,Object,\#Obj)$ ist wegen A_1 die einzige Lösung für $P(\#Obj,c,m,d)$. Also müssen $In(x,\#Obj)$ und $In(y,\#Obj)$ gelten. Nach Axiom A_{18} muß es dann x_1,l_1,y_1 sowie x_2,l_2,y_2 geben, die $P(x,x_1,l_1,y_1)$ bzw. $P(y,x_2,l_2,y_2)$ wahr machen. □

Mit Hilfe der Axiome A_2 bis A_3 kann die eindeutige Benennung jedes Objektes einer Objektbank durch einen Funktionsausdruck erreicht werden. Dazu seien für eine Objektbank OB vier partielle Funktionen (siehe Abb. 4-5) definiert.

Alle vier Funktionen sind wohldefiniert. Wegen Axiom A_2 kann es für ein $l \in$ LAB höchstens ein o mit $P(o,o,l,o) \in$ OB geben. Analog folgt die Wohldefiniertheit für f,g,h aus A_3 und A_4. Offenbar liefert d in seinem Definitionsbereich Identifikatoren von Individualobjekten, f liefert Identifikatoren von Attributen, g von Klassifikationsobjekten, und h von Spezialisierungsobjekten.

$$d : \mathrm{LAB} \to \mathrm{ID}$$
$$f : \mathrm{ID} \times \mathrm{LAB} \to \mathrm{ID}$$
$$g : \mathrm{ID} \times \mathrm{ID} \to \mathrm{ID}$$
$$h : \mathrm{ID} \times \mathrm{ID} \to \mathrm{ID}$$

$$d(l) = \begin{cases} o, & P(o,o,l,o) \in \mathrm{OB} \\ \bot, & \text{sonst} \end{cases}$$

$$f(x,l) = \begin{cases} o, & P(o,x,l,y) \in \mathrm{OB} \text{ und } l \notin \{in, isa\}, o \neq x \\ \bot, & \text{sonst} \end{cases}$$

$$g(x,y) = \begin{cases} o, & P(o,x,in,y) \in \mathrm{OB} \text{ und } o \neq x \\ \bot, & \text{sonst} \end{cases}$$

$$h(x,y) = \begin{cases} o, & P(o,x,isa,y) \in \mathrm{OB} \text{ und } o \neq x \\ \bot, & \text{sonst} \end{cases}$$

Abb. 4-5: Zugriffsfunktionen für Objekte

LEMMA 4-2

Sei (OB,AX,IC) eine Objektbanktheorie. Ferner sei „$\preceq$“ eine Totalordnung auf ID, und es gelte

$$\forall\, o,x,l,y\; P(o,x,l,y) \Rightarrow (x \preceq o) \wedge (y \preceq o) \qquad (A_{33})$$

Dann gibt es für jeden Objektidentifikator $o_1 \in \mathrm{OID(OB)}$ einen Funktionsausdruck t über $\{d,f,g,h\}$ und ID mit Wert o_1, in dem außer den Funktionssymbolen nur Objektnamen vorkommen.

Beweis (konstruktiv). Das Logikprogramm aus Abbildung 4-6 berechnet nach der SLDNF-Strategie den gesuchten Ausdruck. Variablen werden durch Großbuchstaben repräsentiert. Eingabe ist ein Identifikator `OID`, Ausgabe ist der gesuchte Funktionsausdruck `T`, falls `OID` der Identifikator eines Objektes von OB ist. Ansonsten wird „kein OID“ ausgegeben. Die Objektbank OB wird durch das Prädikat `P` repräsentiert.

Die Terminierung ist wie folgt zu sehen. Regel 1 enthält keine rekursiven Aufrufe. Für die rekursiven Aufrufe `REF(X,T)` in den Regeln 2 bis 4 gilt, daß `X` an einen Wert gebunden ist (wg. Aufruf von `P`), und daß `X`$\preceq$`O` (wg. A_{33}). Man nehme an, es gäbe eine unendliche Berechnung. Dann muß es eine unendliche Kette von Aufrufen `REF(`x_1,t_1`)`,`REF(`x_2,t_2`)`,...,`REF(`x_i,t_i`)`,`REF(`x_{i+1},t_{i+1}`)`,... geben mit $x_{i+1} \preceq x_i$ für alle $i \geq 1$. Der Fall $x_{i+1}{=}x_i$ kann nicht auftreten, da dann ein Individualobjekt vorgelegen

```
REF(O,d(L)) :-                % Regel 1: Individualobjekte
  P(O,O,L,O).

REF(O,f(T,L)) :-              % Regel 2: Attribute
  P(O,X,L,Y),
  L <> in, L <> isa, O <> X,
  REF(X,T).

REF(O,g(T1,T2)) :-            % Regel 3: Klassifikationsobjekte
  P(O,X,in,Y),
  O <> X,
  REF(X,T1), REF(Y,T2).

REF(O,h(T1,T2)) :-            % Regel 4: Spezialisierungsobjekte
  P(O,X,isa,Y),
  O <> X,
  REF(X,T1), REF(Y,T2).

REF(O,"kein OID").            % Regel 5: Kein Objekt mit diesem Identifikator

?- read(OID),REF(OID,T),write(T).
```

Abb. 4-6: Programm zur Berechnung von Funktionsausdrücken für Objekte

hätte (Axiome $A_{17} - A_{27}$) und Regel 1 das Programm terminiert hätte. Also gilt $x_i \preceq x_{i+1}$ und $x_i \neq x_{i+1}$ für alle i. Also enthält die Objektbank unendlich viele Objekte, was ein Widerspruch zur Endlichkeitsannahme von OB (Def. 4-1) ist. Zur Korrektheit: Falls die Eingabe `O1` kein Objektdentifikator ist, so trifft keine der Regeln 1 bis 4 zu, sondern nur Regel 5, die die richtige Ausgabe liefert. Falls `O1` in OB existiert, so muß eine der Regeln zutreffen. Diese Eigenschaft gilt wegen der referentiellen Integrität für alle rekursiven Aufrufe. Also muß das Programm mit einem Aufruf von Regel 1 terminieren. Aus der Konstruktion der Funktionsausdrücke folgt: der Ausgabeterm `T` enthält nur Ausdrücke über Objektnamen. Der Wert von `T` ist `O1`, da die Regeln in dem Programm der Definition der Funktionen in Abb. 4-5 entsprechen. □

Korollar

Zu jedem Funktionsausdruck t über den Funktionen d, f, g, h und ID gibt es genau ein $o \in \mathrm{OID(OB)} \cup \{\perp\}$ mit $t = o$ (d.h. o ist Wert von t).

Beweis. Diese Aussage folgt sofort aus der Definition der Funktionen in Abb. 4-5 sowie den Axiomen A_2, A_3 und A_4. □

Die Totalordnung auf ID und die Eigenschaft A_{33} sind für Implementierungen von Objektbanken in O-Telos einfach zu realisieren: für jedes neue Objekt wird durch

einen Zähler der nächsthöhere Objektidentifikator generiert. Die eindeutige Benennung von Objekten durch einen Funktionsausdruck ist eine wünschenswerte Eigenschaft, da Objektidentifikatoren meist vom System generiert werden, und mithin dem Benutzer nicht bekannt sind. Abbildung 4-7 zeigt, daß es auch Objektbanken gibt, die A_{33} nicht erfüllen. Sie stellt Objektmengen der Form $\{P(o_1, o_2, l_1, o_2), P(o_2, o_1, l_2, o_1)\}$ und $\{P(o_1, o_3, l_1, o_2), P(o_2, o_1, l_2, o_3), P(o_3, o_2, l_3, o_1)\}$ dar. Obwohl die Axiome von O-Telos durch diese „Schleifenobjekte“ nicht verletzt sind, kann auf sie offenbar ohne Schaden verzichtet werden, da sie nur deshalb auftreten, weil Attribute Objektidentität besitzen. Formel A_{33} wird in die Menge der Axiome von O-Telos aufgenommen.

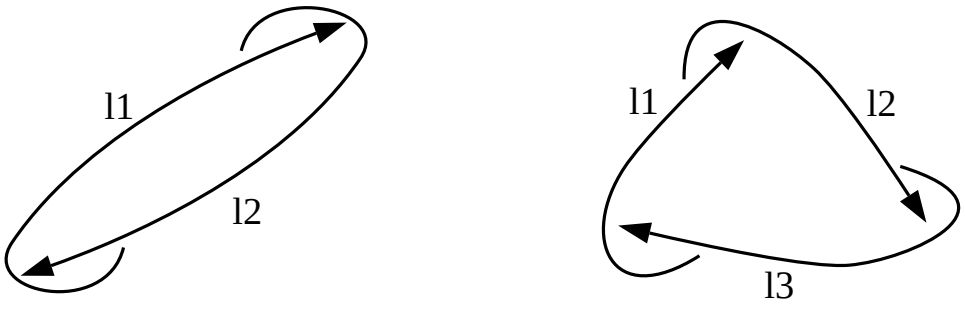

Abb. 4-7: Objektbank, die A_{33} verletzt

Beispiele von Funktionsausdrücken sind mit der Objektbank von Abbildung 4-1

$$d(Person) = \#Pers$$
$$f(d(Patient), takes) = \#takes$$
$$g(f(d(Jack), drug1), f(d(Patient), takes)) = \#in3$$

Eine weitere Eigenschaft ist die *lokale Eindeutigkeit der Attributklasse*, die aus Axiom A_{16} folgt.

LEMMA 4-3

Sei (OB,AX,IC) eine Objektbanktheorie, $x \in \mathrm{OID(OB)}$ ein Objektidentifikator, m ein Attributname und y ein Objektidentifikator derart, daß $A(x, m, y) \in cons(\mathrm{OB} \cup \mathrm{AX})$. Dann gibt es ein eindeutig bestimmtes Objekt $P(a_s, c_s, m, d_s)$ mit:

$$A(x, m, y) \Leftrightarrow \exists o, l\ P(o, x, l, y) \wedge In(o, a_s)$$

und für alle anderen $P(a, c, m, d)$ mit $In(o, a)$ für ein $P(o, x, l, y) \in \mathrm{OB}$ gilt:

$$In(o, a_s) \Rightarrow In(o, a)$$

Beweis. Sei $M = \{c_1, ..., c_n\}$ die Menge aller Identifikatoren von Klassen von x, die ein Attribut namens m haben, und $B = \{a_1, ..., a_n\}$ die Menge der Identifikatoren dieser Attribute. Da $A(x, m, y)$ gilt und es nur aus Axiom A_7 gefolgert werden kann (siehe Vollständigkeitsaxiom, Kap. 2), muß es mindestens ein Objekt $P(a, c, m, d)$ und ein Objekt $P(o, x, l, y)$ geben mit $In(o, a)$. Also folgt aus Axiom A_{13} $In(x, c)$. Also enthalten M und B zumindest je ein Element nämlich c bzw. a. Außerdem existiert nach den Axiomen A_{14}, A_{16} für je zwei $a_1, a_2 \in B$ ein $a_3 \in B$ mit $Isa(a_3, a_1)$ und $Isa(a_3, a_2)$. Da *Isa* zudem eine partielle Ordung ist, gibt es genau ein „kleinstes Element" a_k mit $Isa(a_k, a_i)$ für alle $a_i \in B$. Also ist a_k mit $P(a_k, c_k, m, d_k) \in$ OB einziger Kandidat für a_s. Bleibt zu zeigen: a_k erfüllt die Äquivalenz von Lemma 4-3. Die Richtung von rechts nach links: Es gelte $P(o, x, l, y) \wedge In(o, a_k)$. Nun ist $a_k \in B$, also gilt wegen A_7 $A(x, m, y)$. Richtung von links nach rechts: Es gelte $A(x, m, l)$. Wegen $c_k \in M$ ist Axiom A_8 anwendbar, und es folgt die Behauptung[5]. □

Als Korollar zu obigem Beweis gilt: $Isa(c_s, c)$ für alle $c \in M$ (Axiom A_{16}). Lemma 4-3 ist eine Konsequenz aus der strikten Variante multipler Generalisierung, die durch die Axiome $A_8, A_{14}, A_{15}, A_{16}$ festgelegt ist. Diese Art der multiplen Vererbung schließt Mehrdeutigkeiten aus. Es gibt immer ein „kleinstes" Attribut a_s, das die Bedeutung von $A(x, m, y)$ voll trägt, d.h. die Zugehörigkeit zu allen anderen Attributklassen namens m ist eine Konsequenz aus der Zugehörigkeit zu a_s.

Die oben definierten Begriffe der Klassifikation (Prädikat *In*) und Spezialisierung (Prädikat *Isa*) sind intuitiv mit den Element- und Teilmengenbeziehungen der Mengenlehre verwandt. Von dort ist das Phänomen der **Paradoxien** bekannt. Ein Beispiel ist die Russel'sche Menge

$$M = \{x |\ x \notin x\},$$

für die sowohl die Annahme $M \in M$ als auch ihre Negation $M \notin M$ zu einem Widerspruch führen. Können solche unerwünschen Fälle auch mit O-Telos auftreten? In der Tat ist dies nicht der Fall. Die zugrundegelegte Semantik der relationalen und deduktiven Datenbanken ist von vorneherein so eingeschränkt, daß nur über endliche Bereiche quantifiziert wird

[5] Mit Blick auf die Erweiterung der Objektbanktheorie um deduktive Regeln stellt Axiom A_8 scheinbar eine Einschränkung der Allgemeinheit dar: wenn $In(x, c_k)$ und $A(x, m, y)$ gilt, so muß es auch ein abgespeichertes Attribut $P(o, x, l, y)$ mit Kategorie m geben. Die Folgerung von $A(x, m, y)$ aus einer deduktiven Regel wäre also nur dann zugelassen, wenn $A(x, m, y)$ bereits mit A_7 aus der extensionalen Objektbank ableitbar ist. Dieses Problem wird bei der Einführung deduktiver Regeln so gelöst werden, daß solche Folgerungsprädikate formal verschieden zu $A(x, m, y)$ geschrieben werden.

(Bereichsabschlußaxiom). Syntaktisch wird dieses Axiom durch die Beschränkung auf Bereichsformeln (Def. 2-2) sichergestellt. Zudem müssen alle deduktiven Regeln stratifizierbar sein. Mit diesen Voraussetzungen ist die *Endlichkeit und eindeutige Berechenbarkeit* der Extensionen aller Prädikate durch die Fixpunktsemantik der perfekten Modelle (Kap. 2) garantiert. O-Telos erfüllt die Voraussetzungen, da alle als deduktive Regeln interpretierten Axiome – also insbesondere das Axiom A_{12} zur Vererbung der Klassenzugehörigkeit – wegen der Abwesenheit negativer Literale trivialerweise stratifizierbar sind. Hinzukommende deduktive Regeln mit Folgerungsprädikaten *In* und *Isa* sind problemlos, da auch sie bereichsbeschränkt und stratifizierbar sein müssen (Def. 2-3). Eine vorsichtigere Haltung wird in der älteren Telos-Variante [CCI87,KMSB89,MBJK90] eingenommen. Dort gibt es die Systemklassen *Token* (Objekte, die keine Instanzen haben können), *SimpleClass* (Objekte, deren Instanzen auch Instanz von Token sind), *MetaClass* (Objekte, deren Instanzen auch Instanz von *SimpleClass* sind) usw., die zur Einstufung (fast) aller Objekte in eine Klassifikationshierarchie dienen. Jene Objekte wie z.B. *Object*, die nicht in diese Hierarchie eingeordnet werden können, werden der Systemklasse *OmegaClass* zugewiesen. Es sei hier darauf hingewiesen, daß ohne die Stratifikationsannahme selbst diese Einordnung keine Paradoxien verhindert. Ein Beispiel ist in die folgende Objektbank mit einer deduktiven Regel. Das Objekt *OmegaClass* sei dabei durch den Identifikator $\#OmCl$ referenziert.

OB:

$$P(\#isaO, \#OmCl, isa, \#Indiv) \ ...$$

R:

$$\forall x \ In(x, \#Attr) \wedge \neg In(x, \#Indiv) \Rightarrow In(x, \#OmCl)$$

Ein Attributobjekt x, das kein Individuum ist, gehört nach der Regel zur Klasse *OmegaClass*. Wegen der Spezialisierungsbeziehung und dem Axiom A_{12} ist dasselbe Objekt x aber Instanz von *Individual*, was ein Widerspruch ist. Das Paradoxieproblem wird also nicht durch die Einführung von Klassifikationsebenen gelöst. Wenn man die Regel sowie das Axiom A_{12} genauer betrachtet, so erkennt man eine Verletzung der Stratifikationsbedingung für das Prädikat $In(x, \#OmCl)$.

Die Stratifikationsannahme und das Bereichsabschlußaxiom garantieren eine formale Paradoxiefreiheit. Dennoch sind Objektbanken aufschreibbar, die der intuitiven Interpretation von Klassen als Mengen von Instanzen scheinbar zuwiderlaufen. Beispiele sind in Abbildung 4-8 aufgeführt.

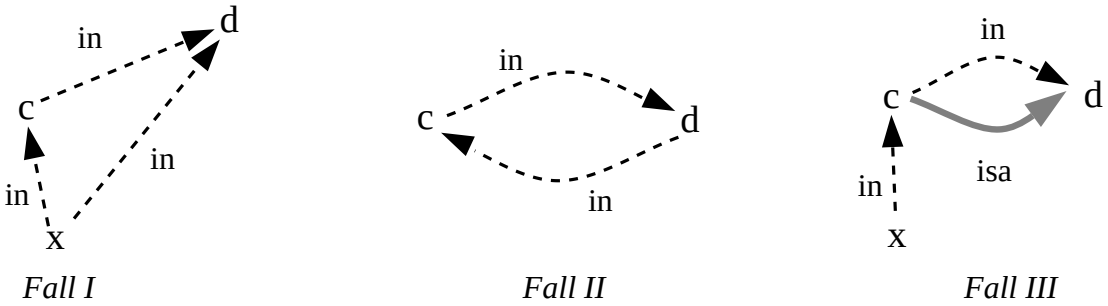

Abb. 4-8: Beispiele nicht-paradoxer Objektbanken

In Fall I gilt:

$$ext(In) \supseteq \{In(x,c), In(c,d), In(x,d)\}$$

Das Ungewöhnliche an der Konstruktion ist, daß x sowohl in c als auch in einer Klasse d von c ist. Es besteht aber formal kein Grund zum Verbot. Tatsächlich ist die Klasse *Object* ein Beispiel dafür, daß solche Beziehungen Sinn machen. Fall II zeigt eine zirkuläre Klassifikationsbeziehung. Hier ist

$$ext(In) \supseteq \{In(c,d), In(d,c)\}$$

Die Forderung der Endlichkeit der Extensionen wird nicht verletzt. Ein Beispiel für die Sinnhaftigkeit solcher Zyklus sind die Systemklassen *Object* und *Individual*. Der letzte Fall ist ähnlich zu dem ersten Fall. Auch hier gilt:

$$ext(In) \supseteq \{In(x,c), In(c,d), In(x,d)\}$$

Offensichtlich sind die Konstruktionen in der Art der Beispiele von Abbildung 4-8 typisch für Systemklassen, mit denen die Eigenschaften des Datenmodells beschrieben werden. Man kann sie durch geeignete Integritätsbedingungen auf genau diese Klassen beschränken – wie im ursprünglichen Telos durch die Systemklassen *Token* etc. geschehen. Diese Arbeit hebt jedoch auf die generischen Fähigkeiten von O-Telos ab, d.h. seine Fähigkeit spezialisierte Datenmodelle durch Metaklassen zu beschreiben (siehe auch Kap. 7). Daher verzichten wir auf eine solche Beschränkung. In der Implementierung des Objektmodells (Kap. 9) sind beide Varianten verfügbar.

Alle Axiome von O-Telos sind bereichsbeschränkt. Dadurch ist es möglich, die Axiome als Regeln und Integritätsbedingungen einer speziellen deduktiven Datenbank (OB,R,IC) anzusehen. R enthalte z.B. die Axiome $A_5, A_6, A_7, A_9, A_{10}, A_{12}, A_{17}, A_{19}, A_{21}, A_{23}$ und A_{25}. Die Axiome A_{28} bis A_{32} können als vordefinierte Objekte von OB realisiert werden.

Die restlichen Axiome sind dann Integritätsbedingungen. Die extensionale Datenbank besteht also nur aus einer einzigen Relation P. Der Umgang mit den Axiomen ist dann wie folgt eingeschränkt: Die Axiome, die als Regeln dargestellt sind, werden nur zur Ableitung von Fakten aus dem Bedingungsteil verwendet (siehe Kap. 2). In diesem Fall werden Lösungen für die Prädikate In, Isa, A aus der Objektbank abgeleitet. Hinzu treten die Vererbung der Klassenmitgliedschaft zu Oberklassen und die Zugehörigkeit zu den vier Systemklassen. Die als Integritätsbedingungen benutzten Formeln müssen gemäß Definition 2-3 immer ableitbar sein, d.h. sie werden wie Anfragen an die Datenbank behandelt, die das Ergebnis *true* liefern. Es sollte bemerkt werden, daß das Prädikat P niemals auf der rechten Seite einer Implikation auftritt. Folglich werden keine Objekte abgeleitet.

4.4. Übertragung relational-deduktiver Techniken

Logische Formeln in deduktiven Datenbanken dienen der Ableitung von Daten aus der extensionalen Datenbank. Bei Anfragen werden die Antworten abgeleitet, bei Integritätsbedingungen deren Wahrheitswert, bei deduktiven Regeln die Extension des Folgerungsprädikats und bei Sichten deren Inhalt. In O-Telos besteht die extensionale Datenbank aus einer einzigen vierstelligen Relation von Objekten. Es ist eine Grundsatzentscheidung, ob abgeleitete Daten auch Objekte sein dürfen, insbesondere ob sie eine Objektidentität haben können. In der vorliegenden Arbeit fällt folgende Entscheidung:

Abgeleitete Information hat keine Objektidentität.

Als Begründung wird auf die Argumente in [ULLM89,ULLM91] verwiesen. Das einfache Beispiel der transitiven Hülle der Pfadrelation in einem Graphen mit Zykeln zeigt, daß es unendlich viele Pfade gibt. In stratifizierbaren deduktiven Datenbanken halten alle „Programme“, d.h. Regelmengen mit einem Ziel, nach endlich vielen Schritten. Gesteht man jedoch jedem berechneten Pfad Objektidentität zu, so kann ein Programm zur Berechnung aller Pfade für Graphen mit Zykeln nicht halten. Will man die wesentliche Eigenschaft der Terminierung erhalten, so muß man an einer bestimmten Stelle mit dem Prinzip brechen, daß jede Information ein Objekt ist. In dieser Arbeit wird diese Grenze exakt durch die extensionale Objektbank (Def. 4-1) beschrieben.

Eine erste Konsequenz ist, daß das Prädikat $P(o, x, l, y)$ nicht als Folgerungprädikat einer deduktiven Regel auftreten darf. Zu beachten ist, daß diese Forderung auch durch die Axiome von O-Telos eingehalten wird. Eine weitere Folgerung ist, daß alle Objektidentifikatoren, die in abgeleiteten Daten auftauchen, bereits in der extensionalen Objektbank vorkommen. Bei der umgekehrten Entscheidung besteht das Hauptproblem in der geeigneten Erfindung von Objektidentifikatoren für die abgeleitete Information. Nach [ULLM89]

ist jedoch der Objektidentifikator eher als Adresse eines Datums anzusehen, mithin der Implementierungsebene zuzuschreiben. Die Bildung der Adresse eines neuen Datums ist willkürlich, mithin logikfern. In ILOG [HY90] wird der Identifikator abgeleiteter Objekte durch eine Skolemfunktion $f(a_1, ..., a_k)$ beschrieben, wobei $a_1, ..., a_k$ die Identifikatoren der Objekte sind, von denen das abgeleitete Objekt abhängt. Diese Skolemfunktionen müssen allerdings immer noch auf konkrete Identifikatoren abgebildet werden. Wie [AK89,HY90] zeigen, gehen Anfragesprachen mit Erfindung von Identifikatoren über die Ausdrucksfähigkeit von deduktiven Regeln ohne diese Eigenschaft hinaus. Wenn die Erfindung neuer Objekte nicht besonders eingeschränkt wird, so wird die Regelsprache Turing-vollständig, da ein beliebig zugreifbarer und unendlich großer Speicher simulierbar ist. Ein solcher Weg wird beispielsweise in [KMS90] mit der Regelsprache RDL1 vorgeschlagen. Wir werden allerdings diese Verallgemeinerungen deduktiver Regeln nicht weiter betrachten, da sie entweder die Entscheidbarkeit kompromittieren oder aber die Optimierbarkeit negativ beeinflussen.

Wie oben bereits für die Axiome erwähnt, ist der Begriff „deduktive Datenbank“ auf triviale Weise auf Objektbanken in O-Telos anwendbar, da O-Telos der Grenzfall einer relationalen Datenbank mit genau einer Relation ist:

DEFINITION 4-2

Sei OB eine extensionale Objektbank, R eine Menge deduktiver Regeln, und IC eine Menge von Integritätsbedingungen mit folgenden Eigenschaften:

1) Die zulässigen Prädikate der Formeln in IC sind P, In, Isa, A,„=“ und „$\preceq$“.
2) Als Folgerungsprädikate deduktiver Regeln sind P und Isa nicht zugelassen. Wenn eine Regel ein von P, In, Isa, A,„=“ und „$\preceq$“ verschiedenes Prädikat im Bedingungsteil enthält, so sind auch In und A als Folgerungsprädikate dieser Regel verboten.

Dann heißt (OB,R,IC) eine **äußere deduktive Objektbank**. Wir schreiben statt $L \in cons(\mathrm{OB} \cup \mathrm{AX}_a \cup \mathrm{R})$ auch kurz $\mathrm{OB} \cup \mathrm{R} \vdash_a L$ mit

$$\mathrm{AX}_a = \{A_5, A_6, A_7, A_9, A_{10}, A_{12}, A_{17}, A_{19}, A_{21}, A_{23}\}.$$

Bei [KLW90] werden die Axiome des objektorientierten Datenmodells in den Regeln des Kalküls kodiert. Im Falle von O-Telos ist dies nicht nötig, da die Prädikatenlogik erster Stufe nicht verlassen wird. Das Verbot von P als Folgerungsprädikat ist oben begründet. Grundsätzlich könnte $Isa(c, d)$ auch durch deduktive Regeln ableitbar sein. Aus Gründen der Überschaubarkeit ist es jedoch sinnvoll, die Extension von $Isa(c, d)$ allein auf den Axiomen A_9 bis A_{11} beruhen zu lassen. Diese Beschränkung ist aus der Rolle von $Isa(c, d)$ als Aussage der Spezialisierung von Klassen her begründet. Würde man $Isa(c, d)$

als Folgerungsprädikat zulassen, so hinge $Isa(c,d)$ von der ganzen Objektbank ab. Von den Grundprädikaten sind also nur Klassenmitgliedschaft $In(x,c)$ und Attributbeziehungen $A(x,m,l)$ deduzierbar. Die Einschränkung der zugelassenen Prädikate für deduktive Regeln ist neben den oben genannten Gründen auch auf die Abgrenzung der Anfrageregeln in der folgenden Definition zurückzuführen.

DEFINITION 4-3

Sei (OB,R,IC) eine äußere deduktive Objektbank und $\psi \in$ R eine deduktive Regel, deren Folgerungsprädikat von A und In verschieden ist. Dann heißt ψ auch **Anfrageregel**.

Anfrageregeln sind also gewöhnliche deduktive Regeln, die aber per Definition 4-2 keinen Einfluß auf die Integrität der deduktiven Objektbank haben können. Sie dienen lediglich der Ableitung von Information.

Definition 4-2 ist eine korrekte Übertragung des Begriffs „deduktive Datenbank“ auf Objektbanken. Die Begriffe „integre Objektbank“ und „Transaktion“ werden unverändert übernommen. Aus folgenden Gründen ist sie jedoch nicht direkt für eine Implementierung tragfähig:

- Die Verfahren zur Integritätskontrolle sowie zur Auswertung deduktiver Regeln und Anfragen sind umso effizienter je mehr Relationen eine Datenbank enthält. Relationen, deren Prädikate nicht unmittelbar oder mittelbar über Regeln in einer Formel vorkommen, tragen zum Ergebnis nicht bei (*domain independence*, siehe z.B. [BRY88]). Da bei O-Telos nur eine extensionale Relation existiert, auf der Transaktionen stattfinden, wäre bei einer direkten Übertragung der deduktiven Verfahren aus Kapitel 2 bei jeder Formel die ganze Objektbank im Suchraum. Insbesondere wäre jede Integritätsbedingung (außer den trivialen wie z.B. *true*) bei jeder beliebigen Einfüge- bzw. Löschoperation zu testen.
- Nur zwei deduzierbare Literale bedeuten wegen der Stratifikationsforderung eine erhebliche Einschränkung. Eine Regel $\forall\, x,y \ldots \neg A(x,left,y) \Rightarrow A(x,right,y)$ wäre verboten. Tatsächlich wäre der Umgang mit Negation auf wenige Fälle beschränkt. Das Konzept der lokalen Stratifikation [CGT90] würde zwar solche Fälle erlauben, kann aber erst zum Zeitpunkt der Regelauswertung getestet werden und wird deshalb als deutlich ineffizienter angesehen.

Der Ursprung dieser Nachteile ist die scheinbar geringe Struktur, die eine Objektbank in O-Telos in sich birgt. Allerdings bedeutet diese Uniformität auch die prinzipielle Gleichbehandlung jeglicher Daten in der Objektbank. Im relationalen Modell wird der

darzustellende Weltausschnitt mit dem Relationenschema angegeben. In O-Telos ist dieses Schema fest. Stattdessen geschieht die Modellierung des Weltausschnitts durch die Angabe von Klassen, d.h. Objekten, die Instanzen haben können. Es zeigen sich zwei Vorteile:

- ▷ Transaktionen in O-Telos beinhalten Änderungen auf jeglicher Art von Objekt: Individualobjekte, Attribute, Klassifikationsobjekte und Spezialisierungsobjekte. Insbesondere können auch Klassen in einer Transaktion manipuliert werden.
- ▷ Eine Transaktion kann aus der Einfügung oder Löschung eines einzelnen Objektes $P(o,x,l,y)$ bestehen. Damit unterstützt das Datenmodell die Manipulation auch einzelner Attribute eines Objektes. Im relationalen Datenmodell ist die kleinste Einheit ein Tupel $R(x_1,...,x_k)$.

Die folgende Definition einer deduktiven Objektbank eliminiert die oben erwähnten Nachteile, indem sie für jedes Objekt zwei Prädikate zur Verfügung stellt.

DEFINITION 4-4

Sei OB eine extensionale Objektbank, die für jedes Objekt $P(p,c,m,d) \in$ OB folgende Axiome erfüllt:

$$\forall\ o\ \mathit{In}(o,p) \Rightarrow \mathit{In}.p(o) \qquad (A_{34})$$

$$\forall\ o,x,l,y\ P(o,x,l,y) \wedge \mathit{In}(o,p) \Rightarrow A.p(x,y) \qquad (A_{35})$$

Ferner sei IC eine Menge von Integritätsbedingungen, die nur die Prädikate $P(o,x,l,y)$, $\mathit{Isa}(c,d)$ sowie Prädikate der Form $\mathit{In}.p(o)$ und $A.p(x,y)$ enthalten, wobei p der Identifikator eines Objektes von OB ist. Die Folgerungsprädikate der Regeln dürfen nicht von der Form $\mathit{In}(x,c)$ bzw. $A(x,m,y)$ sein. Dann heißt das Tripel (OB,R,IC) **innere deduktive Objektbank**. Statt $L \in \mathit{cons}(\mathrm{OB} \cup \mathrm{AX}_i \cup \mathrm{R})$ mit $\mathrm{AX}_i = \mathrm{AX}_a \cup \{A_{34}, A_{35}\}$ schreiben wir kurz $\mathrm{OB} \cup \mathrm{R} \vdash_i L$.

Bei einer inneren deduktiven Objektbank kann jedem Literalvorkommen außer $\mathit{Isa}(c,d)$ und den Folgerungsprädikaten von Anfrageregeln ein Objekt p der Objektbank OB zugeordnet werden, dessen Instanziierung bei Abwesenheit deduktiver Regeln notwendig für die Wahrheit des Literals ist: bei $P(o,x,l,y)$ ist dies $\#\mathit{Obj}$ (siehe Axiom A_{18}), und bei Literalen mit $\mathit{In}.p(o)$ bzw. $A.p(x,y)$ ist dies das Objekt mit Identifikator p. Das Prädikat $\mathit{Isa}(c,d)$ fällt unwesentlich aus dem Rahmen, da hier neben $\#\mathit{Spec}$ (A_{24}, A_6) auch die Instanziierung von $\#\mathit{Obj}$ (A_9) zu Lösungen führt. Im folgenden wird eine partielle Transformation von einer äußeren zu einer inneren deduktiven Objektbank definiert. Dabei werden gewisse „unerwünschte“ Formeln ausgeschlossen.

Wir betrachten unter leichter Beschränkung der Allgemeinheit nur solche äußeren deduktiven Objektbanken (OB,R,IC), bei denen die Bereichsprädikate (Def. 2-2) von der

Form $In(x, c)$ sind. Das Binden von Variablen an Klassen ist üblich für objektorientierte Anfragesprachen (z.B. [BCD89]). In [MBJK90] werden dazu die Abkürzungen $\forall x/C\ \psi$ bzw. $\exists x/C\ \psi$ benutzt. Es sei darauf hingewiesen, daß nach Lemma 4-2 in einer externen Syntax für Formeln anstatt der Identifikatoren auch Ausdrücke über Objektnamen verwendet werden können.

DEFINITION 4-5

Eine äußere deduktive Objektbank (OB,R,IC) heißt **klassenbeschränkt**, falls jede quantifizierte Teilformel eines $\varphi \in \mathrm{R} \cup \mathrm{IC}$ von der Form

$$\forall\ x_1, ..., x_n\ \neg In(x_1, c_1) \vee ... \vee \neg In(x_n, c_n) \vee \psi \quad \text{bzw.}$$
$$\exists\ x_1, ..., x_n\ In(x_1, c_1) \wedge ... \wedge In(x_n, c_n) \wedge \psi$$

ist, wobei $c_i \in \mathrm{OID(OB)}, 1 \leq i \leq n$, Konstanten sind. Die c_i heißen auch **Sorten** der x_i und die Prädikate $In(x, c_i)$ **Sortenprädikate**.

Die Einschränkung der Allgemeinheit besteht darin, daß nicht über Objektnamen quantifiziert werden darf, d.h. in Literalvorkommen $P(o, x, l, y)$ bzw. $A(x, m, y)$ dürfen l, m nur als Konstanten auftreten. Die Allgemeingültigkeit des Ansatzes wird durch die Einschränkung nicht berührt, da die Attributzugriffe $A(x, m, y)$ den Ausdrücken $x.m{=}Y$ in objektorientiertierten Anfragesprachen [MS87,MANO89a,BCD89,KM90a] entsprechen, wobei m einen festen Wert hat. Das Prädikat $P(o, x, l, y)$ ist spezifisch für O-Telos. Mit dieser Einschränkung kann jede Formel auf die geforderte Form gebracht werden, da $In(x, \#Obj)$ nach Lemma 4-1 für alle Objektidentifikatoren x ableitbar ist. Die totalen Funktionen in Abbildung 4-9 dienen der Beschreibung der gesuchten Transformation.

Mit $\mathcal{P}(\mathrm{ID})$ wird die Menge der endlichen Teilmengen von ID bezeichnet. Da $\mathrm{OB} \cup \mathrm{AX}_a \vdash L$ für Grundatome L entscheidbar ist (Kap. 2), sind alle drei Funktionen durch terminierende Programme berechenbar. Aufgrund von Axiom A_{11} muß $min(s)$ eindeutig sein. Die Funktion $classes(v, \varphi)$ liefert entweder $\perp$ oder eine nicht-leere Menge, da für jeden Objektidentifikator v das Prädikat $In(v, \#Obj)$ wahr ist (siehe Lemma 4-1) bzw. $Isa(d, d)$ aufgrund von Axiom A_9 gilt. Definition 4-6 beschreibt die Transformation einer äußeren in die sogenannte zugehörige innere deduktive Objektbank. Dabei werden solche Formeln ausgeschlossen, die entweder Konstanten enthalten, die in der extensionalen Objektbank nicht vorkommen, oder deren Prädikate nicht eindeutig einer Klasse zuordnenbar sind. Letztere Eigenschaft kann man mit einem Typfehler in Argumenten einer Operation konventioneller Programmiersprachen vergleichen: eine Operation `round(REAL):INTEGER` darf z.B. nicht auf eine Zeichenkette "nogood" angewandt werden. Analog macht beispielsweise ein Prädikat $A(x, allergy, y)$ nur für solche Argumente x, y Sinn, die Instanz der beim Attribut mit Namen *allergy* genannten Klassen sind, also *Patient* und *Agent*.

$$min : \mathcal{P}(\mathrm{ID}) \to \mathrm{ID} \cup \{\bot\}$$
$$sort : (\mathrm{VAR} \cup \mathrm{ID}) \times \mathrm{WFF} \to \mathrm{ID} \cup \{\bot\}$$
$$classes : (\mathrm{VAR} \cup \mathrm{ID}) \times \mathrm{WFF} \to \mathcal{P}(\mathrm{ID}) \cup \{\bot\}$$

$$min(\{c_1, \ldots, c_n\}) = \begin{cases} c_i, & Isa(c_i, c_j) \in cons(\mathrm{OB} \cup \mathrm{AX}_a) \text{ für alle } 1 \leq j \leq n \\ \bot, & \text{sonst} \end{cases}$$

$$sort(x, \varphi) = \begin{cases} c, & x \in \mathrm{VAR},\ c \text{ Sorte von } x \text{ in } \varphi, c \in \mathrm{OID(OB)} \\ x, & x \in \mathrm{OID(OB)} \\ \bot, & \text{sonst} \end{cases}$$

$$classes(v, \varphi) = \begin{cases} \{c |\ In(v, c) \in cons(\mathrm{OB} \cup \mathrm{AX}_a)\}, & \text{falls } v \in \mathrm{OID(OB)} \\ \{c |\ Isa(d, c) \in cons(\mathrm{OB} \cup \mathrm{AX}_a)\}, & \text{falls } v \in \mathrm{VAR}, d = sort(v, \varphi) \neq \bot \\ \bot, & \text{sonst} \end{cases}$$

Abb. 4-9: Zuordnung von Variablen zu Objekten

DEFINITION 4-6

Sei $(\mathrm{OB}, \mathrm{R}_a, \mathrm{IC}_a)$ eine klassenbeschränkte äußere deduktive Objektbank, die für jedes $\varphi_a \in \mathrm{R}_a \cup \mathrm{IC}_a$ folgende Bedingungen erfüllt:

a) Für jede Konstante v von φ_a gilt $v \in \mathrm{OID(OB)}$.
b) In jedem Literalvorkommen $In(x, c)$ von φ_a ist c eine Konstante.
c) Zu jedem Vorkommen von $A(x, m, y)$ gibt es ein $c \in classes(x, \varphi_a)$ mit
   - $P(p, c, m, d) \in \mathrm{OB}$ für $p, d \in \mathrm{OID(OB)}$.
   - $In(y, d) \in cons(\mathrm{OB} \cup \mathrm{AX}_a)$, falls y Konstante, bzw. $Isa(sort(y), d) \in cons(\mathrm{OB} \cup \mathrm{AX}_a)$, falls $y \in \mathrm{VAR}$.

Dann heißt die folgende deduktive Objektbank $(\mathrm{OB}, \mathrm{R}_i, \mathrm{IC}_i)$ zu $(\mathrm{OB}, \mathrm{R}_a, \mathrm{IC}_a)$ **zugehörig**.

i) Zu jedem φ_a aus R_a (IC_a) wird ein φ_i in R_i (IC_i) wie folgt gebildet:
   - Jedes Literalvorkommen $In(x, c)$ von φ_a wird in φ_i durch $In.c(x)$ ersetzt.
   - Jedes Vorkommen von $A(x, m, y)$ wird durch $A.p_o(x, y)$ ersetzt, wobei $p_o = min(\{p |\ P(p, c, m, d) \in \mathrm{OB}, c \in classes(x, \varphi_a)\})$.
   - Alle anderen Literalvorkommen bleiben unverändert.
ii) Zu jedem Spezialisierungsobjekt $P(o, q, isa, p)$ werden zwei deduktive Regeln
   $\forall\ x\ In.q(x) \Rightarrow In.p(x)$
   $\forall\ x, y\ A.q(x, y) \Rightarrow A.p(x, y)$
   zu R_i hinzugefügt.

Wegen Annahme a) sind die Verwendungen der Funktionen *classes* und *sort* wohldefiniert. Die Verwendung von *min* in Punkt i) ist wohldefiniert, da die Axiome A_{14}, A_{16} ein

eindeutig bestimmtes Minimum garantieren. Gemäß Definition 4-4 ist $(\mathrm{OB}, \mathrm{R}_i, \mathrm{IC}_i)$ eine innere deduktive Objektbank. Die zusätzlichen deduktiven Regeln in Punkt ii) übertragen die Ableitbarkeit der Klassenzugehörigkeit zu Oberklassen (Axiom A_{12}) und des Attributprädikats A (Axiom A_7) auf die neuen Prädikate der inneren deduktiven Objektbank. Man beachte, daß die Umformungen der Literalvorkommen von der extensionalen Objektbank OB abhängt, nicht aber von ihren deduktiven Regeln.

Auf einen Vergleich der Extensionen der äußeren und der zugehörigen inneren deduktiven Objektbank wird verzichtet. Die äußere deduktiven Objektbank ist wie oben erwähnt durch eine extreme Prädikatsarmut gekennzeichnet, die das Vorkommen von Negation in deduktiven Regeln de facto ausschließt. Die Semantik stützt sich also einzig auf die innere deduktive Objektbank, in der die Klassifikations- und Attributprädikate den Klassen der Objektbank eindeutig zugeordnet sind. Dennoch ist die äußere deduktive Objektbank als Zwischenstufe notwendig, da für es für den Entwerfer einer Objektbank nicht zumutbar ist, logische Formeln mit Objektidentifikatoren statt Objektnamen auszudrücken.

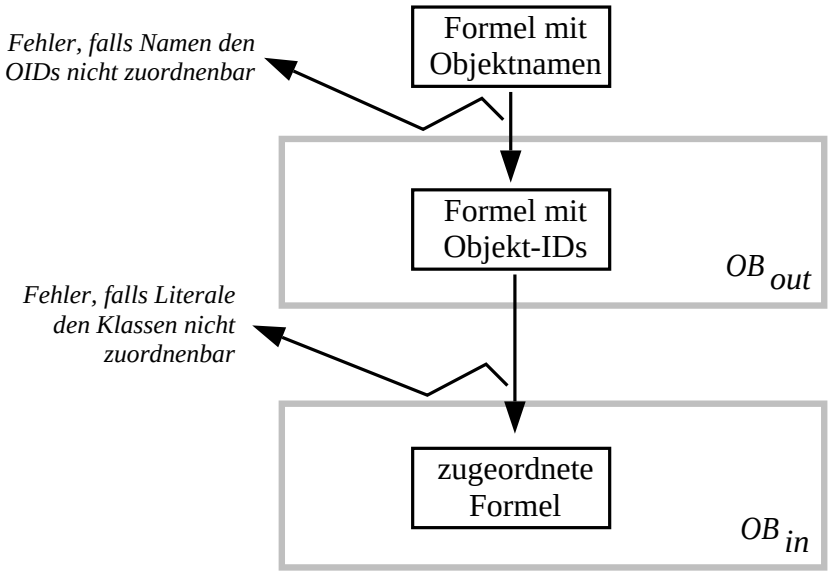

Abb. 4-10: Schrittweise Zuordnung einer Formel zu einer Objektbank

Abbildung 4-10 faßt den Weg einer Formel in eine innere deduktive Objektbank zusammen. Am Anfang steht eine Formel, die anstatt der Objektidentifikatoren deren Namen enthält, z.B. *Patient* statt *#Pat*. Im ersten Schritt werden die Namen gemäß Lemma 4-2 durch die zugehörigen Objektidentifikatoren ersetzt. Falls die Zuordnung nicht gelingt, d.h. die bezeichneten Objekte nicht existieren, muß die Formel zurückgewiesen

werden. Ansonsten kann sie Teil der Formeln der äußeren deduktiven Objektbank werden. Im zweiten Schritt werden die Literale der Formel mit den Anweisungen aus Definition 4-6 den Klassen der Objektbank zugeordnet. Auch hier kann es zu Fehlern kommen, wenn für ein Literalvorkommen keine Klasse gefunden wird. Erst danach wird Stratifikation und Integrität für die geänderte innere deduktive Objektbank geprüft.

4.5. Beispiel

Das Schema der Patientendatenbank aus Kapitel 2 wurde bereits in Abbildung 4-1 in O-Telos übertragen. Die folgenden Formeln stellen die deduktive Regel und die Integritätsbedingung des Beispiels in einer Syntax dar, die nur Objektnamen benutzt (vgl. Abb. 4-10).

$$\forall\ d/Drug,\ s/Symptom,\ a/Agent \qquad (10)$$
$$A(d, component, a) \wedge A(a, effects, s) \Rightarrow A(d, against, s)$$

$$\forall\ p/Patient,\ d/Drug$$
$$A(p, takes, d) \Rightarrow (\exists\ s/Symptom\ A(p, suffers, s) \wedge A(d, against, s)) \wedge \qquad (11)$$
$$(\forall\ a/Agent\ A(d, component, a) \Rightarrow \neg A(p, allergy, a))$$

Die Namen *Drug*, *Symptom* usw. müssen mit Hilfe der Funktionen aus Abb. 4-5 in Objektidentifikatoren umformbar sein. Ansonsten würde die Formel sich auf Objekte beziehen, die nicht Teil der Objektbank sind. Mit der Objektbank aus Abb. 4-1 erhält man folgende klassenbeschränkte Formeln in den Literalen der äußeren deduktiven Objektbank.

R_a:

$$\forall\ d, s, a\ In(d, \#Drug) \wedge In(s, \#Sympt) \wedge In(a, \#Ag) \wedge \qquad (12)$$
$$A(d, component, a) \wedge A(a, effects, s) \Rightarrow A(d, against, s)$$

IC_a:

$$\forall\ p, d\ In(p, \#Pat) \wedge In(d, \#Drug) \wedge A(p, takes, d) \Rightarrow$$
$$(\exists\ s\ In(s, \#Sympt) \wedge\ A(p, suffers, s) \wedge A(d, against, s)) \wedge \qquad (13)$$
$$(\forall\ a\ In(a, \#Ag) \wedge\ A(d, component, a) \Rightarrow \neg A(p, allergy, a))$$

Die zugehörige innere deduktive Objektbank (OB, R_i, IC_i) (vgl. Def. 4-6) wird durch Zuordnung der Literale zu (Klassen-)objekten der extensionalen Objektbank OB gewonnen. Für die Klassifikationsprädikate $In(x, c)$ mit konstanter zweiter Komponente ist dies

trivial. Auch für die Literale $A(x, m, y)$ ist die Zuordnung möglich, da in OB jeweils nur ein Attribut mit dem geforderten Namen existiert.

R_i:

$$\begin{aligned} &\forall\ d, s, a\ In.\#Drug(d) \wedge In.\#Sympt(s) \wedge In.\#Ag(a) \wedge \\ &\quad A.\#comp(d, a) \wedge A.\#effects(a, s) \Rightarrow A\#against(d, s) \\ &\forall\ x\ In.\#Pat(x) \Rightarrow In.\#Pers(x) \end{aligned} \tag{14}$$

IC_i:

$$\begin{aligned} &\forall\ p, d\ In.\#Pat(p) \wedge In.\#Drug(d) \wedge A.\#takes(p, d) \Rightarrow \\ &\quad (\exists\ s\ In.\#Sympt(s) \wedge\ A.\#suff(p, s) \wedge A.\#against(d, s)) \wedge \\ &\quad (\forall\ a\ In.\#Ag(a) \wedge\ A\#comp(d, a) \Rightarrow \neg A.\#allergy(p, a)) \end{aligned} \tag{15}$$

Durch die Vereinfachungsmethode (Abb. 2-1) wird für jedes Literal der Integritätsbedingung ein Trigger generiert (siehe Abbildung 4-11). Die Umformung der Formeln in ihre Normalform (2) bzw. (3) aus Definition 2-2 fehlt aus Gründen der Lesbarkeit.

Der Vergleich zum relationalen Beispiel (Abb. 2-2) zeigt eine erhöhte Anzahl von Triggern, nämlich neun anstatt fünf. Dies rührt von der Trennung der Klassenzugehörigkeit $In.c(x)$ eines Objektes x von seinen Attributen $A.p(x, y)$ her. Der Effekt ist ein Effizienznachteil gegenüber der relationalen Version des Beispiels. Insbesondere Eintragungen von Instanzen der Klassen $\#Pat$, $\#Drug$ usw. eliminieren nur jeweils höchstens einen Allquantor. Andererseits ist zu bemerken, daß nicht mehr über irrelevante Attribute (z.B. das Patientenalter) quantifiziert wird. Eine Änderung dieses Attributs führt also nicht zu einem Test der Integritätsbedingung.

4.6. Diskussion

Der hier erarbeitete Begriff der deduktiven Objektbank zeichnet sich durch zwei Haupteigenschaften aus. Er beeinhaltet die strukturellen Konzepte objektorientierter Datenbanken, und er ist ein Spezialfall der deduktiven Datenbanken. Durch diese Konstruktion können **alle** Algorithmen aus dem Bereich der deduktiven Datenbanken praktisch ohne Veränderung übernommen werden. Das Demonstrationsbeispiel dieser Arbeit ist die Integritätsprüfungsmethode. Das gleiche Argument zählt aber auch für weitere Algorithmen, z.B. die Methode *MagicSet* [BMSU86] zur Optimierung rekursiver DATALOG-Programme, die Nutzung abgespeicherter Sichten zur Anfrageoptimierung

ON Insert($In.\#Pat(p')$) CHECK $\forall\ d\ In.\#Drug(d) \wedge A.\#takes(p', d) \Rightarrow$
$(\exists\ s\ In.\#Sympt(s) \wedge A.\#suff(p', s) \wedge A.\#against(d, s)) \wedge$
$(\forall\ a\ In.\#Ag(a) \wedge A.\#comp(d, a) \Rightarrow \neg A.\#allergy(p', a))$

ON Insert($In.\#Drug(d')$) CHECK $\forall\ p\ In.\#Pat(p) \wedge A.\#takes(p, d') \Rightarrow$
$(\exists\ s\ In.\#Sympt(s) \wedge A.\#suff(p, s) \wedge A.\#against(d', s)) \wedge$
$(\forall\ a\ In.\#Ag(a) \wedge A.\#comp(d', a) \Rightarrow \neg A.\#allergy(p, a))$

ON Insert($A.\#takes(p', d')$) CHECK $In.\#Pat(p) \wedge In.\#Drug(d') \Rightarrow$
$(\exists\ s\ In.\#Sympt(s) \wedge A.\#suff(p', s) \wedge A.\#against(d', s)) \wedge$
$(\forall\ a\ In.\#Ag(a) \wedge A.\#comp(d', a) \Rightarrow \neg A.\#allergy(p', a))$

ON Delete($In.\#Sympt(s')$) CHECK $\forall\ p, d\ In.\#Pat(p) \wedge In.\#Drug(d) \wedge$
$A.\#takes(p, d') \Rightarrow (\exists\ s\ In.\#Sympt(s) \wedge A.\#suff(p, s) \wedge A.\#against(d', s)) \wedge$
$(\forall\ a\ In.\#Ag(a) \wedge A.\#comp(d', a) \Rightarrow \neg A.\#allergy(p, a))$

ON Delete($A.\#suff(p', s')$) CHECK $\forall\ d\ In.\#Pat(p') \wedge In.\#Drug(d) \wedge$
$A.\#takes(p', d) \Rightarrow (\exists\ s\ In.\#Sympt(s) \wedge A.\#suff(p', s) \wedge A.\#against(d, s)) \wedge$
$(\forall\ a\ In.\#Ag(a) \wedge A.\#comp(d, a) \Rightarrow \neg A.\#allergy(p', a))$

ON Delete($A.\#against(d', s')$) CHECK $\forall\ p\ In.\#Pat(p) \wedge In.\#Drug(d') \wedge$
$A.\#takes(p, d') \Rightarrow (\exists\ s\ In.\#Sympt(s) \wedge A.\#suff(p, s) \wedge A.\#against(d', s)) \wedge$
$(\forall\ a\ In.\#Ag(a) \wedge A.\#comp(d', a) \Rightarrow \neg A.\#allergy(p, a))$

ON Insert($In.\#Ag(a')$) CHECK $\forall\ p, d\ In.\#Pat(p) \wedge In.\#Drug(d) \wedge$
$A.\#takes(p, d) \Rightarrow (\exists\ s\ In.\#Sympt(s) \wedge A.\#suff(p, s) \wedge A.\#against(d, s)) \wedge$
$(A.\#comp(d, a') \Rightarrow \neg A.\#allergy(p, a'))$

ON Insert($A.\#comp(d', a')$) CHECK $\forall\ p\ In.\#Pat(p) \wedge In.\#Drug(d') \wedge$
$A.\#takes(p, d') \Rightarrow (\exists\ s\ In.\#Sympt(s) \wedge A.\#suff(p, s) \wedge A.\#against(d', s)) \wedge$
$(In.\#Ag(a') \Rightarrow \neg A.\#allergy(p, a'))$

ON Insert($A.\#allergy(p, a)$) CHECK $\forall\ d\ In.\#Pat(p) \wedge In.\#Drug(d) \wedge$
$A.\#takes(p, d) \Rightarrow (\exists\ s\ In.\#Sympt(s) \wedge A.\#suff(p, s) \wedge A.\#against(d, s)) \wedge$
$(\neg In.\#Ag(a) \vee \neg A.\#comp(d, a))$

Abb. 4-11: Trigger für Integritätsbedingung (15)

[JS91], die effiziente Berechnung der durch eine Transaktion induzierten neuen Fakten [OLIV91], und vieles mehr.

Die objektorientierten Konzepte von O-Telos wurden anhand von Axiomen forma-

lisiert. Grundlage ist eine einheitliche Definition von Objekten, die alle extensionale Information in Quadrupeln darstellt. Insbesondere wird sowohl Klassen als auch Instanzen Objekteigenschaft zugestanden. Tatsächlich werden beide hinsichtlich ihrer Darstellung überhaupt nicht unterschieden. Beide können in Transaktionen und in Anfragen auftauchen. Jederzeit ist eine Änderung von Klassen möglich. Damit wird These 4 aus der Einleitung im Hinblick auf das Schema der Objektbank erfüllt. Wie später noch zu sehen sein wird, kann in gleicher Weise mit den Klassen der Klassen (Metaklassen) umgegangen werden. Dieser Aspekt erlaubt die Darstellung und Manipulation von anderen Daten- und Objektmodellen als Metaklassen von O-Telos.

Durch die Zuordnung von Literalen in Formeln zu Objekten der Objektbank wird quasi für jedes (Klassen-)Objekt ein Prädikat definiert. Der Nachteil einer zu geringen Anzahl von Prädikaten gegenüber den deduktiv-relationalen Datenbanken wird dadurch mehr als wett gemacht. Durch die Möglichkeit der Spezialisierung neigen nämlich Objektbanken dazu, mehr Klassen aufzuweisen als die relationalen Datenbanken Relationen haben. Die strikte Definition von multipler Generalisierung in O-Telos ermöglicht eine monotone Semantik der Spezialisierung von Attributen. Als Konsequenz können Literale in Formeln der kleinsten Attributspezialisierung zugeordnet werden. Durch die übliche Vererbung der Klassenzugehörigkeit gehen keine Lösungen für die Oberklassen dieser Attribute verloren.

Objektbanken erlauben gewöhnlich eine stärkere Strukturierung der Daten als die wertebasierten relationalen Datenbanken. Wenn man diese Strukturierung mit Axiomen ausdrückt, so kann man sie zur Vereinfachung von Anfragen heranziehen. Das allgemeine Verfahren hierzu heißt semantische Anfrageoptimierung. Hier wird es für die Strukturaxiome angewandt.

Kapitel 5: Objektbankstruktur und Formeln

Das relationale Datenmodell stellt vergleichsweise geringe Anforderungen an die Struktur der Daten. Die Beachtung der Stelligkeit der Relationen und die Einhaltung der Komponentenbereiche werden gefordert. Programmiersprachliche Objektbanken stellen zur Überwindung dieser Modellierungsschwäche ein Typsystem mit Konstruktoren zur Verfügung. In O-Telos wird die größere Struktur durch die Vielzahl der fest vorgegeben Axiome ausgedrückt. Das Ziel dieses Kapitels ist die Ausnutzung dieser Axiome zur semantischen Optimierung von logischen Ausdrücken. Der besondere Vorteil liegt in der Tatsache, daß die Axiome für alle Objektbanken gelten. Diese Art der Optimierung verbessert also die Effizienz aller deduktiven Objektbanken unabhängig von den Eigenschaften einer konkreten Anwendung.

5.1. Ausnutzen der Objektidentität

Für ein festes $\#o$ aus ID gibt es nach Axiom A_1 höchstens ein Objekt $P(\#o, \#x, \#l, \#y) \in \mathrm{OB}$. Sei nun φ eine bereichsbeschränkte Formel, die ein Literal $P(\#o, x, l, y)$ enthält, wobei $\#o$ eine Konstante ist und die anderen Komponenten Variablen sind.

Fall I: $\#o \notin \mathrm{OID(OB)}$. Dann gilt wegen der Definition von OID(OB) und dem Bereichsabschlußaxiom, daß das Literal $P(\#o, x, l, y)$ äquivalent durch *false* ersetzt werden kann. Falls $P(\#o, x, l, y)$ Bereichsprädikat der ursprünglichen Formel war, etwa für alle drei Variablen x, l, y, so ist eine weitere Vereinfachung möglich:

$$
\begin{aligned}
& \forall\, x, l, y\ \neg P(\#o, x, l, y) \vee \psi \\
\Leftrightarrow\ & \forall\, x, l, y\ \neg \mathit{false} \vee \psi && \text{(Voraussetzung)} \\
\Leftrightarrow\ & \forall\, x, l, y\ \mathit{true} \vee \psi \\
\Leftrightarrow\ & \mathit{true}
\end{aligned}
$$

Die komplette Teilformel in φ kann also durch *true* ersetzt werden. Wenn x, l, y durch einen Existenzquantor gebunden sind, so kann die Teilformel durch *false* ersetzt werden:

$$\begin{aligned} & \exists\, x,l,y\ P(\#o,x,l,y) \wedge \psi \\ \Leftrightarrow\ & \exists\, x,l,y\ \mathit{false} \wedge \psi \qquad \text{(Voraussetzung)} \\ \Leftrightarrow\ & \exists\, x,l,y\ \mathit{false} \wedge \psi \\ \Leftrightarrow\ & \mathit{false} \end{aligned}$$

Entsprechende Aussagen gelten auch, falls in der Teilformel nur über Teilmengen von $\{x, l, y\}$ quantifiziert wird.

Fall II: $\#o \in \mathrm{OID(OB)}$. Dann gibt es wegen Axiom A_1 genau ein Objekt $P(\#o, \#x, \#l, \#y) \in \mathrm{OB}$. Da das Literal P nicht als Folgerungsliteral einer deduktiven Regel vorkommen darf, ist $P(\#o, x, y, l)$ also äquivalent zur Formel $(x = \#x) \wedge (l = \#l) \wedge (y = \#y)$. Also kann $P(\#o, x, y, l)$ in φ durch diesen Test auf Gleichheit ersetzt werden. Falls $P(\#o, x, y, l)$ Bereichsprädikat war, so können wiederum weitere Vereinfachungen gemacht werden:

$$\begin{aligned} & \forall\, x,l,y\ \neg P(\#o,x,l,y) \vee \psi \\ \Leftrightarrow\ & \forall\, x,l,y\ \neg((x = \#x) \wedge (l = \#l) \wedge (y = \#y)) \vee \psi \qquad \text{(Voraussetzung)} \\ \Leftrightarrow\ & \forall\, x,l,y\ (x \neq \#x) \vee (l \neq \#l) \vee (y \neq \#y) \vee \psi \\ \Leftrightarrow\ & \psi' \end{aligned}$$

Dabei geht ψ' aus ψ durch Substitution aller Vorkommen von x, l, y durch die Konstanten $\#x, \#l, \#y$ hervor. Analog folgt für existenzquantifizierte Formeln:

$$\begin{aligned} & \exists\, x,l,y\ P(\#o,x,l,y) \wedge \psi \\ \Leftrightarrow\ & \exists\, x,l,y\ (x = \#x) \wedge (l = \#l) \wedge (y = \#y) \wedge \psi \qquad \text{(Voraussetzung)} \\ \Leftrightarrow\ & \exists\, x,l,y\ (x = \#x) \wedge (l = \#l) \wedge (y = \#y) \wedge \psi \\ \Leftrightarrow\ & \psi' \end{aligned}$$

Wiederum geht ψ' aus ψ durch Ersetzen aller Vorkommen von x, l, y durch die Konstanten $\#x, \#l, \#y$ hervor. Man beachte, daß φ durch die Ersetzung der Teilformel durch ψ' bereichsbeschränkt bleibt. Insgesamt ist der folgende Satz gezeigt:

SATZ 5-1

Sei $(\mathrm{OB}, \mathrm{R}, \mathrm{IC})$ eine innere deduktive Objektbank, $\varphi \in R \cup \mathrm{IC}$ eine Bereichsformel, in der ein Literal $P(\#o, x, l, y)$ vorkommt mit $\#o \in \mathrm{ID}$. Dann läßt sich φ äquivalent zu einer Bereichsformel umformen, in der $P(\#o, x, l, y)$ durch maximal drei Tests auf Gleichheit ersetzt ist.

Ein ähnliche Aussage ist in [JCV84] zu finden, wo funktionale und Schlüsselabhängigkeiten in relationalen Datenbanken dazu benutzt werden, um die Gleichheit von Variablen in einer Anfrage zu schließen. Als Konzequenz werden die „Tableaus“ (diese entsprechen Hornklauseln) umgeschrieben, so daß gleiche Variablen durch dasselbe Variablensymbol repräsentiert werden. Dies führt im Effekt zur Elimination eines Allquantors.

Als Beispiel betrachte man die Formel

$$\forall\ x, y\ \neg P(\#drug1, x, drug1, y) \vee In.\#Pers(x)$$

Mit der Objektbank von Abbildung 4-1 ist Fall II anwendbar, und wir erhalten

$$\forall\ x, y\ \neg((x = \#Jack) \wedge (drug1 = drug1) \wedge (y = \#QF)) \vee In.\#Pers(x)$$

was zu

$$In.\#Pers(\#Jack)$$

vereinfacht wird. Der Vergleich $(drug1 = drug1)$ ist äquivalent zu *true* und fällt weg. Da y in $In.\#Pers(x)$ nicht auftaucht, fällt auch diese Teilformel weg. Die Variable x wird durch $\#Jack$ ersetzt.

Ein zweites Beispiel zeigt den Fall, in dem $P(o, x, l, y)$ kein Bereichsprädikat ist:

$$\forall\ x, y\ \neg In.\#Pat(x) \vee \neg In.\#Drug(x) \vee P(\#drug1, x, drug1, y)$$

Die Ersetzung von $P(\#drug1, x, drug1, y)$ ergibt hier

$$\forall\ x, y\ \neg In.\#Pat(x) \vee \neg In.\#Drug(y) \vee ((x = \#Jack) \wedge (y = \#QF))$$

Satz 5-1 wird für benutzerdefinierte Regeln und Integritätsbedingungen nur eine geringe Bedeutung haben, da dort erfahrungsgemäß [RD91] das Literal P kaum benutzt wird. Jedoch kann er im Zusammelspiel mit der Vereinfachung von Formeln auf Metaklassenniveau (siehe Kapitel 7) zur Optimierung von abgeleiteten Formeln benutzt werden. Es sei darauf verwiesen, daß der Satz anstatt nur für $P(o, x, l, y)$ auch für beliebige Prädikate $Q(x, y_1, ..., y_k)$ gilt, wenn x in der Extension von Q Schlüsseleigenschaft hat. Diese Aussage betrifft die Optimierung von Formeln, in denen die Prädikate von Anfragen vorkommen.

5.2. Ausnutzen der Attributklassifikation

Das Klassifikationsaxiom A_{13} beschreibt die korrekte Zuordnung von Objekten zu Klassen, insbesondere von Attributen zu ihren Kategorien. In Abbildung 4-1 gibt es zum Beispiel das Attribut *takes*, welches die Klasse *Patient* und die Klasse *Drug* in Beziehung setzt. Jede Instanz dieses Attributs (etwa *#drug1*) muß dann eine Beziehung zwischen einer Instanz von *Patient* (etwa *Jack*) und einer Instanz von *Drug* (etwa *QuasiForte*) sein.

Diese „Attributklassifikation" überträgt sich nach Lemma 4-3 in Abwesenheit deduktiver Regeln auch auf das Prädikat $A(x, m, y)$: aus der Ableitbarkeit von $A(x, m, y)$ kann die Ableitbarkeit von $In(x, c_s)$ und $In(y, d_s)$ geschlossen werden, wobei c_s, d_s sich aus einem eindeutig bestimmten Objekt $P(p_s, c_s, m, d_s) \in \mathrm{OB}$ ergeben. Wenn nun deduktive Regeln hinzukommen, so stellt sich die Frage, ob aus der Ableitbarkeit von $A.p(x, y)$ auch auf die Klassenzugehörigkeit von x und y geschlossen werden kann. In der Tat ist diese Aussage wahr. Sie ist eine Konsequenz der strikten multiplen Generalisierung und der Definition 4-6, die nur solche Formeln zuläßt, in denen die Variablen an die richtigen Sorten gebunden sind. Sei dazu $(\mathrm{OB}, \mathrm{R}_i, \mathrm{IC}_i)$ eine integre innere Objektbank, die einer äußeren Objektbank zugehört. Ferner sei $A.p(x, y) \in ext(A.p)$ mit $P(p, c, m, d) \in \mathrm{OB}$. Dann sind drei Fälle sind möglich.

Fall I: $A.p(x, y)$ gelte wegen Axiom A_{35}. Dann folgt wegen $P(p, c, m, d) \in \mathrm{OB}$ und dem Klassifikationsaxiom $In(x, c)$ und $In(x, d)$. Axiom A_{34} liefert dann die Behauptung.

Fall II: $A.p(x, y)$ gelte wegen einer Regel $\varphi_i \in \mathrm{R}_i$, die gemäß Punkt i) von Definition 4-6 aus einer Regel $\varphi_a \in \mathrm{R}_a$ hervorgegangen ist. Die Regel φ_i muß ein Folgerungsprädikat $A.p(x', y')$ haben. Falls x' Variable der Sorte c', so folgt $Isa(c', c)$ gemäß Definition von *classes* und *min*. Da $In.c'(x)$ wahr sein muß, damit $A.p(x, y)$ durch die Regel ableitbar ist (siehe Def. 4-5), folgt $In.c(x)$. Falls x' Konstante ist, so muß $c \in classes(x', \varphi_a)$ gelten, da sonst φ_a nicht umformbar gewesen wäre. Also gilt $In(x', c)$ wegen der Definition von *classes*, mithin $In.c(x')$. Da $A.p(x, y)$ durch φ_i ableitbar ist, folgt $x = x'$ und $In.c(x)$. Analog zeigt man, daß $In.d(y)$ gilt.

Fall III: $A.p(x, y)$ gelte wegen einer Regel $\forall\, x', y'\ A.q_1(x', y') \Rightarrow A.p(x', y')$, die aufgrund von Punkt ii) aus Definition 4-6 in R_i ist. Dann muß auch $A.q_1(x, y)$ wahr sein mit $P(o_1, q_1, isa, p) \in \mathrm{OB}$ für ein $o_1 \in \mathrm{OID(OB)}$. Nun fällt dieses Faktum entweder in Fall I), Fall II) oder Fall III). Im letzteren Fall kann wieder auf die Ableitbarkeit eines Faktes $A.q_2(x, y)$ geschlossen werden mit $P(o_2, q_2, isa, q_1) \in \mathrm{OB}$. Da es nur endlich viele Objekte gibt muß es ein Index $k \geq 1$ geben, derart daß die Konjunktion

$$A.p(x, y) \wedge A.q_1(x, y) \wedge ... \wedge A.q_k(x, y) \wedge Isa(q_1, p) \wedge Isa(q_2, q_1) \wedge ... \wedge Isa(q_k, q_{k-1})$$

gültig ist, und für q_k keine Regel der Form $\forall\ x', y'\ A.q_{k+1}(x', y') \Rightarrow A.q_k(x', y')$ existiert. Also muß $A.q_k(x, y)$ in Fall I) oder II) fallen. Also folgt $In.c_k(x)$ und $In.d_k(y)$, wobei $P(q_k, c_k, m_k, d_k) \in \mathrm{OB}$ angenommen wird. Mit $q_0 := p$ gilt nun wegen Axiom A_{15} für alle $1 \leq i \leq k$:

$$Isa(c_i, c_{i-1}) \wedge Isa(d_i, d_{i-1})$$

Dabei seien $P(q_i, c_1, m_i, d_i) \in \mathrm{OB}, 0 \leq i \leq k$ die zu den Objektidentifikatoren q_i gehörigen Objekte. Durch Vererbung der Klassenzugehörigkeit zu Oberklassen (siehe Punkt ii) von Definition 4-6) folgt: $In.c_0(x)$ und $In.d_0(y)$. Da $c_0{=}c$ und $d_0{=}d$ ist die Behauptung gezeigt. Die drei Fälle ergeben zusammen den folgenden Satz 5-2.

SATZ 5-2

Sei $(\mathrm{OB}, \mathrm{R}_a, \mathrm{IC}_a)$ äußere deduktive Objektbank, und $(\mathrm{OB}, \mathrm{R}_i, \mathrm{IC}_i)$ die zugehörige innere deduktive Objektbank. Dann gilt für alle $x, y \in \mathrm{OID}(\mathrm{OB}), P(p, c, m, d) \in \mathrm{OB}$:

$$A.p(x, y) \Leftrightarrow A.p(x, y) \wedge In.c(x) \wedge In.d(y)$$

Die Konsequenz aus Satz 5-2 ist, daß Vorkommen von $In.c(x)$ bzw. $In.d(y)$, die in einer Formel $\varphi_i \in \mathrm{R}_i \cup \mathrm{IC}_i$ in Konjunktion mit dem Prädikat $A.p(x, y)$ gestrichen werden dürfen, ohne daß sich die Bedeutung der Formel ändert.

5.3. Beispiel

Das Prinzip der Attributklassifikation führt dazu, daß in praktischen Beispielen fast alle Sortenprädikate $In.c(x)$ aus Regeln und Integritätsbedingungen eliminiert werden können. Mit der Integritätsbedingung (15) aus Kapitel 4 und der zugehörigen extensionalen Objektbank aus Abbildung 4-1 ergeben sich folgende Implikationen:

$$\begin{aligned}
A.\#takes(p, d) &\Rightarrow In.\#Pat(p) \wedge In.\#Drug(d) \\
A.\#suff(p, s) &\Rightarrow In.\#Pat(p) \wedge In.\#Sympt(s) \\
A.\#against(d, s) &\Rightarrow In.\#Drug(d) \wedge In.\#Sympt(s) \\
A\#comp(d, a) &\Rightarrow In.\#Drug(d) \wedge In.\#Ag(a) \\
A.\#allergy(p, a) &\Rightarrow In.\#Pat(p) \wedge In.\#Ag(a)
\end{aligned}$$

Die Integritätsbedingung kann also vereinfacht werden zu

$$\begin{aligned}
&\forall\ p, d\ A.\#takes(p, d) \Rightarrow \\
&\quad (\exists\ s\ A.\#suff(p, s) \wedge A.\#against(d, s)) \wedge \\
&\quad (\forall\ a\ \ A\#comp(d, a) \Rightarrow \neg A.\#allergy(p, a))
\end{aligned} \tag{16}$$

Dies bedeutet insbesondere, daß nur noch fünf Trigger vom Vereinfachungsalgorithmus generiert werden. Sie sind in Abbildung 5-1 zusammengefaßt. Die Anzahl der Trigger entspricht nun auch dem Beispiel aus Kapitel 2 (siehe Abb. 2-2). Allerdings wird hier nicht über irrelevante Attribute – wie das Alter eines Patienten – quantifiziert. Also führen Änderungen an diesem Attribut auch nicht mehr zu Triggerauswertungen.

ON Insert$(A.\#takes(p', d'))$ CHECK $(\exists s\ A.\#suff(p', s) \wedge A.\#against(d', s)) \wedge$
$(\forall a\ A.\#comp(d', a) \Rightarrow \neg A.\#allergy(p', a))$

ON Delete$(A.\#suff(p', s'))$ CHECK $\forall d\ A.\#takes(p', d) \Rightarrow (\exists s\ A.\#suff(p', s) \wedge$
$A.\#against(d, s)) \wedge (\forall a\ A.\#comp(d, a) \Rightarrow \neg A.\#allergy(p', a))$

ON Delete$(A.\#against(d', s'))$ CHECK $\forall p\ A.\#takes(p, d') \Rightarrow (\exists s\ A.\#suff(p, s) \wedge$
$A.\#against(d', s)) \wedge (\forall a\ A.\#comp(d', a) \Rightarrow \neg A.\#allergy(p, a))$

ON Insert$(A.\#comp(d', a'))$ CHECK $\forall p\ A.\#takes(p, d') \Rightarrow$
$(\exists s\ A.\#suff(p, s) \wedge A.\#against(d', s)) \wedge \neg A.\#allergy(p, a')$

ON Insert$(A.\#allergy(p, a))$ CHECK $\forall d\ A.\#takes(p, d) \Rightarrow$
$(\exists s\ A.\#suff(p, s) \wedge A.\#against(d, s)) \wedge \neg A.\#comp(d, a)$

Abb. 5-1: Trigger für Integritätsbedingung (16)

Die hier gezeigte Streichung von Sortenprädikaten ist in der Integritätskomponente der Objektbank ConceptBase (siehe Kapitel 9) realisiert worden. Für eine Objektbank mit 3 Instanzen von *Patient*, 14 Instanzen von *Drug* und 10 Instanzen von *Agent* wurden zwei Testläufe ausgeführt. In Lauf I war die Eliminierung der Sortenprädikate ausgeschaltet, in Lauf II war sie eingeschaltet. Die Ergebnisse sind in Abbildung 5-2 zusammengefaßt. In Lauf II wurde nur etwas mehr als die Hälfte der Trigger generiert. Die Rechenzeiten stehen für den Aufwand zum Testen der Integrität bei der oben erwähnten Anzahl von Eintragungen von Instanzen. Die Zahlen legen nahe, daß der Effizienzgewinn in Lauf II hauptsächlich auf die wesentlich geringere Anzahl von Triggerauswertungen zurückzuführen ist.

5.4. Diskussion

Die Ausnutzung der Objektidentität führt zur Ersetzung von Prädikaten, deren Schlüsselkomponente an einen Wert gebunden ist, durch Tests auf Gleichheit. In einer Reihe von

Testlauf	gen. Trigger	ausgew. Trigger	Rechenzeit
I	9	49	25.7 sec
II	5	12	5.9 sec

Abb. 5-2: Effizienzsteigerung durch Attributklassifikation

Fällen können ähnlich wie bei der Vereinfachung von Formeln Quantoren wegfallen, da die Variablen nur einen einzigen Wert annehmen können.

Die Definition der Axiome von O-Telos betont die Klassifikation von Attributen. Wie Satz 5-2 zeigt, können die Klassen der Argumente eines Attributprädikats $A.p(x,y)$ direkt aus dem Objekt p geschlossen werden. Hieran ist bemerkenswert, daß diese Zuordnung auch in Anwesenheit von multipler Generalisierung und von deduktiven Regeln erhalten bleibt. Die abgeleitete Information erfüllt quasi das Klassenschema, das durch die Taxonomie der Klassen und ihrer Attribute vorgegeben ist.

Die Technik der Formelvereinfachung ist eng verwandt mit der semantischen Anfrageoptimierung (siehe z.B. [JARK84,CGM90]). Dort wird eine Anfrage Q in eine äquivalente Anfrage Q' transformiert, indem die Gültigkeit von Integritätsbedingungen ausgenutzt wird. Speziell für Telos wird in [ITS90] ein resolutionsbasierter Algorithmus vorgeschlagen, der Optimierungen an beliebigen Formeln vornimmt. Allerdings sind diese Optimierungen *anwendungsabhängig*, d.h. abhängig von der „Güte“ der Integritätsbedingungen, die für die jeweilige Objektbank formuliert wurden. Im Gegensatz dazu gelten die Axiome von O-Telos in **jeder** Objektbank. Also ist ihre Anwendbarkeit garantiert.

Eine Stärke des Objektmodells O-Telos besteht in der Unterstützung von Änderungsoperationen auf einzelnen Attributen. Jetzt wird der Fall untersucht, in dem mehrere (möglicherweise alle) Attribute eines Objektes innerhalb einer Transaktion verändert werden. Unter bestimmten Voraussetzungen können solche Operationen besonders effizient ausgewertet werden. Desweiteren wird gezeigt, wie komplex aufgebaute Datenstrukturen mit deduktiven Regeln beschrieben und gewartet werden können. Als Anwendung wird der Bereich der Softwarekonfiguration betrachtet.

Kapitel 6: Komplexe Objekte

Objekte in O-Telos sind sehr kleine Informationseinheiten, nämlich Quadrupel. Ein Vorteil dieser Darstellung ist die uniforme Darstellung von Instanzen, Klassen und Attributen. Änderungen an Klassen, z.B. die Hinzufügung eines Attributs, sind gleichberechtigt gegenüber Operationen an Instanzen. Ein zweiter Vorteil der feinen Granularität der Änderungsoperationen ist, daß kleine Änderungen, etwa nur die Löschung einer einzigen Zugehörigkeitsbeziehung $In(x, p)$ zu einer Klasse p, auch nur solche Regeln und Integritätsbedingungen betreffen, die ein Prädikat $In.p$ bzw. $A.p$ enthalten.

Dieses Kapitel untersucht typische Konstellationen, in denen eine größere Komplexität von Objekten vorteilhafter erscheint. Zum einen ist dies der Fall, wenn in Transaktionen immer mehrere Attribute eines Objektes verändert werden. Mit der Methode von Kapitel 4 wird jede Attributänderung unabhängig verarbeitet. Wie jedoch das relationale Beispiel in Kapitel 2 zeigt, kann es von Vorteil sein, daß die Attribute eines Objektes (Tupels) als eine Einheit angesehen werden. Eine Änderung der ganzen Einheit führt dann zu einer besseren Vereinfachung der zu testenden Integritätsbedingung oder Regel. Zudem werden redundante Auswertungen vermieden. Als Anwendung wird der Einsatz im Bereich der Softwarekonfiguration betrachtet. Als Objekte treten Versionen von *Modulen* auf, die bei kompatiblen *Schnittstellen* miteinander konfiguriert werden können.

Die zweite Situation ist die Integration mit Anwendungsprogrammen. Da in O-Telos der dynamische Aspekt der Operationen auf Objekten anders als z.B. in O_2 bisher ausgeklammert wurde, ist eine Kopplung mit einer Programmiersprache wohl unumgänglich. Auf diesem Feld dominieren Sprachen mit *Datentypen*. Auf einer Familie von Grundtypen können mittels Typkonstruktoren komplexere Typen aufgebaut werden. Interessanterweise reichen Objektidentität und deduktive Regeln auch aus, um diese Datenstrukturen zu beschreiben **und** zu warten. Die Ausgangsidee ist wie bei [SS91b], daß komplexe Objekte Sichten auf die Objektbank sind.

6.1. Aggregation von Änderungsoperationen

Betrachtet wird die Situation, in der innerhalb einer Transaktion mehrere Attribute eines Objektes x gleichzeitig eingetragen bzw. gelöscht werden. Formal kann diese Tatsache als sogenannte **Änderungsregel** aufgeschrieben werden:

$$\forall x, y_1, ..., y_n \; A.p_1(x, y) \wedge ... \wedge A.p_n(x, y_n) \Rightarrow U(x, y_1, ..., y_n) \tag{17}$$

Ein Fakt $U(o, a_1, ..., a_n)$ drückt dann die Tatsache aus, daß die Attribute $a_1, ..., a_n$ von o in einem Schritt verändert werden. Wir definieren für jedes solche U und $\pi \subseteq \{1, ..., n\}$ **abgeleitete Änderungsregeln**:

$$\forall x, y_1, ..., y_k \; A.p_{j_1}(x, y) \wedge ... \wedge A.p_{j_k}(x, y_n) \Rightarrow U_\pi(x, y_1, ..., y_k) \tag{18}$$

Wenn also ein Objekt auf all seinen Attributen geändert wird, so wird es auch auf einer Teilmenge dieser Attribute geändert. Sowohl U als auch die U_π dürfen in keiner anderen deduktiven Regel Folgerungsprädikat sein. Damit gelten für die Formeln (17) und (18) wegen dem Vollständigkeitsaxiom auch deren Umkehrungen. Wir sagen ein Attribut p sei **einwertig**, falls folgende Integritätsbedingung erfüllt ist:

$$\forall \; x, y, z \; A.p(x, y) \wedge A.p(x, z) \Rightarrow (y = z) \tag{19}$$

LEMMA 6-1

Seien $(\mathrm{OB}, \mathrm{R}, \mathrm{IC})$ eine integre innere deduktive Objektbank und $\varphi \in \mathrm{IC}$ eine Formel, die eine Teilformel

$$\phi = A.p_{j_1}(x, y_1) \wedge ... \wedge A.p_{j_k}(x, y_k)$$

enthalte. Ferner enthalte R die Formel (18) als deduktive Regel. Die Formel φ' gehe aus φ durch Ersetzen von ϕ durch $U_\pi(x, y_1, ..., y_k)$ hervor. Dann sind φ und φ' äquivalent.

Beweis. Wenn Teilformel ϕ für eine Substitution ihrer freien Variablen folgt, so folgt wegen der Definition der Regel (18) auch $U_\pi(x, y_1, ..., y_k)$ für diese Substitution. Gilt umgekehrt $U_\pi(x, y_1, ..., y_k)$, so kann dies wegen dem Vollständigkeitsaxiom für U_π nur wahr sein, wenn auch der Bedingungsteil der Regel, also $A.p_{j_1}(x, y) \wedge ... \wedge A.p_{j_k}(x, y_n)$, wahr ist. □

LEMMA 6-2

Sei $(\mathrm{OB}, R, \mathrm{IC})$ und U_π wie in Lemma 6-1. Ferner seien $p_{j_1}, ..., p_{j_k} \in \pi$ einwertige Attribute. Dann hat die erste Komponente von $U_\pi(x, y_1, ..., y_k)$ in $ext(U_\pi)$ Schlüsseleigenschaft, d.h. es gilt

$$U_\pi(x, y_1, ..., y_k) \wedge U_\pi(x, z_1, ..., z_k) \Rightarrow (y_1 = z_1) \wedge ... \wedge (y_k = z_k)$$

für alle $x, y_1, ..., y_k, z_1, ..., z_k \in \mathrm{ID}$

Beweis. Seien $U_\pi(x, y_1, ..., y_k)$ und $U_\pi(x, z_1, ..., z_k)$ zwei Elemente aus $ext(U_\pi)$. Dann müssen wegen dem Vollständigkeitsaxiom aus Kapitel 2 folgende zwei Konjunktionen wahr sein:

$$A.p_{j_1}(x, y_1) \wedge ... \wedge A.p_{j_k}(x, y_k)$$
$$A.p_{j_1}(x, z_1) \wedge ... \wedge A.p_{j_k}(x, z_k)$$

Angenommen $U_\pi(x, y_1, ..., y_k) \neq U_\pi(x, z_1, ..., z_k)$, dann gibt es einen Index $i, 1 \leq i \leq k$ derart, daß $y_i \neq z_i$. Also gilt $A.p_{j_i}(x, y_i) \wedge A.p_{j_i}(x, z_i) \wedge (y_i \neq z_i)$. Dies ist aber ein Widerspruch zur Annahme, daß alle Attribute $p_{j_1}, ..., p_{j_k}$ einwertig sind. □

Aus den Lemmata 6-1 und 6-2 folgt, daß der Integritätstest unter den genannten Voraussetzungen, insbesondere also der Einwertigkeit der Attribute, bei einer gleichzeitigen Änderung mehrerer einwertiger Attribute effizienter ausgewertet werden kann. Sei etwa $\mathrm{Insert}(U(o, a_1, ..., a_n))$ bzw. $\mathrm{Delete}(U(o, a_1, ..., a_n))$ eine solche Änderung. Ferner sei φ eine Integritätsbedingung, die eine Konjunktion $A.p_{j_1}(x, y_1) \wedge ... \wedge A.p_{j_k}(x, y_k)$ enthalte. Dann reicht es, den Einfüge- bzw. Löschtrigger für das Literal $U_\pi(x, y_1, ..., y_k)$ der modifizierten Formel φ' aus Lemma 6-1 auszuführen. Dieser Test ersetzt den Test auf die entsprechenden Operationen für die Literale $A.p_{j_1}(x, y_1), ..., A.p_{j_k}(x, y_k)$.

Beispiel. Wir erweitern das Beispiel aus Abbildung 4-1 um die Attribute *age* und *weight*, die als Eigenschaften der Klasse *Person* definiert werden.

$$P(\#age, \#Pers, age, \#Nat)$$
$$P(\#weight, \#Pers, weight, \#Nat)$$

Die Klasse $\#Nat$ stehe dabei für alle natürlichen Zahlen, die in der Objektbank vorkommen. Wir nehmen für dieses Beispiel ein vordefiniertes Prädikat „$\leq$“ auf natürlichen Zahlen an. Beide Attribute seien einwertig, d.h. es gelten die Integritätsbedingungen

$$\forall\, x, y, z\ A.\#age(x, y) \wedge A.\#age(x, z) \Rightarrow (y = z)$$
$$\forall\, x, y, z\ A.\#weight(x, y) \wedge A.\#weight(x, z) \Rightarrow (y = z)$$

Eine weitere Integritätsbedingung drückt aus, daß Patienten, die jünger als 10 Jahre und leichter als 35 Kilogramm sind, nicht das Medikament *Quasiforte* nehmen dürfen:

$$\forall\, p, a, w\ A.\#age(p, a) \wedge A.\#weight(p, w) \wedge (a \leq 10) \wedge (w \leq 35) \Rightarrow \neg A.\#takes(p, \#QF)$$

In einer Transaktion mögen nun beide Attribute gleichzeitig eingetragen werden. Die zugehörige Änderungsregel lautet:

$$\forall\ p, a, w\ \ A.\#age(p, a) \wedge A.\#weight(p, w) \Rightarrow AW(p, a, w)$$

Damit wird gemäß Lemma 6-1 die Integritätsbedingung umgeschrieben zu

$$\forall\ p, a, w\ \ AW(p, a, w) \wedge (a \leq 10) \wedge (w \leq 35) \Rightarrow \neg A.\#takes(p, \#QF)$$

In der zugehörigen disjunktiven Normalform hat das Prädikat $AW(p, a, w)$ ein negatives Vorzeichen, also wird folgender Trigger generiert:

$$\texttt{ON}\ \text{Insert}(AW(p', a', w'))\ \texttt{CHECK}\ \ (a' \leq 10) \wedge (w' \leq 35) \Rightarrow \neg A.\#takes(p', \#QF)$$

Die zusammengesetzte Änderung bindet in diesem Fall alle Variablen der Integritätsbedingung. Würde die Integritätsbedingung nicht mit $AW(p, a, w)$ umgeschrieben, so wären zwei Trigger auszuwerten:

$$\texttt{ON}\ \text{Insert}(A.\#age(p', a'))\ \texttt{CHECK}$$
$$\forall\ w\ \ A.\#weight(p', w) \wedge (a' \leq 10) \wedge (w \leq 35) \Rightarrow \neg A.\#takes(p', \#QF)$$
$$\texttt{ON}\ \text{Insert}(A.\#weight(p', w'))\ \texttt{CHECK}$$
$$\forall\ a\ \ A.\#age(p', a) \wedge (a \leq 10) \wedge (w' \leq 35) \Rightarrow \neg A.\#takes(p', \#QF)$$

Diese Version ist mehr als doppelt so ineffizient im Vergleich zur Fassung mit dem Prädikat AW, da zwei statt einem Trigger auszuwerten sind. Zudem ist jeder Trigger komplizierter als der Trigger auf $U(p', a', w')$, da ein Quantor mehr vorkommt. Wenn man annimmt, daß insgesamt N Patienten in der Objektbank vorkommen, so ist der Aufwand zur Bestimmung von $A.\#age(p', a)$ bzw. $A.\#weight(p', w)$ mit $log(N)$ zu veranschlagen. Für den Integritätstest dieses Beispiels ist also eine Effizienzsteigerung um den Faktor $2 \cdot log(N)$ zu erwarten.

Wenn eine Transaktion nur einzelne Attribute ändert, so sind weiterhin die Trigger für die entsprechenden Literale (im Beispiel $\text{Insert}(A.\#age(p', a'))$ und $\text{Insert}(A.\#weight(p', w'))$ zu gebrauchen.

6.2. Komplexe Objekte in der Softwarekonfiguration

Programme und Programmsysteme sind komplexe Gebilde mit einer Vielzahl von Eigenschaften. Die Vielzahl der Dokumente, die während der Erstellung eines großen Programms anfallen und die untereinander abhängig sind, erfordern eine Verwaltung und Kontrolle. In [RJG*91] wird ein Software-Prozeßmodell CAD^o beschrieben, das die

Konfiguration von Programmen mittels Entwurfsentscheidungen beschreibt. Eine typische Fragestellung in diesem Zusammenhang ist, ob zwei Programmstücke zueinander passen. Wir betrachten einen stark vereinfachten Ausschnitt aus dem Prozeßmodell, um die Rolle von komplexen Objekten in dieser Umgebung zu untersuchen.

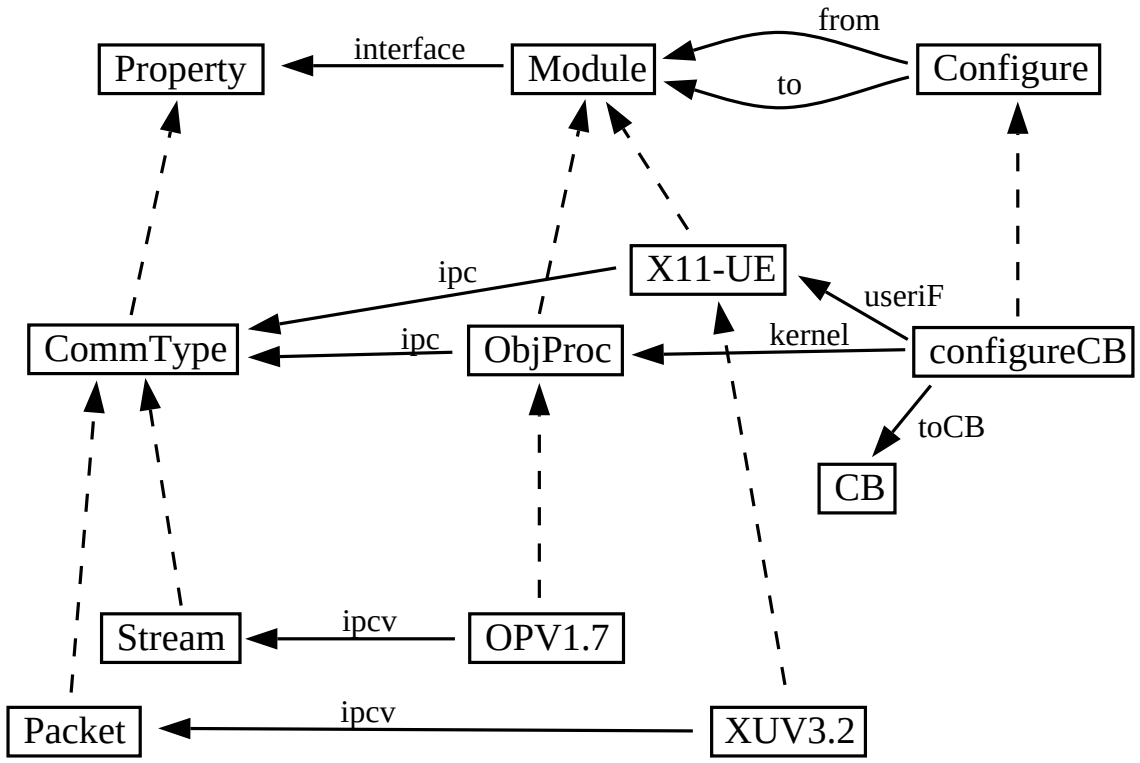

Abb. 6-1: Vereinfachte Softwaredatenbank nach [RJG*91]

Die Module einer **Softwaredatenbank** (Abb. 6-1) werden durch Angabe ihrer Schnittstelle spezifiziert. Eine Implementierung eines Moduls ist formal eine Instanz dieses Moduls, wobei jede Eigenschaft des Moduls genau einmal implementiert wird. Module können mehrere Instanzen haben, die dann auch **Versionen** genannt werden. Für große Programmsysteme ist die Anzahl der Versionen der Module sehr groß, da potentiell jede Änderung an dem Text einer Implementierung eine neue Version ist. Die Architektur des Programmsystems wird durch **Konfigurationsklassen** angegeben. Sie beschreiben im wesentlichen die Import-Relation zwischen Modulen. Die **Konfigurationsentscheidungen**, also die Instanzen der Konfigurationsklassen geben an, welche Modulversionen tatsächlich zusammengebaut wurden.

Um dem Entwickler eine Hilfestellung zu geben, welche Module zusammenpassen können, werden sogenannte Kompatibilitätsbedingungen formuliert. Sie drücken notwendige Bedingungen aus, die Modulversionen erfüllen müssen, um in einer Konfigurationsentscheidung vorzukommen. Für das Beispiel aus Abbildung 6-1 mögen etwa die Module *X11-UE* und *ObjProc* nur dann zusammen konfigurierbar sein, wenn beide in der Eigenschaft *ipc* übereinstimmen.

$$\forall\ c, a_1, a_2, i_1, i_2\ A.\#kernel(c, a_1) \wedge A.\#userIF(c, a_2) \wedge \\ A.\#ipc1(a_1, i_1) \wedge A.\#ipc2(a_2, i_2) \Rightarrow (i_1 = i_2)$$

Aufgrund der Einwertigkeit der Attribute können neue Konfigurationen nur so eingetragen werden, indem alle beteiligten Modulversionen aufgezählt werden. Also kann die Integritätsbedingung mit einer Änderungsregel

$$\forall\ c, a_1, a_2\ A.\#kernel(c, a_1) \wedge A.\#userIF(c, a_2) \Rightarrow U_1(c, a_1, a_2)$$

umgeschrieben werden zu

$$\forall\ c, a_1, a_2, i_1, i_2\ U_1(c, a_1, a_2) \wedge A.\#ipc1(a_1, i_1) \wedge A.\#ipc2(a_2, i_2) \Rightarrow (i_1 = i_2)$$

Da U_1 in der zugehörigen disjunktiven Normalform negativ vorkommt, wird für dieses Literal ein Insert-Trigger generiert:

$$\texttt{ON}\ \text{Insert}(U_1(c', a_1', a_2'))\ \texttt{CHECK}\ \ \forall\ i_1, i_2\ A.\#ipc1(a_1', i_1) \wedge A.\#ipc2(a_2', i_2) \Rightarrow (i_1 = i_2)$$

Bei jeder Auswertung des Triggers sind a_1' und a_2' Konstanten. Jetzt kann zusätzlich die Einwertigkeit der Attribute dieser Modulversionen ausgenutzt werden, indem die (eindeutig bestimmten) Werte für i_1 und i_2 eingesetzt werden. De facto werden also bei Eintragung einer Konfigurationsentscheidung alle Quantoren eliminiert. Dieses Beispiel zeigt, daß die speziellen Anforderungen einer Softwaredatenbank besonders effektiv von deduktiven Objektbanken erfüllt werden. Bei einer relationalen Modellierung würden in der Kompatibilitätsbedingung alle Module einer Konfiguration auftauchen, was zu unnötigen Tests führt, wenn irrelevante Modulversionen geändert werden. Zudem ist es nur schwer möglich, die Komplexität der Module geeignet darzustellen.

In [MJJG91] werden die Konfigurationsklassen zusätzlich als deduktive Regeln interpretiert. In dieser Darstellung wird das Verfahren der Abduktion (siehe z.B. [KM90b]) anwendbar. Bei der Abduktion werden in einer Transaktion keine Fakten extensionaler Prädikate eingetragen, sondern solche intensionaler Prädikate. Durch eine Rückwärtsverfolgung der Regeln – in unserem Fall der Konfigurationsregeln – wird die Änderung auf eine Änderung der extensionalen Datenbank zurückgeführt. Auf diese Weise läßt

sich sowohl die Erstellung einer neuen Modulversion als auch die Suche nach alternativen Implementierungen beschreiben.

6.3. Komplexe Objekte und Regeln

Komplexe Objekttypen wurden in Kapitel 3 als Elemente solcher Typsysteme definiert, die einige Basistypen enthalten und unter den Typkonstruktoren kartesisches Produkt, Potenzmenge und Listenbildung abgeschlossen sind. Diese Typen spiegeln die Datentypen wieder, die in den klassischen Programmiersprachen (z.B. [WIRT79]) vorkommen. Alle Werte eines Typs genügen dem Format, das durch die Typdeklaration vorgegeben ist. Die Formatierung der Werte ist die Voraussetzung für die Implementierung von Funktionen (Operationen, Methoden), die Elemente eines Typs T_1 in Elemente eines Typs T_2 abbilden. Ein Beispiel ist eine Operation $InsertNode(node, tree)$, die eine Baumdatenstruktur in eine Baumdatenstruktur überführt, indem ein neuer Knoten eingefügt wird.

Objektbanken erheben den Anspruch, Anwendungsprogrammen genau die Datenstrukturen zu liefern, die sie weiterverarbeiten. Abbildung 6-2 zeigt den Ansatz des Objektbanksystems PRIMA [HM88,HMS91].

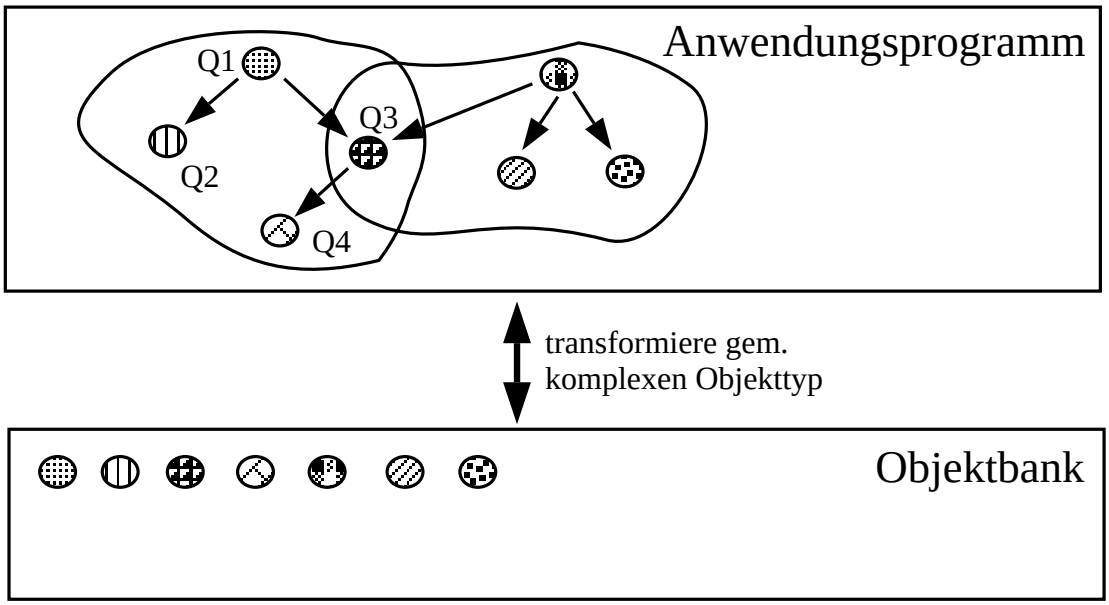

Abb. 6-2: Komplexe Objekte in PRIMA nach [HM88]

In der Objektbank ist eine Ansammlung von Objekten gespeichert. Für ein Anwendungsprogramm werden aus den Objekten komplexe Datenstrukturen konstruiert. Die

Beziehungen zwischen den Objekten in der Anwendungsebene sind durch Zeiger realisiert. Die Abbildung von der Objektbank zu den komplexen Objekten im Anwendungsprogramm wird durch eine *Anfrage* beschrieben. Daraus folgen zwei Änderungsaufgaben:

1) Eine Änderung auf der Objektbank ist auf eine Änderung des komplexen Objektes abzubilden.
2) Eine Änderung des komplexen Objektes ist auf eine Änderung in der Objektbank abzubilden.

Wir konzentrieren uns auf die erste Teilaufgabe, die man auch als *Sichtenwartung* bezeichnet. Für den zweiten Teil, die *Sichtenänderung*, sei für den objektorientierten Aspekt auf [SS91b,SLT91] und für den deduktiven Aspekt auf [KM90b] verwiesen.

O-Telos besitzt das Konzept der Attribute. Damit können in gewisser Weise komplexe Objekte beschrieben werden, jedoch fehlt bisher eine Möglichkeit auszudrücken, wo die *Grenze* des komplexen Objektes ist. Ein erster Schritt in diese Richtung sind die Anfrageregeln (Def. 4-3). Die Extensionen ihrer Folgerungsprädikate enthalten flache Tupel der Form $Q(y_1, ..., y_k)$. In logikbasierten Ansätzen für Objektbanken werden komplexe Objekte meist durch Zulassung komplexer Terme als Argument der Prädikate eingeführt (siehe z.B. [CCT90]). Um eine eindeutige Semantik zu erhalten und die Terminierung von Anfragen zu garantieren, werden die zulässigen Terme eingeschränkt, z.B. durch eine Ausdehnung des Begriffs der Stratifizierung auf die Terme [NT89,AG91]. Definition 6-1 beschreibt eine Alternative, die ohne die Einführung von Termen auskommt.

DEFINITION 6-1

Seien $Q_1, ..., Q_k$ die Folgerungsprädikate von Anfrageregeln. Dann heißt eine Anfrageregel

$$\forall\ v_1, ..., v_m\ Q_1(x_1, t_{11}, ..., t_{n_1 1}) \wedge Q_2(x_2, t_{12}, ..., t_{n_2 2}) \wedge ... \wedge Q_k(x_k, t_{1k}, ..., t_{n_k k}) \Rightarrow CO(v_1, ..., v_m)$$

eine (komplexe) **Objektsicht** genau dann, wenn für alle $x_i, 1 < i \leq k$, gilt: x_i taucht mindestens einmal als t_{rj} in einem $Q_j(x_j, t_{1j}, ..., t_{n_j j})$ mit $j < i$ auf. Falls x_1 Variable ist, so soll $x_1 = v_1$ gelten.

Die $v_1, ..., v_m$ sind genau die freien Variablen der Matrix der Anfrageregel. Mit dem Sichtenbegriff verbindet sich die Annahme, daß die Extension der Sicht abgespeichert wird. Im Falle der Objektsicht nehmen wir dies analog für die Extension $ext(CO)$ an. Jedes Element $s = CO(o_1, o_2, ..., o_m)$ steht für eine Substitution σ_s der freien Variablen der Objektsicht. Wenden wir diese Substitution auf die Prädikate Q_i an, so erhalten wir

die Fakten, die zu der Lösung s beigetragen haben. Wir definieren bei vorgegebener Objektsicht und deren Extension

$$nodes(Q_i) := \{\sigma_s(Q_i(x_i, t_{11}, ..., t_{n_1 1}))|\ s \in ext(CO)\}$$

als die Menge der **Lösungsknoten** der Objektsicht. Die Elemente von $nodes(Q_1)$ heißen auch **Wurzelknoten**.

Eine Objektsicht ist eine nicht-rekursive Regel. Rekursion bzw. Selbstbezüglichkeit kommt aber auf zwei Arten ins Spiel. Einerseits kann ein Q_i mittel- oder unmittelbar durch rekursive Regeln definiert sein. Andererseits kann sich ein Q_j auf ein Q_i mit $i \leq j$ beziehen wie in folgendem einfachen Beispiel:

$$\forall\ x_1, x_2\ Q_1(x_1, a, x_2) \wedge Q_2(x_2, x_1) \Rightarrow CO_1(x_1, x_2)$$

Aus den Lösungsknoten kann auf einfache Weise eine Datenstruktur über Tupeln und Mengen aufgebaut werden. Dazu werden alle Elemente aus $nodes(Q_i)$ mit gleicher erster Komponente mittels des Mengengruppierungsoperators aus [NT89] zusammengefaßt.

$$CQ_i(o_i, M_{1i}, ..., M_{n_i i}) \quad \text{mit}$$
$$M_{ji} := \{p_{ji}|\ Q_i(o_i, ..., p_{ji}, ...) \in nodes(Q_i)\}$$

Die erste Komponente besitzt nun auf CQ_i Schlüsseleigenschaft. Wenn wir annehmen, daß Objektidentifikatoren einfach auf Hauptspeicheradressen abzubilden sind (siehe [MAIE86,GALL90]), so ist durch die CQ_i ein Wert eines Datentyps gegeben, der mit Tupel- und Mengenkonstruktor gebildet wird. Die ersten Komponenten der CQ_i werden als Adressen von n_i-stelligen Tupeln (*Records*) interpretiert. Die Komponenten dieser Records sind wiederum Mengen von Objektidentifikatoren. Falls sie gemäß Definition der Objektsicht als erste Komponente eines anderen CQ_j auftauchen, so sind sie als *Zeiger* anzusehen. Ansonsten sind sie *Werte*. Man beachte, daß durch die Schlüsseleigenschaft und die Verzeigerung der Komponenten x_i in Definition 6-1 ein zusammenhängender Graph entsteht, in dem jedes Element $CQ_i(o_i, M_{1i}, ..., M_{n_i i})$ durch Verfolgen der Zeiger eines Wurzelknotens erreichbar ist.

Abbildung 6-3 zeigt die Beziehung einer Objektsicht mit einer deduktiven Objektbank. Eine Änderung an der extensionalen Objektbank wird über Trigger an die deduktiven Regeln und Integritätsbedingungen weitergereicht. Wenn durch eine Transaktion ein neues Fakt $Q_i(o_i, p_{1i}, ..., p_{n_i i})$ ableitbar wird, so ist davon die Objektsicht betroffen. Existiert das Fakt dort bereits, so braucht nichts gemacht zu werden. Ansonsten wird der Bedingungsteil der Objektsicht für Insert$(Q_i(o_i, p_{1i}, ..., p_{n_i i}))$ ausgewertet. Daraus ergeben sich endlich viele Lösungen für das Folgerungsprädikat CO, die der Extension $ext(CO)$ hinzugefügt werden.

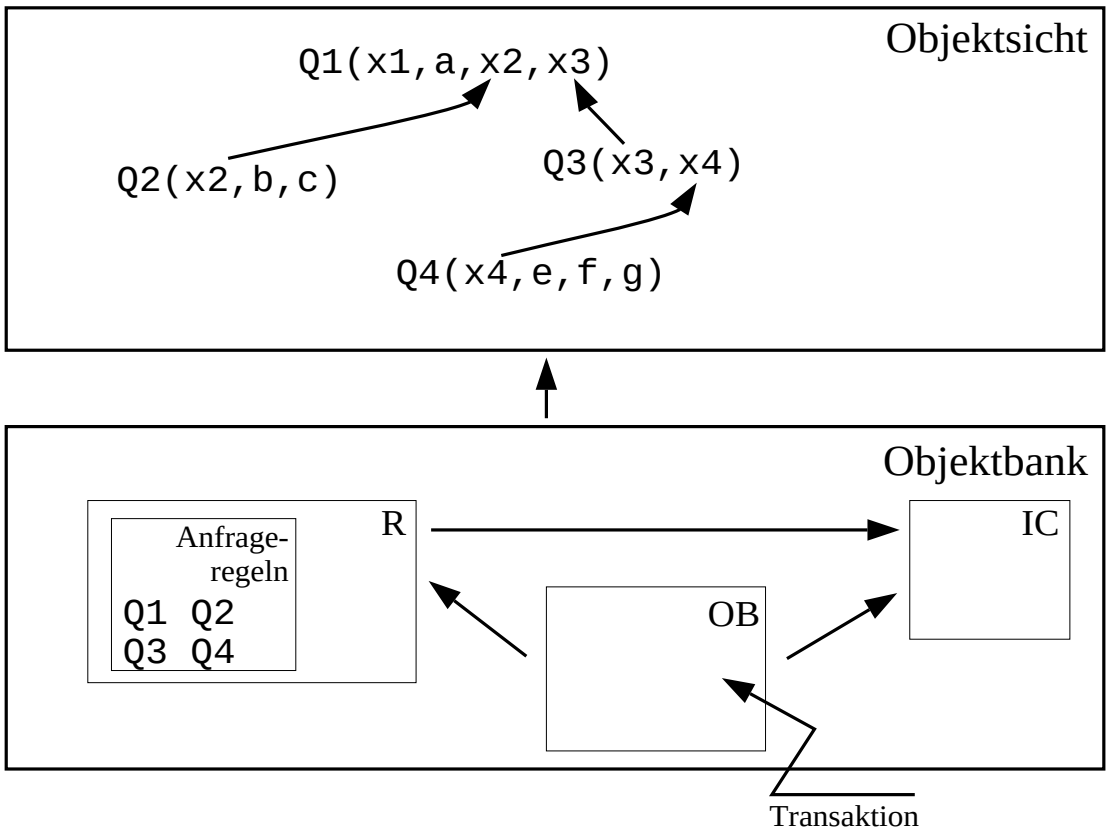

Abb. 6-3: Objektsichten und ihre Wartung

Für den Fall der Löschung sind entsprechende Aktionen auszuführen. Da alle beteiligten Formeln deduktive Regeln sind, können Verfahren aus dem Bereich der deduktiven Datenbanken zur effizienten Berechnung der aus einer Transaktion folgenden Änderungen [OLIV91,KÜCH91] an den Extensionen der abgeleiteten Prädikaten benutzt werden.

Beispiel. Man betrachte eine Anfrageregel, die für einen Patienten alle Wirkstoffe der Medikamente angibt, die er einnimmt.

$$\forall\ p, d, a\ A.\#takes(p, d) \wedge A.\#comp(d, a) \Rightarrow Q_3(p, a)$$

Drei weitere Anfrageregeln Q_1, Q_2, Q_4 seien vorgegeben, aus denen folgende Objektsicht gebildet wird:

$$\forall\ x_1, x_2, x_3, x_4, a, b, c, e, f, g\ Q_1(x_1, a, x_2, x_3) \wedge Q_2(x_2, b, c) \wedge$$
$$Q_3(x_3, x_4) \wedge Q_4(x_4, e, f, g) \Rightarrow CO(x_1, x_2, x_3, x_4)$$

Man beachte, daß die Variablen x_i in der Objektsicht die Verkettung der Lösungsknoten angeben. In diesem Fall zeigt ein Wurzelknoten auf einen Knoten vom „Typ“

Q_1. Dieser hat neben einem Wert a zwei Zeiger auf Lösungsknoten der Typen Q_2 und Q_3. Während Q_2 nur aus Werten besteht, zeigt Q_3 auf Knoten des Typs Q_4. Die Lösungsknoten der Objektsicht seien schon berechnet mit dem folgenden Ergebnis:

$$Q_1(\#drSm, \text{“Dr. Smith“}, \#KlinAc, \#Sam)$$
$$Q_2(\#KlinAc, 5100, \text{“Klinikum Aachen“})$$
$$Q_3(\#Jack, \#Aspi), Q_3(\#Jack, \#Fen)$$
$$Q_4(\#Aspi, 17, 3), Q_4(\#Fen, 7, 21)$$

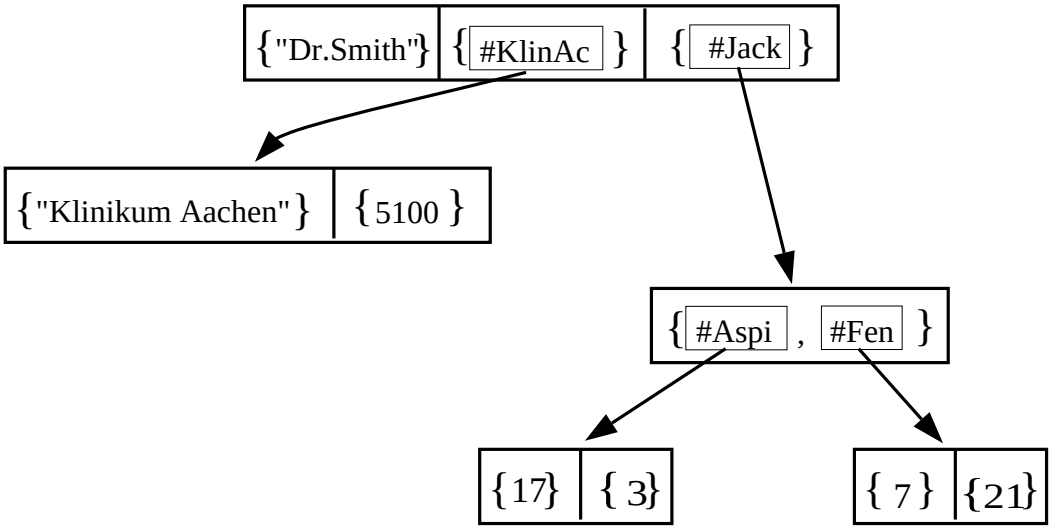

Abb. 6-4: Darstellung einer Objektsicht als Datenstruktur

Durch Gruppierung um die erste Komponente ergeben sich Tupel, in denen diese Komponente Schlüsseleigenschaft besitzt. In obigem Beispiel ist dies insbesondere bei Q_3 zu sehen: die Komponenten $\#Aspi$ und $\#Fen$ werden zu einer Menge zusammengefaßt, auf die mit dem Schlüssel $\#Jack$ zugegriffen werden kann.

$$CQ_1(\#drSm, \{\text{“Dr. Smith“}\}, \{\#KlinAc\}, \{\#Sam\})$$
$$CQ_2(\#KlinAc, \{5100\}, \{\text{“Klinikum Aachen“}\})$$
$$CQ_3(\#Jack, \{\#Aspi, \#Fen\})$$
$$CQ_4(\#Aspi, \{17\}, \{3\})$$
$$CQ_4(\#Fen, \{7\}, \{21\})$$

Abbildung 6-4 zeigt die verzeigerte Datenstruktur, die aus der den CQ_i gewonnen werden kann. Man beachte, daß die ersten Komponenten der Lösungsknoten nicht dargestellt werden. Sie sind implizit als Zeiger zwischen den Teildatenstrukturen vorhanden.

6.4. Diskussion

Objektkomplexität kann in einer deduktiven Objektbank nicht ohne Bezug zu den logischen Formeln gesehen werden. Das erste Ergebnis dieses Kapitels bezieht sich auf die Effizienz der Formelauswertung im Objektmodell O-Telos: wenn Attribute eines Objektes einwertig sind, so kann bei einer gleichzeitigen Änderung einer beliebigen Teilmenge dieser Attribute auch eine gleichzeitige Vereinfachung aller Attributprädikate $A.p(x, y)$ in einer Integritätsbedingung stattfinden, sofern die Prädikate in einer Konjunktion auftreten (Lemmata 6-1 und 6-2). Da im relationalen Datenmodell die Komponenten (Attribute) eines Tupels nicht nur einwertig (1. Normalform), sondern auch notwendig sind, ist der Integritätstest in O-Telos auch bei den aggregierten Anderungsoperationen *mindestens* so effizient wie im deduktiv-relationalen Fall. Wie Kapitel 5 zeigte, ist die Granularität der unterstützten Änderungsoperationen bis auf die Einheit eines einzelnen Attributs verfeinerbar. Insgesamt ist also die änderungsorientierte Formelauswertung in O-Telos als effizienter anzusehen als im relationalen Modell.

Als Anwendung wurde der Kompatibilitätstest bei der Softwarekonfiguration vorgeführt. Man beachte, daß durch die abgeleiteten Änderungsregeln genau die Attributprädikate vereinfacht werden, die in der Integritätsbedingung vorkommen. Das Problem der irrelevanten Attribute bei Formel im relationalen Datenmodell (Kap. 2) taucht hier also gar nicht erst auf.

Mit den aggregierten Änderungsoperationen wird die Eintragung bzw. Löschung von Objekten unterstützt, die mehr als eine einzelne Beziehung ausdrücken. Das zweite Ergebnis in Bezug auf komplexe Objekte beinhaltet einen Weg, wie man beliebig große Datenstrukturen als *Sicht* auf die Objektbank definieren kann und ihre Wartung automatisieren kann. Die Idee ist, ein komplexes Objekt als spezielle deduktive Regel aufzuschreiben. Im Bedingungsteil tauchen Folgerungsprädikate von Anfrageregeln als sogenannte Lösungsknoten auf. Sie fungieren quasi als Tupelkonstruktor. Die Verschachtelung dieser Tupel geschieht dann mittels der ersten Komponenten der Tupel. Aus der Extension des Folgerungsprädikats der Objektsicht wird die komplexe Datenstruktur dadurch gewonnen, daß Lösungsknoten mit gleicher Komponente zusammengefaßt werden. Das Resultat ist eine Menge verschachtelter Datenstrukturen, die aus Tupel- und Mengenkonstruktoren aufgebaut sind.

Der deduktive Rahmen wird bei dem Vorschlag nicht verlassen. Als Folge ist die Berechenbarkeit und die endliche Größe einer Objektsicht garantiert. Ein Anwendungsprogramm braucht lediglich einen Datentyp zu spezifizieren, der auf die Definition einer Objektsicht paßt. Die Umformung der Anfrageergebnisse in diese Datenstruktur ist automatisierbar. Mit der Methode der internen Ereignisse [OLIV91] kann eine Änderung der extensionalen Objektbank effizient an die Objektsicht weitergereicht werden (*Sichtenwartung*). Nicht behandelt wurde hier der umgekehrte Weg: ein Anwendungsprogramm bildet eine komplexe Datenstruktur und speichert sie in der deduktiven Objektbank ab. Dieses Richtung nennen man auch Sichtenänderung oder *view update*. Einige Ergebnisse hierzu sind in [SS91b] enthalten.

Wenn man Klassen als Objekte ansieht, so folgt daraus, daß diese Objekte wiederum Instanz anderer Klassen sein können. Diese nennt man Metaklassen. Die Frage lautet nun: Kann man für diese Metaklassen auch Regeln und Integritätsbedingungen formulieren, die nicht in Paradoxien enden? Die Antwort darauf ist positiv. Zusätzlich wird ein Ansatz zur zweistufigen Vereinfachung von Formeln vorgestellt, der eine große Klasse von Formeln auf Metaklassenniveau der effizienten Auswertung erschließt. Hierzu zählen auch die Axiome des Objektmodells, insbesondere Regeln zur Vererbung und Attributkategorien, die sonst in der Implementierung der Objektbank verborgen wären.

Kapitel 7: Aussagen über Klassen

Das Objektmodell O-Telos drückt die Zugehörigkeit von Objekten zu Klassen explizit durch die Klassifikationsobjekte sowie durch die Prädikate $In(x, c)$ bzw. $In.c(x)$ aus. Tatsächlich kann die Zugehörigkeit zu einer Klasse sogar durch deduktive Regeln ausgedrückt werden. Betrachten wir drei Objekte x, c, mc einer Objektbank, für die Folgendes gilt:

$$In.c(x) \wedge In.mc(c)$$

Das Objekt mc heißt dann **Metaklasse** von x. Es mag noch ein weiteres Objekt mmc geben mit $In.mmc(mc)$, das dann **Metametaklasse** von x genannt wird. Diese Instanzierungshierarchie kann beliebig hoch sein [MBJK90], wobei die Höhe allerdings durch die endliche Größe der Objektbank beschränkt ist. Kapitel 4 zeigte, daß über die Instanzen x einer gegebenen Klasse c gemeinsame Eigenschaften in Form von deduktiven Regeln und Integritätsbedingungen formuliert werden können. Es fragt sich nun, ob manche dieser Eigenschaften nicht auch auf der Höhe der Metaklassen formuliert werden können. Sie brauchen dann für ganze Klassen von Klassen nur einmal spezifiziert werden. Ein Beispiel hierfür ist das Axiom A_{12}, das die Vererbung der Klassenzugehörigkeit beschreibt. Es ist äquivalent zu der Formel:

$$\forall\, p, x, c, d\ In(c, \#Obj) \wedge In(d, \#Obj) \wedge In(x, d) \wedge P(p, d, isa, c) \Rightarrow In(x, c)$$

Die Formel quantifiziert über Objekte x, für die $\#Obj$ Metaklasse ist. Nach den Voraussetzungen von Definition 4-6 ist eine solche Formel keine zulässige deduktive Regel, da in ihr die Prädikate $In(x, c)$ und $In(x, d)$ vorkommen, deren zweite Komponenten Variablen sind.

Nachfolgend soll die Menge der interessierenden Formeln genau beschrieben werden. Eine Reihe von Beispielen zeigt die Ausdrucksstärke solcher Formeln. Im Lösungsvorschlag

wird eine schrittweise Vereinfachung der Formeln beschrieben, die die ursprüngliche Formel durch eine Menge speziellerer Formeln äquivalent ersetzt.

7.1. Metaprädikate und Metaformeln

Der Klassenbegriff in O-Telos stützt sich allein auf das Klassifikationsprädikat $In(x, c)$ bzw. $In.c(x)$. Es gibt daher keine Probleme, für eine Klasse mc von c – d.h. $In.mc(c)$ ist wahr – deduktive Regeln und Integritätsbedingungen zu formulieren, wenn diese die Vorbedingungen von Definition 4-6 erfüllen. Die folgende Klasse von Formeln erfüllt sie nicht.

DEFINITION 7-1

Sei (OB, R, IC) eine deduktive Objektbank, die eine Formel $\varphi \in$ R∪IC enthalte mit:
- in φ gibt es ein Prädikatsvorkommen $In(x, c)$, wobei c eine Variable ist, oder
- in φ gibt es ein Prädikatsvorkommen $A(x, m, y)$, wobei x durch kein Bereichsprädikat $In(x, c)$ mit konstantem c gebunden ist.

Dann heißt dieses Prädikatsvorkommen **Metaprädikat** und φ **Metaformel**.

Metaformeln verlassen keineswegs den Bereich der Prädikatenlogik erster Stufe. Der Name beruht auf der Tatsache, daß über die Klassen c von Objekten x quantifiziert wird. Die Sorte von c ist also Metaklasse der durch x bezeichneten Objekte. Zwei Beispiele aus dem Sprachumfang von Telos [MBJK90][6] zeigen, die Art solcher Metaformeln. Das erste Beispiel ist eine Integritätsbedingung, die für bestimme Attribute p einer Klasse angibt, daß jede Instanz der Klasse einen Wert für dieses Attribut haben muß. Dazu nehmen wir an, daß die Objektbank ein Attribut $P(\#Nec, \#Obj, necessary, \#Obj)$ enthalte.

$$\begin{aligned} \forall\ p, c, m, d, x\ & In.\#Nec(p) \wedge In(x, c) \wedge P(p, c, m, d) \Rightarrow \\ & \exists\ y\ In(y, d) \wedge A(x, m, y) \end{aligned} \tag{20}$$

Metaprädikate sind hier $In(x, c)$, $In(y, d)$ und $A(x, m, y)$. Eine zweite Integritätsbedingung definiert allgemein den Begriff „einwertiges“ Attribut (vgl. Kap. 6). Wir nehmen wiederum die Existenz eines Attributs $P(\#Single, \#Obj, single, \#Obj)$ an.

$$\begin{aligned} \forall\ p, c, m, d, x\ & In.\#Single(p) \wedge In(x, c) \wedge P(p, c, m, d) \Rightarrow \\ & (\ \forall\ y, z\ In(y, d) \wedge In(z, d) \wedge A(x, m, y) \wedge A(x, m, z) \Rightarrow (y = z)\ ) \end{aligned} \tag{21}$$

[6] In [MBJK90] sind diese Beispiele ohne Berücksichtigung der Deduktion des Attributprädikats A(x,m,y) formuliert. Die hiesigen Darstellungen sind also insofern allgemeiner.

In Formel (21) sind $In(x,c)$,$In(y,d)$,$In(z,d)$,$A(x,m,y)$ und $A(x,m,z)$ Metaprädikate. Mithin ist eine deduktive Objektbank, die eine der beiden Formeln enthält nicht in eine innere deduktive Objektbank transformierbar, die Vorraussetzung für nicht-triviale Stratifizierbarkeit und effiziente Auswertung ist. Beide Integritätsbedingungen sind jedoch in Anwendungen so häufig gebraucht, daß sie oft in der Implementierung des Objektbanksystems fest einkodiert sind, etwa im Datenbanksystem SIM [JGF*88]. In relationalen Datenbanken ist die Einwertigkeit durch die erste Normalform [CODD73] – eine Integritätsbedingung dieses Datenmodells – festgeschrieben.

Eine Analyse beider Formeln zeigt, daß die variable zweite Komponente in den Metaprädikaten der Form $In(x,c)$ jeweils durch ein Bereichsprädikat $P(p,c,m,d)$ gebunden ist. Wären c und d Konstanten, so wären auch die Prädikate der Form $A(x,m,y)$ keine Metaprädikate mehr. Lemma 7-1 zeigt, daß Formeln durch eine endliche Menge von einfacheren Formeln äquivalent ersetzt werden können.

LEMMA 7-1

Sei $(\mathrm{OB},\mathrm{R},\mathrm{IC})$ eine deduktive Objektbank, $\varphi \in \mathrm{R} \cup \mathrm{IC}$ eine Formel der Form $\forall x_1,...,x_k\ E(x_1,...,x_k) \Rightarrow \psi$, die das Prädikat E kein zweites Mal enthalte. Dann gilt:

$$\varphi \quad \Leftrightarrow \quad \psi_{e_1} \wedge ... \wedge \psi_{e_s}$$

Dabei sei $ext(E) = \{e_1,...,e_s\}$ und ψ_{e_i} die Formel, die aus ψ durch die Substitution der Variablen $x_1,...,x_k$ mit den entsprechenden Komponenten von $e_i = E(o_1^i,...,o_k^i)$ hervorgeht.

Beweis. Sei φ eine solche Formel. Nach dem Vollständigkeitsaxiom (Kap. 2) ist φ äquivalent zu der Formel

$$\begin{aligned}\forall x_1,...,x_k\ (((x_1 = o_1^1) \wedge ... \wedge (x_k = o_k^1)) \vee \\ ((x_1 = o_1^2) \wedge ... \wedge (x_k = o_k^2)) \vee \\ ... \\ ((x_1 = o_1^s) \wedge ... \wedge (x_k = o_k^s))) \\ \Rightarrow \psi\end{aligned}$$

Durch Anwendung des Distributivgesetzes und Ausnutzen der Äquivalenz der Formel $\forall x\ P(x) \wedge E(x)$ mit $(\forall x\ P(x)) \wedge (\forall x\ E(x))$ ist diese Formel gleichbedeutend zu

$$\begin{aligned}(\forall x_1,...,x_k\ ((x_1 = o_1^1) \wedge ... \wedge (x_k = o_k^1)) \Rightarrow \psi) \wedge \\ (\forall x_1,...,x_k\ ((x_1 = o_1^2) \wedge ... \wedge (x_k = o_k^2)) \Rightarrow \psi) \wedge \\ ... \\ (\forall x_1,...,x_k\ ((x_1 = o_1^s) \wedge ... \wedge (x_k = o_k^s)) \Rightarrow \psi)\end{aligned}$$

Das ist aber äquivalent zu der Formel $\psi_{e_1} \wedge ... \wedge \psi_{e_s}$, wie im Lemma 7-1 definiert. □

Lemma 7-1 zeigt, daß bestimmte Formeln durch eine endliche Formelmenge $\{\psi_{e_1}, ..., \psi_{e_s}\}$ äquivalent ersetzt werden können. Intuitiv wird das quantifizierte Prädikat E durch seine Extension ersetzt. Es sollte beachtet werden, daß die Teilformeln ψ_{e_i} auch durch Anwendung des Vereinfachungalgorithmus aus Abbildung 2-1 erhalten werden kann. Das Prädikat $E(x_1, ..., x_k)$ taucht nämlich in der zugehörigen disjunktiven Normalform als negatives Literal auf.

7.2. Anwendungen

Als Beispiel wird das Lemma auf die Integritätsbedingungen für *necessary*- und *single*-Attribute angewandt. Die Rolle des Prädikats E spielt für Formel (20) das Prädikat $In.\#Nec(p)$ und für Formel (21) das Prädikat $In.\#Single(p)$. Mit der Objektbank aus Abbildung 4-1 soll das Attribut *against* von *Drug* zu beiden Attributklassen *necessary* und *single* Instanz sein, d.h. es gilt

$$ext(In.\#Nec) = \{In.\#Nec(\#against)\} \text{ und}$$
$$ext(In.\#Single) = \{In.\#Single(\#against)\}$$

Jedes Medikament soll also gegen genau ein Symptom wirken. Die Anwendung von Lemma 7-1 auf Formel (20) ergibt die äquivalente Formel:

$$\forall\ c, m, d, x\ In(x, c) \wedge P(\#against, c, m, d) \Rightarrow \exists\ y\ In(y, d) \wedge A(x, m, y) \qquad (22)$$

Jetzt ist Satz 5-1 anwendbar, da $P(\#against, c, m, d)$ nur durch die Substitution mit $P(\#against, \#Drug, against, \#Sympt)$ wahr werden kann. Also kann weiter vereinfacht werden zu:

$$\forall\ x\ In(x, \#Drug) \Rightarrow \exists\ y\ In(y, \#Sympt) \wedge A(x, against, y) \qquad (23)$$

Diese Formel enthält überhaupt keine Metaprädikate mehr und kann gemäß Definition 4-6 transformiert werden zu:

$$\forall\ x\ In.\#Drug(x) \Rightarrow \exists\ y\ In.\#Sympt(y) \wedge A.\#against(x, y) \qquad (24)$$

Die Optimierung mit Satz 5-2 eliminiert ein Klassifikationsprädikat und als Ergebnis erhalten wir die Formel

$$\forall\ x\ In.\#Drug(x) \Rightarrow \exists\ y\ A.\#against(x, y) \qquad (25)$$

Analog erhalten wir bei Formel (21) für die obige Extension von *In.#Single* die äquivalente Formel:

$$\forall\ x, y, z\ A.\#against(x, y) \wedge A.\#against(x, z) \Rightarrow (y = z) \tag{26}$$

Dies ist genau die Formel wie sie für einwertige Attribute in Formel (19) aufgestellt wurde. Wenn weitere Attribute Instanzen von *#Nec* oder *#Single* sind, so sind dafür entsprechende Formeln einzutragen.

Am Schluß dieses Unterkapitels soll noch das Axiom A_{12} aus Kapitel 4 betrachtet werden:

$$\forall\ p, x, c, d\ In(x, d) \wedge P(p, d, isa, c) \Rightarrow In(x, c) \tag{A_{12}}$$

Es ist ebenfalls eine Metaformel wegen der beiden Prädikate $In(x, c)$ und $In(x, d)$. Wir wenden Lemma 7-1 mit $P(p, d, isa, c)$ in der Rolle des Prädikats E an. Mit dem Patientenbeispiel gilt: $ext(P(p, d, isa, c)) = \{P(\#isa1, \#Pat, isa, \#Pers)\}$. Also ist die Formel äquivalent zu

$$\forall x\ In.\#Pat(x) \Rightarrow In.\#Pers(x)$$

Dies ist die gleiche Art Formel, wie sie per Definition 4-6 für Spezialisierungsobjekte eingetragen wurde.

7.3. Das Verfahren der schrittweisen Vereinfachung

Lemma 7-1 eröffnet in Verbindung mit den Optimierungstechniken aus Kapitel 5 eine Möglichkeit, gewisse Metaformeln in eine Menge äquivalenter Formeln umzuformen. Die Kardinalität dieser Menge ist genau die Kardinalität der Extension des Prädikats E. Abbildung 7-1 zeigt, wie aus einer Metaformel auf der Ebene einer Metaklasse durch Anwendung des Lemmas 7-1 für jede Instanz der Metaklasse eine spezielle Formel generiert wird.

In jeder Anwendung des Lemmas wird dabei mindestens ein Metaprädikat zu einem Nicht-Metaprädikat umgeformt. Es wurde bereits erwähnt, daß der Vereinfachungsalgorithmus zur Berechnung der äquivalenten Formeln wiederverwendet werden kann. Der Algorithmus in Abbildung 7-2 realisiert diese Idee, indem neue Formeln zunächst daraufhin untersucht werden, ob sie Metaprädikate enthalten. Falls nicht, so läuft alles wie im ursprünglichen Algorithmus. Ansonsten wird versucht, Lemma 7-1 anzuwenden. Die generierten Formeln $\psi_{e_1}, ..., \psi_{e_s}$ können wiederum Metaformeln sein. Daher werden sie in Schritt 5) nicht direkt eingetragen, sondern der gleichen Prozedur wie die Eingabeformel φ unterzogen. Die Trigger in Schritt 7) sorgen dafür, daß Änderungen an der Extension des

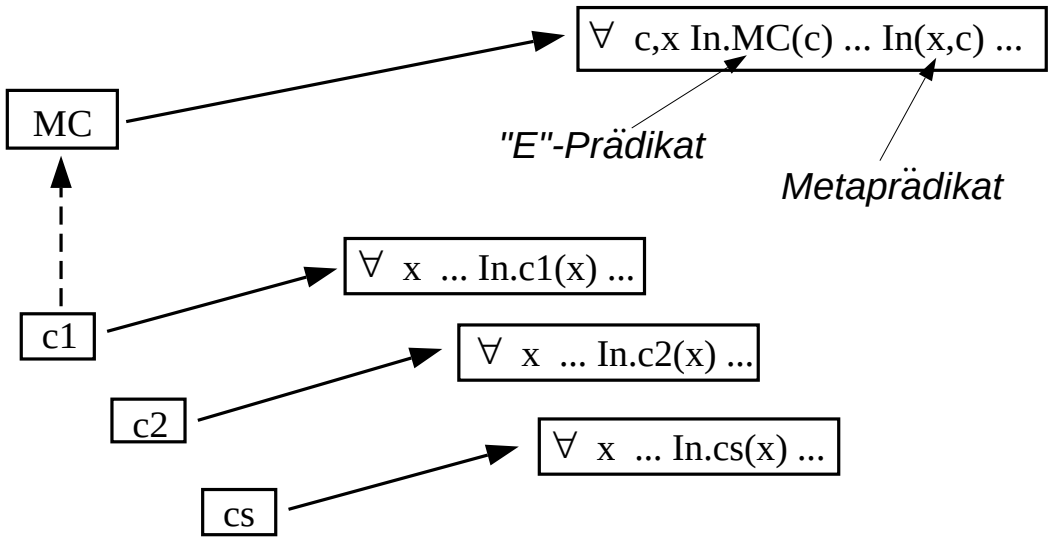

Abb. 7-1: Schrittweise Ersetzung von Metaprädikaten

Prädikats E auch zu entsprechenden Änderungen an der generierten Formelmenge führen. Bei Einfügung ist ein neues $\psi_{e'}$ einzutragen. `Metasimplify` steht hier für eine Ausführung des beschriebenen Algorithmus. Wenn ein Fakt e' aus der Extension von E gestrichen wird, so kann die hierfür generierte Formel $\varphi_{e'}$ ebenfalls gestrichen werden (`Remove`).

1) Setze $F = \{\varphi\}$.
2) Falls $F = \emptyset$, dann STOP. Sonst: Wähle ein beliebiges $\varphi \in F$ und forme es in seine Bereichsform φ' um. Entferne φ aus F.
3) Falls φ' keine Metaformel, so führe die Schritte 2) und 3) vom Algorithmus in Abb. 2-1 aus.
4) Ansonsten forme φ' in die Form $\forall x_1, ..., x_k \; E(x_1, ..., x_k) \Rightarrow \psi$ aus Lemma 7-1 um, falls möglich. Wenn dies nicht möglich ist, so kann φ nicht akzeptiert werden.
5) Füge die Formeln $\psi_{e_1}, ..., \psi_{e_s}$ gemäß der Extension des „E“-Prädikats der Menge F hinzu.
6) Bilde e' aus dem Prädikat $E(x_1, ..., x_k)$, indem alle Variablen durch durch neue Konstanten $x'_1, ..., x'_k$ ersetzt werden. Bilde $\psi_{e'}$ aus ψ durch Substitution der x_i mit x'_i. Füge die Trigger
   `ON` Insert(e') `Metasimplify(`$\psi_{e'}$`)` und
   `ON` Delete(e') `Remove(`$\psi_{e'}$`)`
   dem Objektbanksystem hinzu.
7) Gehe nach 2).

Abb. 7-2: Algorithmus zur schrittweisen Vereinfachung

Zu Schritt 4) ist anzumerken, daß es sowohl vorkommen kann, daß die Umformung nicht möglich ist, als auch daß mehr als eine Umformung existiert. Ein Beispiel für den ersten Fall ist die Formel

$$\exists c\ In.\#Obj(c) \wedge (\forall x\ In.\#Obj(x) \Rightarrow In(x,c))$$

Hier ist es nicht möglich die Formel so umzuformen, daß sie mit einem Allquantor beginnt. Ein Beispiel für den zweiten Fall ist das umgeschriebene Axiom A_{12} aus der Einleitung dieses Kapitels. Dort sind die drei Prädikate $In(c, \#Obj)$, $In(d, \#Obj)$ und $Isa(d, c)$ jeweils Kandidaten für das „E"-Prädikat. Allerdings ist $Isa(d, c)$ zu bevorzugen, da es gleich zwei Quantoren eliminiert, und die beiden anderen Prädikate eine Extension in der Größe der Objektbank haben.

7.4. Metaformeln in der Anwendungsmodellierung

Die Kategorien für die Notwendigkeit und Einwertigkeit von Attributen sind vor allem bei semantischen Datenmodellen wohlbekannt. Lemma 7-1 zeigt, daß solche Spracheigenschaften auch innerhalb eines einfacheren Datenmodells, hier O-Telos, als Integritätsbedingungen spezifiziert und dann automatisch in effiziente Darstellungen überführt werden können. Dieser Aussage ist wichtig für das Ziel der **Erweiterbarkeit** des Schemas einer Objektbank. Allein die Klassifikation eines Attributs unter eine der Kategorien *necessary* bzw. *single* reicht zum Ausdruck der gewünschten Eigenschaft. In diesem Kapitel soll diese Sprachmodellierungsfähigkeit an einer größeren Anwendung, einem erweiterten Entity-Relationship-Modell (ERM) [CHEN76,SS83a], vorgeführt werden. Von dem ursprünglichen ERM wird die Aufteilung in *Entity*- und *Relationship*-Typen übernommen. Zwei Ergänzungen sollen hinzugefügt werden:

I) *Entity*-Typen dürfen auch Attribute zu *Entity*-Typen anstatt nur zu *Domains* haben. Auf diese Weise sind binäre Beziehungen besonders elegant darstellbar.

II) Es gibt einen neuen Typ *ER*, der sowohl Entity- als auch Relationship-Typ ist. Eine solche Erweiterung wurde in [SCHE91] vorgeschlagen, um für das ERM komplexe Objekte (Entities mit Entities als Attributen) definieren zu können.

In O-Telos werden diese Begriffe als Klassen dargestellt, deren Instanzen konkrete Typen wie z.B. *Angestellter* sind. Da diese Typen wiederum Instanzen haben können, etwa den Angestellten *Peter*, stehen die Objekte *Entity*-Typ usw. auf Metaklassennivieau. Das obere Drittel von Abbildung 7-3 zeigt die Darstellung dieser Metaklassen und ihrer Attribute. Der neue Typ *ER* wird kurz dadurch modelliert, daß er sowohl zu Entity als auch zu Relationship Unterklasse ist (multiple Generalisierung). Also sind wegen dem

Axiom A_{12} alle Instanzen von ER auch Instanz der beiden Oberklassen und müssen deren Integritätsbedingungen erfüllen.

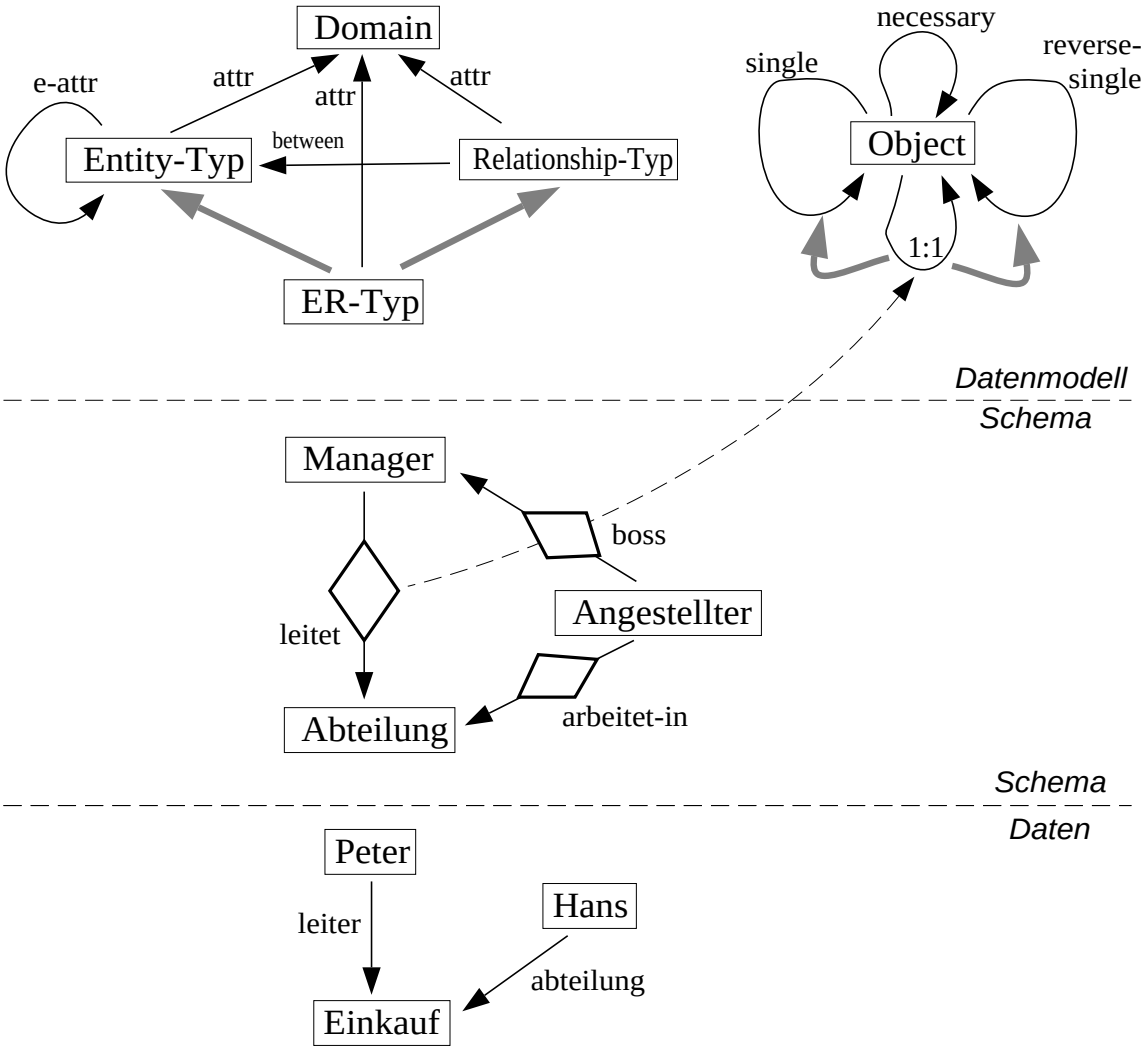

Abb. 7-3: Das erweiterte ER-Modell

Im ERM wird verlangt, daß bei einer Beziehung zwischen Entitäten all diese Entitäten existieren. Dies kann kurz durch eine Unterklassenbeziehung zwischen *between* und *necessary* ausgedrückt werden: jedes Attribut p, das als Instanz von des Attributs *between* auftritt, ist per Vererbung auch Instanz von *necessary*, also muß es die entsprechende Integritätsbedingung (24) erfüllen.

Binäre Beziehungen im ERM können die Kardinalitätsschranken 1:1, 1:n und n:m haben. In der O-Telos-Darstellung von Abb. 7-3 können die Schranken durch Kombination

von Attributkategorien ausgedrückt werden. Dazu wird zunächst das Inverse der Kategorie *single* definiert. Sei also $P(\#RSingle, \#Obj, reversesingle, \#Obj)$ in der Objektbank mit der neuen Integritätsbedingung

$$\begin{aligned} \forall\ p, c, m, d, y\ & In.\#RSingle(p) \wedge P(p, c, m, d) \wedge In(z, d) \Rightarrow \\ & (\ \forall\ x, z\ In(x, c) \wedge In(z, c) \wedge A(x, m, y) \wedge A(z, m, y) \Rightarrow (x = z)\ ) \end{aligned} \tag{27}$$

Durch den Algorithmus in Abb. 7-2 und die gleichen Optimierungsschritte wie für die Formel (26) erhalten wir die für jedes p mit $In.\#RSingle(p)$ die spezielle Integritätsbedingung:

$$\forall\ x, y, z\ A.p(x, y) \wedge A.p(z, y) \Rightarrow (x = z) \tag{28}$$

Damit reicht es, ein Attribut $P(\#oneone, \#Obj, 1{:}1, \#Obj)$ als Spezialisierung beider Attributkategorien *single* und *reversesingle* einzufügen. Die Formeln für die Schranken 1:n und n:m werden hier weggelassen. Abbildung 7-3 zeigt die Kategorie 1:1 als Attribut der Systemklasse *Object* zusammen mit einem Beispiel für zwei Entity-Typen *Angestellter* und *Manager*, die durch ein 1:n-Beziehung *boss* verbunden sind. Zudem ist ein Entity-Typ *Abteilung* aufgeführt, der durch ein 1:n-Attribut *arbeitet-in* mit *Angestellter* und ein 1:1-Attribut *leitet* mit *Manager* in Beziehung steht. Die Zugehörigkeit der Attribute zur Attributklasse *e-attr* wird durch die Raute symbolisiert. Das Attribut *boss* kann durch eine deduktive Regel[7] abgeleitet werden:

$$\forall\ e, d, m\ A.\#arbeitet(e, d) \wedge A.\#leit(d, m) \Rightarrow A.\#boss(e, m)$$

Sei $A.\#boss(\#Peter, \#Hans)$ eine neue Lösung (siehe Abb. 7-3). Dann sind gemäß den spezialisierten Formeln für die Kategorie *single* zwei Trigger zu testen, nämlich:

$$\forall\ z\ A.\#boss(\#Peter, z) \Rightarrow (\#Hans = z) \quad \text{und}$$
$$\forall\ y\ A.\#boss(\#Peter, y) \Rightarrow (\#Hans = y)$$

Dies stellt die Eindeutigkeit des Attributs *boss* sicher. Der zweite Trigger ist äquivalent zum ersten und kann weggelassen werden. Die automatische Entdeckung solcher Symmetrien ist eine weitere Möglichkeit der Optimierung, die jedoch in dieser Arbeit nicht weiter von Interesse sein soll.

Man möge nun annehmen, daß für bestimmte Anwendungen eine neue Kategorie *asymmtric* (29) sinnvoll ist. Sie soll die übliche Asymmetrie binärer Beziehungen ausdrücken. Ein Beispiel ist die Beziehung *boss*: niemand darf der Chef seines Chefs sein.

[7] Die Objektidentifikatoren sind so gewählt, daß die Zuordnung zu den Objektnamen in Abbildung 7-3 eindeutig möglich ist. Auf eine explizite Angabe der Objekte wird verzichtet.

Wiederum sei ein neues Attribut $P(\#Asym, \#obj, asymetric, \#Obj)$ in der Objektbank.

$$\forall\ p,c,m,d,x,y\ In.\#Asym(p) \wedge P(p,c,m,d) \wedge In(x,c) \wedge In(y,d) \wedge A(x,m,y) \Rightarrow \neg(In(y,c) \wedge A(y,m,x)) \tag{29}$$

Für ein Attribut $P(\#Boss, \#Ang, boss, \#Mng)$ mit $In.\#Asym(\#Boss)$ wird dann mit dem schrittweisen Vereinfachungsalgorithmus und nach Anwendung der Optimierungsschritte folgende Integritätsbedingung generiert:

$$\forall\ x,y\ A.\#boss(x,y) \Rightarrow \neg A.\#boss(y,x) \tag{30}$$

Die Formel (30) entspricht der Definition asymetrischer Beziehungen in der Algebra. Es ist zu bemerken, daß sie aus der Formel (29) automatisch abgeleitet ist, in der über alle möglichen asymetrischen Attribute quantifiziert wird.

7.5. Diskussion

Deduktive Objektbanken bauen sowohl auf logik- als auch auf objektorientierten Mustern auf. Beide Wurzeln bergen eine Metaebene. In der Logikprogrammierung werden *Metaprogramme* benutzt, um das Verhalten spezialisierter Auswerter deklarativ zu beschreiben [BRUY90]. In objektorientierten Programmiersprachen dienen *Metaklassen* dazu, gewisse Eigenschaften der Programmiersprache innerhalb der Sprache selbst zu beschreiben und die Erweiterung der Sprache zu ermöglichen [COIN87].

Für deduktive Datenbanken wurde die Abstraktion auf die Metaebene benutzt, um die Gemeinsamkeiten verschiedener Methoden zur Auswertung rekursiver deduktiver Regeln zu beschreiben [BRY90]. Dieses Kapitel zeigt, daß deduktive Objektbanken mit dem Konzept der Metaklassen diese Abstraktionsdimension auch für ihre Regeln und Integritätsbedingungen besitzen. Zwei Eigenschaften sind wesentlich:

I) Objektbanken sind endlich. Wegen des Bereichsabschlußaxioms sind deshalb auch die Extensionen der Folgerungsprädikate deduktiver Regeln endlich.

II) In O-Telos wird die Klassenzugehörigkeit durch ein zweistelliges Prädikat $In(x,c)$ ausgedrückt. Also ist es möglich, in *Metaformeln* über die Klasse c zu quantifizieren, ohne die Logik erster Stufe zu verlassen.

Die erste Eigenschaft wird in Lemma 7-1 genutzt, um für allquantifizierte Formeln eine Quantorenelimination durch Aufzählung der Extension eines Bereichsprädikats zu erreichen. Das Ergebnis ist eine Menge äquivalenter Formeln. Von Nutzen ist diese Umformung in erster Linie für Formeln, die über Klassen quantifizieren. Mit Hilfe des inkrementellen Algorithmus aus Abbildung 7-2 kann die Abbildung zwischen den Metaformeln und

ihrer äquivalenten Darstellung *automatisiert* werden. Das Objektbanksystem muß nur unwesentlich erweitert werden, da der Vereinfachungsalgorithmus wiederverwendet wird. Eine Änderung der Extension des ersetzten Prädikats E ist auch zur Laufzeit möglich. Sie führt zur Einfügung bzw. Löschung von abgeleiteten Formeln.

Die Beispiele der Attributkategorien *single* und *necessary* belegen den Nutzen der Metaformeln. Anstatt diese Kategorien fest zu programmieren, kann es einem automatisierten Verfahren überlassen werden, aus der prädikatenlogischen Definition der Kategorien spezialisierte Integritätsbedingungen zu generieren. Mit Hilfe der in Kapitel 5 beschriebenen Optimierungstechniken gelangt man zu effizienten Darstellungen. Das Beispiel eines Vererbungsaxioms zeigt, daß das Verfahren auch auf deduktive Regeln auf Metaklassennivieau Anwendung finden kann.

Ähnlich wie Metaklassen erlauben es Metaformeln, auf elegante Weise neue Datenmodelle zu beschreiben. Die Definition eines erweiterten Entity-Relationship-Modells ist durch die Einführung von lediglich drei Attributskategorien sowie deren Spezifikation mittels Integritätsbedingungen gelungen. Als besonders ausdrucksstark erweist sich die Möglichkeit der multiplen Generalisierung in O-Telos: die neue Kategorie „1:1“ wird kurz als Unterklasse zweier bestehender Kategorien beschrieben.

Eine offene Frage ist, ab wann es sich nicht mehr „lohnt“, die Metaformeln umzuformen. Sicherlich ist dies der Fall, wenn die Extension des ersetzten Prädikats die Größe der Objektbank erreicht. Für die Beispiele dieses Kapitels kann davon ausgegangen werden, daß nur relativ wenige Formeln generiert werden, nämlich in der Größenordnung der Anzahl der Klassen. Da Klassen die Rolle der Relationen zur Einteilung der Daten spielen, ist der Speicheraufwand gering im Vergleich zur Größe der Objektbank. Eine weitere Frage ist die Auswahl des zu ersetzenden Prädikats im Falle mehrerer Kandidaten. Als Faustregel wird das Prädikat mit der kleinsten Extension empfohlen. Schließlich bleibt noch zu klären, inwiefern die mehrfache Generalisierung zu Attributkategorien mit Metaformeln (wie z.B. mit der Kategorie „1:1“ geschehen) zu optimalen Vereinfachungen führen. Im allgemeinen ist dies nicht der Fall: in den Oberklassen kann beispielsweise die gleiche Formel mehrfach vorkommen, was zu doppelten Auswertungen führt. Bis zu einem gewissen Grad [SCHM89] können solche redundanten Auswertungen durch Einsatz von Theorembeweisern vermieden werden.

Die Gelegenheit zur Abbildung von Metaformeln hängt eng mit der Existenz von Metaklassen zusammen. In Objektmodellen ohne diese Ebene können diese Formeln gar nicht vorkommen. In solchen Umgebungen muß entweder auf feste Kodierung der Kategorien oder der Anwender muß für jede seiner Klassen die spezialisierten Formeln selbst definieren.

Die Fähigkeit zur *Meta-Modellierung* ist eine nachträgliche Bestätigung der Entscheidung, die Wissensrepäsentationssprache Telos zur Basis des Objektmodells O-Telos zu machen. Durch das Vorhandensein der Metaklassenebene ist ein generisches Objektmodell geschaffen, in dem spezialisierte Objektmodelle durch eine Anzahl von Metaklassen beschrieben werden. Falls die Axiome des spezialisierten Objektmodellen durch Metaformeln in der Form von Lemma 7-1 hinschreibbar sind, so kann die Generierung effizienter Testprozeduren automatisiert werden. Die Spezifikation und Erweiterung des Entity-Relationship-Modells mittels weniger Metaklassen und Attributkategorien belegt, daß zumindest die Prototypen maßgeschneiderter Datenmodelle, z.B. das oben beschriebene erweiterte ER-Modell, quasi *ad hoc* implementierbar sind.

Eine große Anwendung von Meta-Modellierung ist die Schemaintegration verteilter relationaler Datenbanken. In [KLEM91] wurde mittels O-Telos eine Methode zur Gewinnung eines gemeinsamen Schemas beschrieben. Das in den Schemata der Basisrelationen enthaltene Wissen wird vom Benutzer interaktiv um Spezialisierungsbeziehungen erweitert. Um die relationalen Schemata und ihre Transformation in das gemeinsame Schema überhaupt beschreiben zu können, wird eine neue Klasse *BaseRelation* eingeführt. Da Relationen Elemente (Instanzen) haben, ist diese Klasse auf der Ebene der Metaklassen. Das gemeinsame Schema wird als Sicht auf die Basisrelationen in O-Telos beschrieben. Für die logische Formulierung der Sichten werden Anfragen benutzt. Da sowohl die Basisrelationen als auch die integrierenden Sichten in O-Telos vorliegen, können beide in gleicher Form abgefragt und manipuliert werden.

Metaformeln werden in Abhängigkeit von der Größe der Extension des „E“-Prädikats auf eine Vielzahl vereinfachter Formeln abgebildet. Zur Bewältigung des Verwaltungsaufwandes für diese generierten Formeln wird ihre Speicherung in der Objektbank vorgeschlagen (Kap. 8). Kapitel 9 ist dem Stand der Implementierung im Objektbanksystem ConceptBase gewidmet.

Deklarative Formeln kann man als Spezifikationen ihrer ausführbaren Darstellungen ansehen. Dieses Kapitel wendet ein Software-Prozeßmodell auf die logischen Formeln an, um sowohl ihre Einfügung und Löschung in der Objektbank als auch ihre Auswertung zu beschreiben. Gerade diese Anwendung eröffnet einen Weg, die in den vorigen Kapiteln vernachlässigte „aktive“ Komponente von Objektbanken doch noch zu erschließen.

Kapitel 8: Ein Modell zur Auswertung und Verwaltung

Logische Formeln in Objektbanken sind Gegenstand vielfältiger Operationen. Bei Eintragung werden sie in ein internes Format überführt, von dem aus die Trigger abgeleitet werden. Falls es sich um Metaformeln handelt, so werden in Abhängigkeit von der Objektbank vereinfachte Formeln generiert. Im Falle der Löschung einer Formel muß der für sie generierte Code ebenfalls gelöscht werden. Daneben wird der Code der Formeln auf Anstoß durch Transaktionen ausgewertet. Diese Operationen haben gemeinsam, daß der Grund ihrer Ausführung die Einfügung beziehungsweise Löschung eines Objektes oder eines Formeltextes ist. Es liegt daher nahe, diese Übereinstimmung zu abstrahieren und zum allgemeinen Prinzip in Objektbanken zu machen.

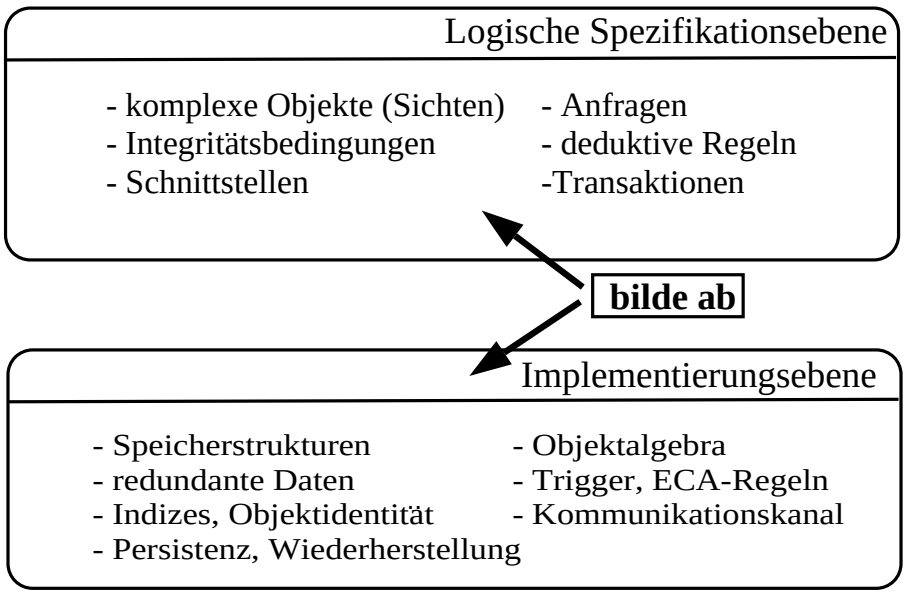

Abb. 8-1: Zweistufige Architektur eines Objektbanksystems

Der Hintergrund der Überlegungen ist eine zweistufige Architektur für Objektbanksysteme (vgl. Abb. 8-1). In der oberen Ebene sind die *Spezifikationen von Eigenschaften* der Objektbank angesiedelt. Hierunter fallen insbesondere deduktive Regeln,

Integritätsbedingungen und Anfragen, aber auch Transaktionen und Spezifikationen von Schnittstellen zu Anwendungsprogrammen. Die untere Ebene stellt die *Implementierungen der Spezifikationen* zur Verfügung. Zwischen den Ebenen gibt es eine enge Beziehung, die **Implementierungsbeziehung** [WIRS86].

Die Umformung der Objekte der oberen Ebene in Objekte der unteren Ebene ist vom Standpunkt der Programmentwicklung ein Implementierungsprozeß. Für solche Implementierungsprozesse wurden in der Literatur Prozeßmodelle entwickelt, die die Objekte und die auf ihnen möglichen Transformationen einordnen. Das D.O.T.-Modell [JR88,JJR89a,JJR89b] dient speziell der Beschreibung der Entwicklung von Datenbankanwendungen. Es unterscheidet Objekte, (Transformations-)Entscheidungen auf den Objekten und Werkzeuge, die die Entscheidungen ausführen. Das D.O.T.-Modell ist bereits in Telos formuliert. Es kann auf der Metaklassenebene Teil der Objektbank sein.

Die Verwaltung und Auswertung von Formeln ist insofern ein Spezialfall eines Software-Entwicklungsprozesses, bei dem die möglichen „Programmspezifikationen“, nämlich Bereichsformeln, alle **automatisch** implementierbar sind. Im weiteren soll aus dieser Idee ein Modell entwickelt werden, das in einheitlicher Weise die Formelmanipulationen in deduktiven Objektbanken beschreibt. Im Mittelpunkt steht der Begriff der Operation mit dem jegliche Art von Abbildungen, Transformationen, Auswertungen abgedeckt sein soll. Die Prinzipien der Klassifikation, Spezialisierung und Attributierung werden benutzt, um

a) die Eingabe- und Ausgabeparameter zu charakterisieren und
b) ähnlich wie bei ECA-Regeln (Kap. 3) festzulegen, wann die Operation aufgerufen wird und wie ihre Parameter zugewiesen werden.

Um einen Implementierungsprozeß mittels D.O.T. (siehe Abb. 8-2) zu beschreiben, werden für die Klasse *Decision* Instanzen eingetragen, die die möglichen Transformationen in der Entwurfsumgebung benennen. Ihre Ein- und Ausgabeparameter werden durch die Attributkategorien *from* und *to* auf Instanzen der Klasse *Objekt* festgelegt. Als Instanzen von *Tool* sind die bekannten Werkzeuge (Compiler, Theorembeweiser usw.) aufgelistet. Sie können mit der Kategorie *by* den Entscheidungen zugeordnet werden.

8.1. Formalisierung der Implementierungsebene

Für die Zwecke der Formelverwaltung und -auswertung wird das Objektmodell O-Telos um eine Klasse *Operation* erweitert. Sie hat zwei Attribute *onInsert* sowie *onDelete*, die Spezialisierungen der Kategorie *from* sind, also Eingabeparameter der Operation. Die Instanzen der Kategorien sollen kurz **Triggerattribute** genannt werden.

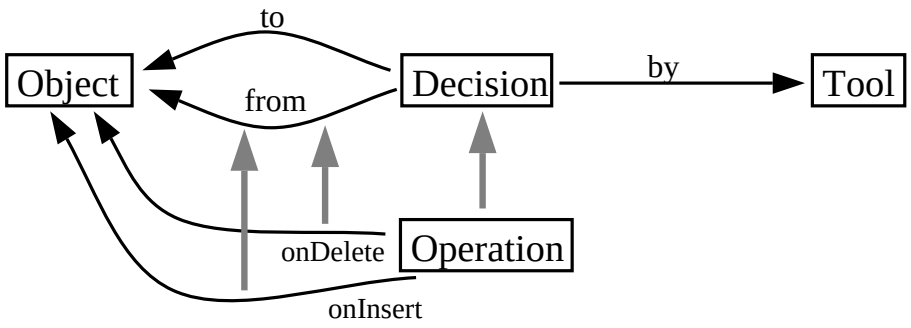

Abb. 8-2: Vereinfachtes D.O.T.-Modell mit Operationen

Ähnlich wie bei den Kategorien aus Kapitel 7 verbirgt sich hinter den Kategorien *onInsert* sowie *onDelete* eine Bedeutung, die über zwei Klassifikationsstufen reicht. Es ist allerdings keine Formel, sondern eine „aktive“ Eigenschaft, die vom Objektbanksystem verlangt wird. Die auslösenden Ereignisse werden durch die Elemente einer Transaktion bestimmt. Abbildung 8-3 definiert eine Zuordnung von Transaktionen zu Ereignissen auf den 3 Prädikatsarten $A.p(x,y)$, $In.c(x)$ und $Isa(c,d)$. Die Zuordnung folgt der Definition der Prädikate aus den Axiomen A_5 bis A_7 und der Definition 4-4.

1) Sei $T = < op_1, op_2, ..., op_s >$ eine Transaktion in dem deduktiven Objektbanksystem.
2) Generiere aus T eine Menge E von *Ereignissen*:
   - 2.1) Falls op_i=$OP(P(o,x,in,c))$, so füge die Ereignisse $OP(In.c(x))$ und $OP(A.c(x_1,y_1))$ der Menge E hinzu, wobei $P(x,x_1,l,y_1) \in$ OB und $OP \in \{\text{Insert}, \text{Delete}\}$.
   - 2.2) Falls op_i=$OP(P(o,c,isa,d))$, so füge $OP(Isa(c,d))$ der Menge E hinzu.
   - 2.3) Sonst: mache nichts

Abb. 8-3: Von Transaktionen zu Ereignissen auf Prädikaten

Es sei nun angenommen, daß in dem Objektbanksystem eine Prozedur

$$\text{CALL}(op, [x_1, ..., x_k])$$

implementiert ist, wobei op der Identifikator einer Operation ist und $[x_1, ..., x_k]$ eine Liste von Identifikatoren. Anschaulich wird die Operation op auf die Argumente $x_1, ..., x_k$ angewandt. Die Aufrufzeitpunkt und die Argumente sind in Abbildung 8-4 definiert.

1) Sei $E = \{e_1, ..., e_s\}$ eine Menge von Ereignissen auf dem deduktiven Objektbanksystem.
2) Führe für jedes Ereignis e_i folgende Schritte aus:
   - 2.1) Falls e_i = Insert($In.c(x)$), dann bestimme alle Attribute $P(p, op, In\text{-}Event, c)$, die Instanz der Kategorie *onInsert* sind und generiere einen Aufruf CALL($op, [x]$).
   - 2.2) Falls e_i = Insert($A.c(x, y)$), dann bestimme alle Attribute $P(p, op, A\text{-}Event, c)$, die Instanz der Kategorie *onInsert* sind und generiere einen Aufruf CALL($op, [x, y]$).
   - 2.3) Falls e_i = Insert($Isa(c, d)$), dann bestimme alle Attribute $P(p, op, Isa\text{-}Event, \#Spec)$, die Instanz der Kategorie *onInsert* sind und generiere einen Aufruf CALL($op, [c, d]$).
   - 2.4) Falls e_i = Delete(P), dann verfahre analog zu Schritt 2.2 bis 2.3 mit der Kategorie *onDelete*
   - 2.5) Sonst: mache nichts

Abb. 8-4: Generierung von Operationsaufrufen durch Ereignisse

Die Eintragung bzw. Löschung der Zugehörigkeit eines Objektes x zu einer Klasse c führt also automatisch zum Aufruf CALL($op, [x]$) der an die Klasse c durch Triggerattribute gebundenen Operationen. Um Änderungen an der Extension der Attributprädikate $A.c(x, y)$ weiterzureichen, wird in Abb. 8-3 regelmässig auch ein Ereignis auf diesen Prädikaten generiert. Dieses Ereignis wird jedoch nur dann zu einem Aufruf führen, falls c ein Triggerattribut mit Namen *A-Event* hat.

In der Tat ist die Klasse *Operation* mit ihren beiden Attributen *onInsert* sowie *onDelete* eine objektorientierte Aufschreibung der Trigger aus Kapitel 2. Der Trigger wird durch ein Attribut zwischen einer Operation *op* und einer Klasse c dargestellt. Das Ereignis ist durch den Namen dieses Attributs (*A-Event* oder *In-Event*) sowie durch die Klassifikation zu der Kategorie *onInsert* bzw. *onDelete* beschrieben. Bedingungsteil und Aktionsteil sind mit dem Aufruf CALL($op, [x_1, ..., x_k]$) auf eine Normalform zusammengefaßt.

Die folgenden zwei Unterkapitel zeigen, wie dieses allgemeine Aufrufschema sowohl zur Verwaltung wie auch zur Auswertung von Formeln einsetzbar ist.

8.2. Formeln als Objekte

Bisher wurden deduktive Regeln und Integritätsbedingungen unterschiedlich zu den Objekten einer deduktiven Objektbank (OB, R, IC) angesehen. Wenn aber Klassen zu jeder Zeit eingetragen und gelöscht werden dürfen, so muß man dies auch für die Formeln

der deduktiven Objektbank erlauben, da sie ja Eigenschaften der Objekte der Klassen ausdrücken. Es liegt also nahe, die Formeln als spezielle Individualobjekte anzusehen, die wie andere Objekte auch manipuliert werden können. Der Name des Formelobjektes enthält den Text der Formel. Um die Formelobjekte einzuordnen, werden zwei neue Klassen *IntegrityConstraint* und *DeductiveRule* definiert. Für diese beiden Klassen gibt es je zwei Operationen, die bei Eintragung bzw. Löschung eines Formelobjektes aufgerufen werden.

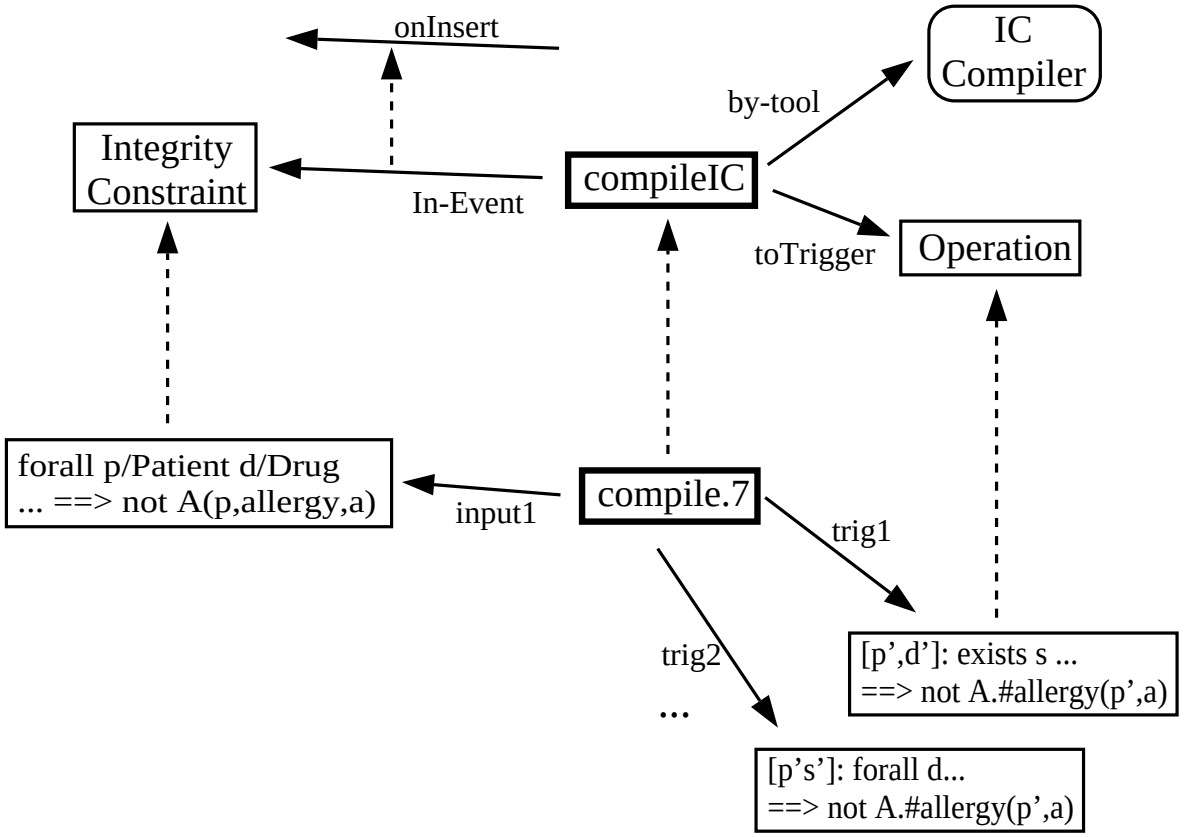

Abb. 8-5: Eintragung von Integritätsbedingungen

Abbildung 8-5 zeigt die Spezifikation der Operation *compileIC* für Integritätsbedingungen. Der Eingabeparameter *In-Event* löst bei Eintragung einer Instanz von *IntegrityConstraint* (Identifikator #IC) die Auswertung der Operation aus. Das Ergebnis ist in der Klasse *Operation*, d.h. die Auswertung erweitert den Code des Objektbanksystems um zusätzliche Operationen. Diese Operationen sind genau die Trigger wie sie im Vereinfachungsalgorithmus (Abb. 2-1) generiert werden. Die Klasse *compileIC* ist über

ein Attribut der Kategorie *by* mit der Klasse *IC-Compiler* verbunden. Auf diese Weise können Operationen den Komponenten des Objektbanksystems zugeordnet werden.

Nehmen wir nun an, eine Integritätsbedingung, etwa die Integritätsbedingung (11) aus Unterkapitel 4.5, soll neu in die deduktive Objektbank eingetragen werden. In Textform lautet ihr voller Text:

```
forall p/Patient d/Drug
  A(p,takes,d) ==> (exists s/Symptom A(p,suffers,s) and A(d,against,s)) and
  (forall a/Agent A(d,component,a) ==> not A(p,allergy,a))
```

In die Objektbank wird diese Formel als sogenanntes **Formelobjekt**

$$P(\#ic1, \#ic1, \texttt{"forall p/Patient d/Drug ... ==> not A(p,allergy,a)"}, \#ic1)$$

eingetragen und der Klasse *IntegrityConstraint* zugeordnet. Die Transaktion bewirkt ein Ereignis Insert($In.\#IC(\#ic1)$). Da auf *IntegrityConstraint* ein Triggerattribut der Kategorie *onInsert* zeigt, wird nach Abbildung 8-4 folgender Aufruf generiert:

$$\text{CALL}(\#compIC, [\#ic1])$$

Hier sei wiederum *#compIC* der Identifikator der Operation namens *compileIC*. Der Aufruf enthält alle Daten, die für die Übersetzung der Formel benötigt werden: durch den Identifikator *#ic1* kann auf den Text der Formel zugegriffen werden. Zudem ist bekannt, daß sich Operation *compileIC* in der Komponente *IC-Compiler* befindet.

Die Details der Implementierung von *compileIC* sind an dieser Stelle nicht von Belang (siehe [KRÜG89,JK90]), sondern nur die Tatsache, daß bei erfolgreicher Übersetzung als Ausgabe die Trigger für den Integritätstester generiert werden. In Abbildung 8-5 sind zwei der Trigger dargestellt. Die restlichen sind wie im Beispiel aus Abbildung 5-1. Die neuen Operationen enthalten analog zu den Formelobjekten ihren Code als Namen. Das vorangestellte Literal dient der Parameterzuordnung bei der Auswertung des Triggers.

Neben dem dynamischen Aspekt des Operationsaufrufs wird das D.O.T.-Modell auch zur Verwaltung der Abhängigkeiten zwischen der Formelspezifikation und dem generierten Code benutzt. Hierzu wird einfach die Instanz der Operation *compileIC* mit den aktuellen Werten der Ein- und Ausgabeattribute gebildet. In Abbildung 8-5 ist das Objekt *compile.7* ein Beispiel für eine solche Aufzeichnung. Da alle Komponenten Objekte sind, können sämtliche Abhängigkeiten in der Objektbank gespeichert werden.

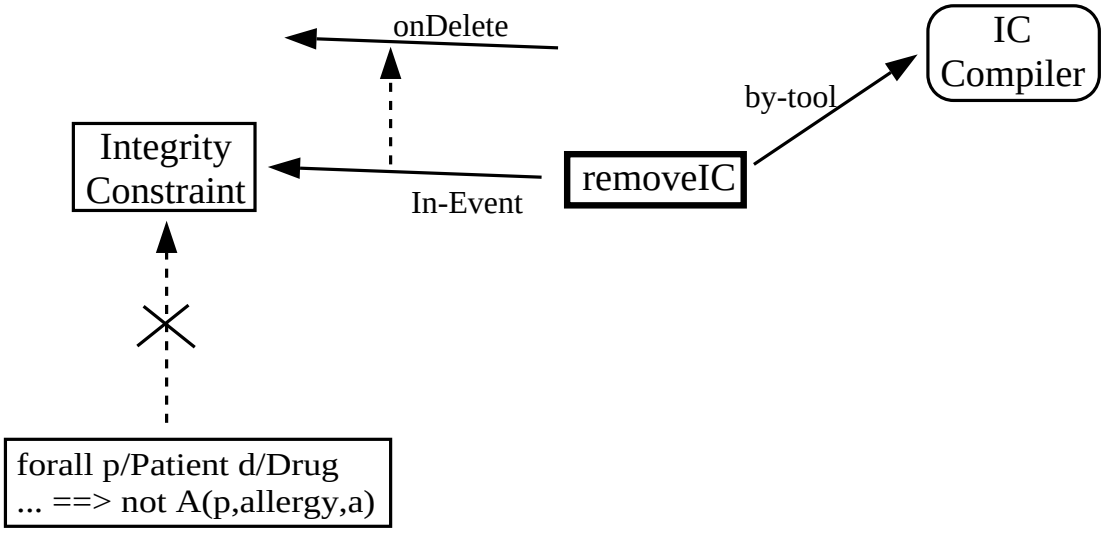

Abb. 8-6: Löschung von Integritätsbedingungen

Formeln können wie jedes Objekt auch wieder aus der Objektbank entfernt werden. Die hierfür zuständige Operation *removeIC* (Abb. 8-6) benutzt für diesen Zweck die vorher abgespeicherten Abhängigkeiten zwischen der Formel und den Triggern. Der Anstoß ist die Löschung eines Formelobjektes. Die Operationen *compileIC* und *removeIC* bilden zusammen die Schnittstelle eines **inkrementellen** Formelverwaltungssystems.

Die Behandlung deduktiver Regeln ist aufwendiger, da bei jeder Transaktion die Änderungen an den Extensionen der abgeleiteten Prädikate zu bestimmen sind. Bei Eintragung und Löschung der Regeln ist neben der Generierung bzw. Löschung des Codes der Regeln auch eine vollständige Berechnung der Extension des Folgerungsprädikats erforderlich, um den Effekt auf Integritätsbedingungen und Objektsichten zu berechnen. Nach der Methode von [OLIV91] können aus einer Transaktion die *internen* Ereignisse auf den abgeleiteten Prädikaten effizient berechnet werden. Da sie das gleiche Format wie die Ereignisse aus Abbildung 8-3 haben, muß an der Darstellung der Trigger als Operationen nichts geändert werden.

Das hier entwickelte Modell ist auch in der Lage, die Übersetzung von Metaformeln zu kontrollieren. Das Problem ist hier, daß Änderungen an der Extension des Prädikats E der Metaformel (vgl. Abb. 7-2) zu einem erneuten Aufruf des Formelübersetzers führen müssen. Diese Aufgabe kann durch Triggerattribute der Kategorien *onInsert* bzw. *onDelete*, die auf die zu E gehörende Klasse zeigen, spezifiziert werden.

8.3. Aktivierung der Formelauswertung

Die Literale in Integritätsbedingungen und deduktive Regeln werden nach dem Verfahren von Definition 4-6 den Klassen der Objektbank zugeordnet. Es macht also Sinn, die Trigger für Integritätsbedingungen direkt diesen Klassen zuzuordnen. Immer wenn eine Instanz dieser Klasse eingetragen (gelöscht) wird, gibt es eine neue Lösung in der Extension des Prädikat (bzw. eine potentielle Löschung in der Extension) und der Code des Triggers ist auszuwerten. Abbildung 8-7 zeigt als Beispiel den Trigger für das Attribut $A.\#takes(p, d)$, der von *IC-Compiler* generiert worden ist. Der volle Text der vereinfachten Form ist der Abbildung 5-1 zu entnehmen.

Das Triggerattribut ist Instanz der Kategorie *onInsert* und hat den Namen *A-Event*. Er zeigt an, daß die Operation für die Einfügung eines Fakts $A.\#takes(x, y)$ zuständig ist. Als Beispiel betrachte man die Objektbank von Abb. 4-1 ohne das Objekt $P(\#in3, \#drug1, in, \#takes)$. Genau dieses Objekt werde in einer Transaktion eingetragen. Dann gibt es nach der Definition in Abbildung 8-3 ein Ereignis Insert$(A.\#takes(\#Jack, \#QF))$. Also wird folgender Operationsaufruf generiert:

$$\text{CALL}(\#trigger1, [\#Jack, \#QF])$$

Hierbei ist $\#trigger1$ der systemgenerierte Objektidentifikator der Operation mit Namen `"[p',d']: exists s ..."`. Durch Einsetzen der Parameter p'=$\#Jack$ und d'=$\#QF$ entsteht die zu testende vereinfachte Form der Integritätsbedingung. Die Darstellung des Triggers als Operation erfüllt also genau den gewünschten Zweck, nämlich den automatisierten Aufruf bei Transaktionen. Die Abbildung zeigt auch ein Attribut *by-tool* mit Ziel *IC-Evaluator*. In der Implementierung des Objektbanksystems bietet es sich an, dieses Werkzeug als Interpreter des Operationsnamens zu implementieren. Dieser Weg ist im Objektbanksystem ConceptBase (Kap. 9) beschritten worden.

Die Repräsentation der Trigger als Objekte erlaubt es, Auswirkungen von Schemaänderungen auf Formeln automatisch zu erkennen. Die Triggerattribute zeigen auf Klassen der Objektbank. Wird eine solche gelöscht, so kann auch das Triggerattribut nicht mehr Teil der Objektbank sein, da sonst die referentielle Integrität (siehe Lemma 4-1) verletzt wäre. Dies macht Sinn, da die Prädikate eng mit den Klassen der Objektbank zusammenhängen: wenn eine Klasse c nicht mehr in der Objektbank ist, so sind die Prädikate $In.c(x)$ und $A.c(x, y)$ quasi ohne Existenzberechtigung. Alle Formeln, die Vorkommen dieser Prädikate aufweisen, müssen gelöscht bzw. re-compiliert werden. Die Repräsentation des abgeleiteten Codes für die Formeln erlaubt es, genau die betroffenen Formeln aufzufinden.

Ein scheinbares Problem stellen die Anfrageregeln dar. Diese Regeln können Prädikate $Q(x, y_1, ..., y_k)$ enthalten, die zunächst keiner Klasse der Objektbank zugeordnet

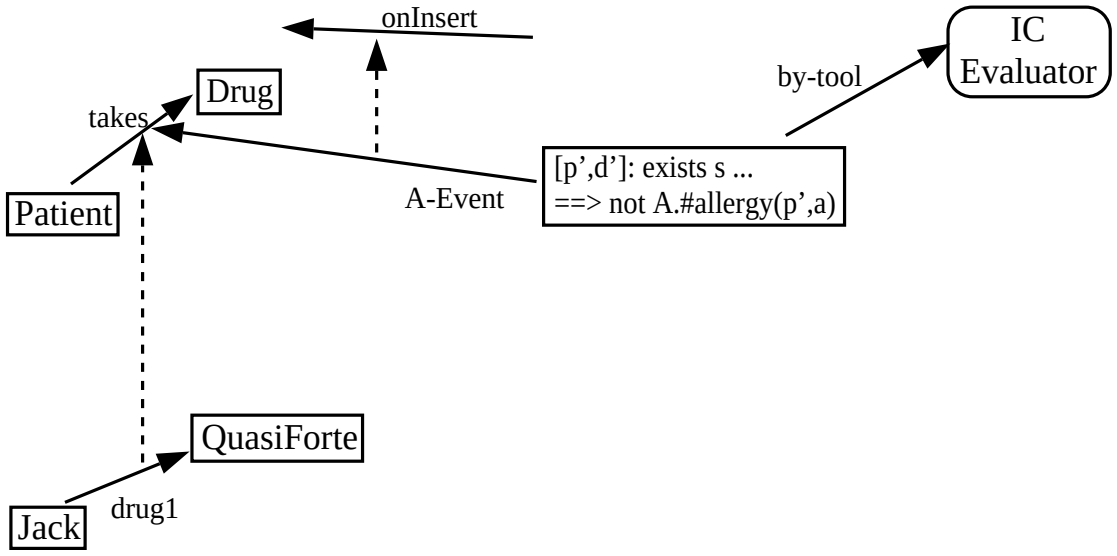

Abb. 8-7: Aktivierung eines Triggers zur Formelauswertung

werden können. Nach [STAU90,JS91] können Anfrageregeln aber als Übersetzungen von sogenannten **Anfrageklassen** angesehen werden. Anfrageklassen sind Teil der Objektbank. Die Fakten der Extension des Anfrageprädikats Q werden als Instanzen der Anfrageklasse angesehen. Also können die Triggerattribute zur Übersetzung einer Anfrage sowie zur inkrementellen Auswertung der Anfrage auf Anstoß durch Transaktionen an die Anfrageklasse angebunden werden. Jede Anfrageklasse Q bekommt dann einen eigenen Triggerattributnamen *Q-Event*. Es folgt, daß die Trigger der Form

$$\texttt{ON}\ \text{Insert/Delete}(Q(x, y_1, ..., y_k))\ \texttt{CHECK}\ \varphi$$

nun ebenfalls als Instanzen der Klasse *Operation* darstellbar sind. Der Algorithmus von Abb. 8-4 wird für Ereignisse Insert/Delete$(Q(x, y_1, ..., y_k))$ dahingehend erweitert, daß in solchen Fällen ein Aufruf CALL$(op, [x, y_1, ..., y_k])$ generiert wird. Damit sind Anfrageprädikate den Prädikaten *In.c*, *A.p* und *Isa* gleichgestellt.

8.4. Zentrale und dezentrale Auswertung

Die als Objekte dargestellten Trigger werden durch die Elemente von Transaktionen angestoßen. Es stellt sich nun die Frage, zu welchem Zeitpunkt der Aufruf stattzufinden hat. Im Fall der Integritätsbedingungen ist die klassische und einfachste Methode die **zentrale** Kontrolle durch das Objektbanksystem: ein Anwendungsprogramm liefert eine Transaktion (Def. 2-4) an das Objektbanksystem ab. Dieses bestimmt die davon

betroffenen Trigger und stellt wie in Kapitel 2 besprochen die Integrität der Objektbank sicher. Diese Art der Kontrolle nennt man auch **reaktiv** [JS89].

Ein weitergehender Ansatz ist die Verlagerung der Triggerauswertungen in die Anwendungsprogramme, den Entstehungsorten der Transaktionen. Der Code der Programme wird von vorneherein so implementiert, daß keine integritätsverletzenden Transaktionen entstehen können. Diesen Ansatz nennt man auch **präventiv** oder **dezentral**. Für die Klasse der dynamischen Integritätsbedingungen, d.h. Bedingungen über zulässige Sequenzen aufeinanderfolgender Datenbankzustände, schlägt [LIPE88] eine Transformation der Integritätsbedingungen in sogenannte Transaktionsspezifikationen vor. Eine solche Spezifikation besteht aus einer Parameterdeklaration, einem Test auf diesen Parametern, einer Fallunterscheidung mit Vor- und Nachbedingungen sowie einer Menge von Invarianten (*frame conditions*).

Um die Einhaltung der Integrität der Datenbank zu garantieren, müssen die Transaktionsspezifikationen in beweisbar korrekte Implementierungen überführt werden. Dieser Ansatz wurde in [SSS88] und [BMSW90] untersucht. Bei beiden ist die Implementierung einer Transaktionsspezifikation mit erheblichem Beweisaufgaben verbunden, die nicht alle automatisierbar sind. Am Ende steht allerdings eine Implementierung, die keine der in der Spezifikation enthaltenen Integritätsbedingungen verletzen kann (*satisfaction by design*).

Wenn alle Anwendungsprogramme nach dieser Methode entwickelt werden, so entfällt der zentralisierte Test im Datenbanksystem. Die Methode birgt auch prinzipiell eine neue Optimierungsdimension, da die zentralisierte Kontrolle durch Verlagerung der Integritätstestphase an das (zeitliche) Ende des Anwendungsprogrammes simuliert werden kann [JS90]. Als Beispiel sei der einfache Test von Parameterwerten genannt. Nach der traditionellen reaktiven Methode wird erst die gesamte Transaktion erzeugt bevor eine Inkonsistenz festgestellt wird. Durch die Verzahnung von Anwendungsprogramm und Integritätsbedingungen kann schon durch die Parameter festgestellt werden, daß die zu generierende Transaktion aufgrund der Parameter niemals integer sein wird. Hierdurch erspart man unnötige Rechenzeit und kann frühzeitig Fehlerkorrekturen anstoßen.

8.5. Verallgemeinerung des Operationsbegriffs

Die Spezifikation der Formelübersetzung und -löschung zeigt, daß auch die Schnittstelle konventionell implementierter Operationen mit dem Schema aus Abbildung 8-2 dargestellt werden können. Tatsächlich ist dieses Schema eng mit abstrakten Datentypen verwandt:

- ▷ Die Sorten werden durch Klassennamen repräsentiert.

- ▷ Den Operationen des abstrakten Datentyps entsprechen die Instanzen der Klasse *Operation*. Ihre Funktionalität, d.h. die Sorten ihrer Ein- und Ausgabeparameter, ist durch die Attribute der Kategorien *from* und *to* gegeben.
- ▷ Schließlich wird eine Menge von Operationen einer Instanz der Klasse *Tool* zugeordnet. Der Name eines solchen Tools steht dann für den abstrakten Datentyp selbst.

Eine vollständige Übereinstimmung ist nicht gegeben, da nicht alle Axiome, die man üblicherweise mit abstrakten Datentypen assoziiert, in Regeln und Integritätsbedingungen ausdrückbar (und somit automatisch implementierbar) sind. Die Axiomensprache für abstrakte Datentypen (siehe etwa [WIRS86]) ist ja dazu entworfen, beliebige Operationen zu spezifizieren, also ist sie im allgemeinen unentscheidbar. Trotzdem können „einfache“ Axiome innerhalb der deduktiven Datenbank formuliert werden. Der Rest der Axiome kann als uninterpretierte Zeichenkette den Instanzen von *Tool* zugeordnet werden.

Ein weiterer Unterschied ist die feste Stelligkeit von Operationen in abstrakten Datentypen. In O-Telos müssen Attribute einer Klasse von einer Instanz nicht notwendig belegt werden. Die Attributkategorien *necessary* und *single* schaffen hier Abhilfe. Indem die Kategorie *by* aus Abbildung 8-2 als Instanz beider Kategorien deklariert wird, erreicht man zudem die eindeutige Zuordnung der Operationen zu den Instanzen von *Tool*.

Wir stellen also fest, daß auch die nicht automatisch implementierbaren Operationen einer deduktiven Objektbank in ihr selbst spezifizierbar sind.

Da die Spezifikation der Operationen Klassen sind, liegt es nahe, darüber nachzudenken, welche praktische Bedeutung die darunterliegende Ebene der Instanzen hat. Am Beispiel der Operation *compileIC* wurde gezeigt, daß die Instanzen der Operationen als Aufzeichnungen der *Operationesaufrufe* anzusehen sind. Durch Belegung der *from/to*-Attribute kann die Beziehung zwischen Ein- und Ausgabe materialisiert werden. In dem Fall der Formelübersetzung wird wie oben gesehen durch Speicherung solcher Instanzen die Wartung des automatisch generierten Codes unterstützt.

8.6. Werkzeugintegration

Unbeachtet blieben bisher die Instanzen der Klasse *Tool*. Im folgenden wird argumentiert, daß gerade diese Facette des D.O.T.-Modells eine Ausdehnung des Integritätsbegriffs von Objektbanken auf die Spezifikation komplexer Objektbanksysteme erlaubt. Nehmen wir an, die verfügbaren Operationen des Objektbanksystems werden ihren Komponenten, den Instanzen von *Tool* mit Hilfe des *by*-Attributs zugeordnet. Der Komponente *IC-Compiler* sind beispielsweise die Operationen *compileIC* und *removeIC* zugeordnet. In gleicher Weise können alle Komponenten des Objektbanksystems dargestellt werden. Das

Ergebnis ist eine (unvollständige) Spezifikation des Systems. Wenn man sie als **Teil der Objektbank** abspeichert, so hat man eine Dokumentation des Systems, die zur Laufzeit mit der gewöhnlichen Anfragesprache (z.B. [STAU90]) abrufbar ist. Das System ist also selbsterklärend[8].

Die erste Folgerung ist, daß auch die **Änderungen an der Spezifikation** des Objektbanksystems in Bezug zu ihrer Funktionalität gesetzt werden kann. Eine Löschung der Operation *compileIC* beispielsweise betrifft direkt die Klasse *IntegrityConstraint* und indirekt alle ihre Instanzen sowie den daraus abgeleiteten Code. Allein die referentielle Integrität von O-Telos reicht schon aus, um unerlaubte Löschungen von Werkzeugen zu erkennen. Darüber hinaus können Integritätsbedingungen, zum Beispiel für Kompatibilitätsaussagen über Versionen von Modulen (siehe Kap. 6), formuliert werden. Solche Aussagen werden typischerweise während des Entwurfsprozesses aufgestellt (siehe [RJG*90,ROSE91]).

Eine weitere Konsequenz ist, daß Werkzeuge wie *IC-Compiler* prinzipiell instanziierbar sind. Wenn die Instanzen von Operationen als Aufrufe dieser Operationen anzusehen sind, so ist es natürlich, die Instanzen der Werkzeuge als **Werkzeugprozesse** zu interpretieren. Also ist das Modell auch in der Lage, die momentane Konfiguration der Prozesse im Objektbanksystem vergröbert darzustellen. Als Beispiel betrachte man ein Werkzeug *X11-GE*, das dazu dienen möge, Informationen über ein Objekt graphisch darzustellen.

Abbildung 8-8 spezifiziert den Grapheditor als Werkzeug mit einer Operation *gedit*, die ein beliebiges Objekt als Eingabe hat. Eine Ebene tiefer ist ein Aufruf dieses Werkzeugs zu sehen. Der Werkzeugprozeß *X11-GE#4* bearbeitet zur Zeit den Aufruf CALL(*#gedit*, [*#Pat*]). Daneben gibt es einen weiteren Prozeß gleichen Typs für das Objekt *#Jack*.

Interaktive Werkzeuge wie der Graph-Editor werden nach dem Stand der Technik [HB89] mit Hilfe eines Zeigegeräts („Maus“) aus Menüs ausgewählt, die bei Anwahl eines Elements auf dem Bildschirm erscheinen (*point-and-click*). Da in unserem Fall alle erforderlichen Informationen in der Objektbank enthalten sind, kann der Inhalt eines Menüs durch eine einfache deduktive Regel beschrieben werden:

$$\forall\, x, c, op\ In(x, c) \wedge A.\#from(op, c) \Rightarrow A.\#menu(x, op)$$

Offensichtlich ist dies eine Metaformel, die jedoch mit der Technik von Kapitel 7 in normale Regeln transformierbar ist, indem sie für die Extension von $A.\#from(op, c)$ – d.h.

[8] Für Programmiersprachen nennt man eine solche Eigenschaft auch *Reflexivität* [COIN87]. Dort benutzt man sie zur Erweiterung der Ausdrucksmittel einer Programmiersprache.

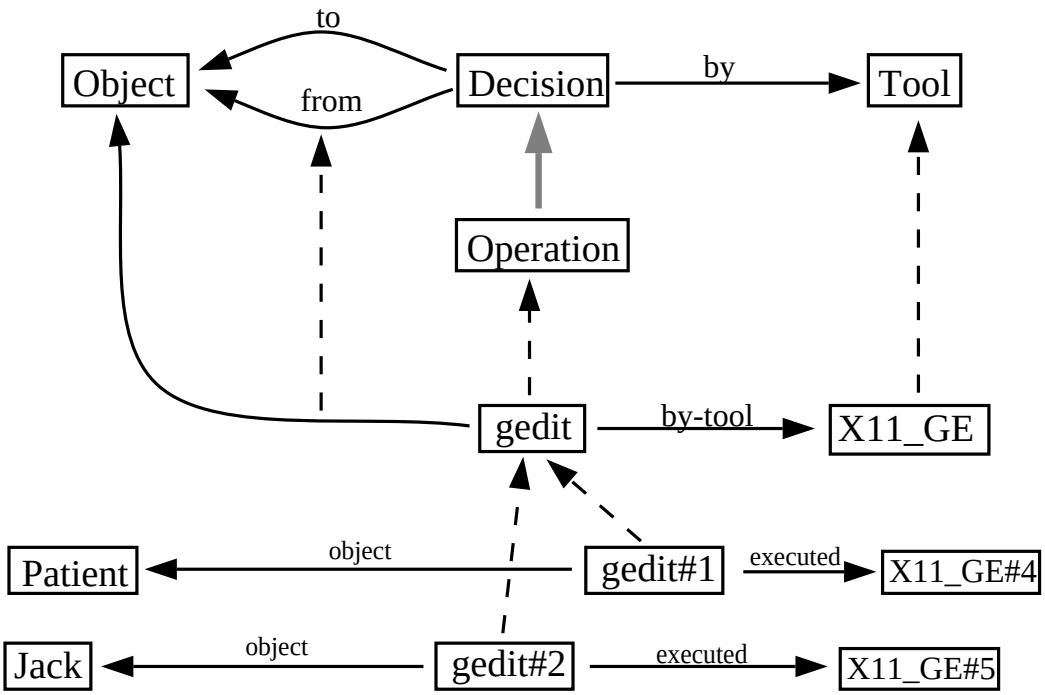

Abb. 8-8: Aufruf eines Graph-Editors

für die Liste der auf Objekte der Klasse c anwendbaren Operationen op – spezialisiert wird. Die deduktive Komponente ermöglicht also die Spezifikation von Menübeschriftungen interaktiver Werkzeuge. Die Inhalte der Menüs können auch zur Laufzeit an Veränderungen in der Objektbank, zum Beispiel die Hinzunahme eines neuen Werkzeugs samt seiner Operationen, angepaßt werden. Ein Beispiel hierzu wird in Kapitel 9 gegeben, wo auch die implementierungsnahen Aspekte der Werkzeugintegration behandelt werden.

8.7. Diskussion

Weder die automatisch generierten Trigger noch der Mechanismus der ereignisgesteuerten Auswertung ist neu. Das erste Konzept stammt aus dem Bereich der deduktiven Datenbanken (Kap. 2), das zweite aus dem Bereich der aktiven Datenbanken (Kap. 3). Der Beitrag ist vielmehr die Ausnutzung der Kombination. Durch die enge Kopplung von Prädikaten an Klassen, die in Kapitel 4 geleistet wurde, ist es möglich, die Trigger direkt als Objekte darzustellen. Ein Entwickler hat durch diesen Trick beispielsweise die Möglichkeit, sich alle Trigger anzuschauen, die an Attributen der Klasse *Patient* anknüpfen. Es ist sogar möglich, Integritätsbedingungen und deduktive Regeln *über* diese automatisch generierten Objekte zu formulieren. Ein Beispiel ist, die Kategorien *onInsert* und *onDelete* für Operationen wechselseitig auszuschließen. Man könnte es auch durch Integritätsbedingungen verbieten, gewissen Klassen Triggerattribute zuzuordnen.

Die Implementierung der Trigger, genauer: der zu ihnen gehörenden Operationen, kann in einer beliebigen Programmiersprache erfolgen. In [BM91] etwa wird über die automatische Generierung von CO2-Prozeduren aus Integritätsbedingungen berichtet. CO2 ist eine objektorientierte, persistente Erweiterung der Sprache C für das Objektbanksystem O_2. Die Abbildung funktionaler Anfragen in die gleiche Sprache beschreiben [BL91]. Mit solchen Zielsprachen erhöht sich wesentlich die praktische Bedeutung der hier vorgestellten Ansätze für kommerzielle Umgebungen.

Wenn alle Formeln in eine imperative Programmiersprache abbildbar sind, so könnte man argumentieren, daß diese Sprache ja für alle Zwecke der Anwendungsentwicklung reiche. Dieser Standpunkt wird beispielsweise von Vertretern der Datenbankprogrammiersprachen [ATKI91] ausdrücklich propagiert. Es ist jedoch Tatsache, daß ein großer Teil der Methoden in objektorientierten Datenbanken nur dem Test von Bedingungen oder der Ableitung von Daten gemäß einfacher Umformungsanweisungen gewidmet ist. Dieser Teil des Codes von programmiersprachlichen objektorientierten Datenbanken kann mit den hier vorgeschlagenen Verfahren automatisch aus deklarativen Spezifikationen generiert werden. Wegen der aus der Theorie der deduktiven Datenbanken garantierten Berechenbarkeit und Korrektheit ist der generierte Code unter dem Aspekt der *Softwarequalität* sogar höher einzustufen als handgeschriebene Programme. Auch der Aspekt der Effizienz spricht eher für automatisch generierten Code, da er wegen der Turing-Unvollständigkeit der Anfragesprache deduktiver Datenbanken in viel stärkerem Ausmaß der Optimierung unterworfen werden kann.

Nach dem oben beschriebenen Stand der Technik scheint eine zentrale Kontrolle der Triggerauswertungen besser geeignet als eine dezentrale. Die Begründung ist wie folgt.

- Bei dezentraler Kontrolle würde jede Änderung an der Menge der Integritätsbedingungen eine Re-Implementierung der betroffenen Transaktionsspezifikationen erfordern. Da der Implementierungsprozeß aber (noch) nicht automatisierbar ist, wären der Aufwand wohl so hoch, daß Änderungen an Integritätsbedingungen nur sehr selten vorkommen würden. Dies widerspricht der Zielsetzung der vorliegenden Arbeit nach einem Objektbanksystem, in dem Änderungen an Klassen gleichberechtigt zu Änderungen an den Instanzen der Klassen sein sollen.
- Bei einem dezentralen Kontrollregime gibt es den Anspruch, alle Anwendungsprogramme so zu implementieren, daß sie niemals nicht-integre Transaktionen generieren. Wäre dies nicht der Fall, so müßte man für die „unsicheren“ Anwendungsprogramme eine zentrale Kontrolle bereitstellen, die bei den „sicheren“ Programmen auszuschalten wäre.

- ▷ Integritätsbedingungen, Regeln und Sichten werden durch ihre Prädikate eng an die Klassen der Objektbank gebunden (vgl. Transformation der Prädikate der inneren deduktiven Objektbank in Kap. 4). Zudem bestehen Abhängigkeiten zwischen Regeln und Integritätsbedingungen, welche die Folgerungsprädikate deduktiver Regeln enthalten. Daher ist die zentrale Verwaltung und Wartung dieser Abhängigkeiten eher machbar als eine Verteilung auf Anwendungsprogramme.

Zur Verallgemeinerung des Operationsbegriffs wurden einzig die vorher definierten Sprachkonzepte verwandt. Ähnlich wie beim Beispiel des in O-Telos dargestellten erweiterten Entity-Relationship-Modells erweist sich die *Beschreibungsstärke* von O-Telos als Vorzug. In anderen Formalisierungen, z.B. die F-Logik [KLW90], sind Operationen eigene Arten von Termen.

Ein Manko des vorgestellten Operationsbegriffs ist die fehlende Berücksichtigung der Spezialisierung. In den programmiersprachlichen Objektbanken wird das Überladen von Operationssymbolen, d.h. die Definition des gleichen Operationsnamens für mehrere Klassen, als wesentliches Merkmal von Objektbanken angesehen. Da wir Operationen als Individualobjekte modelliert haben, muß der Name eindeutig sein. Es kommt noch hinzu, daß eine Verfeinerung von Operationen für Unterklassen nicht erklärt ist.

Die hiesige Verwendung des D.O.T.-Modells muß von seiner Verallgemeinerung CAD^o [ROSE91] zur kombinierten Teamunterstützung und Konfigurationsverwaltung unterschieden werden. Dort werden (Programm-)Dokumente und ihre abstrakten Eigenschaften verwaltet. Hier jedoch stehen die aktiven Eigenschaften und namentlich der Aufruf ihrer Operationen im Mittelpunkt des Interesses.

Alle Werkzeugprozesse haben eindeutige Objektidentifikatoren. Sie bieten sich als Adressen für die **Kommunikation** zwischen den Werkzeugen an. Diese Idee ist vom Autor in dem Objektbanksystem *ConceptBase* implementiert worden. Das folgende Kapitel ist diesem System sowie seiner Anwendung in einer heterogenen Entwurfsumgebung gewidmet.

Um die Realisierbarkeit der vorgeschlagenen Verfahren nachzuweisen, wurden sie in der Objektbank ConceptBase implementiert. ConceptBase folgt einer *Client-Server*-Architektur, wobei die kommunizierenden Werkzeuge auf miteinander verbundenen Rechnern residieren. Interessanterweise ist das gleiche Prozeßmodell wie im vorigen Kapitel anwendbar. Die Fähigkeit zur Integration heterogener Werkzeuge wird anhand des Projektes DAIDA demonstriert.

Kapitel 9: ConceptBase und seine Rolle in DAIDA

CML, der Vorgänger der Sprache Telos, wurde im Jahr 1986 zur Repräsentation von Anforderungsanalysen entwickelt [STAN86]. Etwa zur gleichen Zeit wurden die deduktiven Integritätsprüfungsmethoden [DECK86,BDM88] vorgeschlagen. In dieser Situation startete Ende 1986 das Projekt DAIDA [JMSV91] der Europäischen Gemeinschaft zur Unterstützung der Datenbankanwendungsentwicklung, an dem auch die Universität Passau beteiligt war. Die Aufgabe von Passau war die Bereitstellung einer *globalen Entwurfsdatenbank*, die alle relevanten Daten aus den verschiedenen Sprachumgebungen der Anwendungsentwicklung aufzunehmen hat.

Nach einigen Experimenten mit einer Prototypimplementierung von CML [GALL86] stellte sich heraus, daß diese den Anforderungen bezüglich der Erweiterbarkeit der Sprache nicht voll genügte, da in ihr keine Metaklassen implementiert waren. Auch die Behandlung der Attributkategorien war unterschiedlich, indem sie nicht als eigenständige Objekte sondern nur als Attribute von Klassen auftauchten.

Diese Bestandsaufnahme war die Grundlage für die Entscheidung, eine eigene Implementierung der Sprache unter dem Namen ConceptBase zu starten [JJR87,JJR88, EJJ*89,JARK91]. Der Rest des Kapitels widmet sich zuerst der Implementierung von ConceptBase. Zum Teil geht diese Implementierung über die Sprache O-Telos hinaus. Dies betrifft den eingebauten Zeitkalkül und die Textdarstellung (engl.: *frame representation*) der Objekte. Für die Realisierung der Operationen aus dem letzten Kapitel wurde eine Client-Server-Architektur entwickelt, die die Verteilung der Komponenten (Werkzeuge) in einem weltweiten Rechnernetz gestattet. Andererseits sind auch einige Konzepte, die in den Kapiteln 4 bis 8 erarbeitet wurden, noch nicht vollständig implementiert.

Im Anschluß wird über die Anwendung des Systems im Projekt DAIDA berichtet. In der Zwischenzeit wurde es in einer Reihe weiterer Projekte innerhalb und außerhalb Europas [JJM90,MR91,RD91]) eingesetzt. DAIDA ist aber für diese Arbeit am signifikantesten, da hier viele verschiedene Umgebungen integriert wurden: Zum einen wurden die heterogenen Sprachen durch Ausnutzung der Metaebene von O-Telos in ein gemeinsames

Modell überführt. Zum anderen wurden die heterogenen Werkzeuge mit Hilfe der Client-Server-Architektur von ConceptBase integriert. Anhand des Falls eines zu integrierendes neuen Werkzeuges hat sich die Erweiterbarkeit des hier propagierten Objektbanktyps erwiesen.

9.1. Die Kernarchitektur von ConceptBase

Die Implementierungsstrategie von ConceptBase ist in Kapitel 8 durch die Zweiebenenarchitektur beschrieben worden. Neue Funktionalitäten werden innerhalb der Objektbank spezifiziert. Anschließend erfolgt die Implementierung, deren Dokumentation mit Hilfe des Softwareprozeß-Datenmodells ebenfalls Teil der Objektbank wird. Diese Strategie setzt allerdings voraus, daß zumindest ein Kern der Sprache O-Telos schon vorhanden ist. Dieser Kern, der sogenannte *PropositionProcessor*, wurde vom Autor als schneller Prototyp in der Programmiersprache Prolog [BIM90] realisiert. Die Schnittstelle dieser Komponente ist durch die folgenden Prolog-Prozeduren beschrieben.

create_proposition(p)	Diese Prozedur hat als Argument ein Tupel $P(o, x, l, y, t, st)$. Es wird in die Prolog-Datenbasis als Fakt eingetragen. Zudem wird ein Index auf der zweiten Komponente aufgebaut. Das Tupel enthält über die Definition 4-1 hinausgehend zwei zusätzliche Komponenten, die Angaben über die Gültigkeitszeit machen (siehe unten).
retrieve_proposition(p)	Das Argument p ist ebenfalls ein Tupel $P(o, x, l, y, t, st)$, bei dem jedoch die Argumente auch Variablen sein dürfen. Durch Backtracking [KOWA79] werden alle Fakten der Datenbasis zurückgegeben, die auf das Eingabetupel passen.
delete_proposition(p)	Das Objekt p wird – falls vorhanden – aus der aktuellen Objektbank entfernt. Tatsächlich verändert die Prozedur nur die letzte Komponente des Objekts (siehe unten).

Die obigen Prozeduren steuern den Umgang mit einer extensionalen Objektbank. Darauf aufbauend stellt die Komponente *LiteralProcessor* eine Prozedur *prove_literal*(l) zur Verfügung, wobei l eines der Prädikate A, *In* und *Isa* darstellt (vgl. Abb. 9-1).

Die Axiome des Objektmodells sind in der Komponente *TelosAxioms* zusammengefaßt. Aus Effizienzgründen sind sie direkt in Prolog kodiert. Das in ConceptBase realisierte Objektmodell ist ein Vorläufer des in Kapitel 4 entwickelten Objektmodells. Es fehlt die strikte Definition der multiplen Generalisierung und bei Individualobjekten wird der Objektname als Identifikator benutzt.

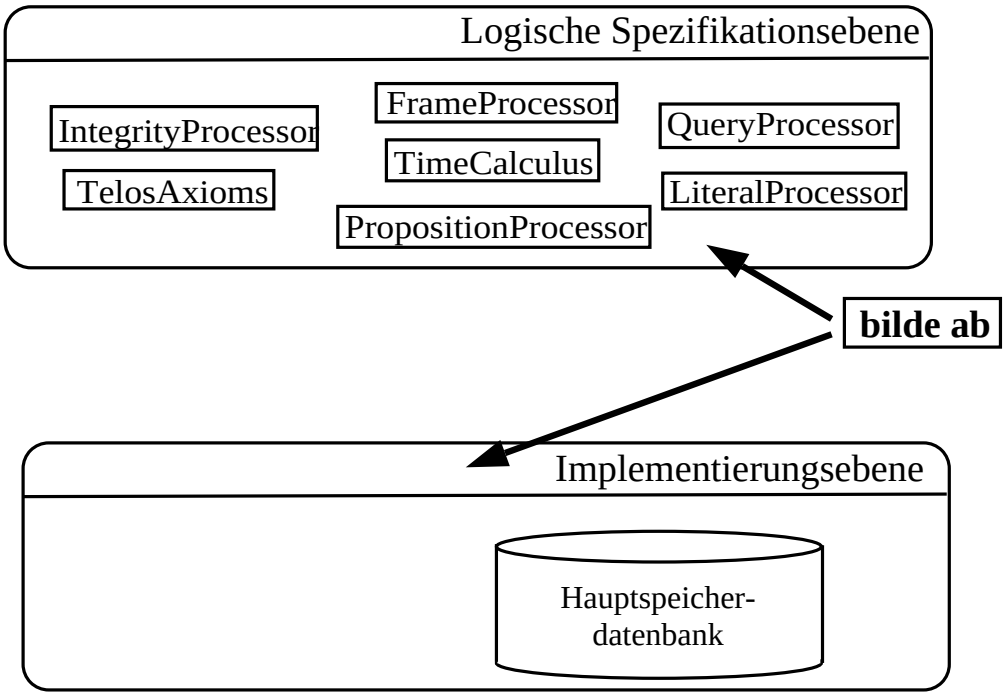

Abb. 9-1: Architektur des Kerns von ConceptBase

Über dem Kernsystem ist eine Textschnittstelle *FrameProcessor* angesiedelt, die ein Objekt zusammen mit seinen Attributen und Klassen in textueller Form repräsentiert. In dieser Form sind – unter Ausnutzung von Lemma 4-2 – alle Objektidentifikatoren durch Objektnamen ersetzt. Aus Gründen der Lesbarkeit werden Individualobjekte direkt durch ihren Namen referenziert. Ein Attribut namens l eines Objektes x wird als $x!l$ geschrieben, Klassifikationsobjekte $P(o, x, in, c)$ als x->c, und Spezialisierungsobjekte als c=>d. Die genaue Syntax ist hier nicht von Interesse (siehe [MBJK90, JARK91]). Stattdessen wird als Beispiel die Objektbank von Abbildung 4-1 angegeben.

```
Individual Person.

Individual Patient isA Person with
  attribute
    takes: Drug;
    suffers: Symptom;
    allergy: Agent
end Patient.

Individual Drug with
  attribute
    against: Symptom;
    component: Agent
end Drug.

Individual Symptom.
```

```
Individual Agent with
  attribute
    effects: Symptom
end Agent.

Individual QuasiForte in Drug.

Individual Jack in Patient with
  takes
    drug1: QuasiForte
end Jack.
```

Die Attribute eines Objektes werden in der Textform einer Attributkategorie zugeordnet, indem sie unter dem Namen dieser Kategorie geschrieben werden. Wegen Lemma 4-3 ist eine Zuordnung dieses Namens zu einem passenden Objektidentifikator eindeutig möglich. Alternativ könnte man beispielsweite für das Attribut *takes* von *Jack* schreiben:

```
Object!attribute Jack!drug1 in Patient!takes
```

9.2. Integration von zeitlicher Information

Wie schon erwähnt weist das realisierte Objektmodell zwei Zeitkomponenten [MBJK90] auf. Für ein Objekt $P(o, x, l, y, t, st)$ wird t als **Gültigkeitszeit** bezeichnet: „Die mit dem Objekt o verbundene Aussage über die Beziehung l zwischen x und y ist wahr innerhalb von t“. Die **Systemzeit** st gibt an, in welchem Zeitintervall o als Bestandteil der aktuellen Objektbank angesehen wird. Die Gültigkeitszeit wird vom Datenbankbenutzer für ein Objekt angegeben. Die Systemzeit hingegen wird immer vom Objektbanksystem zum Zeitpunkt der Einfüge- bzw. Löschtransaktion zugewiesen.

In der Literatur [SOO91] bezeichnet man Datenbanken mit Systemzeit als **Rollback-Datenbanken**. Sie erlauben den Zugriff auf alte Zustände der Datenbank, indem mit der Anfrage auch der Zeitpunkt des gewünschten Zustandes spezifiziert wird. Rollback-Datenbanken benötigen keinen physikalischen Löschbefehl. Statt der Löschung wird das Systemzeitintervall geschlossen. Das folgende Beispiel zeigt zwei Objekte, von denen das erste ein rechtsoffenes Zeitintervall hat, d.h. es ist im aktuellen Objektbankzustand, das zweite wurde am 13.10.1991 gelöscht.

$P(takes, \#Pat, takes, \#Drug, Always, [21.9.1991/17{:}34{:}11; –])$

$P(\#drug1, \#Jack, drug1, \#QF, Always, [6.10.1991/12{:}07.41; 13.10.1991.8{:}55{:}37])$

Systemzeitintervalle können ähnlich wie Intervalle auf der Zahlachse direkt miteinander verglichen werden. Dazu im Gegensatz dürfen für Gültigkeitszeitintervalle Identifikatoren eingesetzt werden. Ihre Interpretation bezieht sich nicht auf den Datenbankzustand, sondern auf die Gültigkeit der Aussage in dem modellierten Weltausschnitt. Datenbanken

mit Gültigkeitszeiten für die Daten heißen auch **historische Datenbanken**. Als Beispiel betrachten wir die Aussage, daß Rom in der Antike eine mächtige Stadt war:

$$P(\#rm, \#Rom, in, \#Mächtig, \#Antike, [17.11.1991/23{:}11{:}02; -])$$

Das Zeitintervall „Antike" wird durch den Identifikator $\#Antike$ dargestellt. Zwischen solchen Zeitintervallidentifikatoren sind 13 elementare Beziehungen [ALLE83] definiert: *before, after, meets, met_by, overlaps, overlapped_by, starts, started_by, during, contains, finishes, finished_by* und *equal*. Die erlaubten Beziehungen sind nicht-leere Teilmengen aus diesen elementaren Beziehungen, z.B. t_1 $\{before, meets\}$ t_2. Die Interpretation ist disjunktiv[9]:

$$(t_1 \ before \ t_2) \vee (t_1 \ meets \ t_2)$$

Im ganzen sind $2^{13} - 1$ Beziehungen zwischen Zeitintervallen erklärt. Ein Kalkül [ALLE83] beschreibt, wie daraus die transitive Hülle berechnet werden kann. Da der zugehörige Algorithmus jedoch NP-schwierig [VK86] ist, muß man sich in praktischen Implementierungen mit eingeschränkten Versionen begnügen. Eine Möglichkeit ist die Transformation in einen Kalkül auf den Anfangs- und Endpunkten der Zeitintervalle [VK86]. Dadurch gehen einige der im ursprünglichen Kalkül erlaubten Beziehungen verloren. Dafür arbeitet der Algorithmus jetzt in kubischer Zeit. In ConceptBase wurde eine Variante dieses Algorithmus implementiert [WENI89]. Ein Teil der transitiven Hülle wird zum Zeitpunkt der Eintragung bzw. Löschung von Zeitintervallbeziehungen berechnet. Die Idee besteht darin, Zusammenhangskomponenten von Zeitintervallen zu definieren und dort alle Beziehungen im voraus zu berechnen. Beziehungen zwischen zwei Intervallen aus verschiedenen Zusammenhangskomponenten hingegen werden erst zum Anfragezeitpunkt berechnet. Nach [WENI89] ergaben sich aus dieser Heuristik Effizienzsteigerungen bis zum Faktor 100 und mehr.

Zeitintervalle sind in ConceptBase als Objekte, ihre Beziehungen als Attribute dargestellt. Der Kalkül ist in der Komponente *TimeCalculus* enthalten, der ein Prädikat $TimeRel(t_1, r, t_2)$ zur Verfügung stellt. ConceptBase ist sowohl eine historische als auch eine Rollback-Datenbank. Solche Datenbanken heißen auch **temporal** [SOO91]. Ein Problem ist die Interaktion mit dem Datenmodell Telos. Grundsätzlich haben alle Objekte – also auch Klassen, Klassifikationsobjekte, Spezialisierungsobjekte und Attribute – eine Gültigkeitszeit. Als Folge sind alle Aussagen in der Objektbank mit einer Gültigkeitszeit behaftet. Insbesondere die Axiome werden komplizierter [KMSB89] und ineffizienter in

[9] Es besteht ein Bezug zu den Zeitintervallen in der Netzplantechnik [KOMP77]. Dort werden die Disjunktionen per Optimierung nach frühesten Endzeitpunkten aufgelöst.

der Auswertung. Als Beispiel betrachte man die temporale Version des Vererbungsaxioms A_{12} nach [WENI89]:

$$\forall\, p,x,c,d,t_1,t_2,st_2,t_3\ \ In(x,d,t_1) \wedge P(p,d,isa,c,t_2,st_2) \wedge I(t_1,t_2,t_3) \Rightarrow In(x,c,t_3)$$

Das neue Prädikat $I(t_1,t_2,t_3)$ ist wahr, wenn t_3 das größte Zeitintervall ist, daß sowohl in t_1 als auch in t_2 enthalten ist. Nun mag es sein, daß es in der Objektbank gar kein solches Zeitintervall t_3 bestimmbar ist, z.B. wenn $t_1\{before,overlaps\}\ t_2$ gilt. Auch wenn es ein Zeitintervall t_3 gibt, kann es durch eine spätere Änderung an den Zeitintervallbeziehungen wieder ungültig werden. Die Tatsache, ob ein Objekt Instanz einer Klasse ist, hängt also grundsätzlich von dem Netz aller Zeitintervallbeziehungen ab.

9.3. Der Integritätstester und Anfrageauswerter

Die Komponente *IntegrityProcessor* stellt die Operationen zum Verwalten und Testen von Integritätsbedingungen bereit. Die erste Implementierung in ConceptBase [KRÜG89] orientiert sich hauptsächlich an der Methode aus [BDM88] und adaptiert sie für die Sprache Telos. Zur Verwaltung wird das Modell aus Kapitel 8 herangezogen. Der Code der vereinfachten Formeln wird als Prolog-Term repräsentiert und von einem sehr kompakten Prolog-Interpreter ausgewertet. Eintragungen und Löschungen von Integritätsbedingungen dürfen wie jede andere Transaktion zur Laufzeit des Systems stattfinden. Eine neue Integritätsbedingung wird bei Eintragung gegen die ganze Objektbank getestet. Bei Einfügung einer Regel muß zunächst die gesamte Extension ihres Folgerungsprädikats auf Verletzung von Integritätsbedingungen getestet werden. Bei Löschen einer Regel wird entsprechend die Löschung ihrer Extension geprüft. Nach der Kategorisierung von Kapitel 8 handelt es sich um eine zentrale, reaktive Kontrolle. Eine dezentrale Kontrolle erfordert das Vorhandensein eines inkrementellen Übersetzers für Anwendungsprogramme, der bei jeder Änderung der Integritätsbedingungen eine Anpassung derjenigen Anwendungsprogramme vornimmt, die Transaktionen generieren.

In einem zweiten Schritt wurde die Komponente vom Autor auf beliebige klassenbeschränkte Formeln erweitert. Durch die striktere Form der Vererbung ist nun eine eindeutige Zuordnung aller Prädikate zu Klassen der Objektbank möglich. Ein wichtiger Vorteil ist die Steigerung der Effizienz mittels Ausnutzung der Attributklassifikation. Die Optimierung von Formelauswertungen durch Änderungsregeln (Kap. 6) wurde ebenso wie die schrittweise Vereinfachung von Metaformeln (Kap. 7) bisher noch nicht implementiert.

Der Integritätstester arbeitet in Kombination mit dem Auswerter für deduktive Regeln. Diese Regeln werden auch für die Komponente *QueryProcessor* [STAU90] von ConceptBase berücksichtigt. Die Anfragekomponente betont die Zweigesichtigkeit von Anfragen: Zum

einen sind sie deduktive Regeln mit einem eigenen Folgerungsprädikat, zum anderen sind sie Klassen der Objektbank. Eine Darstellung als Klasse macht das Erlernen einer speziellen Syntax unnötig: eine Anfrageklasse ist lediglich eine Instanz von *QueryClass*. Die Umformung dieser Objekte in deduktive Regeln geschieht in kanonischer Weise (siehe [STAU90,JS91]).

Eine neue Kategorie *parameter* dient dazu, Attribute einer Anfrageklasse zu spezialisieren bzw. einen festen Wert für sie vorzuschreiben. Bei der zugehörigen Anfrageregel bewirkt die Parametrisierung eine Verschärfung eines Prädikats bzw. die Substitution einer Variablen durch einen Wert aus der Sorte dieser Variable. In [STAU90] wird gezeigt, daß die so erhaltenen Anfragen von den ursprünglichen Anfragen subsumiert werden. Parametrisierte Anfragen haben den Vorteil, daß sowohl der Code der Anfragen als auch ihre Extensionen im vorhinein generiert werden können.

Für die Auswertung wurden zwei Varianten realisiert. Die erste arbeitet nach der Strategie *SLDNF* [KOWA79], die auch von den meisten Prolog-Interpretern benutzt wird. Die zweite Variante folgt der *MagicSet*-Strategie [BMSU86], die für rekursive deduktive Regeln mit Selektionen als die effizienteste Auswertungsmethode gilt.

Anfragen und deduktive Regeln können zu einer *Rollback*-Zeit ausgewertet werden. Dazu wird eine Anfrage Q mit einem Zeitpunkt tp attributiert, der den Suchraum auf die Objektbank

$$\mathrm{OB}_{tp} = \{P(o,x,l,y,st) \in \mathrm{OB} |\ tp\ during\ st\}$$

eingeschränkt. Die letzte Komponente der Tupel in OB_{tp} steht für die Systemzeit des Objektes, also den Zeitraum zu dem das Objekt für Anfragen sichtbar sein soll. Zugriffe auf vergangene Objektbankzustände sind damit sehr einfach implementierbar. Anfragen sowie deduktive Regeln und Integritätsbedingungen, die in Abhängigkeit von der Gültigkeitszeit formuliert sind, sind noch nicht implementiert.

9.4. Das Speichersubsystem

Der erste Prototyp von ConceptBase [JJR88] speichert Objekte zusammen mit einem Index auf der Quelle des Objektes als Prolog-Fakten ab. Diese Speicherart wurde bis zur 3. Version [JARK91] beibehalten. Wegen der mangelhaften Indizierung ist sie jedoch nur für Objektbanken mit bis zu 10000 Objekten hinreichend effizient. Es kommt hinzu, daß Persistenz, Mehrbenutzerbetrieb und Wiederherstellung (Kap. 3) kaum realisierbar sind.

Eine alternative Lösung wurde in [GALL90] entwickelt. Die Speicherung erfolgt nun mit der Hauptspeicherdatenbank GEODE [BBD*90]. Die Grundidee einer Hauptspeicherdatenbank ist die Vorstellung, daß alle Daten im (virtuellen) Hauptspeicher liegen.

Ein Hintergrundprozeß sorgt dafür, daß veränderte Seiten in konsistenter Weise auf einem persistenten Speichermedium gesichert werden. Mit der Hauptspeichervorstellung vebindet sich der direkte Zugriff auf die Daten durch ihre *Adresse*. Das System GEODE beinhaltet ein Verfahren, mit dem sogenannte persistente Adressen vergeben werden. Solche Adressen werden während der gesamten Lebenszeit der Datenbank nur einmal zugeteilt. Sie bieten sich also als Objektidentifikatoren an.

Die Datenstrukturen in GEODE werden aus Grundtypen (String, Integer, ...) und den Konstruktoren für Tupel, Mengen und Listen gebildet. Für die Objekte von ConceptBase gibt es ein festes Format: für jedes Objekt $P(o, x, l, y)$ werden unter die Adresse o die Komponenten x, l, y abgespeichert. Zusätzlich werden Zeiger zu allen Objekten angelegt, die von o aus erreichbar sind und von denen aus o erreicht werden kann [GALL90]. Diese Zeiger werden als Index für der Auswertung von Prädikaten genutzt.

Experimente zeigen, daß die neue Speicherstruktur eine Komplexitätsklasse besser ist als die alte. Für die Suche nach einem Fakt wird nur noch logarithmische anstatt linearer Zeit benötigt. Die aktuelle Implementierung ersetzt den PropositionProcessor mit den oben aufgeführten Operationen. Eine weitere Verbesserung ist zu erwarten, wenn vom Speichersystem direkt die Prädikate $In.c(x)$ und $A.p(x, y)$ angeboten werden. Die Extensionen dieser Prädikate – ohne Berücksichtigung deduktiver Regeln – sind nämlich redundant abgespeichert.

9.5. Physische Werkzeugintegration

Im vorigen Kapitel wurde die Klasse *Operation* mit ihren Attributen als Beschreibung eines abstrakten Datentypen angesehen. Das System ConceptBase wird nach diesem Muster erweitert, indem neue Operationen innerhalb der Objektbank spezifiziert und außerhalb derselben implementiert werden. Im Gegensatz zu programmiersprachlichen Objektbanken hat ConceptBase keine eingebaute Programmiersprache. Um dennoch die Erweiterbarkeit des Systems zu demonstrieren, wurde im Rahmen dieser Arbeit eine *Client-Server*-Architektur entworfen und realisiert. Der *Server* ist das Kernsystem, mit dem Objekte eingetragen, gelöscht und abgefragt werden können. *Klienten* sind beliebige Programmsysteme, die über einen Kanal mit dem Server kommunizieren. Die Programmsysteme sind innerhalb der Objektbank als Instanzen von *Tool* zusammen mit ihren Operationen dargestellt. Abbildung 9-2 zeigt die Architektur mit einigen Werkzeugen der Benutzerschnittstelle von ConceptBase.

Auch das Kernsystem ist als Instanz der Klasse *CB-Server* als Werkzeug in der

Objektbank eingetragen. Für die Kommunikation werden zwei Datenformate ausgetauscht:

$$message(s, r, op, args)$$

$$answer(s, c, v)$$

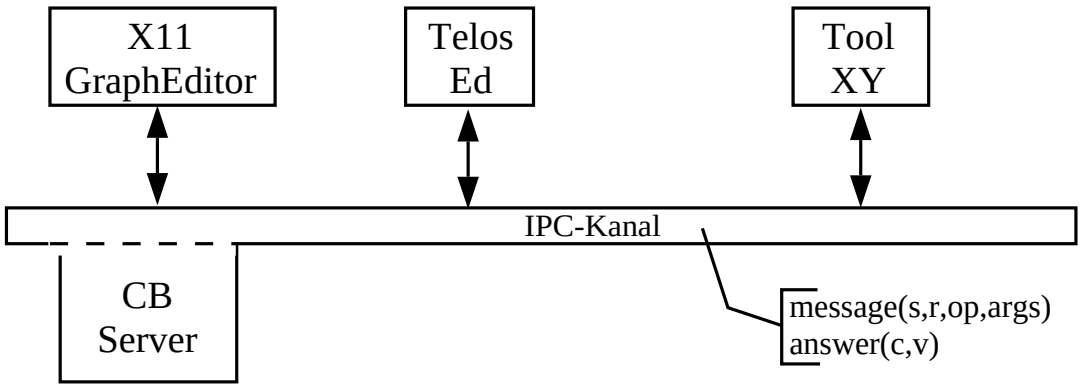

Abb. 9-2: *Client-Server*-Architektur von ConceptBase

In der ersten Datenstruktur bezeichnet s den Sender der Nachricht, r den Empfänger, op den Namen der aufgerufenen Operation und $args$ deren Argumente. Die Nachrichten werden immer zuerst an den Server verschickt, der die Rolle eines Postamtes übernimmt. Sowohl s als auch r müssen in der Objektbank als Werkzeuge bekannt sein. Falls das Kernsystem selber der Empfänger ist, so führt er die Operation mit

$$\text{CALL}(op, args)$$

aus (vgl. Kap. 8) und gibt eine Antwortdatenstruktur zurück. In ihr ist s der Identifikator des antwortenden Werkzeuges, c dient der Fehleranzeige, und v ist die eigentliche Antwort. Wenn ein anderes Werkzeug Empfänger ist, so wird die Nachricht in seine Warteschlange geschrieben. Beispiele für den Nachrichtenaustausch zwischen Werkzeugen der Benutzerschnittstelle und dem Kernsystem von ConceptBase sind im Anhang 1 enthalten.

Als Protokoll zur Übertragung findet IPC (*inter-process-communication*) [SUN90] Verwendung. Mit IPC können die Werkzeuge auch weltweit verstreut sein. Die Nachrichten werden über das gleiche Netz wie die elektronische Post ausgetauscht. In Experimenten wurde auf ein Kernsystem in Passau von Benutzerschnittstellen in Kanada und Australien zugegriffen.

9.6. Die Anwendung von ConceptBase in DAIDA

Das Projekt DAIDA [JMSV90,JMSV91] hatte als Ziel die Unterstützung der Entwicklung von Datenbankanwendungen von der ersten (informellen) Spezifikation bis zum lauffähigen Programm. Auf dem Weg dahin gibt es eine ganze Reihe unterschiedlicher Sprachen und Methoden, die zusammengenommen eine Entwickungsumgebung bilden. Die Abbildung 9-3 zeigt die anfängliche „Wasserfallarchitektur“ von DAIDA.

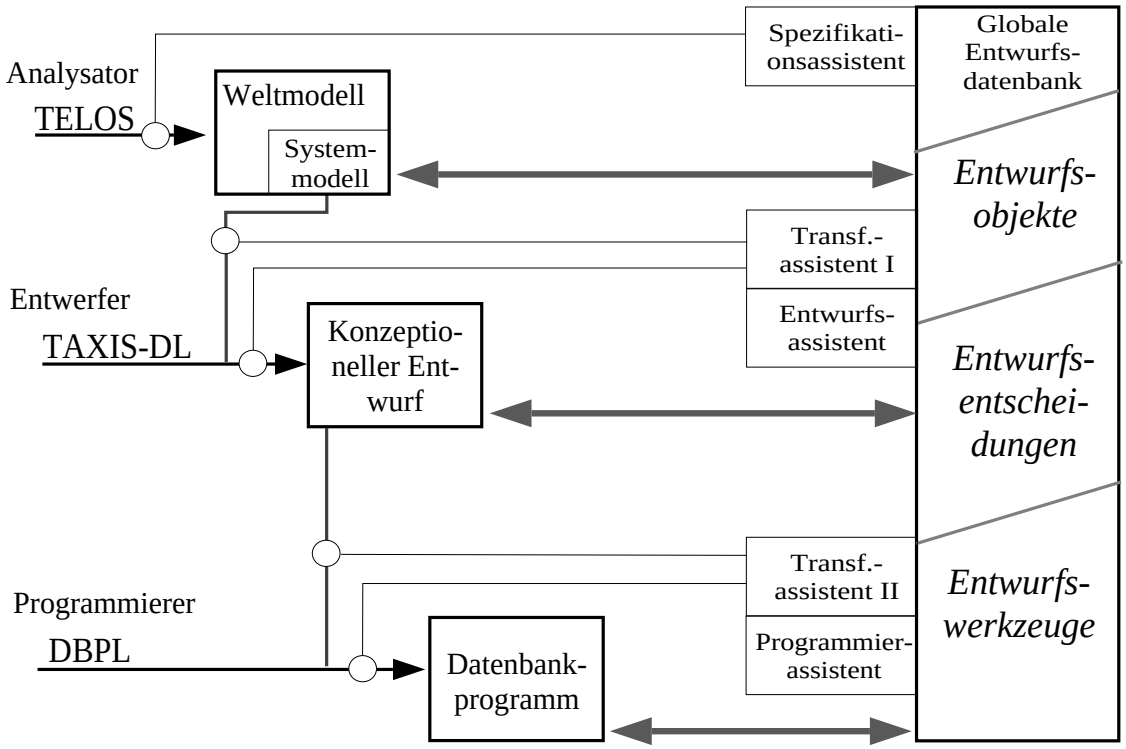

Abb. 9-3: Architektur des DAIDA-Projektes nach [JMSV91]

Der Entwurfsprozeß startet mit einem sogenannten Welt- und Systemmodell, in dem die Anforderungen an das System und seine Beziehung zur Welt in der Sprache Telos niedergeschrieben werden. Das Werkzeug für diese Teilaufgabe heißt Spezifikationsassistent. Hier wird Telos zur Wissensrepräsentation eingesetzt. Wenn die Analysephase beendet ist, wird ein Teil der Beschreibung als Systemspezifikation identifiziert und durch den Transformationsassistenten I in einen Entwurf abgebildet. Die Zielsprache ist Taxis-DL, eine Variante von Taxis, die Datenbankprozeduren mit prädikativen Vor- und Nachbedingungen spezifiziert. Nach Fertigstellung des Entwurfs wird dieser in ein lauffähiges Datenbankprogramm abgebildet (Transformationsassistent II). Das generierte Programm kann dann in einer Programmierumgebung noch weiter verfeinert werden.

Während des Prozesses entsteht eine Vielzahl unterschiedlichster Dokumente, die verwaltet werden müssen. Diese Rolle als „globale Entwurfsdatenbank“ übernimmt ConceptBase. Alle anderen Teilumgebungen stehen mit ConceptBase in Verbindung. Sie können Anfragen stellen über die abgespeicherten Dokumente (Entwurfsobjekte), über anwendbare Transformationen (Entwurfsentscheidungen) und über die Assistenten (Entwurfswerkzeuge), die die Abbildungen ausführen. Es wird erwartet, daß die Assistenten ihre Ergebnisse in der globalen Entwurfsdatenbank abspeichern. Eine zweite Funktion der Entwurfsdatenbank ist die Unterstützung der Wartung des entwickelten Programmsystems. Diese Wartung ist umso genauer möglich je detaillierter ein Dokument in der Entwurfsdatenbank abstrakt beschrieben und in Beziehung zu anderen Dokumenten gesetzt wird.

Das Software-Prozeßmodell D.O.T. dient als Schema der Entwurfsdatenbank. Die einzelnen Teilsprachen wurden innerhalb dieses speziellen Datenmodells abstrakt repräsentiert. Für die Spezifikationsebene gibt es keine Probleme, da die gleiche Sprache Telos verwendet wurde. Die Dokumente der Sprache Taxis-DL können ebenfalls fast 1:1 in ConceptBase dargestellt werden, da diese Sprache als Einschränkung von Telos auf bestimmte fest eingebaute Klassen und Attributkategorien anzusehen ist. Es bleibt die Datenbankprogrammiersprache DBPL [SEM88]. Sie basiert auf der MODULA-2 [WIRT91] und führt das Konzept der Persistenz orthogonal ein: alle Variablen in einem Modul mit dem Schlüsselwort DATABASE sind persistent. In der Entwurfsdatenbank werden DBPL-Dokumente durch ihre persistenten Variablen und die Signatur der exportierten Prozeduren repräsentiert.

Etwa in der Mitte des Projektes stellte sich heraus, daß die Transformation von dem konzeptionellen Entwurf einer Datenbankanwendung in ein lauffähiges Programm nicht in einem Schritt möglich war, sondern daß eine weitere Sprache AM der sogenannten *abstrakten Maschinen* [AGMS88] zwischenzuschalten war. Anhand dieses Fallbeispiels [WNS89,JMW*90] soll berichtet werden, wie das Schema der Entwurfsdatenbank angepaßt wurde und wie die neue Umgebung der Sprache AM mittels ConceptBase in die Gesamtumgebung integriert wurde.

Abstrakte Maschinen ähneln algebraischen Spezifikationen [WIRS86], indem sie eine Struktur aus einem Zustand, Operationen auf dem Zustand und Invarianten über die Operationen beschreiben. Allerdings ist die Semantik nicht algebraisch sondern mengentheoretisch definiert. Eine abstrakte Maschine wird dazu benutzt, um eine Softwaresystem formal zu spezifizieren. Sie kann als *Verfeinerung* einer anderen Maschine erklärt werden, indem ihr Zustand spezieller strukturiert wird und/oder ihre Operationen verfeinert werden. Im ersten Fall muß der alte Zustand aus dem neuen abbildbar sein. Im zweiten Fall

müssen alle Nachbedingungen (Folgerungen aus den Invarianten), die die alte Operation erfüllt, auch von der neuen Operation eingehalten werden.

Der Theorembeweiser *B-Tool* [ABRI86] unterstützt den Entwickler bei dem Konsistenzbeweis (Erfüllung der Invarianten) und dem Verfeinerungsbeweis (eine Maschine ist korrekte Verfeinerung einer anderen). Das B-Tool basiert auf sogenannten verallgemeinerten Substitutionen: in dem Kalkül gibt es sogenannte *Joker*, die für prädikatenlogische Teilformeln stehen. Für sich genommen ist das B-Tool ein Theorembeweiser ohne Theorien (Menge wahrer Sätze) und ohne Beweistaktiken. Für die Anwendung der abstrakten Maschinen stehen vordefinierte Theorien (Mengen, partielle Funktionen) bereit. Für einen Beweis muß der Entwickler dem B-Tool eine Taktik vorgeben. Eine Taktik bestimmt die Reihenfolge, in denen die Sätze der Theorien angewendet werden sollen, um eine Behauptung zu beweisen. Wenn für eine Behauptung keine weiteren Sätze mehr anwendbar sind, so wird sie als *Lemma* abgelegt und als unbewiesen markiert. Für das Projekt DAIDA wurden Theorien und Taktiken entwickelt, die an den Bereich der Datenbankanwendugen angepaßt sind [WNS89,BMSW90].

Das B-Tool verwaltet seine Dokumente – Theorien, Taktiken, Beweisverpflichtungen – nur innerhalb einer Sitzung. Bei Beendigung des Programms können diese Daten nur auf Dateien gesichert werden, die höchstens durch ihren Dateinamen aufeinander bezogen sind. Aus Sicht von der globalen Entwurfsdatenbank fehlt vor allem der Bezug zu der höher Sprache Taxis-DL, der unbedingt hergestellt werden mußte, um die Entwicklung von der ersten Spezifikation bis zum endgültigen Programm lückenlos zu dokumentieren.

Die folgende Textdarstellung zeigt die Modellierung von abstrakten Maschinen in ConceptBase. Die ersten beiden Attribute dienen dazu abstrakte Maschinen zu dem Entwurf in Taxis-DL in Beziehung zu setzen. Mit den anderen Attributen wird die Struktur der abstrakten Maschine aufgeschrieben.

```
Object AbstractMachine isA BtoolRule with
 dependson
      dependsonAm: AbstractMachine;
      dependsonTdl: Taxis_DL_Design
 attribute
      basicsets: AM_BAS;
      initializations: AM_INI;
      contexts: AM_CTX;
      variables: AM_VRB;
      invariants: AM_INV;
      operations: AM_OPN;
      others: AM_OTH
  constraint
      inv4ops: $ forall am/AbstractMachine
                 (exists v/AM_VRB A(am,variables,v) ==>
                  exists i/AM_INI A(am,initializations,i)) $
end AbstractMachine
```

Die Integritätsbedingung besagt, daß es einen Variablendeklarationsteil in einer abstrakten Maschine nur dann geben darf, wenn auch ein Initialisierungsteil existiert.

Zur Verwaltung der Dateien, die die Originaltexte enthalten, gibt es ein Objekt *ImplementationDesign*, das durch ein Attribut mit einer abstrakten Maschine verbunden ist.

```
Object ImplementationDesign with
  justification
      creation: ModifyImplementationDesign
  objectsource
      sourcefile: String
  objectsemantic
      itsdescription: AbstractMachine
end ImplementationDesign
```

Die erlaubten Operationen darauf werden als Instanz der Klasse *Decision* modelliert und in Beziehung zu dem unterstützenden Werkzeug gesetzt. Die vollständige Beschreibung ist im Anhang 2 enthalten.

```
Object ModifyImplementationDesign in Decision with
  part
      verification: Proof
  by
      assistant: DBPL_MAP
end ModifyImplementationDesign

Object MapToImplementationDesign in Decision
                          isa ModifyImplementationDesign with
  from
      given: ConceptualDesign
  to
      mapped: ImplementationDesign
  part
      verification: TransformationProof
  rule
      justrule: $ forall map/MapToImplementationDesign
                      des/ImplementationDesign
                  A(map,mapped,des) ==> A(des,creation,mod) $
 MapToImplementationDesign

Object DBPL_MAP in Tool isa Btool
```

Die Entscheidung `MapToImplementationDesign` ist eine Spezialisierung von `Modify-ImplementationDesign`, die ein Objekt der Klasse `ConceptualDesign` als Eingabe hat und ein Objekt der Klasse `ImplementationDesign` produziert. Wie bei der Oberklasse können Teile der Ausführung der Entscheidung aus einem Beweis mit dem Werkzeug `Btool` bestehen. In diesem Fall sind jedoch nur Beispiel von `TransformationProof` erlaubt, d.h. Beweise über die Korrektheit der Abbildung von `ConceptualDesign` nach `ImplementationDesign`. Die deduktive Regel dient der automatischen Ableitung des Attributs `creation` von `ImplementationDesign`.

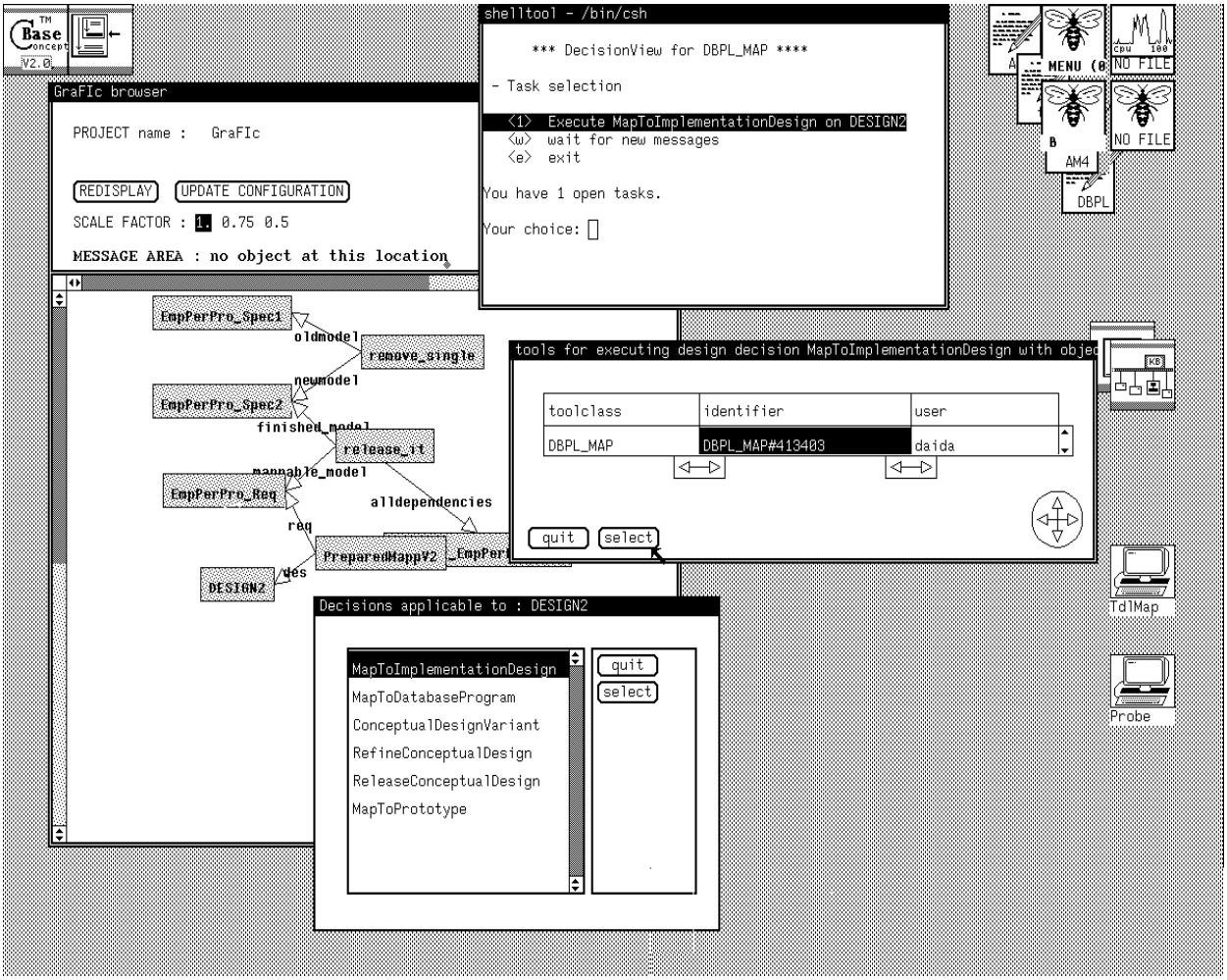

Abb. 9-4: Aktivierung der Operation eines externen Werkzeugs

Für die Funktion von ConceptBase als Entwurfsdatenbank ist nun entscheidend, daß diese Beschreibung ausreicht, um die neue Sprache und die sie unterstützenden Werkzeuge in die Gesamtumgebung zu integrieren. Abbildung 9-4 zeigt einen Zwischenstand, in dem eine erste Spezifikation *EmpPerPro_Spec1* durch mehrere Transformationsentscheidungen in einen konzeptionellen Entwurf *DESIGN2* überführt wurde. Für dieses Objekt wird eine Anfrage an ConceptBase nach den anwendbaren Operationen gestellt[10]. Aus der Antwort wird die Lösung *MapToImplementationDesign* ausgewählt. Eine weitere Anfrage an ConceptBase ergibt, daß das Werkzeug *DBPL_MAP* diese Operation ausführen kann. Zusätzlich werden die momentanen Instanzen von *DBPL_MAP* angegeben. Wie in Kapitel 9 erläutert stehen solche Instanzen für aktive Werkzeugprozesse, die mit dem Objektbankssystem verbunden sind. Der Entwickler wählt die angezeigte Instanz aus und

[10] Dies ist die Implementierung der durch deduktive Regeln spezifizierten Menüs aus Kapitel 9.

das Werkzeug erhält die Nachricht

$$message(CBserver\#413401, DBPL\_MAP\#413403,$$
$$MapToImplementationDesign, [DESIGN2]).$$

Die anderen Werkzeuge der DAIDA-Umgebung sind in der gleichen Weise beschrieben und integriert. Die Fallstudie zeigt, daß das in Kapitel 9 beschriebene Konzept der Werkzeugintegration nicht nur auf objektbanknahe Komponenten wie den Formelübersetzern und -auswertern anwendbar ist, sondern auch die Integration heterogener Programmsysteme zu leisten vermag.

9.7. Diskussion

Die Implementierung der deduktiven und objektorientierten Konzepte hatte nicht allein den Zweck der Validierung an einem praktischen System. Die frühe Fertigstellung eines ersten Prototypen [JJR88] gab auch Impulse für die Weiterentwicklung des Objektmodells O-Telos. Am Anfang war beispielsweise das Axiom zur Vererbung der Klassenzugehörigkeit mit dem Folgerungsprädikat $P(o, x, in, c)$ formuliert. Die abgeleitete Information hatte also Objektidentität. Die Generierung von Objektidentifikatoren war jedoch sehr aufwendig. In keinem Fall wurde die Objekteigenschaft der abgeleiteten Information benötigt. Diese Beobachtung war der Anstoß für das Verbot von P als Folgerungsprädikat in deduktiven Regeln.

Die feste Integration der Gültigkeitszeit in das Objektmodell von ConceptBase muß im nachhinein kritisch beurteilt werden. Der Kalkül steht außerhalb der prädikatenlogischen Axiomatisierung von O-Telos. Eine Integration mit Integritätsbedingungen und vor allem der änderungsorientierten Auswertung deduktiver Regeln ist schwierig, da verschiedenartige Auswertungsparadigmen zu koppeln wären. Bei Anwendungen wurde zudem festgestellt, daß die Möglichkeiten des Zeitkalküls fast nie genutzt wurden [KCK*91]. Diese Beobachtung wird auch von [ABD*90,MANO89a,MANO89b] gestützt, die eine Zeitkomponente als Bestandteil von Objektbanken nicht als notwendig erachten. Der Autor schlägt vor, die Gültigkeitszeit als Erweiterung des Objektbanksystems zuzulassen, jedoch aus dem Objektmodell zu entfernen. Sofern es die Anwendung erfordert, *kann* dann den Objekten über eine neue Attributkategorie *validtime* ein Zeitintervall zugeordnet werden.

Die Systemzeit ist hingegen positiv anzusehen. Die zulässigen Zeitintervalle für diese Komponente sind feste Abschnitte auf dem Zeitstrahl und werden ohne Zutun des Benutzers vom System zugewiesen. Durch Hinzufügen eines neues Axioms

$$\forall\ o, x, l, y, st\ P'(o, x, l, y, st) \Rightarrow P(o, x, l, y)$$

bleiben die restlichen Axiome von O-Telos unberührt. Bei Anfragen muß dann ein Rollback-Zeitpunkt spezifiziert werden, der den Objektbankzustand spezifiziert, auf dem sie ausgewertet werden soll. In der Implementierung der Anfragekomponente von ConceptBase ist diese Funktion bereits realisiert (vgl. Anhang 1).

Der Einsatz von ConceptBase als globale Entwurfsdatenbank in dem Projekt DAIDA ist eine Validierung der Werkzeugintegration aus Kapitel 8. Nicht nur die von den Umgebungen manipulierten heterogenen Objekte können mittels O-Telos in einem gemeinsamen Rahmen repräsentiert werden, sondern auch die Architektur des Gesamtsystems. Die Teilwerkzeuge tauchen in der Objektbank als Instanzen der Klasse *Tool* zusammen mit den von ihnen angebotenen Operationen. Mit dieser Darstellung ist es möglich, die Evolution des Gesamtsystems als Änderung der Objektbank zu modellieren. In der gleichen Weise, in der Klassen zu jeder Zeit eingetragen bzw. gelöscht werden können, können auch Werkzeuge in das Gesamtsystem eingebaut oder wieder entfernt werden. Die physische Realisierung mit dem weitverbreiteten Kommunikationsprotokoll IPC erlaubt die weltweite räumliche Verteilung der zu integrierenden Werkzeuge. Diese Eigenschaft gewinnt mit der zunehmenden Verteilung von Arbeitsgruppen in Projekten eine immer größere Bedeutung. Das Konzept der Werkzeugintegration hat die gleiche Wurzel – nämlich das D.O.T.-Softwareprozeßmodell – wie das CAD^o-Modell [ROSE91] zur Beschreibung von Konfiguration/Versionierung von Programmmodulen und zur Verteilung von Teilaufgaben an Gruppenmitglieder. Eine Kopplung der Werkzeugintegration mit CAD^o ist daher unproblematisch. Die Entwicklungsumgebungen der verteilten Gruppen können sich als Werkzeuge in eine gemeinsame Objektbank einschalten und Nachrichten austauschen.

Eine weitere Anwendung ist die Schemaintegration heterogener Datenbanken. Am Beispiel relationaler Datenbanken wurde in [KLEM91] gezeigt, daß mit O-Telos die Schemata der Teildatenbanken in ein gemeinsames Schema überführbar sind. Zudem wurden die verteilten Datenbanksysteme mittels der werkzeugintegrierenden Fähigkeiten von ConceptBase zusammengeführt: nachdem sie sich als Werkzeuge in ConceptBase eingeschaltet haben, können Anfragen und Änderungsoperationen von ConceptBase aus an die Teildatenbanken weitergeleitet werden. Wie in DAIDA kombinieren sich also die darstellenden Fähigkeiten von O-Telos mit der physischen Integration von Softwareumgebungen.

Kapitel 10: Rückblick und Ausblick

Am Schluß der Arbeit werden die zentralen Ergebnisse dieser Arbeit rückblickend zusammengefaßt. Es kann wohl nicht behauptet werden, daß das am Anfang beschriebene Ziel der deduktiven Objektbank in vollem Umfang erreicht wurde. Es wurden jedoch einige wesentliche Schritte unternommen und in einer lauffähigen deduktiven Objektbank realisiert. Im Ausblick wird ein breiterer Rahmen aufgezeigt, um sinnvolle nächste Schritte zu identifizieren.

10.1. Rückblick

Am Anfang dieser Arbeit stand die Frage, ob es eine Verschmelzung deduktiver und objektorientierter Datenbanken gibt, die mehr als nur die Summe der Teile ist. Die Antwort besteht aus einer Reihe von Ergebnissen, die in die Definition einer deduktiven Objektbank münden. Ein wichtiges Ziel war die Wiederverwendung der Verfahren aus dem Bereich der deduktiven Datenbanken.

- Mit Blick auf die Wiederverwendung deduktiver Methoden wurde aus der Wissensbeschreibungssprache Telos ein **Objektmodell O-Telos** herausgearbeitet, das zugleich ein Grenzfall des relationalen Datenmodells ist. Es besteht aus nur einer einzigen Relation, der aber eine Vielzahl zusätzlicher Axiome gegenüberstehen. Wenn man die Axiome ignoriert, so können mit O-Telos beliebige Graphen aus Knoten und Kanten beschrieben werden. In gewisser Weise ist O-Telos also die Verallgemeinerung aller anderen Objektmodelle, da in ihnen Objekte als spezielle Graphen dargestellt werden (siehe [BEER90] und Kapitel 3). Die Axiome von O-Telos definieren formal die Konzepte der Objektidentität, Klassifikation, Spezialisierung und Attributierung. Der Kern der Axiome wurde von der Ausgangsprache Telos übernommen. Hinzu tritt vor allem eine strikte Axiomatisierung der Spezialisierung von Attributen, die Mehrdeutigkeiten bei multipler Generalisierung ausschließt.
- Zwar kennt O-Telos nur eine einzige Relation in seiner extensionalen Datenbank, jedoch ermöglichen es die Axiome dieses Objektmodells, für jede Klasse eigene Prädikate zu definieren. Aus der ursprünglichen Armut an Prädikaten wird ein Reichtum: die Anzahl der Klassen schließt deren Attribute mit ein, die weit mehr Prädikate als Relationen im relationalen Datenmodell ausmachen. Durch die **Transformation** der deduktiven Regeln und Integritätsbedingungen auf diese Vielzahl von Prädikaten wird sowohl das Effizienzproblem als auch das **Stratifikationsproblem** gelöst, das für deduktive Datenbanken mit wenigen Relationen existiert. Das Ergebnis ist eine **deduktive Objektbank**. Sie vereinigt nicht nur deduktive und objektorientierte Konzepte, sondern unterstützt als Granularität der Transaktionen Änderungen

einzelner Attribute. Relational-deduktive Datenbanken hingegen sind meist auf die Granularität eines ganzen Tupels einer Relation festgelegt. Integritätsbedingungen und deduktiven Regeln sind voll mit den Axiomen des Objektmodells integriert, da diese in derselben prädikatenlogischen Sprache formuliert sind. Insbesondere wird die Vererbung der Klasenzugehörigkeit durch deduktive Regeln spezifiziert (und damit realisiert), die zusammen mit den benutzerdefinierten Regeln stratifiziert werden. Mehrdeutige Interpretationen sind dadurch ausgeschlossen.

- ▷ Mit den Axiomen wurden für Objektbanken wichtige Eigenschaften gezeigt. So kann es nie einen Verweis auf ein Objekt geben, das nicht existiert (**referentielle Integrität**). Desweiteren ist jedes Objekt notwendig Instanz der Klasse *Object*. Diese Klasse steht also exakt für die ganze extensionale Objektbank.

- ▷ Eine Neuerung von O-Telos gegenüber seinem Vorgänger Telos ist die durchgängige Unterscheidung von **Objektname** und **Objektidentifikator**. Es wurde gezeigt, daß jedes Objekt neben seinem Objektidentifikator auch durch einen Term aus Objektnamen referenzierbar ist. Die in dieser Arbeit zahlreich verwendeten Graphen sind also keine bloßen Veranschaulichungen sondern genaue Repräsentationen von Objektbankausschnitten.

- ▷ Die Axiome des Objektmodells gelten für alle Objektbanken. Es liegt also nahe, die zusätzliche Struktur von Objektbanken zur **semantischen Optimierung** von logischen Ausdrücken heranzuziehen. Als Beispiele wurden Objektidentität und die Attributklassifikation vorgeführt. Letztere führt zu einer beträchtlichen Reduktion von Triggern, die für den Test einer Transaktion auf Integrität auszuwerten sind. Praktische Experimente haben ergeben, daß fast alle Sortenprädikate, die quantifizierte Variablen an Klassen binden, eliminiert werden können. Wesentlich für die Möglichkeit der Optimierung ist die in Kapitel 4 vorgestellte enge Kopplung von logischen Formeln an die Klassen der Objektbank.

- ▷ Eine weiteres Ergebnis ist die Darstellung **komplexer Objekte als deduktive Regeln**. Anders als z.B. in [CCT90,AG91] werden keine komplexen Terme in die logische Sprache eingeführt. Stattdessen ergibt sich der Aufbau des komplexen Objektes aus den Prädikaten im Bedingungsteil einer speziellen Regel und der Verkettung derer Argumente. Antworten der Objektbank sind mittels Aggregation als komplexe Datenstruktur darstellbar und folglich von Programmiersprachen mit solchen Datenstrukturen direkt weiterverarbeitbar. Zumindest auf der Seite der Datenstrukturen entfällt also die vielgescholtene Sprachlücke *impedance mismatch*.

- ▷ Die logische Sprache von O-Telos enthält Prädikate, die einzelne Attributsbeziehungen ausdrücken. Diese feine Granularität ist positiv bei der Formelauswertung, wenn

nur diese Attribute geändert werden. Wenn allerdings alle Attribute eines Objektes in einer Transaktion geändert werden, so kann es zu unnötigen Doppelauswertungen kommen. In Kapitel 6 wurde gezeigt, daß auch dieser Nachteil gegenüber den relational-deduktiven Datenbanken nur scheinbar existiert. Schon die Einwertigkeit von Attributen genügt, um **gleichzeitige Änderungen** auf diesen Attributen zu aggregieren. Als Resultat folgt, daß die änderungsorientierte Formelauswertung mit O-Telos in keinem Fall schlechter, vielfach aber effizienter als bei relational-deduktiven Datenbanken ist.

- Das relationale Datenmodell unterscheidet streng zwischen dem Schema einer Datenbank (die erlaubten Relationen) und der eigentlichen Datenbank (die aktuellen Relationen). In O-Telos ist dieser Unterschied zwar formal mit dem Schema aus genau einer Relationen noch vorhanden, jedoch de facto aufgehoben, da Klassen als gewöhnliche Objekte die Rolle der Relationen übernehmen. Hieraus ergeben sich neue Ausdrucksmöglichkeiten für logische Formeln, nämlich Quantifizierungen über mehrere Klassenebenen (sogenannte **Metaformeln**). Eine große Teilmenge dieser Metaformeln kann durch Einsetzung der gesamten Extension eines der vorkommenden Prädikate in Formeln transformiert werden, die nur noch über eine Klassenebene quantifizieren. Das Verfahren der **schrittweisen Vereinfachung** erlaubt es, Konzepte, die sonst im Datenmodell spezifiziert werden, als Integritätsbedingungungen und deduktive Regeln effizient zu realisieren. Als Beispiel wurde eine Erweiterung des Entity-Relationship-Modells vorgeführt. Besonders deutlich werden die Vorteile bei der Definition neuer Attributkategorien wie etwa der Eindeutigkeit: die Eigenschaften bereits bestehender Kategorien können einfach durch multiple Generalisierung zusammengefaßt werden. Zu keinem Zeitpunkt wird der Rahmen der Prädikatenlogik erster Stufe verlassen.

- Ein deduktives Objektbanksystem besteht aus zwei Ebenen. In der oberen Ebene werden Objekte und ihre Eigenschaften deklarativ **spezifiziert** und in der darunterliegenden Schicht **implementiert**. Deduktive Regeln und Integritätsbedingungen sind in dieser Einordnung Spezifikationen, die **automatisch implementierbar** sind. Indem der Text der Formeln als Name eines Objektes repräsentiert wird, können sowohl die Verwaltung der Formeln und des aus ihnen generierten Codes als auch ihre änderungsorientierte Auswertung *innerhalb* der Objektbank beschrieben werden. Hierzu wird das Softwareprozeßmodell D.O.T. [JJR89a] um zwei Kategorien erweitert, denen eine „aktive“ Semantik in Form des Anstoßes von Operationen zugeordnet ist. Das in O-Telos formalisierte Modell von Operationen kann auch zur Spezifikation beliebiger Operationen herangezogen werden. Da hier Operationen nur hinsichtlich ihrer Ein- und Ausgabeargumente spezifiziert werden und das dynamische Binden

an die Implementierung (siehe Kap. 3) fehlt, ist dieses Modell kein voller Ersatz für die Methoden programmiersprachlicher Objektbanken. Es kann aber für die **Integration heterogener Werkzeuge** eingesetzt werden.

- ▷ Die voranstehenden Ergebnisse wurden für die Implementierung der deduktiven Objektbank ConceptBase eingesetzt. Die Strategie dieser Arbeit war es, theoretische Resultate möglichst früh an einem Prototypen zu validieren. Auf diese Weise fand eine wechselseitige Befruchtung zwischen Entwicklung der Theorie und praktischer Umsetzung von Teilergebnissen statt (siehe Kapitel 9). Das praktische Ergebnis dieser Arbeit ist das **Kernsystem eines lauffähigen deduktiven Objektbanksystems**. Eine Fallstudie sowie zahlreiche externe Benutzer belegen die Einsatzfähigkeit des Systems, im besonderen der Werkzeugintegration.

Die Originalität der Arbeit liegt in der vollständigen Übernahme aller Konzepte und Algorithmen der deduktiven Datenbanken. Andere Autoren gehen von einem vollentwickelten Objektmodell aus und versuchen, für diese kompliziertere Theorie deduktive Regeln (kaum Integritätsbedingungen) zu definieren. Dieser Ansatz hat den Vorteil, daß Konzepte wie Operationen und Objektkomplexität von vorneherein berücksichtigt sind. Allerdings sind die Verfahren deduktiver Datenbanken höchstens nach umfangreicher Anpassung an das Objektmodell wiederverwendbar. Je weiter das Objektmodell von dem relationalen Datenmodell entfernt ist, desto schwieriger wird die Übertragung der dort entwickelten Algorithmen. Als Beispiel sei hier die Anfragesprache von O_2 genannt.

Wegen des Umfangs des Gebiets hat der Autor einen Schwerpunkt auf Integritätsbedingungen und deduktive Regeln gelegt. Objektorientierte Sprachmuster werden innerhalb des beweistheoretischen Rahmens von [REIT84] als neue Axiome hinzugefügt. Es ist bemerkenswert, daß durch diese Axiome die ursprünglich für das relationale Datenmodell entwickelten Algorithmen noch verbessert werden. Zudem steigt wegen der Metaklassen die Ausdrucksfähigkeiten logischer Formeln.

Man könnte argumentieren, daß eine programmiersprachliche Objektbank wegen ihrer Turing-Vollständigkeit eine bessere Antwort auf die Aufgaben der Anwendungsentwicklung geben. Hiergegen ist einzuwenden, daß viele Teilaufgaben von ihrer Natur her als deduktive Regeln oder Integritätsbedingungen spezifiziert werden können. Durch Bereitstellung der Komponenten für die Übersetzung und Auswertung dieser Formeln kann der Anwendungsprogrammierer von ständig wiederkehrenden Aufgaben, wie der Programmierung von Testprozeduren zur Einhaltung der Datenbankintegrität, entlastet werden. Die Erfahrung lehrt auch, daß bei Handprogrammierung Fehler durch Vergessen von Fällen gemacht werden, oder daß Dinge getestet werden, die gar nicht getestet zu werden brauchen.

10.2. Umblick

Die Ergebnis dieser Arbeit beruht auf der Idee, deduktive Objektbanken als Spezialfall der deduktiven Datenbanken aufzufassen. Die Erweiterungen hinsichtlich der komplexen Objekte, der Metaklassen und der Operationen wurden wie in einem Baukastensystem durch Hinzunahme neuer Klassenobjekte und Formelobjekte erreicht, ohne die Begriffswelt der deduktiven Datenbanken zu verlassen. Jetzt soll über den Tellerrand der deduktiven Datenbanken geblickt werden, um diese Arbeit in einen größeren Kontext einordnen zu können.

Bezüglich der zweistufigen Architektur sind Ähnlichkeiten mit den *erweiterbaren Datenbanksysteme* [FREY87,LLPS91] festzustellen. Am Beispiel des Anfrageoptimierers eines Datenbanksystems wird argumentiert, daß eine fest implementierte Optimierungsstrategie nicht in der Lage ist, anwendungsabhängige Zugriffsverfahren und -häufigkeiten zu berücksichtigen. Als Ausweg wird die deklarative Beschreibung der Optimierungsstrategie vorgeschlagen: Eine Anfrage in SQL wird in einen Ausdruck einer erweiterten relationalen Algebra transformiert. Für solche Ausdrücke sind Äquivalenzen bekannt. [FREY87] gibt eine Sprache von Termersetzungsregeln an, die mit Anwendbarkeitsbedingungen attributiert werden können. Wenn ein Optimierer auf diese Weise beschrieben ist, so ist es leicht, die Menge der Termersetzungsregeln an die jeweilige Anwendung anzupassen, um eine möglichst effiziente Abbildung einer SQL-Anfrage in einen auszuwertenden Algebraausdruck zu erreichen. Neben den Regeln benötigt der Optimierer dann noch eine Kostenfunktion, die den bestmöglichen Algebraausdruck in einer Äquivalenzmenge beschreibt. Der Ansatz zeigt, daß auch die in dieser Arbeit nicht behandelte Abbildung einer logischen Formel in eine Algebra in einer deklarativen Sprache aufgeschrieben und inkrementell verändert werden kann. Da eine Algebra typischerweise mehr Operatoren aufweist als die Prädikatenlogik Junktoren, sind weit mehr Äquivalenzen zwischen Ausdrücken zu erwarten. Zudem können die Operatoren an die tatsächliche physischen Zugriffspfade angepaßt werden.

Die Ähnlichkeit der Optimierung der Auswertung von Integritätsbedingungen mit der Transformation von Logikprogrammen wird in [SMSB90] untersucht. Eine deduktive Datenbank wird als spezielles Logikprogramm angesehen, in dem Integritätsbedingungen in Klauselform vorliegen (vgl. auch [SK88]). Wenn man nun weiß, daß bestimmte Änderungen an der Datenbank häufig vorkommen (meist die Einfügung oder Löschung von Tupeln), so macht es Sinn, den Teil des Programms, der die Integritätsbedingungen darstellt, umzuformen. Die Technik besteht in einer Reformulierung der Klauseln, indem bestimmte Literale l in die *Metaebene* $p(l)$ erhoben werden, um sie von anderen Vorkommen von l zu unterscheiden. Bei einer Änderung auf l wird dann nur ausgewertet, ob $p(l)$ wahr ist.

Die aktiven Datenbanken [DBB*88] sind als Nachfolger der relationalen Datenbanken konzipiert worden. Sie bieten als Programmierumgebung eine Sprache von Produktionsregeln an. Wegen der Ähnlichkeit mit den Triggern zur Integritätsprüfung (Kap. 2) und zu der hier verfolgten zweistufigen Objektbankarchitektur kann eine aktive Datenbank aber auch als Implementierungsumgebung für deduktive Objektbanken fungieren. Sowohl Trigger zur Integritätsprüfung als auch zur Sichtenwartung [CW90,CW91] können automatisch aus deklarativen Darstellungen gewonnen werden. Wie die Sprache RDL1 [KMS90] zeigt, ist eine Erweiterung deduktiver Regeln zu Produktionsregeln auch eine Möglichkeit, Turing-Vollständigkeit ohne Inkaufnahme einer Sprachlücke (*impedance mismatch*) zu erreichen. Um allerdings die garantierte Terminierung deduktiver Regeln sowie die für sie entwickelten effizienten Verfahren beizubehalten, muß vom Objektbanksystem eine Trennung der Regeln in einen entscheidbaren Teil (deduktive Regeln) und einen potentiell unentscheidbaren Teil (allgemeine Regeln) vorgenommen werden.

10.3. Ausblick

Ist jetzt die endgültige Definition einer deduktiven Objektbank gefunden? Diese Frage ist sicher mit einem Nein zu beantworten. Diese Arbeit stellt zur Entwicklung von Anwendungen lediglich die Ausdrucksstärke von DATALOG$^\neg$ bereit. Dies reicht aber bekanntlich bei weitem nicht an die benötigte Turing-Vollständigkeit heran. Nach Stand der deduktiven Datenbankenforschung ist es durchaus unklar (siehe [ULLM91]), wie eine erweiterte deduktive Sprache auszusehen hat. Sollen Prädikate, die Turing-vollständig programmiert wurden in deduktiven Regeln zugelassen werden? Wenn ja, verlöre man die Terminierungsgarantie. Oder sollen die Prädikate aus den deduktiven Regeln innerhalb eines Programmes aufrufbar sein? Dann wäre eine globale Optimierung schwierig. Die Antwort auf diese Fragen sind Gegenstand aktueller und zukünftiger Forschung.

Wenn man die tatsächlich benutzten Sprachen für die Programmierung von Datenbankanwendungen auflistet, so machen die imperativen Sprachen und vor allem Cobol den überwiegenden Anteil aus. Eine deduktive Objektbank, die diese Realität ignoriert, ist sicherlich auf eine kleine Marktnische beschränkt. Von dem Impetus dieser Arbeit sind die Kopplung solcher Programme mit dem Mittel der Werkzeugintegration und die Generierung von Triggern in diesen Sprachen naheliegende Vorschläge. Die Werkzeugintegration läßt die Programmierparadigmen der Umgebungen unabhängig voneinander bestehen. Der Informationsaustausch könnte jedoch mit den komplexen Objektsichten recht eng gestaltet werden. Die Machbarkeit ist allerdings noch durch Fallstudien, die über das DAIDA-Beispiel hinausgehen, nachzuweisen. Die Generierung von Triggern in imperativen Sprachen macht deren Einbau in komplette Anwendungsprogramme möglich.

Wenig Beachtung fanden in dieser Arbeit implementierungsnahe Speicher- und Auswertungstechniken für deduktive Objektbanken. Sie sind für akzeptable Antwortzeiten jedoch unverzichtbar. Im Rahmen von ConceptBase wurde in [GALL90] eine Speichertechnik auf der Basis einer Hauptspeicherdatenbank realisiert. Eine Testumgebung für die Abbildung der logischen Ausdrücke auf eine Algebra, die auf den Speicherstrukturen operiert, steht also bereit. Die Optimierung der Algebraausdrücke mit Techniken aus [FREY87,GD87] ist eine der nächsten Herausforderungen.

Als letzter Punkt soll die Beziehung zu den Konzeptsprachen genannt werden. In dieser Arbeit wurden Anfragen mit deduktiven Regeln definiert. Sie besitzen aber auch eine natürliche Interpretation als Klassen [STAU90,JS91]: die Attribute der *Anfrageklasse* repräsentieren die Komponenten des Folgerungsprädikates der Anfrageregel, ihre Integritätsbedingungen drücken Zugehörigkeitsbedingungen zur Anfrageklasse aus. Die Antwort auf die Anfrage ist die Menge der Instanzen der Anfrageklasse. Eine solche Sicht ist schon sehr ähnlich zu den Klassen in Konzeptsprachen. Die Implementierungen der Konzeptsprachen bieten einen Klassifizierer an, der automatisch die Subsumtion zweier Konzepte (Klassen) entscheidet. Leider müssen die Sprachmittel stark eingeschränkt werden, um die Entscheidbarkeit sowie die Effizienz zu garantieren. Dennoch ist es lohnend, derart eingeschränkte Anfrageklassen gegen die Klassen der Objektbank auf Subsumtionsbeziehungen zu untersuchen. Man erhält dann eine – bezogen auf das Spezialisierungsnetz – optimale Suchraumeinschränkung für die Auswertung der Anfrage. Außerdem könnte auf diese Weise die Wiederverwendbarkeit abgespeicherter Antworten früherer Anfragen entschieden werden. In [KLEM91] werden heterogene Datenbanken mittels Anfrageklassen integriert. Da die Ausschnitte der einzelnen Datenbanken durch Anfragklassen repräsentiert werden, kann man also auch die Anfrageauswertung in dieser verteilten Umgebung mit einem Klassifizierer optimieren: Datenbanken, die nicht als Unterklasse einer gestellten Anfrage auchtauchen, brauchen nicht zugegriffen zu werden.

Kapitel 11: Literatur

[AB91] Abiteboul,S., Bonner,A. (1991). Objects and views. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Denver Colorado, 238-247.

[ABD*90] Atkinson,M., Bancilhon,F., DeWitt,D., Dittrich,K., Maier,D., Zdonik,S. (1990). The object-oriented database system manifesto. In [KNN90], 223-240.

[ABIT90] Abiteboul,S. (1990). Towards a deductive object-oriented database language. In [KNN90]; auch in *Data & Knowledge Engineering* 5, 1990, 2663-287.

[ABRI74] Abrial,J.R. (1974). Data semantics. In Klimbie, Koffeman (Hrsg.): Data Base Management, North-Holland Publ., 1-60.

[ABRI86] Abrial,J.R. (1986). An informal introduction to B. Arbeitspapier, Paris, November 1986.

[ADV90] Committee for Advanced DBMS Function (1990). Third-generation database system manifesto. Memorandum UCB/ERL M90/28, University of California.

[AG91] Abiteboul,S., Grumbach,S. (1991). A rule-based language with functions and sets. *ACM Transactions on Database Systems* 16(1), März 1991, 1-30.

[AGMS88] Abrial,J.R., Gardiner,P., Morgan,C., Spivey,M. (1988). Abstract machines: part 1 - part 4. Arbeitspapiere, 26 Rue des Plantes, Paris 75014, Juni 1988.

[AH87] Abiteboul,S., Hull,R. (1987). IFO: a formal semantic database model. *ACM Transactions on Database Systems* 12(4), Dezember 1987, 525-565.

[AK89] Abiteboul,S., Kanellakis,P.C. (1989). Object identity as a query language primitive. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Portland, Oregon, 159-173.

[ALLE83] Allen,J. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM* 26(11), 823-843.

[AM91] Ahad,R., McLeod,D. (1991). A performance optimization technique for an object-oriented functional data model. *Information Systems* 16(2), 103-123.

[ATKI91] Atkinson,M. (1991). A vision of persistent systems. In [DKM91], 453-459.

[BBMR89] Borgida,A., Brachman,R.J., McGuinness,D.L., Resnick,L.A. (1989). CLASSIC: a structural data model for objects. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Portland, Oregon, 58-67.

[BB*90] Brodie, M., Bancilhon,F. et al. (1990). Next generation database management systems technology. In [KNN90], 335-348.

[BBB*88] Bancilhon,F., Barbedette,G., Benzaken,V. et al. (1988). The design and implementation of O_2, an object-oriented database system. In [DITT88], 1-22.

[BBD*90] Bellosta,M.J., Bessede,A., Darrieumerlou,C., Gruber,O. (1990). GEODE: concepts and facilities. Technischer Bericht, INRIA, Rocquencourt, Frankreich, 1990.

[BBH*90] Baader,F., Bürckert,H.-J., Hollunder,B., Nutt,W., Siekmann,J.H. (1990). Concept logics. In Lloyd (Hrsg.) Computational Logic. Symposium Proceedings, *Basic Research Series*, Springer Verlag, November 1990, 177-201.

[BCD89] Bancilhon,F., Cluet,S., Delobel,C. (1989). A query language for the O_2 object-oriented database system. Rapport Technique Altaïr 35-89.

[BDM88] Bry,F., Decker,H., Manthey,R. (1988). A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. *Proc. EDB-T'88*, Venedig, 488-505.

[BEER90] Beeri,C. (1990). A formal approach to object-oriented databases. *Data & Knowledge Engineering* 5, 353-382.

[BGN89] Beck,H.W., Gala,S.K., Navathe,S.B. (1989). Classification as a query processing technique in the CANDIDE semantic data model. *Proc. 5th Int. Conf. on Data Engineering*, 1989, 572-581.

[BIM90] BIM (1990). ProLog by BIM - 3.0. Vol. I: reference manual, Vol. II: user manual. BIM sa/nv, Everberg, Belgien.

[BL84] Brachman,R.J., Levesque,H.J. (1984). The tractability of subsumption in frame-based description languages. *Proc. AAAI*, 1984, 34-37.

[BL86] Brachman,R.J., Levesque,H.J. (1986). What makes a knowledge base knowledgable? A view of databases from the knowledge level. *Proc. First Int. Conf. on Expert Database Systems*, Menlo Park, Kalifornien, 69-78.

[BL91] Boucelma,O., Le Maitre,J. (1991). An extensible functional query language for an object-oriented database system. In [DKM91].

[BM90] Blakeley,J.A., Martin,N.L. (1990). Join index, materialized view, and hybrid-hash join: a performance analysis. *Proc. 6th Int. Conf. on Data Engineering*, Los Angeles, 256-263.

[BM91] Bouzeghoub,M., Métais,E. (1991). Semantic modeling of object oriented databases. *Proc. VLDB'91*, Barcelona, Spanien, 3-14.

[BMSU86] Bancilhon,F., Maier,D., Sagiv,Y., Ullman,J. (1986). Magic sets and other strange ways to implement logic programs. *Proc. 5th ACM Symp. Principles of Database Systems (PODS)*, 1-15.

[BMSW90] Borgida,A., Mertikas,M., Schwidt,J.W., Wetzel,I. (1990). Specification and refinement of database applications. Technischer Bericht ESPRIT 892 (DAIDA), Universität Hamburg.

[BORG85] Borgida,A. (1985). Features of languages for the development of information systems at the conceptual level. *IEEE Software* 2(1), Januar 1985, 63-72.

[BORG92] Borgida,A. (1992). From types to frames: natural semantics specifications for description logics. Erscheint in: *Int. J. Intelligent and Cooperative Information Systems* 1(1), 1992.

[BR86] Bancilhon,F., Ramakrishnan,R. (1986). An amateur's introduction to recursive query processing strategies. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Washington D.C., 16-52.

[BRAC83] Brachman,R.J. (1983). What IS-A is and isn't: an analysis of taxonomic links in semantic networks. *IEEE Computer* 16(10), Oktober 1983, 30-36.

[BRUY90] Bruynooghe,M. (Hrsg.,1990). Proceedings of the Second Workshop on Meta-programming in logic (META-90). K.U. Leuven, Department of Computer Science, Belgien, April 1990.

[BRY88] Bry,F. (1988). Logical rewritings for improving the evaluation of quantified queries. *Proc. 2nd Int. Symp. on Mathematical Fundamentals of Data Base Theory*, Visegrád, Ungarn.

[BRY90] Bry,F. (1990). Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data & Knowledge Engineering* 5, 1990, 289-312.

[BS85] Brachman,R.J. Schmolze,J.G. (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science* 9(2), April 1985, 171-216.

[CCC*90] Cacace,F., Ceri,S., Crespi-Reghizzi,S., Tanca,L., Zicari,R. (1990). Integrating object-oriented data modeling with a rule-based programming paradigm. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Atlantic City, New Jersey, 225-236.

[CCI87] Crete Computer Institute (1987). CML: an informal introduction. Arbeitspapier, CCI-FORTH, Heraklion, Griechenland, Februar 1987.

[CCT90] Ceri,S., Cacace,F., Tanca,L. (1990). Object orientation and logic programming for databases: a season's flirt or a long-term marriage? In Schmidt, Stogny (Hrsg.): Next Generation Information System Technology, *LNCS 504*, Springer-Verlag, 1990, 124-143.

[CD91] Cluet,S., Delobel,C. (1991). Towards a unification of rewrite based optimization techniques for object-oriented queries. *Proc. VII'eme journées Bases de Données Advancées*, Lyon, Frankreich.

[CGM90] Chakravarthy,U.S., Grant,J., Minker,J. (1990). Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems* 15(2), Juni 1990, 162-207.

[CGT90] Ceri,S., Gottlob,G., Tanca,L. (1990). Logic programming and databases. Springer-Verlag.

[CODD70] Codd,E.F. (1970). A relational model for large shared data banks. *Comm. of the ACM* 13(6), Juni 1970, 377-387.

[CODD79] Codd,E.F. (1979). Extending the relational model to capture more meaning. *ACM Transactions on Database Systems* 4(4), Dezember 1979, 397-434.

[COIN87] Cointe,P. (1987). Metaclasses are first class: the ObjVlisp model. *OOPSLA'87 Conference Proceedings, Special Issue of SIGPLAN Notices* 22(12), Dezember 1987, 156-167.

[CHEN76] Chen,P.P-S. (1976). The entity-relationship model: toward a unified view of data. *ACM Transactions on Database Systems* 1(1), März 1976, 9-36.

[CKPR73] Colmerauer,A., Kanoui,H., Persero,R., Roussel,P. (1973). Un systeme de communication homme-machine en francais. Forschungsbericht, Groupe Intelligence Artificielle, Université Aix-Marseille II, Frankreich, 1973.

[CKW89] Chen,W., Kifer,M., Warren,D.S. (1989). HiLog as a platform for database languages. *Proc. 2nd Int. Workshop on Database Programming Languages*, Gleneden Beach, Oregon, Juni 1989, 315-329.

[CW85] Cardelli.L., Wegner,P. (1985). On understanding types, data abstraction, and polymorphism. *Computing Surveys* 17(4), Dezember 1985, 471-522.

[CW90] Ceri,S., Widom,J. (1990). Deriving production rules for constraint maintenance. *Proc. 16th VLDB Conf.*, Brisbane, Australien, 566-577.

[CW91] Ceri,S., Widom,J. (1991). Deriving production rules for incremental view maintenance. *Proc. 17th VLDB Conf.*, Barcelona, Spanien, 577-589.

[DATE90] Date,C.J. (1990). An introduction to database systems. Band I, 5. Ausgabe, Addison-Wesley, 1990.

[DBB*88] Dayal,U., Blaustein,B., Buchmann,A., Chakravarthy,U., Hsu,M., Ledin,R., MacCarthy,D., Rosenthal,A., Sarin,S., Carey,M.J. Livny,M., Jauhari,R. (1988). The HiPAC project: combining active databases and timing constraints. *SIGMOD Record* 17(1), März 1988, 51-70.

[DBM88] Dayal,U., Buchmann,A., McCarthy,D. (1988). Rules are objects too: a knowledge model for an active object-oriented database system. In [DITT88], 129-143.

[DD86] Dittrich,K., Dayal,U. (Hrsg.,1986). Proceedings of the 1986 international workshop on object-oriented database systems. IEEE Computer Society Press, Washington, D.C.

[DD88] Duhl,J., Damon,C. (1988). A performance comparison of object and relational databases using the Sun benchmark. *OOPSLA'88 Conference Proceedings*, 153-163.

[DECK86] Decker,H. (1986). Integrity enforcement on deductive databases. *Proc. First Int. Conf. on Expert Database Systems*, Menlo Park, Kalifornien, 381-395.

[DECK91] Decker,H. (1991). On the declarative, operational and procedural semantics of disjunctive computational theories. *Proc. 2nd Int. Workshop on the Deductive Approach to Information Systems and Database*, Report LSI/91/30, Universitat Politecnica de Catalunya, Spanien, 149-173.

[DGG90] Dittrich,K.R., Geppert,A., Goebel,V. (1990). The data definition language of NO^2. Report ITHACA.Unizh.90.X.4#1, Universität Zürich.

[DPS86] Deppisch,U., Paul,H.-B., Schek,H.-J. (1986). A storage system for complex objects. In [DD86], 183-195.

[DITT88] Dittrich,K. (Hrsg.,1988). Advances in object-oriented database systems – 2nd international workshop on object-oriented database systems. *LNCS* 334, Springer-Verlag.

[DKM91] Delobel,C., Kifer,M., Masunaga,Y. (Hrsg.,1991). Deductive and object-oriented databases – Proceedings 2nd international conference (DOOD'91). *LNCS* 566, Springer-Verlag.

[DW83] Dilger,W., Womann,W. (1983). Semantic networks as abstract data types. *Proc. 8th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, Karlsruhe, August 1983, 321-324.

[DW89] Das,S.K., Williams,M.H. (1989). Integrity checking methods in deductive databases: a comparative evaluation. In Williams (Hrsg.): Proceedings of the Seventh British National Conference on Databases, Cambridge University Press, 1989, 85-115.

[EJJ*89] Eherer,S., Jarke,M., Jeusfeld,M., Miethsam,A., Rose,T. (1989). A KBMS for database software evolution: ConceptBase V2.0 user manual. Report MIP-8936, Universität Passau.

[EM85] Ehrig,H., Mahr,B. (1985). Fundamentals of algebraic specification 1. *EATCS Monographs on Theor. Comp. Science* 6, Springer-Verlag.

[ER91] Eick,C.F., Raupp,T. (1991). Towards a formal semantics and inference rules for conceptual data models. *Data & Knowledge Engineering* 6(4), Juli 1991, 297-317.

[FERB89] Ferber,J. (1989). Computational reflection in class based object oriented languages. *OOPSLA'89 Conference Proceedings, Special Issue of SIGPLAN Notices* 24(10), Oktober 1989, 317-326.

[FREY87] Freytag,J.C. (1987). A ruled-based view of query optimization. *Proc. ACM-SIGMOD Int. Conf. on Management of Data 1987*, 173-180.

[GALL86] Gallagher,J. (1986). Overall design of CML support system. Arbeitspapier, ESPRIT-Projekt 107 (LOKI), SCS Technische Automation und Systeme GmbH, Hamburg.

[GALL90] Gallersdörfer,R. (1990). Realisierung einer deduktiven Objektbank durch abstrakte Datentypen. Diplomarbeit, Universität Passau.

[GD87] Graefe,G., DeWitt,D.J. (1987). The EXODUS optimizer generator. *Proc. ACM-SIGMOD Int. Conf. on Management of Data 1987*, 160-172.

[GEBH87] Gebhardt,F. (1987). Semantisches Wissen in Datenbanken - ein Literaturbericht. *Informatik-Spektrum* 10(2), April 1987, 79-98.

[GKS90] Gottlob,G., Kappel,G., Schrefl,M. (1990). Semantics of object-oriented data models – the evolving algebra approach. In Schmidt, Stogny (Hrsg.): Next Generation Information System Technology, *LNCS 504*, Springer-Verlag, 1990, 144-160.

[GM78] Gallaire,H., Minker,J. (Hrsg.,1978). Logic and data bases. Plenum Press.

[GPvG90] Gyssens,M., Paredaens,J., van Gucht,D. (1990). A graph-oriented object model for database end-user interfaces. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Atlantic City, New Jersey, 24-33.

[GR83] Goldberg,A., Robson,D. (1983). Smalltalk-80: the language and its implementation. Addison-Wesley.

[GRAE90] Graefe,G. (1990). Volcano, an extensible and parallel query evaluation system. Report CU-CS-481-90, University of Colorado at Boulder, Juli 1990.

[GREE84] Greenspan,S. (1984). Requirements modeling: the use of knowledge representation techniques for requirements specification. Ph.D. Dissertation, Dept. Computer Science, University of Toronto, Ontario.

[GTW78] Goguen,J.A., Thatcher,J.W., Wagner,E.G. (1978). An initial algebra approach to the specification, correctness, and implementation of abstract data types. In Yeh (Hrsg.): Current Trends in Programming Methodology, Vol. 4: Data Structuring, Prentice-Hall, 80-149.

[GUTT75] Guttag,J.V. (1975). The specification and application to programming of abstract data types. Ph.D. thesis, University of Toronto, Dept. of Computer Science, Technical Report CSRG-59.

[GV89] Gardarin,G., Valduriez,P. (1989). Relational databases and knowledge bases. Addison-Wesley.

[HB89] Hayes,F., Baran,N. (1989). A guide to GUIs – your complete guide to 12 state-of-the-art graphical user interface. *BYTE* 14(7), Juli 1989.

[HI85] Hsu,A., Imielinski,T. (1985). Integrity checking for multiple updates. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, 152-168.

[HK87a] Hudson,S.E., King,R. (1987). Object-oriented database support for software environments. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, 491-503.

[HK87b] Hull,R., King,R. (1987). Semantic database modeling: survey, applications, and research issues. *ACM Computing Surveys* 19(3), September 1987, 201-260.

[HK89] Hudson,S.E., King,R. (1989). Cactis: a self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Transactions on Database Systems* 14(3), September 1981, 291-321.

[HM88] Hübel,C., Mitschang,B. (1988). Object-Orientation within the PRIMA-NDBS. In [DITT88], 98-103.

[HMS91] Härder,T., Mitschang,B., Schöning,H. (1991). Query processing for complex objects. *Data & Knowledge Engineering* 6, 1991.

[HY90] Hull,R., Yoshikawa,M. (1990). ILOG: declarative creation and manipulation of object identifiers. *Proc. 16th VLDB Conf.*, Brisbane, Australien, 455-468.

[IBM78] IBM Corporation (1978). Information management system - virtual storage general information manual. IBM Form No. GH20-1260.

[ITS90] Illarramendi,A., Topaloglou,T., Sbattella,L. (1990). Query optimization for KBMSs: temporal, syntactic and semantic transformations (revised). Report KRR-TR-89-10, University of Toronto, Dept. of Computer Science, Oktober 1990.

[JACK90] Jackson, M.S. (1990). Beyond relational databases. *Information and Software Technology* 32(4), Mai 1990, 258-265.

[JARK84] Jarke,M. (1984). External semantic query simplification: a graph-theoretic appoach and its implementation in Prolog. *Proc. 1st Int. Workshop Expert Database Systems*, Kiawah Island, South Carolina, 467-482.

[JARK91] Jarke,M. (Hrsg.,1991). ConceptBase V3.0 user manual. Report MIP-9106, Universität Passau.

[JCV84] Jarke,M., Clifford,J., Vassiliou,Y. (1984). An optimizing Prolog front end to a relational query system. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, 1984, 296-306.

[JGF*88] Jagannathan,D., Guck,R.L., Fritchman,B.L., Thompson,J.P., Tolbert,D.M. (1988). SIM: a database system based on the semantic data model. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Chicago, Illinois, 46-55.

[JJ89] Jarke,M., Jeusfeld,M. (1989). Rule representation and management in ConceptBase. *SIGMOD Record* 18(3), September 1989, 46-51

[JJ91] Jeusfeld,M., Jarke,M. (1991). From relational to object-oriented integrity simplification. In [DKM91], 460-477; auch als *Aachener Informatik-Berichte* 91-19, RWTH Aachen.

[JJM90] Jarke,M., Jeusfeld,M., Miethsam,A. (1990). Redundancy in integrity management for deductive object bases: a preliminary investigation. ESPRIT BRA Compulog deliverable D.2.1.b, Universität Passau.

[JJR87] Jarke,M., Jeusfeld,M., Rose,T. (1987). A global KBMS for database software evolution: design and development strategy. Report MIP-8722, Universität Passau.

[JJR88] Jarke,M., Jeusfeld,M., Rose,T. (1988). A global KBMS for database software evolution: documentation of first ConceptBase prototype. Report MIP-8819, Universität Passau.

[JJR89a] Jarke,M., Jeusfeld,M., Rose,T. (1989). A software process data model for knowledge engineering in information systems. *Information Systems* 15(1), 1990, 85-116; auch als Report MIP-8910, Universität Passau.

[JJR89b] Jarke,M., Jeusfeld,M., Rose,T. (1989). Software process modeling as a strategy for KBMS implementation. In [KNN90], 531-552; auch als Report MIP-8933, Universität Passau.

[JK84] Jarke,M., Koch,J. (1984). Query optimization in database systems. *Computing Surveys* 16(2), 1984.

[JK90] Jeusfeld,M., Krüger,E. (1990). Deductive integrity maintenance in an object-oriented setting. Report MIP-9013, Universität Passau.

[JMSV90] Jarke,M., Mylopoulos,J., Schmidt,J.W., Vassiliou,Y. (1990). Information systems development as knowledge engineering: a review of the DAIDA project. *Programirovanie* 17(1), Moskau, Akademie der Wissenschaften, UdSSR, 3-33; auch als Report MIP-9010, Universität Passau.

[JMSV91] Jarke,M., Mylopoulos,J., Schmidt,J.W., Vassiliou,Y. (1991). DAIDA: an environment for evolving information systems. Erscheint in *ACM Trans. Information Systems*; auch als *Aachener Informatik-Berichte* 91-18, RWTH Aachen.

[JMW*90] Jeusfeld,M., Mertikas,M., Wetzel,I., Jarke,M., Schmidt,J.W. (1990). Database application development as an object modeling activity. *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, Australien, 1990, 442-454; auch als Report MIP-9006, Universität Passau.

[JR88] Jarke,M., Rose,T. (1988). Managing knowledge about information system evolution. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Chicago, Illinois, 303-311.

[JS89] Jarke,M., Simon,E. (1989). Integrity control in data and knowledge bases: a survey. ESPRIT BRA Compulog deliverable D.2.1.a, Universität Passau.

[JS91] Jeusfeld,M., Staudt,M. (1991). Query optimization in deductive object bases. Erscheint in Freytag, Vossen, Maier (Hrsg.): Query Processing for Advanced Database Applications, Morgan Kaufmann, 1992; auch als *Aachener Informatik-Berichte* 91-26, RWTH Aachen.

[KBC*88] Kim,W., Ballou,N., Chou,H.-T., Garza,J.F., Woelk,D., Banarjee,J. (1988). Integrating an object-oriented programming system with a database system. *OOPSLA'88 Conference Proceedings*, 142-152.

[KBG*91] Kim,W., Ballou,N., Garza,J.F., Woelk,D. (1991). A distributed object-oriented database system supporting shared and private databases. *ACM Trans. Database Systems* 9(1), Januar 1991, 31-51.

[KCK*91] Kramer,B.M., Chaudhri,V.K., Koubarakis,M., Topaloglou,T., Wang,H., Mylopoulos,J. (1991). Implementing Telos. *SIGART Bulletin* 2(3), Juni 1991, 77-83.

[KHOS90] Khoshafian,S. (1990). Insight into object-oriented databases. *Information and Software Technology* 32(4), Mai 1990, 274-289.

[KL89] Kifer,M., Lausen,G. (1989). F-Logic: a higher-order language for reasoning about objects, inheritance, and scheme. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Portland, Oregon, 134-146.

[KLEM91] Klemann,A. (1991). Schemaintegration relationaler Datenbanken. Diplomarbeit, Universität Passau.

[KLW90] Kifer,M., Lausen,G., Wu,J. (1990). Logical foundations of object-oriented and frame-based languages. Reihe Informatik 3/1990, Universität Mannheim.

[KM90a] Kemper,A., Moerkotte,G. (1990). Access support in object bases. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Atlantic City, New Jersey, 364-374.

[KM90b] Kakas,A,C., Mancarella,P. (1990). Database updates through abduction. *Proc. 16th VLDB Conf.*, Brisbane, Australien, 650-661.

[KMS90] Kiernan,G., de Maindreville,C., Simon,E. (1990). Making deductive database a practical technology: a step forward. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Atlantic City, New Jersey, 237-246.

[KMSB89] Koubarakis,M., Mylopoulos,J., Stanley, M., Borgida,A. (1989). Telos: features and formalization. Technical Report KR-89-04, University of Toronto, Ontario.

[KNN90] Kim,W. Nicolas,J.-M., Nishio, M. (Hrsg.,1990). Deductive and object-oriented databases, North Holland.

[KOMP77] Kompenhans,K. (1977) Netzplantechnik und Transplantechnik. Peter Hanstein-Verlag, Köln.

[KOWA79] Kowalski,R.A. (1979). Logic for problem solving. North-Holland, 1979.

[KRÜG89] Krüger,E. (1989). Integritätsprüfung in deduktiven Objektbanken am Beispiel von ConceptBase. Diplomarbeit, Universität Passau.

[KÜCH91] Küchenhoff,V. (1991). On the efficient computation of the difference between consecutive database states. In [DKM91], 478-502.

[LHW90] Lyngbæk,P., Wilkinson,K., Hasan,W. (1990). The Iris kernel architecture. *Proc. EDBT'90*, Venedig, 348-362.

[LIPE88] Lipeck,U.W. (1988). Transformation of dynamic integrity constraints into transaction specifications. *Proc. 2nd Int. Conf. on Database Theory*, *LNCS* 326, Springer-Verlag, 322-337.

[LLOY90] Lloyd,J.W. (Hrsg.,1990). Computational logic. Springer-Verlag.

[LLPS91] Lohmann,G.M., Lindsay,B., Piradesh,H., Schiefer,K.B. (1991). Extensions to Starburst: objects, types, functions, and rules. *Comm. ACM* 34(10), 94-109.

[LRV88] Lécluse,C., Richard,P., Velez,F. (1988). O_2, an object-oriented data model. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Chicago, Illinois, 424-433.

[LST86] Lloyd,J.W., Sonnenberg,E.A., Topor,R.W. (1986). Integrity constraint checking in stratified databases. Technical Report 86/5, Department of Computer Science, University of Melbourne.

[LT84] Lloyd,J.W., Topor,R.W. (1984). Making Prolog more expressive. *Journal Logic Programming* 1(3), 225-240.

[LT85] Lloyd,J.W., Topor,R.W. (1985). A basis for deductive database systems. *Journal Logic Programming* 2(2), 93-109.

[LV90] Laenens,E., Vermeir,D. (1990). A fixpoint semantics for ordered logic. *Journal Logic Computat.* 1(2), 159-185.

[MAIE86] Maier,D. (1986). Why object-oriented databases can succeed where others have failed. In [DD86], 227-227.

[MANO89a] Manola,F. (1989). An evaluation of object-oriented DBMS developments. Technical Report TR-0066-10-89-165, GTE Laboratories, Waltham, MA.

[MANO89b] Manola,F. (1989). Applications of object-oriented database technology in knowledge-based integrated information systems. Report, GTE Labaratories, Waltham, Mass.

[MANT90] Manthey,R. (1990). Declarative languages – paradigm of the past or challange of the future? In Schmidt, Stogny (Hrsg.): Next Generation Information System Technology, *LNCS 504*, Springer-Verlag, 1990, 1-16.

[MBJK90] Mylopoulos,J., Borgida,A., Jarke,M., Koubarakis,M. (1990). Telos: a language for representing knowledge about information systems. *ACM Trans. Information Systems* 8(4), Oktober 1990, 325-362.

[MBW80] Mylopoulos,J., Bernstein,P.A., Wong,H.K.T. (1980). A language facility for designing database-intensive applications. *ACM Trans. Database Systems* 5(2), Juni 1980, 185-207.

[MGN89] Manthey,R., Gallaire,H., Nicolas,J.-M. (1989). Can we reach a uniform paradigm for deductive query evaluation? In Brauer, Freska (Hrsg.): Wissensbasierte Systeme, Spinger-Verlag, 17-32.

[MJJG91] Miethsam,A., Jeusfeld,M., Jarke,M., Gocek,M. (1991). Constrained abduction as a basis for configuration management and reusability. Arbeitspapier, RWTH Aachen, September 1991.

[ML90] Moerkotte,G., Lockemann,P.C. (1990). Reactive consistency control in deductive databases. Interner Bericht Nr. 3/90, Universität Karlsruhe, Fakultät für Informatik, Januar 1990.

[MR91] Mylopoulos,J., Rose,T. (1991). Case-based reuse for information system development – the Techne project. *Proc. Int. Workshop on Software Reusability*, Dortmund, 3.-5. Juli 1991.

[MS86] Maier,D., Stein,J. (1986). Indexing in an object-oriented DBMS. In [DD86], 171-182.

[MS87] Maier,D., Stein,J. (1987). Development and implementation of an object-oriented DBMS. In Shriver, Wegner (Hrsg.): Research Directions in Object-Oriented Programming, MIT Press, 355-392.

[MYLO91] Mylopoulos,J. (1991). Conceptual modelling and Telos. Erscheint in Loucopoulos, Zicari (Hrsg.): Conceptual Modelling, Databases and CASE: An Integrated View of Information Systems Development, McGraw Hill.

[NCL*87] Nixon,B.A., Chung,L., Lauzon,D., Borgida,A., Mylopoulos,J., Stanley,M. (1987). Implementation of a compiler for a semantic data model: experiences with Taxis. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, San Francisco, Kalifornien, 118-131.

[NICO79] Nicolas,J.-M. (1979). Logical formulas and integrity constraints: the range restricted property and a simplification method. T-R CERT-LBD/79-1, Toulouse.

[NICO82] Nicolas,J.-M. (1982). Logic for improving integrity checking in relational databases. *Acta Informatika* 18, 227-253.

[NILS82] Nilsson,N. (1982). Principles of artificial intelligence. Springer-Verlag.

[NT89] Naqvi,S., Tsur,S. (1989). A logical language for data and knowledge bases. Computer Science Press.

[OLIV91] Olivé,A. (1991). Integrity constraints checking in deductive databases. *Proc. VLDB'91*, Barcelona, Spanien, 513-524.

[PGB91] Patel-Schneider,P.F., McGuinness,D.L., Borgida,A. (1991). The CLASSIC knowledge representation system: guiding principles and implementation rationale. *SIGART Bulletin* 2(3), Juni 1991.

[RD91] Ramesh,B., Dhar,V. (1991). Process knowledge-based group support for requirements engineering. Arbeitspapier, Naval Postgraduate School, Monterey, CA.

[REIT84] Reiter,R. (1984). Towards a logical reconstruction of relational database theory. In Brodie et al. (Hrsg.): On Conceptual Modelling, Springer-Verlag, 191-233.

[REIT90] Reiter,R. (1990). What should a database know? Technical report KRR-TR-90-5, University of Toronto, Dept. of Computer Science, Juli 1990.

[REHM88] Rehm,S. et al. (1988). Support for design processes in a structurally object-oriented database system. In [DITT88], 80-97.

[RICH78] Richter,M.M. (1978). Logikkalküle. Teubner, Stuttgart, 1978.

[RJG*91] Rose,T., Jarke,M., Gocek,M., Maltzahn,C., Nissen,H. (1991). A decision-based configuration process environment. *Software Engineering Journal* 6(5), Special Issue on Software Process and its Support, Sept. 1991, 332-346.

[ROSE91] Rose,T. (1991). Entscheidungsorientierte Versionen- und Konfigurationenverwaltung. Dissertationsschrift, Universität Passau, 1991.

[SCHE91] Scheer,A.-W. (1991). Architektur integrierter Informationssysteme: Grundlagen der Unternehmensmodellierung. Springer-Verlag.

[SCHM77] Schmidt,J.W. (1977). Some high-level language constructs for data of type relation. *ACM Trans. Database Systems* 2(3), 247-261.

[SCHM89] Schmidt-Schauß,M. (1989). Subsumption in KL-ONE is undecidable. *Proc. First Int. Conf. on Principles of Knowledge Representation and Reasoning*, Toronto, Ontario, 1989, 421-431.

[SCHÖ90] Schöning,H. (1990). Integrating complex objects and recursion. In [KNN90], 573-592.

[SEM88] Schmidt,J.W., Eckhardt,H., Matthes,F. (1988). DBPL report. Memo 111-88, Fachbereich Informatik, Universität Frankfurt, 1988.

[SHIP81] Shipman,D.W. (1981). The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems* 6(1), März 1981, 140-173.

[SIGA91] *SIGART Bulletin* 2(3), Juni 1991.

[SJGP90] Stonebraker,M., Jhingran,A., Goh,J., Potiamos,S. (1990). On rules, procedures, caching and views in database systems. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Atlantic City, New Jersey, 281-290.

[SK88] Sadri,F., Kowalski,R. (1988). A theorem-proving approach to database integrity. In Minker (Hrsg.): Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann Publishers, 313-362.

[SK91] Stonebraker,M., Kemnitz,G. (1991). The Postgres next-generation database management system. *Comm. of the ACM* 34(10), Oktober 1991, 78-93.

[SLR89] Sellis,T., Lin,C.-C., Raschid,L. (1989). Data intensive production systems: the DIPS approach. *SIGMOD Record* 18(3), September 1989, 52-58.

[SLT91] Scholl,M., Laasch,C., Tresch,M. (1991). Updatable views in object-oriented databases. In [DKM91], 189-207.

[SOO91] Soo, M.D. (1991). Bibliography on temporal databases. *SIGMOD Record* 20(1), März 1991, 14-23.

[SRH90] Stonebraker,M., Rowe,L.A., Hirohama,M. (1990). The implementation of Postgres. Memorandum UCB/ERL M90/34, Univ. of California, April 1990.

[SS83a] Schlageter,G., Stucky,W. (1983). Datenbanksysteme: Konzepte und Modelle. Teubner Studienbücher.

[SS83b] Schek,H.-J., Scholl,M. (1983). Die NF^2-Relationenalgebra zur einheitlichen Manipulation externer, konzeptueller und interner Datenstrukturen. In Schmidt (Hrsg.): Sprachen für Datenbanken. *Informatik-Fachberichte* 72, Springer-Verlag, 113-133.

[SS91a] Schmidt-Schauß,M., Smolka,G. (1991). Attribute concept descriptions with complements. *Artificial Intelligence* 48, Februar 1991, 1-26.

[SS91b] Scholl,M., Schek,H.-J. (1991). Supporting views in object-oriented databases. *Data Engineering Bulletin*, Juni 1991.

[SSS88] Stemple,D., Socorro,A., Sheard,T. (1988). Formalizing objects for databases using ADABTL. In [DITT88], 110-128.

[SSU91] Silberschatz,A., Stonebraker,M., Ullman,J. (1991). Database systems: achievements and opportunities. *Comm. of the ACM* 34(10), Oktober 1991, 110-120.

[STAN86] Stanley,M.T. (1986). CML: a knowledge representation language with application to requirements modeling. M.S. thesis, University of Toronto, Ontario.

[STAU90] Staudt,M. (1990). Anfragerepräsentation und -auswertung in deduktiven Objektbanken. Diplomarbeit, Universität Passau.

[STON75] Stonebraker,M. (1975). Implementation of integrity constraints and views by query modification. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Mai 1975.

[SUN90] Sun Microsystems (1990). Network programming guide. Handbuch Nr. 800-3850-10, USA, 27.3.1990.

[SV87] Simon,E., Valduriez,P. (1987). Design and analysis of a relational integrity subsystem. MCC Technical Report No. DB-015-87, Texas.

[SZ90] Shaw,G.M., Zdonik,S.B. (1990). Object-oriented queries: equivalence and optimization. In [KNN90], 281-296.

[TARS41] Tarski,A. (1941). On the calculus of relations. *Journal of Symbolic Logic* 6(3), September 1941, 73-89.

[UKN92] Urban,S.D., Karadimce,A.P., Nannapaneni,R.B. (1992). The implementation and evaluation of integrity maintenance rules in an object-oriented database. Erscheint in: *Proc. 8th Int. Conf. on Data Engineering*, 1992.

[ULLM89] Ullman,J.D. (1989). Principles of database and knowledge-base systems. Volume I & II. Computer Science Press, 1988-1989.

[ULLM91] Ullman,J.D. (1991). A comparison of deductive and object-oriented database systems. In [DKM91], 263-277.

[VK86] Vilain,M., Kautz,H. (1986). Constraint propagation algorithms for temporal reasoning. *Proc. 5th National Conf. on Artificial Intelligence (AAAI)*, Philadelphia, Pa., 377-382.

[vL90] von Luck,K. (1990). KL-One: eine Einführung. IWBS Report 106, IBM Deutschland, Wissenschaftliches Zentrum, Stuttgart, Februar 1990.

[VOLG89] Volger,H. (1989). The semantics of disjunctive deductive databases. Report MIP-8931, Universität Passau.

[VOSS91] Vossen,G. (1991). Data models, database languages and database management systems. Addison-Wesley, Int. Computer Science Series.

[WENI89] Wenig,T. (1989). Zeitbegriffe in deduktiven Objektbanken am Beispiel von ConceptBase. Diplomarbeit, Universität Passau.

[WIRS86] Wirsing,M. (1986). Structured algebraic specifications: a kernel language. *Theoretical Computer Science* 42, 1986, 123-249.

[WIRT79] Wirth,N. (1979). Algorithmen und Datenstrukturen. 2. Auflage, Teubner, Stuttgart.

[WIRT91] Wirth,N. (1991). Programmieren in Modula-2. 2. Auflage, Springer-Verlag.

[WNS89] Wetzel,I., Niebergall,P., Schmidt,J.W. (1989). A mapping assistant for database program development. Arbeitspapier, ESPRIT 892 (DAIDA), Universität Frankfurt, März 1989.

Anhang 1: Beispiellauf von ConceptBase

Der folgende Beispiellauf von ConceptBase [JARK91] wurde am 25. September 1991 ausgeführt. Der Rechner ist eine SPARCstation-1 der Firma SUN mit 12 Megabyte Hauptspeicher. Ausgaben des *Servers* werden in Schreibmaschinenschrift dargestellt. Die Ausgaben wurden von für diese Arbeit unwichtigen Details bereinigt. Die Angaben von Gültigkeits- und Systemzeit wurden eliminiert und die Notation der Literale wurde an die in dieser Arbeit eingeführte Schreibweise behutsam angeglichen. Im Gegensatz zur Notation von Kapitel 4 werden bei Individualobjekten die Namen auch als Objektidentifikatoren benutzt. Die im Verlauf der Sitzung eingetragenen Formeln lauten im Klartext:

```
Individual Agent with
 constraint
   tertiumNonDatur: $ forall a/Agent s/Symptom
                      not (A(a,improves,s) and A(a,makesWorse,s)) $
end Agent

Individual Drug with
 rule
    onSymptoms: $ forall d/Drug s/Symptom
                  (exists a/Agent
                     A(d,component,a) and
                     (A(a,improves,s) or not A(a,makesWorse,s)))
                ==> A(d,against,s) $
end Drug

Individual Patient with
 constraint
    canTake: $ forall p/Patient d/Drug
                A(p,takes,d) ==>
                (exists s/Symptom A(p,suffers,s) and
                              A(d,against,s)) $
end Patient
```

Das Beispiel zeigt, daß auch Negation und Disjunktion in einem Regelrumpf zugelassen sind. Solche Regeln werden nach [LT84] in das Format von DATALOG$^\neg$ überführt.

---

1. **Starten des Datenbank.** Es wird eine Datenbank über Medikamente, Wirkstoffe und Patienten geladen. Da die Objekte neu sind, werden auch sie auf Integrität getestet.

```
ProLog by BIM - release 3.0 - 20-Dec-1990
...
Loading : /home/jarke/CBase/CB_Exe/CBSYSASSTIME.prop ready

  > This is ConceptBase (CBserver) V3.0, 5-Sep-1991
  > Copyright (C) 1988-1991 by
  >    Lehrstuhl Prof. Dr. M. Jarke
  >    University of Passau
  >    P.O. Box 2540
  >    D-8390 Passau / Germany
```

```
   > Operating system: SUN Unix Release 4.1.1
   > Tracing mode: veryhigh
   > Optimization of integrity constraints: structural
   > Default query evaluation paradigm: SLDNF
   > KB update mode: nonpersistent

+++ CBserver ready on host 'tell' under port number 4321 +++

>>> C_interface#7484.1: CALL TELL_MODEL ON
                    [/private.tell/jeusfeld/DISS/Examples/DRUGS/DRUGS]

Loading : /private.tell/jeusfeld/DISS/Examples/DRUGS/DRUGS.sml
   ... stored temporarily (0 error(s), 0.62 sec used)
   ... semantic integrity checked (0 error(s), 2.95 sec used)
+P(#7484.2,Agent,in,CLASS)
+P(#7484.3,Agent,in,SimpleClass)
+P(#7484.6,Symptom,in,CLASS)
+P(#7484.7,Symptom,in,SimpleClass)
+P(#7484.8,Person,in,CLASS)
+P(#7484.9,Person,in,SimpleClass)
+P(#7484.11,#7484.10,in,Necessary)
+P(#7484.12,Patient,in,CLASS)
+P(#7484.14,Patient,in,SimpleClass)
+P(#7484.17,Drug,in,CLASS)
+P(#7484.18,Drug,in,SimpleClass)
+P(#7484.13,Patient,*isa,Person)
+P(Agent,Agent,-,Agent)
+P(Symptom,Symptom,-,Symptom)
+P(Person,Person,-,Person)
+P(Patient,Patient,-,Patient)
+P(Drug,Drug,-,Drug)
+P(#7484.4,Agent,improves,Symptom)
+P(#7484.5,Agent,makesWorse,Symptom)
+P(#7484.10,Person,name,String)
+P(#7484.15,Patient,takes,Drug)
+P(#7484.16,Patient,suffers,Symptom)
+P(#7484.19,Drug,component,Agent)
+P(#7484.20,Drug,against,Symptom)
   ... confirmed in KB
   3.89 sec used for telling
 ready
```

---

2. **Einfügen von Regeln und Integritätsbedingungen.** Die erste Formel drückt aus, daß ein Wirkstoff nicht gleichzeitig ein Symptom verbessern und verschlechtern kann (siehe `MSFOLassertion`). Die Formel wird in Bereichsform transformiert. Aufgrund der Bindungen der Variablen an Klassen können die Literale `In(a,Agent)` und `In(s,Symptom)`, die in Konjunktion mit `A(a,improves,s)` stehen, eliminiert werden.

```
>>> C_interface#7484.1: CALL TELL_MODEL ON
                    [/private.tell/jeusfeld/DISS/Examples/DRUGS/RuleIc]

>>> BDMTransFormula: parseMSFOLassertion --->
   MSFOLassertion(forall(a,Agent,forall(s,Symptom, not and(lit(A(a,improves,s)),
   lit(A(a,makesWorse,s))))))
```

```
>>> BDMTransFormula: miniscopeToRangeform --->
   forall([a,s], [In(a,Agent), A(a,improves,s), In(s,Symptom), A(a,makesWorse,s)],false)

>>> BDMLiteralDeps: ConcernedClass(vars([a,s],[a,s]),[range(a,Agent),range(s,Symptom)],
   A(a,improves,s)) ---> #7484.4

>>> SemanticOptimizer: In(a,Agent) guaranteed by A(a,improves,s)

>>> SemanticOptimizer: In(s,Symptom) guaranteed by A(a,improves,s)

>>> SemanticOptimizer: Optimize by InstanceOf_constraint_1 --->
   forall([a,s], [A(a,improves,s), A(a,makesWorse,s)], false)

>>> BDMTransFormula: RangeToEvaForm --->
   forall([In(_30394,Agent), A(_30394,improves,_30399),
   A(_30394,makesWorse,_30399)], false)
```

---

3. **Vereinfachung der 1. Integritätsbedingung.** In der optimierten Form tauchen nur noch zwei Literale auf. Daraus resultieren zwei vereinfachte Formen, die jeweils für den Einfügungsfall auszuwerten sind.

```
>>> BDMCompile: SimplifyRangeform(Insert, A(a,improves,s)) --->
   forall(nil, [A(a,makesWorse,s)], false)

>>> BDMTransFormula: RangeToEvaForm --->
   implies([A(_31479,makesWorse,_31484)], false) BY A(_31479,improves,_31484)

>>> BDMLiteralDeps: ConcernedClass(vars([a,s],[a,s]), [range(a,Agent),
   range(s,Symptom)], A(a,makesWorse,s)) ---> #7484.5

>>> BDMCompile: SimplifyRangeform(Insert, A(a,makesWorse,s)) --->
   forall(nil, [A(a,improves,s)], false)

>>> BDMTransFormula: RangeToEvaForm --->
   implies([A(_32538,improves,_32543)], false) BY A(_32538,makesWorse,_32543)
```

---

4. **Einfügung der deduktiven Regel.** Die Regel drückt aus, daß ein Medikament gegen ein Symptom wirkt, wenn es einen Wirkstoff enthält der dieses Symptom verbessert oder zumindest nicht verschlechtert. Sie ist ein Beispiel für eine Regel mit negativen Literalen im Bedingungsteil.

```
>>> BDMTransFormula: parseMSFOLassertion --->
   MSFOLassertion(forall(d,Drug, forall(s,Symptom, impl(exists(a,Agent,
   and(lit(A(d,component,a)), or(lit(A(a,improves,s)),
   not lit(A(a,makesWorse,s))))), lit(A(d,against,s))))))

>>> BDMTransFormula: miniscopeToRangeform --->
   rangerule(forall([d,s,a], [In(d,Drug), A(d,component,a), In(a,Agent),
   In(s,Symptom)], and( not A(a,improves,s), A(a,makesWorse,s))), A(d,against,s))
```

```
>>> BDMLiteralDeps: ConcernedClass(vars([d,s,a],[d,s,a]),
   [range(d,Drug),range(s,Symptom), range(a,Agent)], A(d,against,s)) ---> #7484.20

>>> BDMLiteralDeps: ConcernedClass(vars([d,s,a],[d,s,a]), [range(d,Drug),
   range(s,Symptom),range(a,Agent)], A(d,component,a)) ---> #7484.19

>>> SemanticOptimizer: In(d,Drug) guaranteed by A(d,component,a)

>>> SemanticOptimizer: In(a,Agent) guaranteed by A(d,component,a)

>>> SemanticOptimizer: Optimize by InstanceOf_constraint_1 --->
   forall([d,s,a], [A(d,component,a), In(s,Symptom)], and( not A(a,improves,s),
   A(a,makesWorse,s)))

>>> BDMTransFormula: RangeToEvaForm --->
   evarule(exists([In(_43750,Drug), A(_43750,component,_43760), In(_43755,Symptom)],
   or(A(_43760,improves,_43755), not A(_43760,makesWorse,_43755))),
   A(_43750,against,_43755))
```

---

5. **Einfügung der 2. Integritätsbedingung.** Diese Integritätsbedingung besagt, daß ein Patient nur dann ein Medikament nehmen darf, wenn er an einem Symptom leidet, gegen das dieses Medikament wirkt. Man beachte, daß auch hier alle *In*-Literale eliminiert werden können. Deweiteren ist zu sehen, daß erst jetzt vereinfachte Formen für die deduktive Regel generiert werden, da das Folgerungsprädikat $A(d, \mathit{against}, s)$ in der Integritätsbedingung vorkommt.

```
>>> BDMTransFormula: parseMSFOLassertion --->
   MSFOLassertion(forall(p,Patient, forall(d,Drug, impl(lit(A(p,takes,d)),
   exists(s,Symptom, and(lit(A(p,suffers,s)), lit(A(d,against,s))))))))

>>> BDMTransFormula: miniscopeToRangeform ---> forall([p,d], [In(p,Patient),
   A(p,takes,d), In(d,Drug)],
   exists([s],[A(p,suffers,s), In(s,Symptom), A(d,against,s)],true))

>>> BDMLiteralDeps: ConcernedClass(vars([p,d,s],[p,d]), [range(p,Patient),
   range(d,Drug),range(s,Symptom)], A(p,takes,d)) ---> #7484.15

>>> SemanticOptimizer: In(p,Patient) guaranteed by A(p,takes,d)

>>> SemanticOptimizer: In(d,Drug) guaranteed by A(p,takes,d)

>>> BDMLiteralDeps: ConcernedClass(vars([p,d,s],[p,d]), [range(p,Patient),
   range(d,Drug),range(s,Symptom)], A(p,suffers,s)) ---> #7484.16

>>> SemanticOptimizer: In(s,Symptom) guaranteed by A(p,suffers,s)

>>> SemanticOptimizer: Optimize by InstanceOf_constraint_1 --->
   forall([p,d], [A(p,takes,d)], exists([s],[A(p,suffers,s), A(d,against,s)],true))

>>> BDMTransFormula: RangeToEvaForm --->
   forall([In(_50246,Patient), A(_50246,takes,_50251)],
   exists([A(_50246,suffers,_50256), A(_50251,against,_50256)],true))

>>> BDMCompile: SimplifyRangeform(Insert,A(p,takes,d)) --->
   forall(nil,nil, exists([s],[A(p,suffers,s), A(d,against,s)], true))
```

```
>>> BDMTransFormula: RangeToEvaForm --->
   exists([A(_51490,suffers,_51500), A(_51495,against,_51500)],true)
   BY A(_51490,takes,_51495)

>>> BDMCompile: SimplifyRangeform(Delete,A(p,suffers,s)) --->
   forall([d],[A(p,takes,d)],exists([s],[A(p,suffers,s), A(d,against,s)], true))

>>> BDMTransFormula: RangeToEvaForm --->
   forall([A(_52675,takes,_52680)], exists([A(_52675,suffers,_52685),
   A(_52680,against,_52685)],true)) BY A(_52675,suffers,_52928)

>>> BDMCompile: SimplifyRangeform(Delete,A(d,against,s)) --->
   forall([p], [A(p,takes,d)],exists([s], [A(p,suffers,s), A(d,against,s)],true))

>>> BDMTransFormula: RangeToEvaForm --->
   forall([In(_54009,Patient), A(_54009,takes,_54014)],
   exists([A(_54009,suffers,_54019), A(_54014,against,_54019)], true))
   BY A(_54014,against,_54356)
```

---

6. **Vereinfachung der Regel.** In der 2. Integritätsbedingung kommt das Literal `A(d,against,s)` positiv vor, d.h. bei Löschungen dieses Literals muß sie getestet werden. Dies macht die Vereinfachung der deduktiven Regel nötig.

```
>>> BDMCompile: Original rule #7484.53 has to be re-compiled for the case "Delete"
   of its conclusion literal A(d,against,s)

>>> BDMCompile: SimplifyRangeform(Delete,A(d,component,a)) --->
   forall([s], [A(d,component,a),In(s,Symptom)], and( not A(a,improves,s),
   A(a,makesWorse,s)))

>>> BDMTransFormula: RangeToEvaForm --->
   evarule(exists([A(_55978,component,_55988),In(_55983,Symptom)],
   or(A(_55988,improves,_55983),
   not A(_55988,makesWorse,_55983))), A(_55978,against,_55983))
   BY A(_55978,component,_55988)

>>> BDMLiteralDeps: ConcernedClass(vars([d,s,a],[d,s,a]), [range(d,Drug),
   range(s,Symptom),range(a,Agent)], In(s,Symptom)) ---> Symptom

>>> BDMCompile: SimplifyRangeform(Delete, In(s,Symptom)) --->
   forall([d,a], [A(d,component,a), In(s,Symptom)],
   and( not A(a,improves,s), A(a,makesWorse,s)))

>>> BDMTransFormula: RangeToEvaForm --->
   evarule(exists([In(_57590,Drug), A(_57590,component,_57600), In(_57595,Symptom)],
   or(A(_57600,improves,_57595), not A(_57600,makesWorse,_57595))),
   A(_57590,against,_57595)) BY In(_57595,Symptom)

>>> BDMCompile: SimplifyRangeform(Delete, A(a,improves,s)) --->
   forall([d], [A(d,component,a), In(s,Symptom)], and( not A(a,improves,s),
   A(a,makesWorse,s)))

>>> BDMTransFormula: RangeToEvaForm --->
   evarule(exists([In(_59427,Drug), A(_59427,component,_59437), In(_59432,Symptom)],
   or(A(_59437,improves,_59432), not A(_59437,makesWorse,_59432))),
```

```
   A(_59427,against,_59432)) BY A(_59437,improves,_59432)

>>> BDMCompile: SimplifyRangeform(Insert,A(a,makesWorse,s)) --->
   forall([d], [A(d,component,a), In(s,Symptom)], and( not A(a,improves,s),
   A(a,makesWorse,s)))

>>> BDMTransFormula: RangeToEvaForm --->
   evarule(exists([In(_61324,Drug), A(_61324,component,_61334), In(_61329,Symptom)],
   or(A(_61334,improves,_61329), not A(_61334,makesWorse,_61329))),
   A(_61324,against,_61329)) BY A(_61334,makesWorse,_61329)

  ... stored temporarily (0 error(s), 4.17 sec used)
```

---

7. **Test auf Integrität.** Es wird getestet, ob die beiden neuen Integritätsbedingungen wahr sind. Sie werden in die Objektbank übernommen.

```
>>> BDMEvaluation: Evaluating integrity constraint ... forall([In(_55583,Agent),
   A(_55583,improves,_55591), A(_55583,makesWorse,_55591)], false)


>>> BDMEvaluation: Evaluating integrity constraint ...
   forall([In(_55583,Patient), A(_55583,takes,_55591)],
   exists([A(_55583,suffers,_55601), A(_55591,against,_55601)], true))

  ... semantic integrity checked (0 error(s), 11.26 sec used)
+P(#7484.22,#7484.21,in,BDMConstraint)
+P(#7484.21,#7484.21,"$forall a/Agent s/Symptom
  not (A(a,improves,s) and A(a,makesWorse,s))$"],#7484.21)
< ... >

  ... confirmed in KB
  21.85 sec used for telling
 ready
```

---

8. **Einfügen von Instanzen.** Es wird ein Patient Jack mit Symptomen Fieber und Kopfschmerzen eingetragen. Jack nimmt das Medikament QuasiForte. Diese Information stößt die Auswertung der 2. Integritätsbedingung an. Es wird getestet, ob Jack ein Symptom hat, das von QuasiForte behandelt wird.

```
>>> C_interface#7484.1: CALL TELL_MODEL ON
   [/private.tell/jeusfeld/DISS/Examples/DRUGS/Inst1]

Loading : /private.tell/jeusfeld/DISS/Examples/DRUGS/Inst1.sml
  ... stored temporarily (0 error(s), 0.76 sec used)

>>> BDMEvaluation: Test integrity constraints [#7484.70] on literals
   [A(Jack,_16622,QuasiForte), In(#7484.130,#7484.15)] for operation Insert


>>> BDMEvaluation: Evaluating integrity constraint ...
   exists([A(Jack,suffers,_16674), A(QuasiForte,against,_16674)], true)
```

```
  ... semantic integrity checked (0 error(s), 2.68 sec used)
+P(#7484.126,Headache,in,Symptom)
+P(#7484.127,Fever,in,Symptom)
+P(#7484.128,Fentanyl,in,Agent)
+P(#7484.129,Jack,in,Patient)
+P(#7484.131,#7484.130,in,#7484.15)
+P(#7484.133,#7484.132,in,#7484.16)
+P(#7484.135,#7484.134,in,#7484.16)
+P(#7484.136,QuasiForte,in,Drug)
+P(#7484.138,#7484.137,in,#7484.19)
+P(Headache,Headache,-,Headache)
+P(Fever,Fever,-,Fever)
+P(Fentanyl,Fentanyl,-,Fentanyl)
+P(Jack,Jack,-,Jack)
+P(QuasiForte,QuasiForte,-,QuasiForte)
+P(#7484.130,Jack,drug1,QuasiForte)
+P(#7484.132,Jack,indication1,Fever)
+P(#7484.134,Jack,indication2,Headache)
+P(#7484.137,QuasiForte,agent1,Fentanyl)
  ... confirmed in KB
  3.69 sec used for telling
ready
```

---

9. **Löschen eines Attributes.** Zunächst wird die Textrepräsentation des Objektes Jack und seiner Attribute in einen Editor übertragen. Von diesem Editor aus erfolgt das Löschen (UNTELL) des 2. Symptoms von Jack. Von dieser Operation ist wiederum die 2. Integritätsbedingung betroffen. Sie ist wegen des Symptoms Fieber noch erfüllt. Abbildung A1-1 zeigt die Benutzerschnittstelle mit einer graphischen Repräsentation eines Teils der Objektbank. In einem Editor wurde gerade das Löschen des Attributes vorgenommen.

```
>>> X11_GraphBrowser#7484.139: CALL ASK ON [get_object([Jack / objname]),
   ROLLBACK : Now, FORMAT : FRAME]

>>> CBserver#7484.0: RESPONSE IS "Individual Jack in Patient with
  takes,attribute
    drug1 : QuasiForte
  suffers,attribute
    indication1 : Fever;
    indication2 : Headache
end Jack
" {ok}


>>> TelosEd#7484.140: CALL UNTELL ON [
"Jack with
  suffers,attribute
    indication2 : Headache
end Jack
"
]

  ... removed temporarily (0 error(s), 0.32 sec used)
```

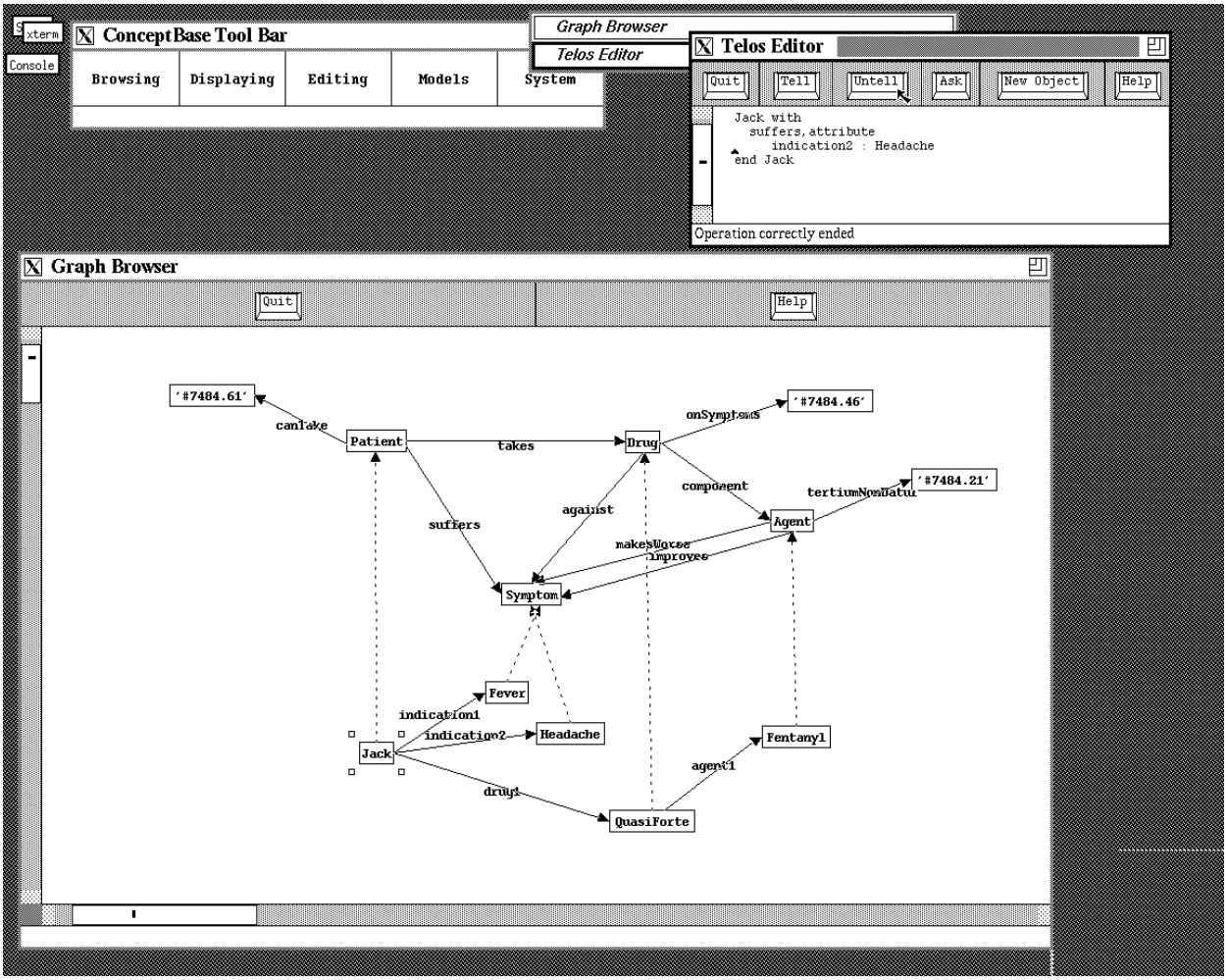

Abb. A1-1: Benutzerschnittstelle nach Schritt 9

```
>>> BDMEvaluation: Test integrity constraints [#7484.78] on literals
   [A(Jack,_7968,Headache),In(#7484.134,#7484.16)] for operation Delete

>>> BDMEvaluation: Evaluating integrity constraint ...
   forall([A(Jack,takes,_8020)], exists([A(Jack,suffers,_8030),
   A(_8020,against,_8030)],true))

  ... semantic integrity checked (0 error(s), 0.83 sec used)
-P(#7484.135,#7484.134,in,#7484.16)
-P(#7484.134,Jack,indication2,Headache)
  ... transactiontime is: millisecond(1991,9,25,15,41,51,0)
  ... confirmed in KB
  1.17 sec used for untelling
```

---

10. **Einfügung bei Fentanyl.** Als nächster Schritt bekommt Fentanyl ein Attribut, das besagt, daß es das Symptom Fieber verschlechtert. Diese Information geht in das negativ vorkommende Literal `A(a,makesWorse,s)` der Regel ein und bewirkt die Löschung des abgeleiteten Fakts `A(QuasiForte,against,Fever)`. Diese Löschung

wiederum betrifft die 2. Integritätsbedingung, deren Test nun fehlschlägt. Abbildung A1-2 zeigt, wie sich diese Situation an der Benutzerschnittstelle darstellt. Es öffnet sich ein Fehlerfenster mit entsprechenden Erklärungen.

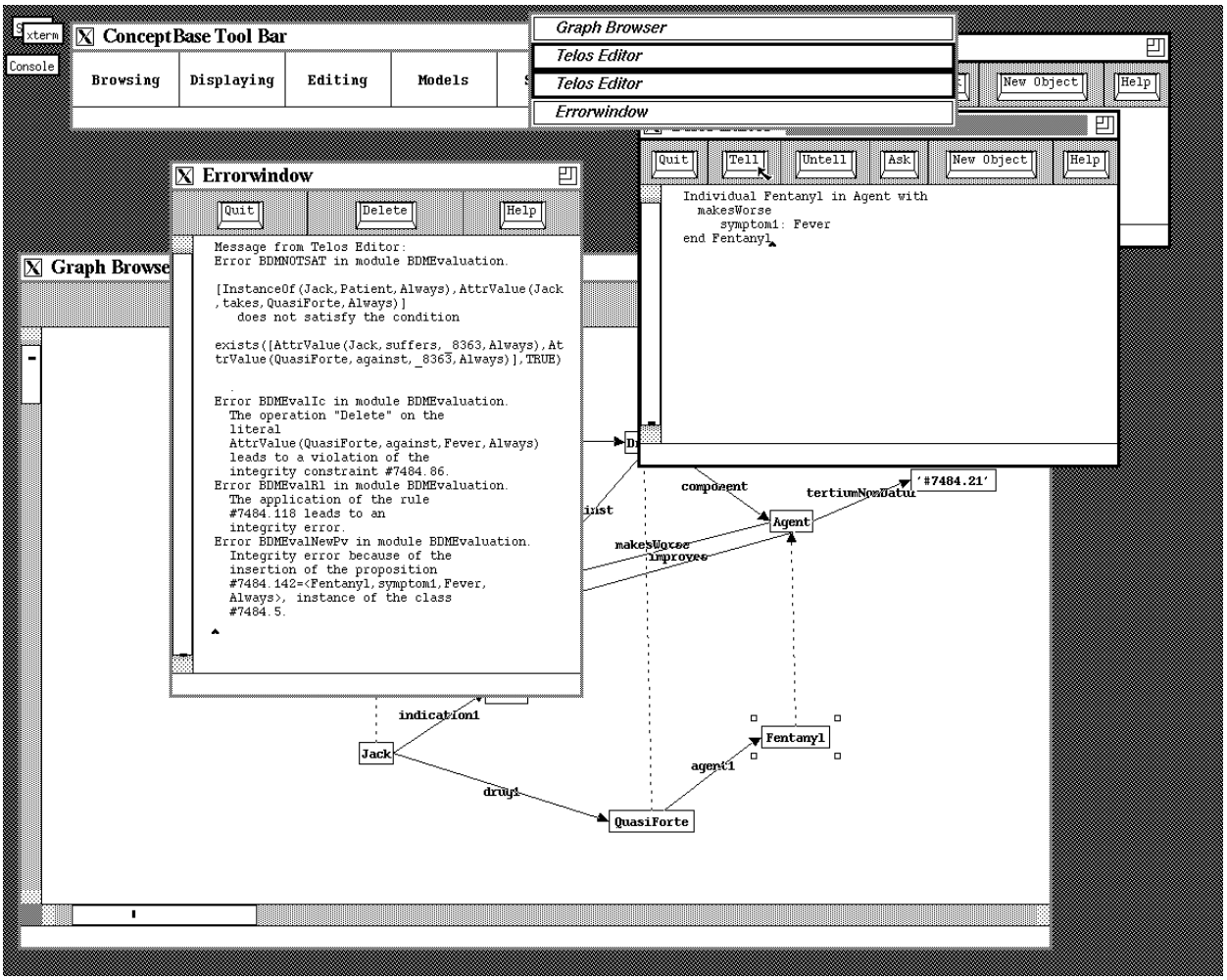

Abb. A1-2: Benutzerschnittstelle nach Schritt 10

```
>>> X11_GraphBrowser#7484.139: CALL ASK ON [get_object([Fentanyl / objname]),
   ROLLBACK : Now,FORMAT : FRAME]

>>> CBserver#7484.0: RESPONSE IS "Individual Fentanyl in Agent with
end Fentanyl
" {ok}

>>> TelosEd#7484.141: CALL TELL ON [
"Individual Fentanyl in Agent with
  makesWorse
    symptom1: Fever
end Fentanyl
```

```
"
]
   ... stored temporarily (0 error(s), 0.29 sec used)

>>> BDMEvaluation: Test integrity constraints [#7484.38] on literals
   [A(Fentanyl,_8127,Fever), In(#7484.142,#7484.5)] for operation Insert

>>> BDMEvaluation: Evaluating integrity constraint ...
   implies([A(Fentanyl,improves,Fever)], false)

>>> BDMEvaluation: Evaluating condition of deductive rule ...
   exists([In(_8163,Drug), A(_8163,component,Fentanyl), In(Fever,Symptom)],
   or(A(Fentanyl,improves,Fever), not A(Fentanyl,makesWorse,Fever)))

>>> BDMEvaluation: Evaluating condition of deductive rule ...
   exists([In(_8163,Drug,), A(_8163,component,Fentanyl), In(Fever,Symptom)],
   or(A(Fentanyl,improves,Fever), not A(Fentanyl,makesWorse,Fever)))

>>> BDMEvaluation: Test integrity constraints [#7484.86] on literals
   [A(QuasiForte,against,Fever)] for operation Delete

>>> BDMEvaluation: Evaluating integrity constraint ...
   forall([In(_8345,Patient), A(_8345,takes,QuasiForte)],
   exists([A(_8345,suffers,_8363), A(QuasiForte,against,_8363)], true))

   ... semantic integrity checked (1 error(s), 1.47 sec used)
~P(#7484.143,#7484.142,in,#7484.5)
~P(#7484.142,Fentanyl,symptom1,Fever)
   ... temporary information retracted
```

Das Protokoll der Sitzung zeigt auch, daß die Benutzerschnittstelle mittels der Werkzeugintegration (Kap. 8 und 9) mit dem Kernsystem verbunden sind. Das Kernsystem selber ist auch eine Werkzeug (`CBserver#7484.0`). Die Nachrichten an das Kernsystem werden im Klartext des Protokolls in der Form

```
>>> /SENDER/: CALL /OPERATION/ ON [/ARGUMENTLISTE/]
```

ausgegeben. Die Anfrage des Werkzeuges `X11_GraphBrowser#7484.139` in Schritt 10 enthält ein Beispiel, wie in der Abfragesprache eine Rollback-Zeit angegeben wird. Das Zeitintervall `Now` steht dabei für den Zeitpunkt, zu dem die Anfrage das Kernsystem erreicht.

Anhang 2: Ausschnitt aus der DAIDA-Entwurfsdatenbank

Der folgende Text enthält die Klassen, die in der DAIDA-Entwurfsdatenbank die Manipulation von abstrakten Maschinen repräsentieren. Die Syntax weicht leicht von den Angaben in Kapitel 9 ab. Die Objekte *INDIVIDUALCLASS* und *ATTTRIBUTECLASS* sind Spezialisierungen von *Object*. Die Schreibweisen *DesignObject*, *DesignDecision* und *DesignTool* stehen in der Diktion von Kapitel 8 für *Object*, *Decision* und *Tool*.

```
(*
*
* File:       ImplementationDesign.sml
* Version:    V2.0
* Creation:   19-Apr-1989, Manfred Jeusfeld (UPA)
* Last change: 16-Nov-1989, Manfred Jeusfeld (UPA)
* -------------------------------------------------------------------------
*
* This file contains the GKBMS definitions of the implementation
* design (which is based on so-called abstract machines).
*
*)



                (* -------------------------- *)
                (*  D e s i g n O b j e c t s  *)
                (* -------------------------- *)


INDIVIDUALCLASS ImplementationDesign IN DesignObject ISA DAIDA_DesignObject WITH
  justification
      creation: ModifyImplementationDesign
  objectsource
      sourcefile: String
  objectsemantic
      itsdescription: AbstractMachine
END

INDIVIDUALCLASS InitialImplementationDesign IN DesignObject
                                    ISA ImplementationDesign,Initial_DO
WITH
  justification
      creation: MapToImplementationDesign
  objectsemantic
      itsdescription: InitialAbstractMachine
END

INDIVIDUALCLASS RefinedImplementationDesign IN DesignObject
                                    ISA ImplementationDesign,Refined_DO
WITH
  justification
      creation: RefineImplementationDesign
END

INDIVIDUALCLASS BaselineImplementationDesign IN DesignObject
                                   ISA ImplementationDesign,Baseline_DO
WITH
  justification
      creation: ReleaseImplementationDesign
```

```
END


INDIVIDUALCLASS BtoolRule IN DesignObject ISA SimpleClass

INDIVIDUALCLASS ProofObligation IN DesignObject

INDIVIDUALCLASS Lemma IN DesignObject ISA ProofObligation

INDIVIDUALCLASS TrueOrFalse IN DesignObject ISA Lemma

INDIVIDUALCLASS BtoolTheory IN DesignObject ISA SimpleClass WITH
  objectsource
      sourcefile: String
  part
      rules: BtoolRule
END

INDIVIDUALCLASS BtoolTactic IN DesignObject ISA SimpleClass WITH
  objectsource
      sourcefile: String
END


INDIVIDUALCLASS BuiltinTheory IN DesignObject ISA BtoolTheory

INDIVIDUALCLASS MappingAssistantTheory IN DesignObject ISA BtoolTheory

INDIVIDUALCLASS AuxiliaryTheory IN DesignObject ISA BtoolTheory

INDIVIDUALCLASS ApplicationTheory IN DesignObject ISA BtoolTheory



              (* ------------------------------ *)
              (*  O b j e c t S e m a n t i c s  *)
              (* ------------------------------ *)

INDIVIDUALCLASS Type IN MetametaClass ISA Literals

INDIVIDUALCLASS BooleanType IN Type ISA SimpleClass

INDIVIDUALCLASS TRUE IN BooleanType,ProofObligation ISA Token

INDIVIDUALCLASS FALSE IN BooleanType ISA Token

INDIVIDUALCLASS VariableType IN Type ISA SimpleClass

INDIVIDUALCLASS SetType IN Type ISA SimpleClass

INDIVIDUALCLASS StringType IN Type ISA SimpleClass

INDIVIDUALCLASS RelationType IN Type ISA SimpleClass

INDIVIDUALCLASS FunctionType IN Type ISA SimpleClass

INDIVIDUALCLASS TotalFunctionType IN Type ISA FunctionType

INDIVIDUALCLASS PartialFunctionType IN Type ISA FunctionType

INDIVIDUALCLASS ClassType IN Type ISA SimpleClass
```

```
INDIVIDUALCLASS PowersetType IN Type ISA SimpleClass

INDIVIDUALCLASS BasicSetType IN Type ISA SimpleClass


INDIVIDUALCLASS AbstractMachine IN MetaClass ISA BtoolRule WITH
 dependson
      dependsonAm: AbstractMachine
      dependsonTdl: TDL_Design
 attribute
      basicsets: AM_BAS
      initializations: AM_INI
      contexts: AM_CTX
      variables: AM_VRB
      invariants: AM_INV
      operations: AM_OPN
      others: AM_OTH
  constraint
      inv4ops: $ forall am/AbstractMachine
                 (exists v/AM_VRB A(am,variables,v)
                  ==>
                  exists i/AM_INI A(am,initializations,i)) $

END


INDIVIDUALCLASS InitialAbstractMachine IN MetaClass
                              ISA AbstractMachine WITH
  attribute
      tobeproven: ProofObligation
END



INDIVIDUALCLASS AM_DataClass IN MetaClass ISA SimpleClass

INDIVIDUALCLASS AM_BAS IN MetaClass ISA SimpleClass

INDIVIDUALCLASS AM_CTX IN MetaClass ISA SimpleClass WITH
  attribute
      basic_set: AM_BAS
      is_def_by: String
END

INDIVIDUALCLASS AM_VRB IN MetaClass ISA SimpleClass WITH
  attribute
      is_eq_to: AM_VRB
      is_defined: BooleanType (* it will be a AMALLiterals *)
      inv_on_vrbtype: Type
END

INDIVIDUALCLASS AM_INV IN MetaClass ISA SimpleClass

INDIVIDUALCLASS AM_OPN IN MetaClass ISA SimpleClass

INDIVIDUALCLASS AM_INI IN MetaClass ISA SimpleClass

INDIVIDUALCLASS AM_OTH IN MetaClass ISA SimpleClass
```

```
                    (* ------------------ *)
                    (*  AM - refinement   *)
                    (* ------------------ *)

ATTRIBUTECLASS AbstractMachine!dependsonAm WITH
  attribute
      tobeproven: ProofObligation
END

ATTRIBUTECLASS AbstractMachine!basicsets WITH
  dependson
      dependsonBas: AbstractMachine!basicsets
END

ATTRIBUTECLASS AbstractMachine!contexts WITH
  dependson
      dependsonCtx: AbstractMachine!contexts
END


ATTRIBUTECLASS AbstractMachine!contexts!dependsonCtx WITH
  attribute
      tobeproven: ProofObligation
END

ATTRIBUTECLASS AbstractMachine!variables WITH
  dependson
      dependsonVrb: AbstractMachine!variables
END

ATTRIBUTECLASS AbstractMachine!invariants WITH
  dependson
      dependsonInv: AbstractMachine!invariants
END

ATTRIBUTECLASS AbstractMachine!operations WITH
  dependson
      dependsonOpn: AbstractMachine!operations
END

ATTRIBUTECLASS AbstractMachine!operations!dependsonOpn WITH
  attribute
      tobeproven: ProofObligation
END

ATTRIBUTECLASS AbstractMachine!initializations WITH
  dependson
      dependsonIni: AbstractMachine!initializations
END

ATTRIBUTECLASS AbstractMachine!initializations!dependsonIni WITH
  attribute
      tobeproven: ProofObligation
END

ATTRIBUTECLASS AbstractMachine!others WITH
  dependson
      dependsonOth: AbstractMachine!others
END
```

```
INDIVIDUALCLASS AbstractMachineDependencies IN DecisionDescription WITH
  dependencies
      dep_bas: AbstractMachine!basicsets!dependsonBas
      dep_ctx: AbstractMachine!contexts!dependsonCtx
      dep_vrb: AbstractMachine!variables!dependsonVrb
      dep_inv: AbstractMachine!invariants!dependsonInv
      dep_opn: AbstractMachine!operations!dependsonOpn
      dep_ini: AbstractMachine!initializations!dependsonIni
      dep_oth: AbstractMachine!others!dependsonOth
      dep_am:  AbstractMachine!dependsonAm
END


                (* ------------------ *)
                (*  TDL - AM mapping  *)
                (* ------------------ *)


ATTRIBUTECLASS AbstractMachine!basicsets WITH
  dependson
      depsonTDLEntities: TDL_Design!entities
      depsonTDLEnumeratedClass: TDL_Design!enumerated
      depsonTDLAggregateClass: TDL_Design!aggregates
      depsonTDLBasicClass: TDL_Design!basicclasses
END

ATTRIBUTECLASS AbstractMachine!contexts WITH
  dependson
      depsonTDLEnumeratedClass: TDL_Design!enumerated
      depsonTDLAggregateClass: TDL_Design!aggregates
      depsonTDLBasicClass: TDL_Design!basicclasses
END

ATTRIBUTECLASS AbstractMachine!variables WITH
  dependson
      depsonTDLEntities: TDL_Design!entities
      depsonTDLEnt_unchanging: TDL_EntityClass!unchanging
      depsonTDLEnt_changing: TDL_EntityClass!changing
      depsonTDLBasicClass: TDL_Design!basicclasses
END


ATTRIBUTECLASS AbstractMachine!invariants WITH
  dependson
      depsonTDLEnt_unique: TDL_EntityClass!unique
      depsonTDLEnt_invariant: TDL_EntityClass!invariant
END

ATTRIBUTECLASS AbstractMachine!operations WITH
  dependson
      depsonTDLTransaction: TDL_Design!transactions
  attribute
      tobeproven: ProofObligation
END

ATTRIBUTECLASS AbstractMachine!initializations WITH
  attribute,single
      tobeproven: ProofObligation
END
```

```
INDIVIDUALCLASS FirstAbstractMachineDependencies IN DecisionDescription WITH
  dependencies
      dep_bas_ent: AbstractMachine!basicsets!depsonTDLEntities
      dep_bas_enum: AbstractMachine!basicsets!depsonTDLEnumeratedClass
      dep_bas_aggr: AbstractMachine!basicsets!depsonTDLAggregateClass
      dep_bas_bas: AbstractMachine!basicsets!depsonTDLBasicClass
      dep_ctx_enum: AbstractMachine!contexts!depsonTDLEnumeratedClass
      dep_ctx_aggr: AbstractMachine!contexts!depsonTDLAggregateClass
      dep_ctx_bas: AbstractMachine!contexts!depsonTDLBasicClass
      dep_vrb_ent: AbstractMachine!variables!depsonTDLEntities
      dep_vrb_ent_unchang:
        AbstractMachine!variables!depsonTDLEnt_unchanging
      dep_vrb_ent_chang:
        AbstractMachine!variables!depsonTDLEnt_changing
      dep_vrb_bas: AbstractMachine!variables!depsonTDLBasicClass
      dep_inv_ent_unique:AbstractMachine!invariants!depsonTDLEnt_unique
      dep_inv_ent_inv:AbstractMachine!invariants!depsonTDLEnt_invariant
      dep_opn_trans: AbstractMachine!operations!depsonTDLTransaction
      dep_tdl: AbstractMachine!dependsonTdl
END



              (* ------------------------------- *)
              (*  D e s i g n D e c i s i o n s  *)
              (* ------------------------------- *)


INDIVIDUALCLASS ModifyImplementationDesign IN DesignDecision ISA SimpleClass
WITH
  part
      verification: Proof
  by
      assistant: DBPL_MAP
END

INDIVIDUALCLASS MapToImplementationDesign IN DesignDecision
                             ISA ModifyImplementationDesign WITH
  from
      given: BaselineConceptualDesign
  to
      mapped: InitialImplementationDesign
  decisionsemantic
      mappingdescription: FirstAbstractMachineDependencies
  part
      verification: TransformationProof
      (* usage of development-theories of the B-Tool *)
  rule
      justrule: $ forall map/MapToImplementationDesign
                      des/ImplementationDesign
                  A(map,mapped,des) ==> A(des,creation,mod) $
END


INDIVIDUALCLASS RefineImplementationDesign IN DesignDecision
                               ISA ModifyImplementationDesign WITH
  from
      given: ImplementationDesign
  to
      refined: RefinedImplementationDesign
```

```
  decisionsemantic
      refinementdescription: AbstractMachineDependencies
  rule
      justrule: $ forall map/RefineImplementationDesign
                     des/ImplementationDesign
                 A(map,refined,des) ==> A(des,creation,mod) $
END


INDIVIDUALCLASS ReleaseImplementationDesign IN DesignDecision
                               ISA ModifyImplementationDesign WITH
  from
      given: ImplementationDesign
  to
      released: BaselineImplementationDesign
  decisionsemantic
      refinementdescription: AbstractMachineDependencies
  rule
      justrule: $ forall map/ReleaseImplementationDesign
                     des/ImplementationDesign
                 A(map,released,des) ==> A(des,creation,mod) $
END



(* Proof model: *)

INDIVIDUALCLASS ProofStepDescription IN DecisionDescription ISA SimpleClass WITH
  attribute
      tactic: BtoolTactic
      rulesequence: String
END

INDIVIDUALCLASS ProofStep IN DesignDecision ISA SimpleClass WITH
  from
      fromconjecture: ProofObligation
      fromtheories: BtoolTheory
  to
      toconjecture: ProofObligation
  by
      assistant: Btool
  decisionsemantic
      proof: ProofStepDescription
END


INDIVIDUALCLASS Proof IN DesignDecision ISA ProofStep WITH
  to
      toconjecture: TrueOrFalse
  part
      subproofsteps: ProofStep
      subproofs: Proof
END


INDIVIDUALCLASS TransformationProof IN DesignDecision ISA Proof WITH
  to
      totheory: BtoolTheory
END
```

```
                (* ---------------------- *)
                (*  D e s i g n T o o l s  *)
                (* ---------------------- *)

INDIVIDUALCLASS Btool IN DesignTool WITH
  attribute
      preloadedtheories: BtoolTheory
END


INDIVIDUALCLASS DBPL_MAP IN DesignTool ISA Btool
```

Anhang 3: Index der Grundbegriffe