Ein Transaktionskonzept für die deduktive Objektbank *ConceptBase*

(Fehlerreduzierte Version)

Markus Baumeister
geb.: 11.6.1968
Matr. Nr.: 183688

Diplomarbeit am
Lehrstuhl für Informatik V
Prof. Dr. Matthias Jarke
RWTH–Aachen
Betreuer: Dipl.–Inform. Rainer Gallersdörfer

Aachen, Januar 1994

Ich möchte hiermit all denen danken, die mich bei der Erstellung dieser Diplomarbeit unterstützt haben:
Rainer Gallersdörfer und Herrn Professor Dr. Matthias Jarke für ihre Betreuung;
Kai von Thadden, Martin Staudt, Dr. Manfred Jeusfeld und Hans Nissen, dem "harten Kern" der *Concept Base*–Entwickler, für ihre unbegrenzte Bereitschaft mich mit Antworten zu *ConceptBase* und deduktiven Datenbanken zu versorgen
sowie meinen Korrekturlesern Christina Roßmanith, Heike Wiegand–Baumeister und Kai v. Thadden für teilweise in Rekordzeit erarbeitete Verbesserungsvorschläge.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Einführung 1

1.2 Aufgabenstellung 1

1.3 Vorausgesetzte Begriffe 2

1.3.1 Datenbanken 2

1.3.2 Betriebssysteme, Nebenläufige Programmierung 2

1.4 Weiterer Aufbau 3

2 Grundlagen **4**

2.1 Deduktive Datenbanken 4

2.1.1 Anfragen, Regeln 4

2.1.2 Auswertungsstrategien 4

2.1.3 Datenspeicherung 5

2.2 *ConceptBase* 6

2.2.1 Anwendungsgebiet 6

2.2.2 Objektorientiertheit 6

2.2.3 Anfragen, Regeln 7

2.2.4 Datenspeicherung 8

2.2.5 Architektur 10

2.2.6 Messungen 13

2.3 Anforderungen an das Transaktionskonzept 14

3 Transaktionskonzepte **15**

3.1 Klassische Systeme 15

3.1.1 Das ACID-Prinzip und die Serialisierbarkeit 15

3.1.2 Isolationsgrade 17

3.1.3 Optimistische und pessimistische Verfahren 18

3.1.4 Granularität . . . 19
3.1.5 Sperrverfahren . . . 20
3.1.6 Zeitstempelverfahren . . . 23
3.1.7 Sonstiges . . . 24
3.2 Fortgeschrittene Systeme . . . 24
3.2.1 Erweiterung der Serialisierbarkeit . . . 26
3.2.2 Verringerung der Isolation . . . 29
3.2.3 Sonstiges . . . 34
3.3 Deduktive Datenbanken und Transaktionen . . . 37
3.3.1 Bedarf an Transaktionssystemen für deduktive DB . . . 37
3.3.2 Probleme deduktiver Transaktionskonzepte . . . 38
3.4 CB und Transaktionen . . . 40
3.4.1 Designumgebungen und lange Transaktionen . . . 40
3.4.2 Constraints und Integrität . . . 40
3.4.3 Objektidentität und Konflikterkennung . . . 41
3.5 Bewertung . . . 42

4 Das Transaktionssystem für ConceptBase 47

4.1 Entwurf . . . 47
4.1.1 Serialisierbarkeit wiederaufgegriffen – und fallengelassen . . . 47
4.1.2 Gruppen – jedem die seine . . . 51
4.1.3 Konflikte – erstmal finden . . . 53
4.1.4 Optimismus – und wenn's schiefgeht? . . . 55
4.1.5 Zugriffsbeschränkung – Finger weg von meinen Objekten! . . . 56
4.1.6 Zusammenfassung . . . 58
4.1.7 Korrektheit . . . 58
4.2 Implementierung . . . 59
4.2.1 Grobarchitektur . . . 59
4.2.2 Änderungen am Objektspeicher . . . 61
4.2.3 Überblick über die Feinarchitektur . . . 64
4.2.4 Schnittstelle zu *ConceptBase* . . . 68
4.3 Ergebnisse der Implementierung . . . 69
4.3.1 Probleme . . . 69
4.3.2 Nichtimplementierte Konzeptteile . . . 71
4.3.3 Messungen . . . 72
4.3.4 Kontrolle der Anforderungserfüllung . . . 73

5 Zusammenfassung **75**

Literatur **76**

Kapitel 1

Einleitung

1.1 Einführung

Während das Konzept der relationalen Datenbank nach seiner Entwicklung im Jahre 1970 [Cod70] relativ schnell kommerzielle Verbreitung fand, haben deduktive Datenbanken, obwohl bereits in den späten 70er Jahren erstmals entworfen [GM78], bisher lediglich auf dem wissenschaftlichen Sektor eine größere Bedeutung gewonnen.
Deduktive Datenbanken – also die Verknüpfung logischer Ableitungsfähigkeiten mit der Verwaltung größerer Datenmengen durch ein DBMS – eignen sich durch ihre deklarative, logikorientierte Anfragesprache gut für Anwendungen, die Ad–hoc–Anfragen oder transitive Hüllen benötigen. Ihre Verbreitung leidet aber unter der vergleichsweise geringen Anzahl passender Applikationen und den teilweise hohen Anforderungen an die Systemressourcen. Auch die Integration von Systemteilen, die im relationalen Modell bereits gut verstanden werden – wie z.B. einer Transaktionsverwaltung, ist bei deduktiven Datenbanken noch relativ erfolglos gewesen.

ConceptBase[JEG+93, Jar92] ist ein Wissensbankverwaltungssystem, das am Lehrstuhl Jarke in Passau und Aachen entwickelt wurde. Es beruht auf der Kombination einer deduktiven DB mit objektorientierten Konzepten. Obwohl mit Hilfe von *ConceptBase* bereits seit längerem Mehrbenutzeranwendungen z.B. im Bereich Gruppenorientierte Designumgebungen (CoAUTHOR[JS93], DAIDA[Jar93]) entwickelt wurden, fehlte bisher ein nennenswertes Transaktionskonzept, um die Interaktionen der Benutzer auf eine sichere Basis zu stellen.

1.2 Aufgabenstellung

Die Entwicklung und – falls möglich – Implementierung eines Transaktionskonzeptes für *ConceptBase* ist das Ziel dieser Diplomarbeit. Das Transaktionskonzept soll eine koordinierte, gleichzeitige Benutzung der Wissensbank durch mehrere Benutzer ermöglichen. Insbesondere müssen Konflikte, die durch die konkurrierende Bearbeitung des gleichen oder zusammenhängender Objekte entstehen, verhindert oder erkannt und behoben werden. Zudem werden durch Transaktionen inkonsistente Zwischenzustände zwischen TELL-/UNTELL-Operationen, die bisher jeweils als eine Transaktion gelten, ermöglicht.

Berücksichtigt werden muß das Hauptanwendungsgebiet von *ConceptBase*: die Bereitstellung eines Wissensspeichers für Entwurfsumgebungen. Hieraus leitet sich die Forderung nach der Unterstützung langer Transaktionen ab. Um der hohen Systembelastung bei großer Nutzerzahl gerecht werden zu können, sollten Transaktionen parallel – auf Rechnern mit SMP (Symmetric Multi Processor) sogar echt parallel – abgearbeitet werden können.

Die Implementierung des Konzepts soll auf dem von Hubertus Pützer [Püt92] in C++ entwickelten Objektspeicher beruhen, der die bisherige Verwaltung der Daten als ProLog–Fakten ablöst.

Um die Aufgabe zu begrenzen, wird die eigentlich zu einem Transaktionskonzept zu leistende Bereitstellung von Typ 2- bis Typ 4-Recovery nur insofern beachtet, daß sie nicht völlig unmöglich gemacht werden sollte.

1.3 Vorausgesetzte Begriffe

Im weiteren Verlauf der Diplomarbeit werden gewisse Begriffe aus dem Bereich der Datenbanken und der Nebenläufigen Programmierung als bekannt vorausgesetzt. Sie sollen hier kurz vorgestellt werden.

1.3.1 Datenbanken

Der Begriff der **relationalen Datenbank** und des Datenbank Management Systems sollte allgemein bekannt sein. Ein passendes Buch wäre andernfalls z.B. [Dat90]. Aus dem Bereich der **Objektorientierung** bzw. der OODB, aus dem viele der in Kap. 3.2 vorgestellten Transaktionskonzepte stammen, werden besonders die Konzepte der Objektkapselung und der Objektidentität sowie die Klassenhierachie benötigt ([Mey88]).
Obwohl ich in Kap. 3.1 auf die verschiedenen bekannten **Transaktion**skonzepte eingehen werde, sollte ein allgemeines Verständnis der Problematik sowie der klassischen Grundlagen wie z.B. Serialisierbarkeit, ACID, Commit-Verfahren vorhanden sein ([BHG87, VGH93]).
Verschiedene Arten und Anforderungen von **Entwurfsanwendungen** werden u.a. in [BK91] beschrieben.

Eine gewisse Merhfachbelegung existiert beim Begriff "Objekt". Während die offizielle Sprachregelung zu *ConceptBase* "Objekt" äquivalent zu Proposition setzt, bezeichnet "Objekt" in der Literatur eher die Instanz einer Klasse (in *ConceptBase* also eine Menge von Propositionen). In dieser Diplomarbeit wird "Objekt" hauptsächlich im zweiten Sinne verwendet werden, wenn auch gelegentliches Abwandern zur ersten Bedeutung nicht vermieden werden konnte, so z.B. bei der Bezeichnung "Objektserver", der natürlich keine Objekte sondern Propositionen zur Verfügung stellt.

1.3.2 Betriebssysteme, Nebenläufige Programmierung

Um insbesondere die in den Kapiteln zu Implementierung und Architektur behandelten Probleme besser verstehen zu können, wäre vorhandenes Wissen über nebenläufige Programmierung (engl. Concurrency) und damit verbundene Begriffe wie gegenseitiger Ausschluß (engl. kurz Mutex) und andere Synchronisationsmethoden von Vorteil (vgl. [Gos91, Tan92]). Die in dieser Arbeit verwendete Definition von 'Thread', 'LWP' (light weighted Process) und 'Prozeß' stimmt mit der von SUN in der Solaris-Dokumentation [Sun93a] verbreiteten überein, die sich ihrerseits am POSIX–Standard orientiert.

1.4 Weiterer Aufbau

Im nächsten Kapitel werde ich auf die für diese Arbeit wichtigen Besonderheiten deduktiver Datenbanken sowie auf *ConceptBase* eingehen. Das darauffolgende Kapitel gibt eine Übersicht über viele klassische oder speziell für Entwurfsanwendungen entwickelte Transaktionskonzepte. Anschließend werden sie bewertet und es werden Probleme aufgezeigt, die bei der Integration eines Transaktionssystems in eine deduktive Datenbank auftreten können.
Das vierte Kapitel beschreibt das für *ConceptBase* entwickelte Transaktionskonzept: Einerseits die Planvorstellungen, andererseits den tatsächlichen Stand der Implementierung. Anschließend werden die aus der Implementierung gewonnen Erfahrungen vorgestellt. Den Abschluß bildet eine Zusammenfassung der Ergebnisse dieser Arbeit.

Kapitel 2

Grundlagen

2.1 Deduktive Datenbanken

In diesem Kapitel wird ein kurzer Überblick über die wichtigsten Merkmale deduktiver Datenbanken gegeben. Das Hauptaugenmerk liegt auf solchen, die sich auf den Entwurf eines Transaktionskonzeptes auswirken können.

2.1.1 Anfragen, Regeln

Die Anfragesprache deduktiver Datenbanken unterscheidet sich von der relationaler Systeme (z.B. SQL) in 3 Punkten:

1. Sie ist abstrakter: Um Joins, Selektionen oder Projektionen muß sich der Benutzer keine Gedanken machen.
2. Sie ist mächtiger: Es läßt sich die transitive Hülle einer Daten– und Regelmenge berechnen.
3. Fakten, die mit Hilfe der in der (extensionalen) Datenbasis abgespeicherter Regeln berechnet werden können, gelten als ebenfalls in der (intensionalen) Datenbasis liegend.

In deduktiven Datenbanken werden die Daten als Fakten (z.B. `wohnt(gertrud,berlin).`) und als Regeln (`vorfahr(X,Y):- parent(X,Z), vorfahr(Z,Y).` vgl. unten) abgelegt. Obige Notation wurde der bekanntesten deduktiven Datenbanksprache "Datalog$^{\neg}$" entnommen und stimmt bis auf das Verbot von Funktoren in Argumenten mit ProLog überein.

2.1.2 Auswertungsstrategien

Da die Anfragesprache von deduktiven Datenbanken abstrakt ist (im Gegensatz zu SQL, wo der Lösungsweg vorgegeben werden muß), bleiben dem System eine Vielzahl von Optimierungsmöglichkeiten offen. Eine davon ist die Wahl der Auswertungsstrategie. Hierzu gibt es zwei Grundprinzipien (vgl. [Jar91, CGT90, Kem91]):

1. Vorwärtsauswertung (Bottom-Up)
2. Rückwärtsauswertung (Top-Down)

Während die Vorwärtsauswertung durch Ableitung aller möglichen Fakten das kleinste Herbrand–Modell der Datenbasis berechnet und hieraus die Anfrage befriedigt, arbeitet die Rückwärtsauswertung ProLog–artig. Sie versucht die Anfrage durch Resolution auf abgespeicherte Fakten zu reduzieren.

Beispiel 2.1 *Betrachten wir folgende Fakten- und Regelmenge:*

wohnt(gertrud,berlin).	parent(gertrud,werner).	vorfahr(X,Y):-parent(X,Y).
wohnt(werner,berlin).	parent(gertrud,hilde).	vorfahr(X,Y):-parent(X,Z),
wohnt(hilde,essen).	parent(otto,werner).	vorfahr(Z,Y).
wohnt(rudi,frankfurt).	parent(otto,hilde).	
wohnt(dora,frankfurt).	parent(lucy,rudi)	
wohnt(lucy,dortmund).	parent(lucy,dora).	
wohnt(meinold,dortmund).	parent(meinold,otto).	
	parent(meinold,lucy).	

vorfahr_in(X,Y,S):-wohnt(Y,S), vorfahr(X,Y).

und die Anfrage ?vorfahr_in(meinold,Y,berlin).

Ein **Vorwärtsauswerter** berechnet nun als erstes alle herleitbaren “vorfahr“ und “vorfahr_in“ Fakten (also unter anderem vorfahr(gertrud,werner), vorfahr(gertrud,hilde), vorfahr(otto,werner), ..., vorfahr(meinold,otto), vorfahr(meinold,lucy), vorfahr(meinold,werner), vorfahr(meinold,hilde), vorfahr(meinold,rudi), vorfahr(meinold,dora) sowie etwa vorfahr_in(gertrud, werner, berlin), ..., vorfahr_in(meinold, werner, berlin), vorfahr_in(meinold, hilde, essen), vorfahr_in(meinold, rudi, frankfurt),...).
Hieraus wird dann die Antwort Y=“werner“ auf die Anfrage gesucht.

Ein Vorteil dieses Verfahrens liegt in der gesicherten Terminination, da irgendwann ein Fixpunkt der Berechnung erreicht wird. Des weiteren lassen sich die von Vorwärtsauswertern durchgeführten Operationen einfach auf Befehle aus der relationalen Algebra abbilden.

Ein **Rückwärtsauswerter** versucht als erstes das Unterziel wohnt(Y,berlin) zu befriedigen und erhält Y={“gertrud“,“werner“}. Nun versucht er für jeden der beiden zu beweisen, daß sie Vorfahren von meinold sind (die zweite Voraussetzung für vorfahr_in). Dies läßt sich nur bei “werner“ über vorfahr(meinold,werner):- parent(meinold,otto), vorfahr(otto,werner) zeigen.

Rückwärtsauswertung vermeidet insbesondere die bei der Vorwärtsauswertung durchgeführte Berechnung nicht benötigter Fakten und ist darum meist schneller. Allerdings besteht bei linksrekursiven Regeln die Gefahr von Endlosschleifen.

Mischverfahren (wie z.B. Magic Set) versuchen, die Vorteile beider Verfahren zu kombinieren. So führt Magic Set eine Vorwärtsauswertung durch (und umgeht so Probleme mit unendlicher Rekursion), die durch ein ProLog–artiges Weiterreichen der instanziierten Argumente der Anfrage eingeschränkt wird.

2.1.3 Datenspeicherung

Zur Abspeicherung der Fakten wird meist ein bestehendes relationales System genutzt, da eine natürliche Beziehung zwischen einer Menge von Fakten gleichen Namens und gleicher Parameterzahl und einer Relation mit entsprechender Spaltenzahl besteht.

Beispiel 2.2 *Die "*wohnt*"–Fakten im obigen Beispiel würden als Relation etwa so aussehen:*

wohnt

name	**ort**
gertrud	*berlin*
werner	*berlin*
hilde	*essen*
rudi	*frankfurt*
dora	*frankfurt*
lucy	*dortmund*
meinold	*dortmund*

Auf diese Relationen kann nun einfach per Index zugegriffen werden, wenn ein bestimmtes Fakt gesucht wird, oder es können – bei Vorwärtsauswertung – Operationen der Relationalen Algebra angewendet werden.

2.2 *ConceptBase*

2.2.1 Anwendungsgebiet

ConceptBase [JEG+93, Jar92] ist ein experimentelles Wissensbankverwaltungssystem, das die Wissensrepräsentationssprache Telos [KMSJ88, MBJK90] in einer Client/Server–Architektur implementiert und weltweit von mindestens einem Dutzend Forschungseinrichtungen benutzt wird. Telos erlaubt es, zusammenhängendes Wissen in Form eines semantischen Netzes zu bearbeiten, auf dessen Basis objektorientierte Eigenschaften verwirklicht werden.
Die Anwendungsgebiete von *ConceptBase* liegen hauptsächlich in den Bereichen Gruppenarbeit und Entwurfsunterstützung ,wie neben Projekten wie DAIDA [Jar93] oder CoAuthor [Ehe93] auch folgendes Zitat zeigt:

> „*ConceptBase* ist ein experimentelles deduktives Objektbank–Managementsystem, speziell entworfen für die Koordination in Designumgebungen.“[1]

Entwurfsumgebungen erfordern – wie in Kapitel 3.2 erläutert wird – die Möglichkeit langdauernder, interaktiver Transaktionen und erhöhter Kooperation zwischen den Benutzern.

2.2.2 Objektorientiertheit

In *ConceptBase* wurde die von einigen Forschern für unmöglich [Ull91] bzw. nicht wünschenswert (F.Bancilhon in [BBH+89]) erklärte Kombination von deduktiven und objektorientierten Datenbankkonzepten zumindest teilweise verwirklicht. Erreicht wird dies durch den Aufsatz eines objektorientierten Datenmodells auf ein grundsätzlich deduktives System. Objektorientiertheit zeigt sich in zwei Ebenen. Zum einen besitzt jede "Proposition" (die kleinste Speichereinheit; vgl. übernächster Abschnitt) einen eindeutigen Identifikator. Zum anderen

[1] nach [Jar92, Abstract], eigene Übersetzung

sind durch verschiedene Axiome ([JEG+93]) über den Propositionen Klassifikation, Aggregation und Generalisierung definiert. Hierbei wird die Klassenhierarchie in gleicher Weise auf Propositionen abgebildet, wie die Instanzen, so daß Änderungen der Klassenhierarchie (und damit des DB–Schemas) zur Laufzeit leicht möglich sind. Die durch diese Struktur erzeugten Objekte können vom Benutzer entweder grafisch oder in einer 'Frame'–Schreibweise sichtbar gemacht werden. Ein Frame repräsentiert ein Objekt sowie die ihm zugeordneten Attribute und Regeln.

Beispiel 2.3 *Ein oft verwendetes Beispiel (vgl. [Jar92, Kap. 2]) ist das Modell einer Firma mit Angestellen ('Employee'), Abteilungen ('Department') und Abteilungsleitern ('Manager'). Die Klasse Employee und eine Instanz von ihr werden in Frame–Notation folgendermaßen dargestellt:*

```
Class Employee with
   attribute
      name: String;
      salary: Integer;
      dept: Department
end
```

```
Individual Max in Employee with
   name
      n: “Max Müller“
   salary
      brutto: 4000
   dept
      abgestellt_zu: Konstruktion;
      angestellt: Entwicklung
end
```

wobei Konstruktion *und* Entwicklung *jeweils Instanzen der Klasse* Department *sein müssen.*

Im Gegensatz zu einer vollständigen Objektorientierung fehlen aber Datenkapselung und Methoden. Objekte sind also nur „bessere Strukturen“, da der Zugriff auf die einzelnen Attribute unbeschränkt ist. Auch können die vorhandenen Mechanismen zur Integration von Regeln als Attribute oder eigene Klassen die Mächtigkeit eines Methodenkonzepts nicht erreichen.

2.2.3 Anfragen, Regeln

Es gibt mehrere Arten, wie Regeln in *ConceptBase* verwendet werden können (vgl. [Jar92, Kap. 2]):

1. als 'Rules' zur Berechnung von Attributen
2. als 'Constraints' zur Beschränkung der Instanzen von Klassen
3. als Selektionskriterien innerhalb von 'Query-Classes' (Anfragen)

Die Punkte 1 und 2 werden durch das folgende Beispiel demonstriert.

Beispiel 2.4 *Als Ergänzung zur im obigem Beispiel definierten Klasse* Employee*:*[2]

```
Class Employee with
  attribute
    child: Person;
    tax: Integer
  rule,bdmrule
    Taxrule1: $ forall e/Employee i/Integer (exists p/Person
                (e child p) and (e salary i)) ==> (e tax i*0.3) $
    Taxrule2: $ forall e/Employee i/Integer (not exists p/Person
                (e child p) and (e salary i)) ==> (e tax i*0.4) $
  bdmconstraint
    DepSalary: $ forall e/Employee (e dept Entwicklung) and (e salary i)
                 ==> i >= 2000 $
end
```

Die dem Objekt zugeordneten Regeln berechnen die Steuern der Angestellten in Abhängigkeit der Kinderzahl. Die Beschränkung legt eine Untergrenze für das Gehalt der Angestellten der Abteilung Entwicklung *fest. Zu dieser Beschränkung nicht konforme Objekte können nun nicht mehr in die Datenbasis eingefügt werden.*

Wie man sieht, können Regeln teilweise betrachtende Aufgaben ("Observer Functions"; vgl. [KM92, S.179f]) der hier nicht vorhandenen Methoden übernehmen. Objektänderungen ("Mutator Functions") müssen jedoch direkt durch die Applikation mittels 'TELL' bzw. 'UNTELL' (statt über Methoden) vorgenommen werden.

Jede Regel muß dem System vor der ersten Verwendung bekannt gemacht werden. Sie wird neben ihrer Abspeicherung als String in der Propositionen–Datenbasis intern mehrfach umgewandelt bzw. vereinfacht, bevor sie mittels eines ProLog–Fakts in das *ConceptBase* implementierende Programm eingefügt wird. Erst danach kann sie aufgrund von Anfragen oder Änderungen an der Datenbasis (bei 'Constraints') ausgewertet werden.

Für die Auswertung selbst stehen dem Benutzer zwei Verfahren (vgl. Kap. 2.1.2) zur Verfügung, die er für jede Regel seperat selektieren kann:

- Eine ProLog–artige Rückwärtsauswertung
- Eine 'gemischte' Strategie mit Magic Sets

Zu jeder Anfrage kann eine 'Rollback–Time' angegeben werden, die bestimmt, welcher zeitliche Zustand der Datenbank betrachtet werden soll (vgl. nächster Abschnitt).

2.2.4 Datenspeicherung

Propositionen

Wie bereits oben erwähnt, werden die Daten in *ConceptBase* nicht direkt in der obigen 'Frame'–Repräsentation abgespeichert. Die 'Frames' werden vielmehr erst in kleinere Ein-

[2] Dieses Beispiel dient nur der Veranschaulichung. Es ist keineswegs korrektes Telos, da Rechenoperationen und Quantifizierungen über Integervariablen nicht erlaubt sind.

heiten – die sogenannten 'Propositions' – aufgeteilt. Eine Proposition besteht aus fünf Komponenten:

proposition(id, src, label, dest, tt)

id: Ein eindeutiger Identifikator, der in anderen Propositionen als 'src' oder 'dest' Komponente auftauchen kann.

src: Der Beginn der Verbindung zwischen zwei Objekten, die die Proposition repräsentiert. Steht die Proposition für ein Individualobjekt (also für eine Klasse oder Instanz), so ist id=src=dest.

label: Der Name der Verbindung bzw. des Objekts.

dest: Das Ziel der Verbindung.

tt: Das Zeitintervall (möglicherweise rechtsseitig unbeschränkt), in dem die Proposition dem DBMS bekannt ist, Systemzeit genannt. Falls eine Proposition für ungültig erklärt wird (UNTELL), so wird dieses Intervall abgeschlossen und die Proposition in die historische Datenbank verlagert. Dies ermöglicht es, Anfragen über vergangene Zustände der Datenbasis zu stellen.

Wenn man diese Art der Speicherung mit Kapitel 2.1.3 vergleicht, kann man sie auch als Verschärfung der dort angegebenen Ideen verstehen, indem alle Daten in nur einer Relation abgespeichert werden.

Beispiel 2.5 *Der Frame 'Max' aus Beispiel 2.3 wird durch folgende Propositions repräsentiert:*[3]

```
prop(oid_1000,oid_1000,'Max',oid_1000,t)
prop(oid_1001,oid_1000,'*instance_of',oid_200,t)
prop(oid_1002,oid_1000,'n',oid_2000,t)
prop(oid_1003,oid_1002,'*instance_of',oid_202,t)
prop(oid_1004,oid_1000,'brutto',oid_2001,t)
prop(oid_1005,oid_1004,'*instance_of',oid_204,t)
prop(oid_1006,oid_1000,'angstellt',oid_300,t)
...
```

wobei oid_200 *die Klasse* Employee *und* oid_300 *die Instanz* Entwicklung *von* Department *repräsentiert.*

Diese Propositionen werden bei einem 'TELL' in die Datenbasis eingefügt und bei einem 'UNTELL' wieder entfernt[4]. Es ist nicht möglich, eine einmal ge'TELL'te Proposition zu ändern. Auch indirekt, über ein 'UNTELL' mit nachfolgendem 'TELL' der abgeänderten Proposition, ist dies streng genommen unmöglich, da durch die Eindeutigkeit der Identifikatoren[5] die neue Proposition einen anderen Identifikator erhält, also *neu* und nicht *geändert* ist.

[3] Da die Systemzeit nicht weiter interessiert, wird sie als 't' angegeben

[4] genauer: in die historische Datenbasis übertragen

[5] Die Eindeutigkeit kann nur während der Laufzeit des Servers garantiert werden. Um die Domäne der IDs beschränkt halten zu können, müssen diese gelegentlich reorganisiert werden.

Die in Kapitel 2.2.3 vorgestellten Regeln und 'Constraints' werden ebenfalls als Propositionen abgelegt. Hierbei enthält die label–Komponente die unverarbeitete Regel einschließlich der '$'–Zeichen. Um eine Neuübersetzung bei jedem Zugriff zu vermeiden, werden sie außerdem im *ConceptBase*–Kern intern als ProLog–Regeln gespeichert. Diese werden nicht bei jedem Hochfahren des Servers neu berechnet oder zusammen mit den Daten abgelegt, sondern in einem separaten 'OB.rule'–File gesichert. Die Verwaltung der Regeln läuft also fast völlig an den an der Datenspeicherung beteiligten Modulen vorbei.

Objektspeicher

Die Speicherung der Propositionen erfolgte lange Zeit direkt als Fakten im ProLog–Auswerter. Trotz aller Tricks (Erzeugung von "Indizes" durch unterschiedliche Faktennamen für unterschiedliche Label, ...) ist diese Methode sowohl durch mangelnde Geschwindigkeit als auch durch Beschränkungen der Datenbasisgröße gehandicapt. Die von Rainer Gallersdörfer [Gal90] und Hubertus Pützer [Püt93] entworfenen Objektspeicher sollen diesen Problemen abhelfen.

Der Objektspeicher hält weiterhin alle Objekte im Hauptspeicher und überläßt damit die Pufferverwaltung quasi dem Betriebssystem. Lediglich zur Herstellung der Persistenz finden Sekundärspeicherzugriffe statt. Durch die Verwendung von Indexstrukturen kann die Suche nach einer Proposition nun aber in logarithmischer (Baum–Verfahren) oder 'konstanter' Zeit (Hashing) durchgeführt werden.

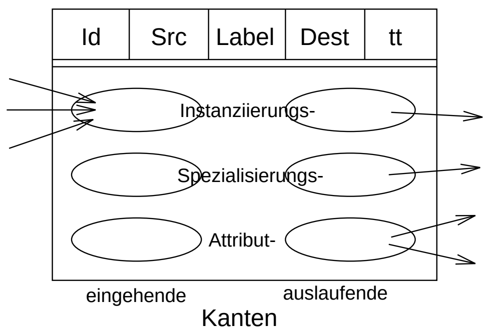

Abbildung 2.1: Schema eines komplexen TelosObjektes (TOB)

Des weiteren werden im Objektspeicher neben den eigentlichen Propositioninformationen noch zusätzlich – und redundant – die Identifikatoren der mit der jeweiligen Proposition verknüpften Propositionen abgespeichert (s. Bild 2.1). Für alle ein- und auslaufenden Kanten existiert – getrennt nach der Art der Kante – eine eigene Menge. Dies ermöglicht einen schnellen Zugriff auch anhand "einlaufender" Propositionen (z.B. finde alle Instanzen einer Klasse) sowie die direkte Anwendung von Mengenoperationen während der Vorwärtsauswertung. Dies wird z.B. von H. Thönnissens Objektalgebra [Thö92] ausgenutzt.

2.2.5 Architektur

In der Grobarchitektur entspricht *ConceptBase* einem typischen Client/Server–System (s. z.B. [And91]). Die verschiedenen Applikationen (Toolbar, GraphBrowser, Editor, ...), die die Schnittstelle zum Benutzer bilden, stellen die Klienten dar. Diese Klienten sind jeweils über ein 'Socket' mit dem als Server fungierenden *ConceptBase*–Kern verbunden. Die Verbindung wird vom "CB–Interface" verwaltet, das den Klienten vordefinierte C–Funktionen für Verbindungsaufbau, Datenmanipulation, Anfragen, ... und dem Server die für ProLog aufbereiteten Daten zur Verfügung stellt. Ein Kommando der Applikation wird dem Server übermittelt, dort abgearbeitet, und das Ergebnis wird zur Applikation zurückgesendet. Befehle verschiedener Applikationen werden streng seriell abgearbeitet. Das kann zur Folge haben, daß ein arbeitaufwendiges Kommando viele nachfolgende Befehle (und damit auch die zugehörigen Applikationen) blockiert.

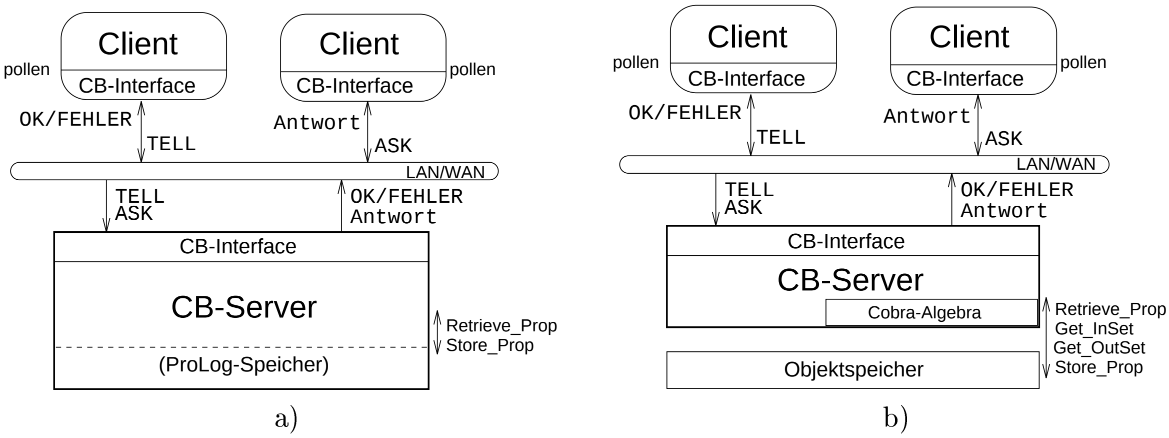

Abbildung 2.2: Grobarchitektur von *ConceptBase* (a) mit ProLog–Speicher und (b) mit Objektspeicher

Durch die Integration des Objektspeichers ändert sich nur wenig an der Architektur. Lediglich die für die Persistenz der Propositionen zuständigen ProLog–Module werden nicht mehr benötigt.

Mehr auf die Einzelheiten eingehende Architekturdiagramme finden sich in verschiedenen Veröffentlichungen [Gal90, Jeu92, JEG$^+$93]. Weil es jedoch keine offizielle Darstellung gibt, unterscheiden sich alle Bilder je nach Standpunkt und Zielsetzung des Autors. In dieser Tradition fortfahrend möchte ich hier die für diese Arbeit wichtigen Architekturteile präsentieren[6].

Zuerst sei kurz Bild 2.3 beschrieben, in dem die Architekturteile gezeigt werden, die bei einem TELL/UNTELL von Bedeutung sind.
Im Modul IPCChannel befindet sich die Hauptschleife, die auf Nachrichten der Klienten wartet und diese an CBserverinterface zur Verarbeitung weiterreicht. Alle Befehle, die sich nicht sofort befriediegen lassen (also TELL, UNTELL, ASK, HYPO-ASK), werden an Funktionen aus TellAndAsk weitergegeben, das sie mit Hilfe von Prädikaten aus Languageinterface von der 'Frame'– in die 'SML'–Repräsentation umwandelt. Die SML–Fragmente werden dann vom ObjectProcessor weiterverarbeitet, indem er z.B. bei einem TELL zuerst den ObjectTransformer veranlaßt, sie temporär abzuspeichern, dann die Integrität vom Modul SemanticIntegrity testen läßt und im Erfolgsfall die temporär gespeicherten Propositionen und Regelinformationen unter Aufruf diverser Untermodule permanent macht.

Die Aufgaben des ObjectTransformers liegen darin, unter Zuhilfenahme von FragmentToProposition aus den SML–Fragmenten Propositionen zu erzeugen, die FragmentToProposition mit

[6]Zur Unterstützung dieser Untersuchung wurde von mir ein kurzes Programm geschrieben, das die ":-import"–Anweisungen der ProLog–Files auswertet und in ein Netz aus *ConceptBase*–Klassen umwandelt

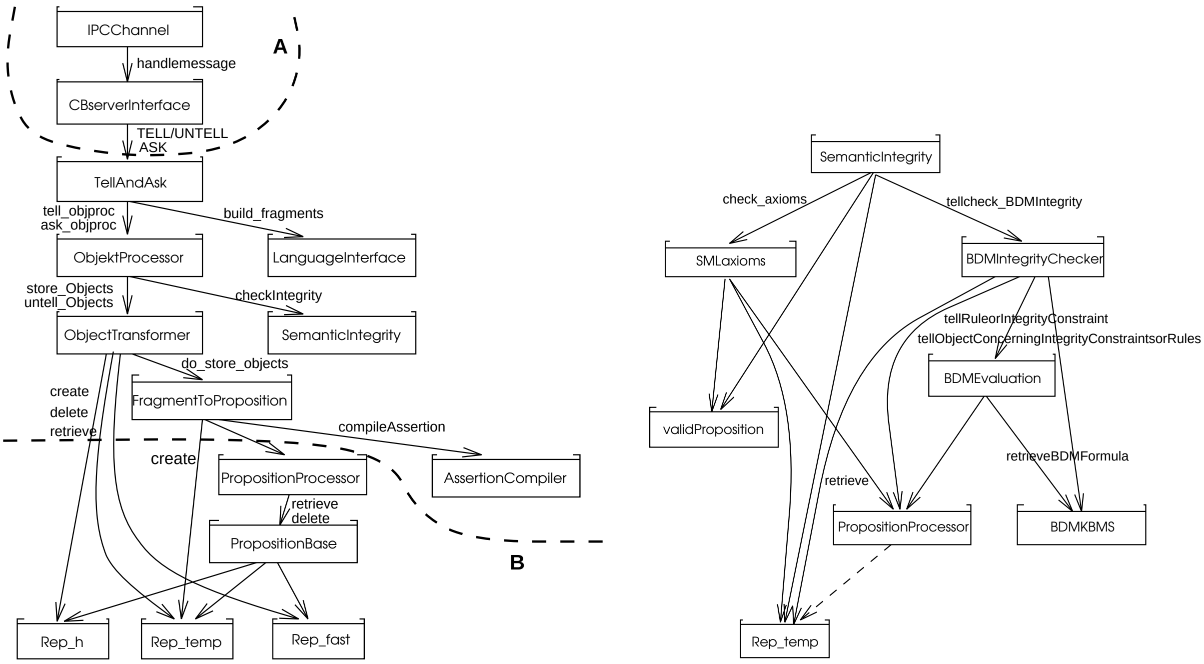

Abbildung 2.3: Teile der Architektur des CB–Kerns bei TELL/UNTELL

dem STORE Prädikat in Rep_temp temporär ablegt. Wenn dies mit allen SML–Fragmenten geschehen ist, ruft ObjectTransformer erneut FragmentToProposition auf, damit nun alle bisher nur in String–Repräsentation gespeicherten 'Assertions' (Regeln) in ein für *ConceptBase* verständliches Format umgewandelt werden. Dies geschieht im AssertionCompiler, dessen Interna den *ConceptBase*–'Freaks' überlassen bleiben sollen.

Änderungen in der Architektur müssen hauptsächlich an zwei Stellen durchgeführt werden: Erstens müssen die obersten Module (Kreis A) an die durch ein Transaktionssystem zusätzlich zur Verfügung stehenden Befehle angepaßt werden. Da alles unterhalb von Linie B durch den Objektspeicher ersetzt wird, erscheint es zweitens sinnvoll, hier die Hauptteile des Transaktionssystems einzufügen, die den Zugriff auf die Propositionen regeln sollen. Wünschenswert ist auch eine direkte Verbindung der beiden Teile, um Informationen wie z.B. die ID der derzeit behandelten Transaktion ohne den Umweg über den ProLog–Teil den Verwaltungsroutinen direkt mitteilen zu können.

Der z.Z. im Anschluß an jedes TELL/UNTELL durchgeführte Integritätstest wird wie gesagt vom Modul SemanticIntegrity durchgeführt. Dies unterscheidet dabei den Test auf Erfüllung der Axiome 13, 14, 15 (teilweise), 16, 3 und 4 von O–Telos (vgl. [JEG$^+$93]), der von SMLaxioms ausgeführt wird, und den Test auf Einhaltung der Benutzer–definierten Beschränkungen in BDMIntegrityChecker. Dieser stützt sich auf die diversen BDM–Module ab, die allesamt BDMKBMS zur Speicherung der vorübersetzten Regeln und 'Constraints' benutzen.

Änderungen in diesem Bereich betreffen zum einen das Aufbrechen der festen TELL→Integritätstest–Abfolge in ein TELL→Axiomtest, COMMIT→Constrainttest. Desweiteren muß der Zugriff auf die mittels BDMKBMS gespeicherten Regeln an und für sich genauso überwacht werden, wie die Zugriffe auf Propositionen über PropositionProcessor.

2.2.6 Messungen

Zur Analyse des derzeitigen Zugriffsverhalten und der Zugriffsgeschwindigkeit wurden verschiedene Messungen durchgeführt, die in Tabelle 2.1 zu sehen sind.

Einerseits interessiert das Verhältnis von Schreib- zu Lese–Zugriffen auf die Datenbasis. Hierzu gibt Tabelle 2.1 sämtliche Zugriffsversuche – auch die nicht erfolgreichen – an, da in hauptspeicherorientierten Datenbanken erfolgreiche und fehlgeschlagende Zugriffe die gleiche Komplexität besitzen. Die angegeben Daten erstaunen auf den ersten Blick, weil Änderungsoperationen mehr Lesezugriffe erfordern als Anfragen. Dies läßt sich aber dadurch erklären, daß bei Änderungen die 'Constraints' überprüft werden müssen, was eine größere Zahl an Leseoperationen verursacht.

Es bleibt festzuhalten, daß an der Propositionsschnittstelle Lesezugriffe bei weitem überwiegen, was beim zu entwerfenden Client/Server–Konzept in der Cache–Strategie berücksichtigt werden muß.

Operation	Read Temp	Read Fast	Write Temp	Write Fast	Del. Temp
Hochfahren CoAuthor	4323	3816	0	0	0
Einfügen Idee	44	175	6	6	6
Einfügen Anmerkung	486	236	10	10	10
einfache Anfrage[7]	38	32	0	0	0
komplexe Anfage[8]	252	210	0	0	0

Tabelle 2.1: Anzahl der Zugriffe auf Propositionen

Um andererseits einen Eindruck von den durch das Transaktionskonzept erzeugten Leistungsverlusten erhalten zu können, sind Daten über die Zugriffsgeschwindigkeit vor Einbau des Transaktionssystems vonnöten. Die in Tabelle 2.2 aufgeführten Daten gelten für das *ConceptBase*–System mit C++–Objektspeicher [Püt93] und wurden auf der C++–Seite der Propositionenschnittstelle ermittelte.

Suchmuster	OID	Src.	Label	Dest.	Zeit pro Zugriff [ms]
1	frei	frei	frei	inst.	0.275
2	frei	frei	inst.	frei	0.037
3	frei	frei	inst.	inst.	0.343
4	frei	inst.	frei	frei	0.150
5	frei	inst.	frei	inst.	0.755
8	inst.	frei	frei	frei	0.128

Tabelle 2.2: Zugriffszeiten ohne TA–System

Die Suchmusternummern entsprechen den intern vergebenen Identifikatoren für die verschiedenen Anfragetypen. Sie geben an, welche Teile der gesuchten Proposition instanziiert sind und welche frei.

[7] Die Anfrage war: [each,_4808,where,AttrValue(Test_STR,createdBy,_4808,_4818)] (Achtung! Dieses Format wird offiziell nicht mehr unterstützt.)

[8] QueryClass ConflictingIdeas: Sucht alle Ideen, an denen sowohl eine zustimmende als auch eine ablehnende Bemerkung hängt (Datenbasis ca. 20 Ideen; 1 Erfolg)

2.3 Anforderungen an das Transaktionskonzept

Um die im folgenden Kapitel vorgestellten Konzepte bewerten zu können, werden die an das für *ConceptBase* zu entwickelnde Transaktionskonzept gestellten Anforderungen hier nochmals zusammengefaßt:

- Gewährleistung der bekannten Transaktionskriterien
- Unterstützung langer Transaktionen
- Kooperation zwischen den Benutzern
- keine Festlegung auf ein bestimmtes Anwendungsgebiet
- Verbesserung der Antwortzeiten bei gleichzeitigen Befehlen mehrerer Benutzern
- geringe Leistungsverluste
- Abwärtskompatibilität zu alten Applikationen

Kapitel 3

Transaktionskonzepte

Nachdem in den beiden vorhergehenden Kapiteln die Aufgabenstellung und die Besonderheiten deduktiver Datenbanksysteme im allgemeinen und von *ConceptBase* im besonderen dargestellt wurden, widmet sich dieses Kapitel den verschiedenen bekannten Transaktionskonzepten. Nach einem kurzen Überblick über die klassischen Verfahren wird ausführlich auf die für diese Diplomarbeit bedeutsameren fortgeschrittenen Konzepte eingegangen. Anschließend werden Probleme, die bei der Kombination deduktiver Datenbanksysteme bzw. *Concept Base*s mit einem Transaktionssystem auftreten können, behandelt. Unter diesen Gesichtpunkten und auf Grundlage der im letzten Kapitel aufgestellten Anforderungen werden zuletzt alle vorgestellten Konzepte auf ihre Verwendbarkeit in *ConceptBase* untersucht.

3.1 Klassische Systeme und Grundlagen

3.1.1 Das ACID-Prinzip und die Serialisierbarkeit

Die meisten in der Literatur vorgestellten (klassischen) Transaktionssysteme werden daran gemessen, inwiefern sie die ACID–Prinzipien [HR83, Gra81] erfüllen. Sie sollen darum hier kurz wiederholt werden:

Atomicity / Atomarität Eine Transaktion wird entweder vollständig oder gar nicht ausgeführt.

Consistency / Konsistenz Eine Transaktion überführt einen konsistenten Zustand der Datenbank in einen anderen konsistenten Zustand.

Isolation Die Zwischenergebnisse gleichzeitig ablaufender Transaktionen dürfen jeweils für die anderen Transaktionen nicht sichtbar sein. Man könnte auch sagen: Keine Transaktion merkt, daß andere Transaktionen parallel zu ihr laufen.

Durability / Dauerhaftigkeit Die Ergebnisse einer erfolgreichen Transaktion werden gegen Versagen der Hard– und Software gesichert und können nur durch eine weitere Transaktion wieder geändert werden.

Im klassischen Fall werden die Punkte 'Consistency' und 'Isolation' normalerweise dadurch erreicht, daß man verlangt, daß die Transaktionen serialisierbar ablaufen. Serialisierbar (s. z.B. [BHG87]) bedeutet vereinfacht, daß die Abfolge der atomaren Aktionen[1] (auch 'Schedule' oder Historie genannt) der Transaktionen äquivalent zu einer Abfolge sein muß, die sich bei serieller Hintereinanderausführung der Transaktionen ergäbe. Verschiedene Definitionen der "Äquivalenz" führen dann zu den unterschiedlichen Versionen der Serialisierbarkeit, die unten behandelt werden.

Die Erhaltung von Konsistenz und Isolation bei Serialisierbarkeit läßt sich anschaulich folgendermaßen begründen:
Unter der Voraussetzung, daß jede Transaktion im Einzelbetrieb die Konsistenz der Daten erhält, bleibt diese auch bei serieller Hintereinanderausführung erhalten. Da aber jeder 'Schedule' paralleler Transaktionen zu einer solchen seriellen Ausführung äquivalent sein muß, erhalten auch serialisierbar ausgeführte Transaktionen die Konsistenz.
Die Isolation ergibt sich (zumindest bei Sichten– und Konflikt–Serialisierbarkeit, vgl. unten) dadurch, daß in der äquivalenten seriellen Auführung jede Transaktion nur Endergebnisse der anderen lesen kann.

Für die Äquivalenz der 'Schedules' von Transaktionen gibt es drei verschiedene Definitionen [Pap86, Kap.2] und [VGH93]:

Zustands–Serialisierbarkeit (ZSR) Dies ist die intuitivste Definiton. Zwei Abfolgen der gleichen Transaktionen sind äquivalent, wenn für alle möglichen Anfangszustände und Semantiken der Transaktionen die Endzustände jeweils gleich sind.

Sichten–Serialisierbarkeit (SSR) ist eine Verschärfung der Zustands–Serialisierbarkeit. Es wird zusätzlich verlangt, daß jede Leseoperation in den beiden 'Schedules' jeweils den Wert von derselben Schreiboperation erhält.

Konflikt–Serialisierbarkeit (KSR) verlangt, daß die Reihenfolge jeweils zweier in Konflikt stehenden Operationen in beiden 'Schedules' gleich bleibt.

Bei der Definition der Konflikt–Serialisierbarkeit wurde der Begriff "Konflikt" verwendet. Zwei Operationen stehen in Konflikt, wenn sie dasselbe Datum behandeln, und wenigstens eine das Datum ändert, also schreibt. Den englischen Bezeichnungen für Lesen (<u>R</u>ead) und Schreiben (<u>W</u>rite) folgend, unterscheidet man R/W–, W/R– und W/W–Konflikte.

In der Praxis wird meist nur die Konflikt–Serialisierbarkeit verwendet, da – wie z.B. [Pap86] zeigt – der Test auf eines der beiden anderen Kriterien NP–vollständig ist. Allerdings ist KSR auch am restriktivsten, d.h. für die Mengen der serialisierbaren 'Schedules' gilt $S_{ZSR} \supset S_{SSR} \supset S_{KSR}$. Es gibt aber sogar Abfolgen, die die Konsistenz wahren, jedoch nicht in ZSR liegen.

Beispiel 3.1 *Der folgende Schedule (aus [Wei88]) ist nicht zustands–serialisierbar, da über die Semantik der Rechenoperationen im Transaktionssystem normalerweise nichts bekannt ist. Er ist aber intuitiv korrekt, da er mit der seriellen Ausführung von T_2–T_1 übereinstimmt:*

[1] meist Lese– und Schreib–Operationen

T_1	T_2
V1:=Read(x);	
V1:=V1+1;	
Write(x,V1);	
	V2:=Read(x);
	V2:=V2+1;
	Write(x,V2);
V1:=Read(x);	
V1:= 2∗V1;	
Write(x,V1);	

Nicht alle Schedules, die korrekt sind, sind also auch serialisierbar. Ein Transaktionskonzept, das auf die (Konflikt–)Serialisierbarkeit als Korrektheitskriterium verzichtet, erlaubt darum eine höhere Parallelität. Ein Verfahren, welches obigen Ablauf als korrekt erkennen würde, wird z.B. in Abschnitt 3.2.1 vorgestellt.

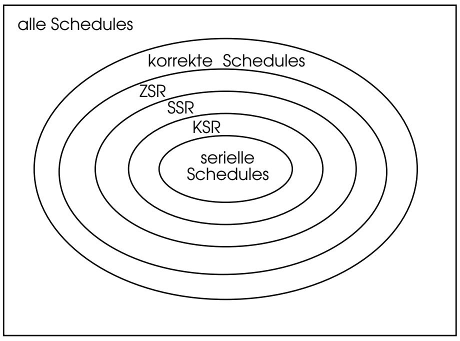

Abbildung 3.1: Enthaltenseinsbeziehung der verschiedenen Arten der Serialisierbarkeit als Venn–Diagramm

3.1.2 Isolationsgrade

Bei der Erwähnung der ACID–Eigenschaften wurde so getan, als ob Isolation ein monolithischer Begriff sei. [GR93] definieren jedoch 4 verschiedene Abstufungen von Isolation zwischen Transaktionen, die in Tabelle 3.1 aufgeführt sind.

Der Isolationsgrad einer Transaktion muß ihr bekannt sein, und sie muß entsprechend programmiert worden sein. So sehen 0°– und 1°–Transaktionen ggf. einen inkonsistenten Zustand, müssen aber dennoch bei ihrem Abschluß einen konsistenten Zustand hinterlassen (vgl. ACID). Ein niedrigerer Isolationsgrad erschwert also einerseits die Transakionsprogrammierung, andererseits benötigt er weniger und kürzere Sperren als ein höherer und läßt so mehr Parallelität zu. Dadurch ermöglichen es die verschiedenen Isolationsgrade, einen Kompromiß zwischen notwendiger Korrektheit/Isolation und gewünschter Leistungsfähigkeit zu wählen[2].

[2] Z.B. laufen die meisten kommerziellen relationalen DBMS per Default unter Isolationsgrad 2.[GR93, S.399f]

	Grad 0 (0°)	**Grad 1** (1°)	**Grad 2** (2°)	**Grad 3** (3°)
Gebräuchlicher engl. Name	Chaos	Browse	Cursor stability	Isolated serializable repeatable reads
Schutz	Andere können unter höherer Isolation laufen	0°und keine 'lost updates'	Keine 'lost updates' Keine 'dirty reads'	Keine 'lost updates' Keine 'dirty reads' 'repeatable reads'
Daten-änderungen	sofort sichtbar	erst bei EOT sichtbar	wie 1°	wie 1°
Schmutzige Daten	überschreibt sie nicht	0° und andere überschreiben eigene nicht	0°,1° und liest keine schmutzigen Daten	0°, 1°, 2° und andere machen gelesene Daten nicht schmutzig
Sperren	Setzt kurze[3] exkl. Sperren auf zu schreibende Daten	Setzt lange[3] exkl. Sperren auf zu schreibende Daten	1° und setzt kurze gemeinsame Sperren auf zu lesende Daten	1° und setzt lange gemeinsame Sperren auf zu lesende Daten

Tabelle 3.1: Isolationsgrade nach [GR93, S.401]

3.1.3 Optimistische und pessimistische Verfahren

Bisher ging es darum, wie verschiedene Transaktionseigenschaften durch Erhaltung der Serialisierbarkeit gesichert werden können. Wann Verletzungen der Serialisierbarkeit erkannt und wie sie verhindert werden sollen, wurde nicht erwähnt. Hierfür gibt es prinzipiell zwei Kategorien von Verfahren, pessimistische und optimistische:

pessimistisch Mögliche Verstöße werden (kurz) bevor sie auftreten erkannt, und die auslösenden Datenzugriffe werden verzögert, bis sie ohne Verletzung der Serialisierbarkeit durchgeführt werden können. Dies ist heutzutage die Standardmethode.
Kurz: Erkennung sofort; Serialisierbarkeit meist durch Blockieren.

optimistisch Erst beim Abschluß einer Transaktion wird geprüft, ob sie sich serialisieren läßt. Ist dies unmöglich, wird sie abgebrochen und muß von vorne beginnen (vgl. [BK91, S.277f] und [Wei88, S.145ff]). Deshalb durchlaufen optimistische Transaktionen normalerweise drei Phasen: Eine Lesephase, in der Daten gelesen und Berechnungen durchgeführt werden, eine Validierungsphase, in der die Einhaltung der Konsistenzkriterien geprüft wird, und eine Schreibphase, in der die in der Lesephase berechneten Änderungen in die Datenbasis eingetragen werden.
Kurz: Erkennung am Transaktionsende; Serialisierbarkeit durch Neustart.

Ihre Namen haben sie dadurch erhalten, daß pessimistische Verfahren davon ausgehen, daß Verstöße häufig auftreten werden und darum verhindert werden müssen, während optimistische Verfahren einen Konflikt für unwahrscheinlich halten und darum den Aufwand für die frühzeitige Erkennung scheuen. Beide Konzepte haben ihre spezifischen Vor- und Nachteile.

[3]Kurze Sperren: Nach der Operation sofort wieder aufgehoben. Sichern Atomarität. Lange Sperren: Bis (annähernd) zum Ende der Transaktion gehalten.

Pessimistische Verfahren:

+ frühzeitige Erkennung der Konflikte
+ mit Arbeitsverlust verbundene Abbrüche der Transaktionen nur bei Deadlock
– Zur Vermeidung von Schneeball–Effekten notwendige Striktheit vermindert die mögliche Nebenläufigkeit

Optimistische Verfahren:

+ Nur–Lese–Transaktionen brauchen sich nicht um Synchronisation zu kümmern
+ Keine Deadlock–Gefahr
+ Beim Konflikttest können die tatsächlichen Ergebnisse der Operationen der Transaktionen berücksichtigt werden (vgl. [Her90])
– Änderungen müssen bis zum Ende der Transaktion in einem lokalen Speicher gehalten werden
– Ein großer Teil des Synchronisierungsaufwandes fällt gesammelt am Schluß an

[ACL85] untersuchen mittels einer Simulation die Performance zweier Sperrverfahren (pessimistisch) und eines optimistischen Verfahrens. Nach ihren Ergebnissen ist das Sperrverfahren dem optimistischen Verfahren immer dann überlegen, wenn die Auslastung des Prozessors oder der Zugriffskapazität auf den Sekundärspeichers hoch ist, da die Neustarts des optimistischen Verfahrens dann eine knappe Resource verbrauchen. Ist die Auslastung hingegen gering, so wird sie durch das Blockieren des Sperrverfahrens weiter verringert, während das optimistische Verfahren (allerdings auf Kosten konkurrierender Hintergrund–Jobs) einen höheren Durchsatz erreicht.
Letzterer Fall tritt – abgesehen von Multiprozessorsystemen – besonders bei interaktiven Anwendungen ein, bei denen zwischen den verschiedenen Datenzugriffen immer eine 'Denkzeit' des Benutzers liegt, so daß die Auslastung niedrig bleibt.
Obwohl [ACL85] also optimistische Verfahren für interaktive Transaktionsanwendungen für geeignet halten, gehen sie nicht auf die Probleme ein, die ein Rücksetzen bei interaktiven Anwendungen mit sich bringt: insbesondere den Verlust menschlicher Arbeitszeit (vgl. Kap. 3.2).

3.1.4 Granularität

Neben der Frage, wann Serialisierbarkeitsverletzungen erkannt werden sollen, besteht auch noch die Frage, auf welcher Datenebene dies geschieht: auf Tupeln, auf Relationen oder auf (Teil-)Datenbanken?

Eine Erkennung auf einer dieser Ebenen entspricht jeweils dem Sperren (oder welches Verfahren jeweils angewendet wird) einer bestimmten Menge von Daten, gibt also die 'Granularität' des Erkennungsmechanismus an. Ein Arbeiten auf niedriger Ebene (also Tupeln) bedingt einen hohen Verwaltungsaufwand, wenn z.B. ein Großteil der in einer Relation vorhandenen Tupel benötigt wird. Es ermöglicht aber andererseits eine höhere Nebenläufigkeit als das

Sperren ganzer Relationen oder Datenbanken, denn bei grober Granularität ist der Zugriff auf eigentlich unbenötigte Tupel für andere Transaktionen nicht mehr möglich.

Da beim Entwurf eines Systems meist nicht vorhergesagt werden kann, welche Granularität die günstigste ist, bieten einige Verfahren (siehe Abschnitt "Multigranulares Sperren") dynamische Granularitätswahl für jede Transaktion an.

Auch in objektorientierten Systemen gibt es das Problem der Granularität, nur daß hier die Ebenen "Individual", "Klasse", "Metaklasse", ... sind und durch komplizierte Instanzierungs– und Spezialisierungsbeziehungen das Problem hinzukommt, die jeweils übergeordneten Ebenen eindeutig zu definieren.

3.1.5 Sperrverfahren

Die Verwendung von Sperren ('Locks') ist heutzutage die in kommerziellen, relationalen Datenbanksystemen am häufigsten genutzte Synchronisationsmethode. Sperren gewährleisten Konfliktserialisierbarkeit und sind prinzipiell pessimistisch. Obwohl man sich theoretisch auch ein auf Sperren basierendes optimistisches Konzept vorstellen kann, wäre es in der Realität völlig sinnlos, da frühzeitig vorhandene Information über Konflikte bis zum Transaktionsende verzögert werden müßte.

Zwei–Phasen–Sperren (2PL)

Das Sperren von Objekte, die man lesen oder ändern will, reicht nicht aus, um Serialisierbarkeit zu erreichen. Vielmehr muß der Fundamentalsatz des Sperrens ([LS87, S.426]) erfüllt werden, der unter anderem fordert, daß eine Transaktion in zwei Phasen zerfällt:

1. Eine Wachstumsphase, in der Sperren angefordert, aber nicht freigegeben werden dürfen;
2. Eine Schrumpfungsphase, in der Sperren freigegeben, aber nicht mehr angefordert werden dürfen.

Natürlich kann zwischen diesen beiden Phasen eine beliebig lange Phase liegen, in der sich die gehaltenen Sperren nicht ändern.

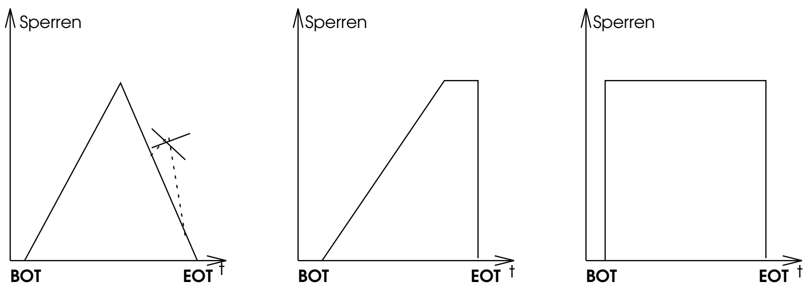

Abbildung 3.2: Normales 2PL, Striktes 2PL und Striktes 2PL mit Voranforderung

Das aus dem Fundamentalsatz abgeleitete Zwei–Phasen–Sperren (2PL) reicht aus, um die Serialisierbarkeit zu garantieren, falls alle Transaktionen erfolgreich abgeschlossen werden. Wird jedoch eine Transaktion abgebrochen, so kann es andere Transaktionen geben, die von der ersten Transaktion geänderte und bereits freigegebene Daten gelesen haben. Weil diese Daten zurückgesetzt werden, haben die anderen Transaktionen einen falschen Wert gelesen und müßten ebenfalls zurückgesetzt werden. Abgesehen vom hierdurch bedingten Mehraufwand kann ein Rücksetzen unmöglich sein, weil andere Transaktionen breits erfolgreich beendet wurden und ihre Ergebnisse wegen der "Dauerhaftigkeit" (ACID) nicht mehr entfernt werden dürfen. Um solche Probleme zu vermeiden, wird meistens das Strikte 2PL–Protokoll angewendet, das alle Sperren erst am Ende der Transaktion freigibt.

'Deadlocks' (Verklemmungen) können auch beim Strikten 2PL noch auftreten, da sich durch die Blockierung bei der Sperrenanforderung Wartezyklen ergeben können (vgl. [Tan92] für die Voraussetzungen von Deadlocks). Werden jedoch alle Sperren am Beginn der Transaktion auf einmal angefordert, und wird die Transaktion bei Mißerfolg neu gestartet, so können sich keine Zyklen bilden. Dieses Verfahren — Striktes 2PL mit Voranforderung — setzt allerdings die Kenntnis der Lese– und Schreibmenge ('read–set' und 'write–set') der Transaktion voraus und führt zu einer weiteren Abnahme von Nebenläufigkeitsmöglichkeiten.

Werden Deadlocks nicht verhindert, so müssen sie (z.B. durch Überprüfung eines Abhängigkeitsgraphen) erkannt und (z.B. durch Abbruch einer beteiligten Transaktion) beseitigt werden.

'Field Calls' und Escrow–Sperren

In den meisten Datenbanken gibt es Bereiche oder einzelne Tupel, die sehr viele Transaktionen lesen und ändern wollen. Um zu verhindern, daß durch diese 'Hotspots' genannten Daten die Parallelität reduziert wird, indem alle Transaktionen außer einer blockiert werden, wurden verschiedene Verfahren erdacht, die mittels Optimismus ein langfristiges Sperren unnötig machen.

[GR93, S.430f] schlagen hierfür sogenannte 'Field Calls' vor. Ein 'Field Call' ist eine Kombination aus einer Test- und einer Änderungsoperation, wobei der Test einmal zum Aufrufzeitpunkt und einmal während der Abschlußbehandlung der Transaktion ausgeführt wird. Die Änderung wird beim Aufrufzeitpunkt zwischengespeichert und nur beim Transaktionsabschluß, wenn der Test erfolgreich war, ausgeführt.

Beispiel 3.2 *Das Holen von 90 Schrauben aus dem Lager läßt sich folgendermaßen formulieren:*

```
if schrauben.anzahl > 90
set schrauben.anzahl:=schrauben.anzahl−90;
```

Ist die zu einem 'Field Call' gehörende Bedingung beim Aufruf nicht erfüllt, so wird ein Fehler zurückgegeben. Ist sie beim Transaktionsabschluß nicht mehr erfüllt, wird die Transaktion zurückgesetzt. Ansonsten wird die Änderung durchgeführt.

Der Vorteil dieser Methode ist, daß nur sehr kurzfristig Sperren auf den Daten gehalten werden müssen. Dafür tauchen Probleme auf, wenn die Transaktion von ihr geänderte Werte lesen will, oder wenn die Transaktion zurückgesetzt werden muß. Auch kann dieses Verfahren nur angewendet werden, wenn alle Änderungsoperationen kommutativ sind, da die Reihenfolge (und damit die Serialisierbarkeit) ihrer Ausführung nicht mehr vorhersagbar ist.

Um das Problem der zurückzusetzenden Transaktionen in den Griff zu kriegen, wurde die Methode um die sogenannten Escrow–Sperren erweitert (vgl. [GR93, S.432f]; ursprünglich [O'N86]). Hierbei wird bei jedem Datum das durch noch nicht ausgeführte Änderungsoperationen verursachte Intervall der möglichen Werte abgespeichert. Bei obigem Beispiel würde z.B. die Untergrenze des Intervalls für schrauben.anzahl um 90 verringert, da bei einer erfolgreichen Ausführung der Transaktion 90 Schrauben weniger vorhanden wären. Die Testoperationen werden auch gegen die entsprechende Intervallgrenze durchgeführt, so daß die Erfüllung der Bedingung am Transaktionsende gewährleistet bleibt.

Multigranularität

Abschnitt 3.1.4 schilderte die Notwendigkeit, Sperren auf unterschiedlichen Datenebenen setzen zu können. Es ergibt sich das Problem, wie eine auf höheren Ebenen sperrende Transaktion merkt, daß tieferliegende Daten bereits von einer anderen Transaktion gesperrt wurden und umgekehrt. Eine Lösung bietet das Multigranulare Sperren mittels 'Intention'–Sperren (siehe hierzu [BHG87, S. 69ff] und [GR93, S. 406ff]). Hierbei werden neben der Sperre auf das eigentliche Objekt noch sogenannte 'Intention'–Sperren auf die übergeordneten Strukturen bis zur Wurzel des Hierarchiebaums[4] gesetzt. Diese Intention–Sperren sind zwar zueinander kompatibel, aber nicht zu den normalen Lese- oder Schreibsperren, so daß verhindert wird, daß sich gleichzeitig vorhandene Sperren in ihren Sperrbereichen überschneiden.

Die vom 2PL bekannte Kompatibilitätsmatrix der Sperrmodi muß um die Modi 'ir', 'iw' und 'riw'[5] erweitert werden. Letztere deckt den häufigen Fall ab, daß eine Struktur gelesen (Lesesperre) und Teile geändert (Intention–Schreibsperre) werden soll.

gewünschte	bestehende Belegung					
Belegung	frei	r	w	ir	iw	riw
r	+	+	−	+	−	−
w	+	−	−	−	−	−
ir	+	+	−	+	+	+
iw	+	−	−	+	+	−
riw	+	−	−	+	−	−

Tabelle 3.2: Kompatibilitätsmatrix für Multigranulares Sperren

DAG–Sperren

Ein Sperrverfahren für baumartig strukturierte Daten, das nicht zwei–phasig ist aber trotzdem Serialisierbarkeit und sogar Deadlock–Freiheit garantiert, stellen [SK80] vor. [Yan82] erweiterte das Konzept auf allgemeine, nicht–zyklische Graphen mit beständiger Struktur. Die Vorgehensweise ist wie folgt:

1. Als erstes kann die Transaktion ein beliebiges Datum sperren.
2. Weitere Sperren können nur erteilt werden, wenn alle direkten Vorgänger im Graphen bereits einmal durch die Transaktion gesperrt waren und mindestens einer es noch ist.

[4] vgl. Abbildung 3.7

[5] die Abkürzungen stehen für 'Intention read', 'Intention write' und 'Read and intention write'

3. Sperren können jederzeit freigegeben und angefordert werden, aber kein Datum darf, nachdem es freigegeben wurde, erneut gesperrt werden.
4. Standardforderungen: Auf ein Datum darf nur zugegriffen werden, wenn es gesperrt ist; ein Datum kann jeweils nur von einer Transaktion gesperrt sein.

Der Vorteil dieses Verfahrens liegt offensichtlich darin begründet, daß es erlaubt, Sperren zu beliebigen Zeitpunkten freizugeben und so eine höhere Parallelität erlaubt. Allerdings stellt es die Rücksetzbarkeit nicht sicher und erfordert im Gegensatz zum Multigranularen Sperren eine Sperre auf jedem behandelten Datum.

3.1.6 Zeitstempelverfahren

Während bei den Sperrverfahren die Serialisierungsreihenfolge der nebenläufigen Transaktionen nicht von vorneherein feststeht, werden sie bei Zeitstempelverfahren in der Abfolge ihres Beginns serialisiert. Hierzu erhält jede Transaktion einen (streng monoton aufsteigenden) Zeitstempel, der für Konflikttests genutzt wird: Bei jedem Zugriff auf ein Datum wird überprüft, ob bereits eine Transaktion mit jüngerem Zeitstempel eine in Konflikt stehende Operation auf diesem Datum ausgeführt hat. Ist dies der Fall, so wird die “zu spät gekommene“ Transaktion abgebrochen und muß neu beginnen. Die jüngsten Zeitstempel für jede Zugriffsart müssen für jedes Datum abgespeichert werden. Der Vorteil dieses Verfahrens liegt in der impliziten Verhinderung von Deadlocks. Die Menge der akzeptierten Operationsreihenfolgen liegt schief zu der von 2PL, so daß sich Aussagen über die Leistungsfähigkeit nur schlecht treffen lassen.

Das bisher dargestellte Konzept entstammt [BHG87]. [BK91] beschreiben unter dem gleichen Namen ein Verfahren, das auf 2PL aufsetzt. Es verhindert Deadlocks, indem es Transaktionen je nach Zeitstempel abbricht, statt sie zu blockieren, wenn sie eine Sperre nicht erlangen können.

Multiversions-Zeitstempelverfahren

Von der Leistungsfähigkeit wesentlich interessanter als die einfachen Zeitstempelverfahren, sind diejenigen, die ältere Versionen eines Datums aufheben, um “zu spät gekommenen“ Transaktionen zumindest noch Lesezugriffe zu erlauben. Dann können Nur–Lese–Transaktionen, ohne blockiert oder abgebrochen zu werden, ihre Aufgabe erfüllen. Ein solches Verfahren ist das 'Multiversion Timestamp Ordering' [BHG87, S.153ff].

Allerdings muß trotz der allzeit möglichen Lesezugriffe die Serialisierbarkeit aufrechterhalten werden. Hierzu wird für jede Versionen eines Datenelements der Zeitstempel der Lesezugriffe gespeichert. Versucht nun eine Transaktion A ein Datum zu ändern (also eine neue Version anzulegen), so wird überprüft, ob zwischen ihrem Zeitstempel TS_A und dem Zeitstempel der nächst jüngeren Version bereits ein Lesezugriff stattgefunden hat. Dieser hätte – falls die neue Version erzeugt würde – fälschlicherweise den Wert der nächst älteren Version gelesen (vgl. Grafik unten). Transaktion A darf also nicht schreiben und muß abgebrochen werden. Gibt es hingegen keinen solchen Lesezugriff, so wird eine neue Version des Datums mit den Änderungen von Transaktion A angelegt. Dieses Verhalten sichert, daß alle Transaktionen in der Reihenfolge ihrer Entstehung – also in der Reihenfolge ihrer Zeitstempel – serialisiert werden können.

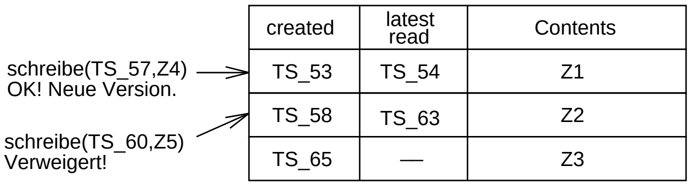

Abbildung 3.3: Schreibzugriff beim Multiversions–Zeitstempelverfahren

Andere Verfahren bewahren nur eine weitere Version des Datums auf (z.B. SAC [Rös91, S.27f]) oder benutzen keine Zeitstempel, sondern Sperren oder Tests des Serialisierbarkeitsgraphens zur Synchronisierung ([BHG87, S.143ff]).

3.1.7 Sonstiges

Geschachtelte Transaktionen

Dieses in [Ree78] und [Mos85] vorgestellte Konzept beruht auf der Idee, zum Aufbau von Transaktionen nicht nur atomare Operationen zu verwenden, sondern auch Untertransaktionen. Eine Transaktion gliedert sich also in eine Baumstruktur aus Untertransaktionen, deren Blätter die atomaren Operationen sind. Die Transaktion an der Wurzel des Baumes fordert Sperren wie bisher von einem globalem Sperrenverwalter an und gibt sie erst an ihrem Ende wieder frei. Nach außen bleibt also ein striktes 2PL–Protokoll gewährleistet. Die Kindertransaktionen hingegen können parallel ablaufen und werden lokal synchronisiert. Sperren erhalten sie von bzw. vererben sie an ihre Mutter. Wenn eine Kindtransaktion abbrechen muß, so macht sie selbständig ihre Änderungen wieder rückgängig, die Entscheidung über das weitere Vorgehen (vollständiger Abbruch, erneuter Versuch, Ausweichen auf eine andere Untertransaktion, ...) bleibt aber der Mutter überlassen.

Die Vorteile dieses Konzeptes liegen in der unabhängigen Rücksetzbarkeit von Teilen der Transaktion bei Fehlschlägen und dem erhöhten Durchsatz durch die Nebenläufigkeit der Untertransaktionen. Zusätzlich unterstützt diese Idee auch noch die Verwendung eines Modulkonzepts zur Transaktionsprogrammierung, was im Programmiersprachenbereich ja bereits zu deutlichen Vereinfachungen geführt hat.

3.2 Fortgeschrittene Systeme

Die im letzten Abschnitt aufgeführten Prinzipien und Verfahren finden hauptsächlich Anwendung bei traditionellen Problemen, wie z.B. Verwaltungsaufgaben und Geschäftsanwendungen. Im Zuge der zunehmenden Computerisierung gewannen aber auch Bereiche wie Büroautomatisierung, Computer–unterstützte Konstruktion und Entwicklung an Bedeutung. Mit den bisherigen Anwendungen haben sie die Verwaltung großer Datenmengen gemeinsam, die die Benutzung eines DBMS wünschenswert erscheinen läßt. Sie unterscheiden sich von ihnen unter anderem in den Anforderungen, die sie an ein System zur Nebenläufigkeitskontrolle stellen [BK91, ELMB92]:

Dauer einer Transaktion: Während z.B. eine Transaktion im Bankbereich in kurzer Zeit das Geld von einem Konto zum nächsten bewegt hat, kann es Stunden oder Tage dauern, bis ein Entwickler mit seinen Ergebnissen zufrieden ist und sie der Allgemeinheit zugänglich machen will. Die Transaktion, in der er den Entwurf durchführt, sollte ebensolange dauern, um ihm die Möglichkeit des Rücksetzens zu geben und die Konsistenz sicherzustellen.

Würde die Transaktion nun Sperren z.B. nach dem 2PL–Verfahren verwenden, so müssten diese natürlich ebenfalls lange gehalten werden. Das ist jedoch nicht wünschenswert, weil es den Zugriff anderer Transaktionen – und damit die Arbeit anderer Entwickler – verzögern würde. Auch ein optimistisches Konzept, das Transaktionen bei Serialisierbarkeitsverletzungen abbricht, kommt nicht in Frage, denn die Transaktionen lassen sich aufgrund ihrer Interaktivität nicht automatisch neu ausführen, so daß sich ein Arbeitsverlust ergeben würde.

Isolation der Benutzer: Entwickler arbeiten meist nicht isoliert voneinander, sondern in einem Team zusammen an einem Projekt, wobei auch unvollständige Ergebnisse (im Datenbanksprachgebrauch also 'uncommited data') ausgetauscht werden. Im klassischen Konzept wird aber davon ausgegangen, daß Transaktionen Konkurrenten statt Partner sind, so daß das ACID–Prinzip zum Schutz der Transaktionen voreinander eine unnatürliche Isolation zwischen den Entwicklern erzwingt [EG89]:

Beispiel 3.3 *Softwareentwickler A und B arbeiten zusammen an einem Projekt. Entwickler A hat während seiner normalen Entwicklung in der projekteigenen Programmbibliothek einen Fehler entdeckt und diesen behoben. Er hat seine sonstige Entwicklung aber bisher noch nicht beendet und seine Transaktion darum noch nicht 'committed'. Entwickler B kann bei einem traditionellen Konzept nun auf die (neue Version der) Bibliothek solange nicht zugreifen, bis A fertig ist, obwohl er sie wahrscheinlich schon vorher zu Testzwecken benötigt.*

Die in traditionellen Ansätzen erwünschte Isolation führt hier also zu unnötigen Verzögerungen, die sich bei etwas Kooperation (A teilt B mit, daß die Änderungen an der Bibliothek soweit fertig sind) vermeiden ließe. [EG89, S.401] folgern daraus sogar die Unbrauchbarkeit des (traditionellen) Transaktionsbegriffs für interaktive Anwendungen:

> „Generally, transactions are not well suited to interactive use, for instance a user with two transactions active in separate windows on the same object would be presented with two different data states — it would be better if the windows showed the same state.“

Diese Probleme wurden bereits vor einigen Jahren von Datenbankentwicklern ([KS88, FO89]) erkannt, die die Verwendung der Serialisierbarkeit als Korrektheitskriterium für interaktive Anwendungen mit langen Transaktionen kritisierten und versuchten, durch verschiedenartige Abweichungen von der traditionellen Sichtweise eine Verbesserung zu erreichen.
Einige dieser Ansätze werden im folgenden vorgestellt. Viele stammen aus dem Bereich der objektorientierten Datenbanken, bei denen sich obige Probleme bei der Entwicklung von CAD/CAM–Systemen frühzeitig ergaben. Die Einteilung der Ansätze erfolgte nach ihrer

stärksten Abweichung von traditionellen Konzepten: Ausdehnung des Serialisierbarkeitsbegriffs, Aufgabe bzw. Verminderung der Isolation zwischen den Transaktionen und sonstige Maßnahmen.

3.2.1 Erweiterung der Serialisierbarkeit

Die in diesem Abschnitt vorgestellten Verfahren beruhen weiter auf der Serialisierbarkeit, versuchen jedoch, durch verschiedene Erweiterungen zusätzliche Parallelität zu erreichen.

Ausnutzung der Kommutativität

Bei der klassischen Definition der (Konflikt-)Serialisierbarkeit wird ein Konflikt lediglich über die Einteilung der beteiligten Befehle in Lese- und Schreiboperationen definiert. In objektorientierten Datenbanken geht man jedoch von Schreib-/Leseoperationen auf die den Objekten zugeordneten Methoden als kleinste Einheiten eines Konflikts über. Dadurch bietet sich eine engere Fassung des Konfliktbegriffs an. Betrachten wir hierzu das folgende Beispiel (aus [Geb92]).

Beispiel 3.4 *Es sei die Klasse "Konto" gegeben mit den Methoden Einzahlung, Auszahlung und Saldo, wobei das Konto beliebig weit überzogen werden darf. Nach der klassischen Konfliktdefinition ergibt sich die linke Kompatibtilitätsmatrix, in der nur zwei Leseoperationen "Saldo" nicht miteinander in Konflikt stehen, da Einzahlung und Auszahlung jeweils Schreiboperationen sind.*

Bei genauerer Überlegung fällt allerdings auf, daß es egal ist, in welcher Reihenfolge zwei Beträge auf das Konto eingezahlt werden, das Ergebnis bleibt das gleiche. Zwischen zwei Einzahlungsoperationen besteht also kein Konflikt. Da das Konto außerdem beliebig weit überzogen werden darf, gilt dasselbe für alle Kombination von Ein- und Auszahlungen. Es ergibt sich also die rechte Kompatibilitätsmatrix.

	Einz.	Ausz.	Saldo
Einz.	–	–	–
Ausz.	–	–	–
Saldo	–	–	+

	Einz.	Ausz.	Saldo
Einz.	+	+	–
Ausz.	+	+	–
Saldo	–	–	+

Tabelle 3.3: Kompatibilitätsmatrizen ohne und mit Beachtung der Kommutativität

In Konflikt stehen also nur noch solche Operationen, die nicht kommutativ sind, wo also – entspechend der mathematischen Definition der Kommutativität – nicht sichergestellt ist, daß $Op_1(Op_2(Datum))=Op_2(Op_1(Datum))$. Unter Verwendung der Kommutativität läßt sich auch die in Beispiel 3.1 angegebene Aktionsfolge serialisieren, wenn man die ersten 3 Befehle einer Transaktion jeweils als eine (atomare) Inkrement–Operation ansieht. Man kann durch Ausnutzung der Kommutativität also mehr Transaktionsabfolgen serialisieren, als nach der klassischen Serialisierbarkeitstheorie möglich wäre.

Im obigen Beispiel tritt jedoch ein Problem auf, wenn das Konto nicht überzogen werden darf. In diesem Fall sind nur noch Einzahlungsoperationen kommutativ, weil in allen anderen Fällen ein negativer Kontostand und damit ein Transaktionsabbruch auftreten kann. Um eine Überziehung ausschließen zu können, muß neben der auszuführenden Aktion noch der Zustand des Datums bekannt sein. Um nun von Fall zu Fall festzustellen, ob zustandsabhängige Kommutativität gewährleistet ist, und darum Zugriff gewährt werden kann, müssten jeweils die in folgender Tabelle angeführten Bedingungen getestet werden. Die Operationen werden also teilweise zweifach ausgeführt, was insbesondere bei komplexeren Methoden einen hohen Aufwand bedeuten kann.

Zustand(z)	Einz.(x)	Ausz.(x)	Saldo
Einz.(y)	+	+ wenn x<z	–
Ausz.(y)	+ wenn y<z	+ wenn x+y<z	–
Saldo	–	–	+

Tabelle 3.4: Kompatibilitätsmatrix für zustandsabhängige Kommutativität

Ein ähnliches kontextbasiertes Verfahren zur Synchronisation beschreiben [WA93]. Allerdings stellt sich bei all diesen Verfahren – und auch bei anderen, wie noch zu sehen sein wird – die Frage, woher die semantischen Informationen, wie hier z.B. die Angaben über die Kommutativität, kommen sollen: Werden sie vom Benutzer bei Erzeugung der Objekte angegeben oder vom System zur Laufzeit ermittelt? Letzteres dürfte in den nächsten Jahren nur unvollständig möglich sein, da das System semantisches Verständnis benötigen würde. Aber auch der Benutzer (genauer der Entwickler einer neuen Methode) ist ggf. überfordert, wenn er die Kommutativität zu hunderten anderer Methoden angeben soll oder es zwar wenig andere Methoden gibt, diese dafür aber unvollständig dokumentiert sind.

Mehrschichten–Transaktionen

Komplexe Datenbanksysteme gliedern sich meist in mehrere Schichten. Die oberste, für den Benutzer sichtbare, enthält Befehle und Anweisungen auf einem hohen Abstraktionsniveau. Je maschinennäher die Schichten werden, desto mehr nähern sich die zur Verfügung gestellten Operationen Schreib-/Lesebefehlen für unstrukturierte Daten an. Eine Synchronisation der Transaktionen wäre auf jeder dieser Ebenen möglich, etwaige Nachteile sollen aber an den beiden Extremen festgemacht werden.

Synchronisation auf der untersten Ebene berücksichtigt die in den Befehlen der oberen Ebenen vorhandene Semantik nicht, obwohl deren Einbeziehung zu über die klassischen Konzepte hinausgehenden Fähigkeiten führt.
Eine auf der obersten Ebene angesiedelte Nebenläufigkeitskontrolle setzt hingegen voraus, daß alle von ihr kontrollierten Befehle atomar sind. Die den einzelnen Methoden zugeordneten Operationen der unteren Ebenen dürften sich also nicht überlappen, was eine deutliche Einschränkung der Parallelität zur Folge hat.

Beeri und Weikum ([BSW88]) entwickelten darum ein den Geschachtelten Transaktionen ähnliches System, dessen Schachtelungsebenen allerdings fest mit den Implementierungsschichten der Datenbank korrelieren: Die Mehrschichten–Transaktionen sowie -Serialisierbarkeit. Diese erweiterte Serialisierbarkeit ist anschaulich[6] folgendermaßen definiert:

[6] Eine formale Definition findet sich z.B. in [WS92].

1. Beginne auf der untersten Ebene n. Ordne hier die Operationen unter Ausnutzung der Kommutativität (vgl. letzter Abschnitt) um, bis es keine Verwebungen ('interleavings') von Operationen mehr gibt, die zu unterschiedlichen Befehlen auf Ebene $n-1$ gehören.
2. Da nun alle Befehle auf Schicht $n-1$ als atomar angesehen werden können, braucht Ebene n nicht mehr betrachtet zu werden. Streiche darum Ebene n ("Reduktion").
3. Lassen sich die obigen Schritte bis zum Erreichen der obersten Ebene ausführen, so ist die ursprüngliche Historie serialisierbar.

Ein Beispiel zeigt Bild 3.4. Allerdings wurde ein Punkt bisher mißachtet: Höhere Ebenen geben ggf. eine bestimmte Anordnung von Operationen vor. Zum einen sollen möglicherweise die Operationen innerhalb einer Transaktion nicht vertauscht werden, zum anderen darf durch niedrigere Schichten eine weiter oben zur Konfliktvermeidung erzeugte Ordnung nicht wieder zerstört werden. Aus diesem Grund geht man auf die sogenannte ordnungserhaltende Serialisierbarkeit über, die ein Vertauschen kommutativer Operationen nur zuläßt, wenn ihr kleinster gemeinsamer Vorfahr ihnen keine besondere Ordnung auferlegt hat.

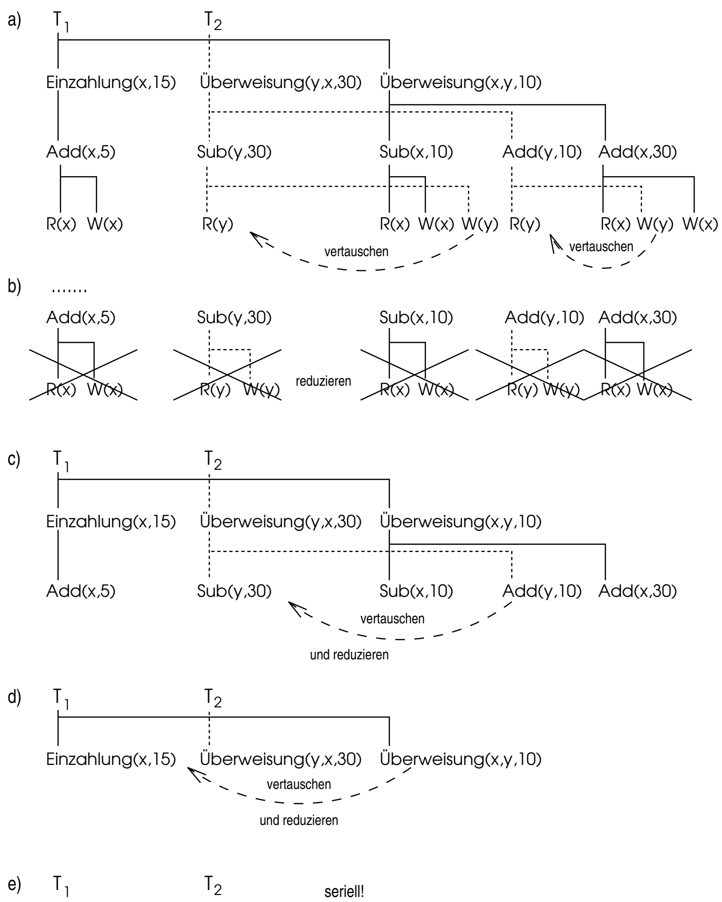

Abbildung 3.4: Vertauschungen und Reduktionen bei Mehrschichten–Serialisierbarkeit

In [BSW88] wird weiter gezeigt, daß die Synchronisation von Mehrschichten–Transaktionen durch unterschiedliche Konzepte in den verschiedenen Ebenen möglich ist, sofern eine von zwei Bedingungen eingehalten wird:

1. Alle verwendeten Konzepte sind ordnungserhaltend serialisierend. (Dies trifft auf die meisten klassischen Konzepte zu.)
2. Alle verwendeten Konzepte beruhen auf der (nicht unbedingt ordnungserhaltenden) Serialisierung von Konflikten und ein Konflikt zwischen zwei Operationen auf Ebene $n-1$ impliziert einen Konflikt zwischen zwei zugehörigen Operationen auf Ebene n (also der darunterliegenden).

Da dieses Konzept einen frühzeitigen Zugriff auf noch nicht bestätigte Daten ('uncommited data') ermöglicht, wird, um das Rücksetzen von Transaktionen zu ermöglichen, ein Verfahren benötigt, wie es z.B. im Abschnitt über Kompensationsfunktionen vorgestellt wird.

3.2.2 Verringerung der Isolation

Die Notwendigkeit, die Korrektheit paralleler Transaktionsausführung durch Serialisierbarkeit zu sichern, ergibt sich bekanntlich nur, wenn lediglich syntaktische Information zur Verfügung steht. Kann die Semantik der Transaktionen oder der abgespeicherten Daten ausreichend genutzt werden, so kann auf die Serialisierbarkeit verzichtet werden. Gleichzeitig kann hierdurch der Isolationsgrad verringert werden, der sonst durch die Serialisierbarkeit fest vorgegeben wird. Dieses Vorgehen führen die folgenden Ansätze vor.

Kompensationsfunktionen und Sagas

Wie man schon bei den 'Field Calls'/Escrow-Sperren gesehen hat, ist es ein Ziel, eine Sperre möglichst kurzfristig zu halten. Bei der Erläuterung des Strikten 2PL wurde aber auch klar, welche Folgewirkungen eine frühe Freigabe von Sperren haben kann:

- Kaskadierendes oder gar nicht mehr mögliches Rücksetzen, falls die Änderungen sofort in der Datenbank vorgenommen werden, oder

- Prüfung der Serialisierbarkeit mit eventuellem Abbruch, wenn die Änderungen erst am Transaktionsende in die Datenbank eingefügt werden.

Um nun das kaskadierende Zurücksetzen zu vermeiden, wird die klassische Forderung der Atomarität: „Eine Transaktion führt alle ihre Operationen aus oder gar keine“ leicht abgewandelt zu „Eine Transaktion führt alle ihre Operationen aus, oder die bereits ausgeführten Operationen werden kompensiert“. Die Kompensation wird durch jeder Transaktionsoperation zugeordnete “Konpensationsfunktionen“ durchgeführt. Diese machen die Effekte der Operationen auf die Datenbasis auf semantischer Basis wieder rückgängig, unabhängig davon, ob inzwischen andere Transaktionen die gleichen Daten verändert haben.

Was bedeutet „auf semantischer Basis rückgängig machen“? Es ist klar, daß ein Datum nicht einfach auf den Wert zurückgesetzt werden kann, den es vor Ausführung einer Transaktion A hat, wenn zwischendurch Transaktion B diesen Wert geändert hat. Generell läßt sich eine Datenbasis nur schwer in den Zustand bringen, den sie hätte, wenn alle Transaktionen außer einer bestimmten ausgeführt worden wären, wie das folgende Beispiel zeigt.

A	B
...	
dat:=dat+1;	
Release_Lock(dat);	...
...	dat:=dat*2;
	COMMIT;
ABORT;	

Das einfache Ausführen der Umkehrfunktion zu dat:=dat+1 dat:=dat-1 stellt keineswegs den gewünschten Zustand her. Vielmehr müßten die Effekte aller sonstigen Transaktionen auf dieses Datum beachtet werden. In obigem Beispiel bedeutet dies, daß die Kompensationsfunktion dat:=dat-2 hieße. Dies ist im allgemeinen Fall natürlich unmöglich zu bestimmen. [KLS90] zeigen, daß das Herstellen eines Datenbankzustands äquivalent zu einem Zustand ohne Ausführung der Transaktion A und ihrer Kompensationsfunktion nur möglich ist, wenn die von der Kompensationsfunktion durchgeführten Operationen zu allen Operationen der anderen Transaktionen auf denselben Daten kommutativ sind. Ein so großes Maß an Kommutativität ist im Normalfall nicht zu erwarten, und so definieren [KLS90] verschiedene schwächere Formen von Kompensationsfunktionen, die nur einen "ähnlichen" Zustand herstellen. "Ähnlich" meint in diesem Zusammenhang, daß bestimmte vorgegebene Prädikate in beiden Zuständen gleichermaßen erfüllt bzw. verletzt sind.
Ein weniger schwerwiegendes Problem stellen Operationen „in der realen Welt“ dar, die sich nicht kompensieren lassen. Populäre Beispiele sind das Abschießen einer Rakete oder die Geldausgabe aus einem Automaten. Lösungen hierfür bieten z.B. [CR92] an.

Ein Verfahren, das Kompensationsfunktionen nutzt, ist "Sagas" aus [GMS87]. Eine 'Saga' ist eine lange Transaktion, die aus vielen Untertransaktionen besteht. Die Untertransaktionen entsprechen den ACID–Prinzipien, geben ihre Änderungen aber sofort nach ihrer Beendigung frei[7]. Die Untertransaktionen müssen darum in ihrer Sicht auf die Datenbank voneinander unabhängig sein, da sich der Datenbankzustand zwischen zwei Untertransaktionen geändert haben kann. Wie bei den Geschachtelten Transaktionen bleibt die Reaktion auf einen Mißerfolg einer Untertransaktion der Entscheidung der Saga überlassen. Beschließt sie abzubrechen, so werden bereits abgeschlossene Untertransaktionen durch die oben beschriebenen Kompensationsfunktionen (hier Kompensationstransaktionen genannt) rückgängig gemacht.

Eine Weiterentwicklung von "Sagas" mittels eines formalen Rahmens findet sich in [CR92]. Zusätzlich eingeführt werden haupsächlich geschachtelte Sagas, die Behandlung nicht kompensierbarer Untertransaktionen und fest vorgegebene Alternativtransaktionen.

Neben "Sagas" nutzen noch viele andere Verfahren das Konzept der Kompensationsfunktionen (z.B. die bereits erwähnten Mehrschichten–Transaktionen, der Tool Kit-Ansatz (s.u.), ConTract (vgl. [WR92]) u.a.), so daß es möglicherweise einmal die Methode für das Rücksetzen objektorientierter Transaktionen werden wird.

Restrukturierung von Transaktionen

Eine weitere Möglichkeit, das Problem der Isolation zu umgehen, ist die von [KP92] vorgeschlagene Restrukturierung von Transaktionen. Sie erlaubt es einem Benutzer, Objekte, auf

[7]vererben also im Gegensatz zu den Geschachtelten Transaktionen ihre Sperren nicht an ihre Mutter

die seine Transaktion Sperren hält, die er aber nicht mehr benötigt, an einen anderen Benutzer "abzugeben". Hierfür wird die ursprüngliche Transaktion in zwei unabhängig serialisierbare aufgeteilt (SPLIT), von denen eine mit der Transaktion eines anderen Benutzes vereint (JOIN) werden kann.

Dieses Konzept erhält trotz der Verminderung der Isolation die Serialisierbarkeit der neustrukturierten Transaktionen aufrecht, so daß im Rahmen der Aufteilung einige Bedingungen erfüllt sein müssen:

$$\mathsf{AWriteSet} \cap \mathsf{BWriteSet} \subset \mathsf{BWriteLast} \tag{3.1}$$
$$\mathsf{AReadSet} \cap \mathsf{BWriteSet} = \emptyset \tag{3.2}$$
$$\mathsf{BReadSet} \cap \mathsf{AWriteSet} = \mathsf{ShareSet} \tag{3.3}$$
$$\mathsf{AWriteSet} \cup \mathsf{BWriteSet} = \mathsf{TWriteSet} \tag{3.4}$$
$$\mathsf{AReadSet} \cup \mathsf{BReadSet} = \mathsf{TReadSet} \tag{3.5}$$

A und B seien hierbei die aus T neu erzeugten Transaktionen. Die Read- und Writesets von A und B müssen vom Benutzer angegeben werden, um dem System mitzuteilen, wie er T aufteilen will. Während Formeln (3.4) und (3.5) nur aussagen, daß die ursprüngliche Transaktion vollständig aufgeteilt werden muß, stecken die eigentlichen Bedingungen in den Formeln (3.1) bis (3.3). (3.1) fordert, daß alle Daten, die sowohl von A als auch von B geändert wurden, zuletzt von B geschrieben wurden. Aus (3.2) folgt, daß A keine von B vorgenommenen Änderungen gesehen haben darf, während der umgekehrte Fall erlaubt ist (3.3). Durch diese drei Voraussetzungen wird sichergestellt, daß A vor B serialisiert werden kann. Man könnte A also nach der Teilung sofort beenden und mit B weiterarbeiten, wodurch bereits fertige Teilergebnisse allgemein zugänglich würden (vgl. Beispiel 3.3).

Verschärft man die Bedingungen dahingehend, daß BWriteLast und ShareSet leer sein müssen, so lassen sich A und B in beliebiger Reihenfolge serialisieren, insbesondere muß B nicht abgebrochen werden, falls dies mit A geschieht. Unter dieser Voraussetzung ist es dann möglich, A (oder auch B) mit der Transaktion eines anderen Benutzers zu vereinigen. Er erhält dadurch nicht nur Zugriff auf alle (nicht unbedingt konsistenten) Objekte, für die A die Zugriffsrechte besitzt. Desweiteren ist er allein für das ordnungsgemäße Beenden der vereinigten Transaktion zuständig. D.h., daß auch die Änderungen an Objekten aus A verlorengehen, wenn der neue Benutzer die vereinte Transaktion abbricht.

Eigene Korrektheits- und Isoliertheitskriterien

Wie bereits angeklungen ist, besitzen fortgeschrittene Anwendungen meist sehr spezifische, aufgabenabhängige Korrektheitskriterien, so daß die Möglichkeit, benutzer–definierte Konsistenzbedingungen anzugeben, wünschenswert ist. Solche Bedingen können zwei Formen annehmen. Am häufigsten findet man Prädikate über den Daten der Datenbank, die erfüllt sind, wenn sich die Datenbank in einem korrekten Zustand befindet. Die andere Methode besteht darin, statt der Korrektheit der Daten diejenige aller Transaktionen aufgrund von Benutzervorgaben ständig zu überprüfen. Dies führt ebenfalls zum Erfolg, da korrekte Transaktionen von einem konsistenten Zustand ausgehend ebenfalls nur konsistente Zustände liefern. Der letzte Ansatz wird im hier vorgestellten Konzept verfolgt, das von [NRZ92] auf Grundlage der Arbeit von [Ska89] entworfen wurde.

In einer baumartigen Struktur aus Transaktionsgruppen (TG) als inneren Knoten und kooperierenden Transaktionen (KT) als Blättern bestimmt jede TG ihre eigenen Korrektheitskriterien für Operationsfolgen. Die Operationen der KT finden auf lokalen Kopien in der jeweiligen TG statt. Eine Propagierung von konsolidierten[8] Operationen in die nächst höhere TG findet erst bei einem 'Checkpoint' der verursachenden KT statt, der nur möglich ist, falls die bisherige Operationsfolge alle Korrektheitskriterien erfüllt.

Die Korrektheitskriterien gliedern sich in zwei Arten:

Patterns: Eine Abfolge von Operationen, die in der Historie vollständig vorkommen muß, damit diese korrekt ist. In einem Software–Projekt vorstellbar wäre z.B. die Bedingung, daß nach Änderungen in einer Prozedur diese von der Qualitätskontolle überprüft werden muß.

Conflicts: Eine verbotene Abfolge. Z.B. könnte es nicht erlaubt sein, eine Prozedur zu ändern, bevor man nicht die neuste Version gelesen hat, um das Überschreiben anderer Modifikationen zu verhindern.

Der Test auf beide Kriterien soll nicht erst nach Abschluß der Historie, sondern gleichzeitig zum Aufruf der Operationen erfolgen, so daß eine Darstellungsform gewählt werden muß, die den Test auf Präfixe zuläßt. Diese Eigenschaft haben u.a. die von [NRZ92] vorgeschlagenen LR(0)–Grammatiken, die sonst hauptsächlich im Compilerbau Verwendung finden (vgl. [ASU86, Kap.4]). Das Alphabet einer Grammatik sind die für das jeweilige 'Pattern' bzw. den jeweiligen 'Conflict' relevanten Operationen. In den folgenden Beispielen werden Operationen als Dreitupel $< A, o, O >$ notiert mit A dem Ausführenden, o der Operation und O dem betroffenen Objekt.

Beispiel 3.5 *Für das oben angeführte Software–Produktions–Beispiel seien (in Anlehnung an [NRZ92]) die folgenden Grammatiken definiert:*

$$
\begin{array}{lll}
P_1 : & & \\
\quad \mathcal{S} & \leftarrow & < any, w, proc > \mathcal{A} \quad | \quad \epsilon \\
\quad \mathcal{A} & \leftarrow & < any, w, proc > \mathcal{A} \quad | \quad < qualctrl, r, proc > \mathcal{B} \\
\quad \mathcal{B} & \leftarrow & < qualctrl, w, report > \mathcal{S} \\
C_1 : & & \\
\quad \mathcal{S} & \leftarrow & < any, w, proc > \mathcal{A} \\
\quad \mathcal{A} & \leftarrow & < any, w, proc > \mathcal{A} \quad | \quad < qualctrl, r, proc > \mathcal{B} \\
\quad \mathcal{B} & \leftarrow & < any, w, proc > \quad | \quad < qualctrl, w, report > \mathcal{S} \\
P_2 : & & \\
\quad \mathcal{S} & \leftarrow & \mathcal{B} \\
\quad \mathcal{B} & \leftarrow & \mathcal{AB} \quad | \quad \mathcal{A} \\
\quad \mathcal{A} & \leftarrow & < any, inc, bug\_count > \mathcal{B} < any, dec, bug\_count > \quad | \\
 & & \quad < any, inc, bug\_count >< any, dec, bug\_count >
\end{array}
$$

[8] z.B. Zusammenfassung mehrerer Schreiboperatione auf demselben Objekt oder gegenseitige Auslöschung sich aufhebender Operationen

$$
\begin{array}{lcl}
C_3: & & \\
\mathcal{S} & \leftarrow & <A,w,O> \quad | \quad <A,r,O>\mathcal{A} \\
\mathcal{A} & \leftarrow & <\overline{A},w,O>\mathcal{S} \quad | \quad <A,r,O>\mathcal{A} \quad | \quad <A,w,O>\mathcal{A}
\end{array}
$$

Die Grammatiken sind in Bild 3.5 als endliche Automaten bzw. Kellerautomaten visualisiert. Das 'Pattern' P_1 sichert die bereits erwähnte Qualitätskontrolle einer Prozedur nach ihrer Veränderung. C_1 verhindert, daß während der Kontrolle die Prozedur erneut geändert wird. In P_2 wird ein Zähler für gefundene Bugs eingeführt. Das Projekt kann erst dann beendet werden, wenn alle im Laufe der Entwicklung gefundene Bugs behoben wurden. Die unter C_3 aufgeführten Produktionen stellen natürlich keine Grammatik dar, da für A und O noch alle möglichen Ausführenden bzw. Objekte substituiert werden müssen, was aber obige Darstellung stark anwachsen lassen würde. C_3 ist eine sogenannte "Synchronisationsgrammatik", die die Datenkonsistenz aufrechterhalten soll. Dazu wird gefordert, daß man von einem Objekt die neuste Version gelesen haben muß, um es ändern zu dürfen.

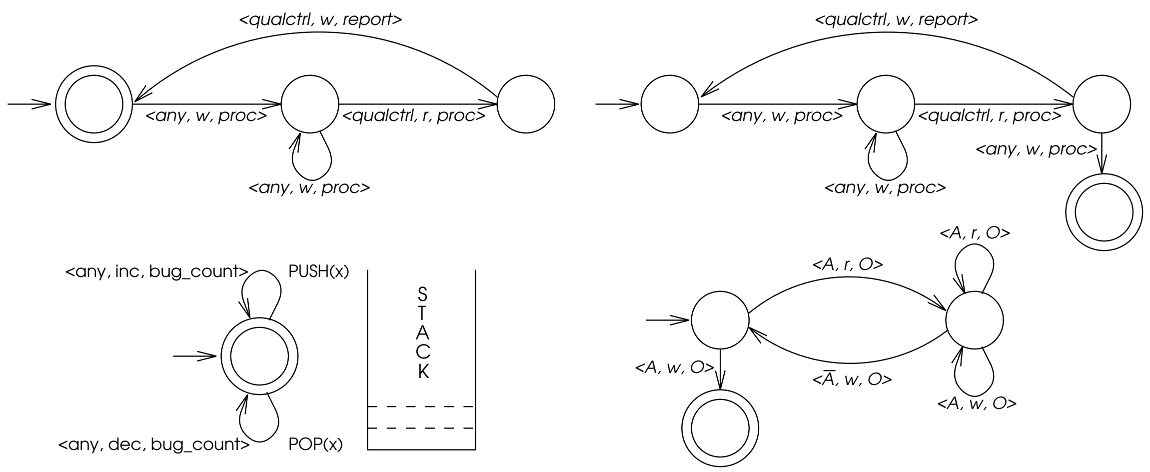

Abbildung 3.5: Darstellung der 'Patterns' und 'Conflicts' als Automaten

Die Grammatiken erlauben es den Mitgliedern einer Transaktionsgruppe, für sie wichtige Operationsfolgen zu fordern und einen ihnen genehmen Isolationsgrad zu wählen. So ließe sich durch entsprechende Produktionen durchaus ein 2PL–artiges Synchronisierungsverfahren spezifizieren, aber auch durch einen Verzicht auf Synchronisationsgrammatiken ein völlig freier Objektzugriff ermöglichen.

Die 'Patterns' unterstützen auch das Zurücksetzen von Transaktionen und sogar einzelner Operationen. Die Abhängigkeiten der Operationen untereinander sind durch die 'Patterns' bekannt. Zusätzlich müssen noch die Schreib-/Leseabhängigkeiten beachtet werden. Nun werden alle von den zurückzusetzenden Operationen abhängigen Operationen ebenfalls rückgängig gemacht. Nach mehrerern Iterationsschritten sind alle direkt oder indirekt abhängigen Operationen zurückgesetzt, und die für die Auswertung der 'Patterns' zuständigen Kellerautomaten müssen auf einen entsprechenden Zustand gesetzt werden.

Als Kritikpunkt bei diesem Konzept erscheint zum einen die Nichtverwendung der aus dem Compilerbau bekannten Fehlerzustände. Durch sie könnten viele 'Conflict'–Grammatiken überflüssig gemacht werden. Durch die zusätzliche Regel

$$\mathcal{B} \leftarrow < any, w, proc > \mathcal{Z}$$

in P_1 mit $\mathcal{Z}$ als Fehlerzustand kann z.B. auf C_1 verzichtet werden.

Abgesehen hiervon besteht zum anderen das Problem, daß die Grammatiken, die zur Beschreibung von Abhängigkeiten und Konflikten genutzt werden, von jemand definiert werden müssen, was mühsam ist und einen im Umgang mit LR(0)–Grammatiken erfahrenen Betreuer erfordert.

3.2.3 Sonstiges

Dynamic Directed Graph

Im Abschnitt 3.1.5 wurden die Probleme des Konzepts der DAG–Sperren bei nicht–zyklenfreien, veränderlichen Graphen beschrieben. Andererseits bietet sich für den Datenbestand von Wissensbanken die Darstellungsweise als Graph an. [CHM92] wandeln darum das DAG–Sperrverfahren um, so daß auch Zyklen und Strukturveränderungen keine Schwierigkeiten mehr bereiten. Hierfür wird der Graph um einige Knoten erweitert, die keine Daten repräsentieren. Ein zentraler Wurzelknoten wird benötigt, um dem Zusammenhang des Gesamtgraphen zu garantieren. Für jede Zusammenhangskomponente des ursprünglichen Graphen wird ein zusätzlicher Knoten eingerichtet, der mit allen Quellknoten der Komponente gerichtet verbunden wird und eine gerichtete Kante vom Wurzelknoten erhält. Außerdem werden sämtliche stark zusammenhängenden Teilgraphen (die Zyklen) identifiziert und gemerkt.

Diese Grapherweiterungen und Informationsansammlungen müssen bei allen Änderungen in der Wissensbank (z.B. Verbindung zweier vorher getrennter Zusammenhangskomponenten; Löschen eines Quellknotens; ...) aktuell gehalten werden, was einen erhöhten Aufwand bei Änderungen bedeutet. Dafür kann nun eine Abwandlung des DAG–Verfahrens angewendet werden. Von dessen 4 Regeln muß nur die zweite neu geschrieben werden:

2. Ein Knoten v kann nur gesperrt werden, wenn alle seine direkten Vorgänger im derzeitigen Graphzustand, die nicht auf der gleichen starken Zusammenhangskomponente wie v liegen, in der Vergangenheit von der Transaktion gesperrt waren und zumindest einer es noch ist.
   Alle Knoten einer starken Zusammenhangskomponente (=Zyklus) werden auf einmal gesperrt, indem alle Zugangsknoten (=Knoten, in die eine Kante von außerhalb der Komponente mündet) gleichzeitig gesperrt werden.

Gruppenstrukturen

Wie z.B. [BKK85] beobachtet haben, werden größere Designaufgaben meist in voneinander unabhängige Projekte aufgeteilt, die jeweils von einer Entwicklergruppe zusammen bearbeitet werden. Während <u>innerhalb</u> der Teilprojekte ein hoher Bedarf an Kooperation und gemeinsamem Zugriff auf Daten besteht, ist dies <u>zwischen</u> den Projekten selten nötig. Zudem ist es einfacher, Zugriffskonflikte zu lösen, die innerhalb eines Entwicklerteams aufgetreten sind, als solche zwischen Entwicklern aus verschiedenen Projekten.

Innerhalb der Projekte wird also eine andere Isolation und Synchronisation benötigt als zwischen ihnen. Diese Tatsache, sowie der sich aus der Projektorganisation natürlich ergebende hierarchische Aufbau, wird durch das Gruppenkonzept [BK91, S.307] berücksichtigt. Es teilt die gesamte Datenmenge in möglichst disjunkte Bereiche auf, von denen jeder die für ein Teilprojekt benötigten Daten enthält. Unterschiedliche lokale und globale Synchronisationsalgorithmen sind möglich (und wünschenswert), sofern durch ihre Zusammenarbeit die Konsistenz nicht gefährdet wird. Über die für die Gruppenverwaltung benötigten Datenstrukturen lassen sich zudem Zugriffsbeschränkungen oder On–Line–Konferenzen unterstützen.

Tool Kit

Ein sehr umfangreiches Konzept, das kein bestimmtes Synchronisationsverfahren verwirklicht, sondern es dem Entwickler ermöglicht, für jede Applikation ein passendes Verfahren einzusetzen, beschreiben Unland und andere [Unl91, US92] in ihrem Tool Kit[9] Ansatz. Die Werkzeuge sind Programmmodule, die die einfache Implementierung anwendungsspezifischer Transaktionsmanager mit unterschiedlichen Transaktionstypen, Synchronisations- und Rücksetztechniken erlauben. Dadurch kann jeder Transaktion ein anderes Synchronisationsprotokoll zugewiesen werden. Die Zusammenarbeit der verschiedenen Transaktionstypen wird durch einige Grundregeln über die verwendbaren Transaktionsmodelle (sozusagen ein Meta–Transaktions–Modell) koordiniert.

Jeder Transaktion ist ein Objektpool zugeordnet, in dem sich eine (logische) Kopie jedes Objekts befindet, auf das die Transaktion eine Sperre hält. Die Transaktionen sind hierarchisch angeordnet und eine Transaktion kann eine Sperre nur über die jeweilige Elterntransaktion erhalten (sogenanntes 'Stepwise Check–out'). D.h. falls ein Objekt benötigt wird, das sich erst in einer indirekten Vorfahrentransaktion finden läßt, müssen die Synchronisationsverfahren aller zwischenliegenden Transaktionen befolgt werden. Hierbei sorgt jede Transaktion mit ihrem eigenen Protokoll für eine korrekte Ausführung auf ihrer Ebene, kann aber ihren Kindern nicht vorschreiben, welches Protokoll sie zu verwenden haben.

Insgesamt wird eine Transaktion durch die Wahl bei mehreren Alternativen charakterisiert:

1. Welches Nebenläufigkeitskontrollverfahren wird verwendet? (optimistisch, pessimistisch, strikt, zwei-phasig, ...)
2. Wird der Zugriff sowie die Rückspielmöglichkeiten oberhalb liegender Objekte für Nachfahr–Transaktionen beschränkt? (Implizit oder explizit?)
3. Dient die Transaktion nur als Durchgangsstation für Daten zwischen ober- und unterhalb liegenden Transaktionen („service transaction“) oder führt sie selber Operationen aus („operational transaction“)?
4. Dürfen Untertransaktionen nebenläufig ausgeführt werden?
5. Wie wird die Rücksetzbarkeit sichergestellt? ([US92] erwägen Kompensationsfunktionen)

Obwohl das Konzept für alle Synchronisationsverfahren verwendbar sein soll, betrachten [US92] lediglich die Zusammenarbeit sperr–orientierter Konzepte genauer. Zur Verbesserung der Kooperation führen sie einige neue Sperrmodi ein. Durch sie werden die möglichen Operationen auf Objekten gegenüber dem reinen Schreib-/Leseansatz differenziert, was die Ausnutzung semantischen Wissens vereinfacht. Die möglichen Operationen sind: (inkonsistentes (**B**)

[9] tool kit = Werkzeugkasten

interner Effekt	externer Effekt B	S	D	U	X
B	+	+	+	+	+
S	+	+	*	*	–
D	+	+	*	!	–
U	+	*	!	!	–
X	+	–	–	–	–

+ = erlaubt
- = verboten
* = möglich
! = nur in ganz besonderen Umgebungen möglich

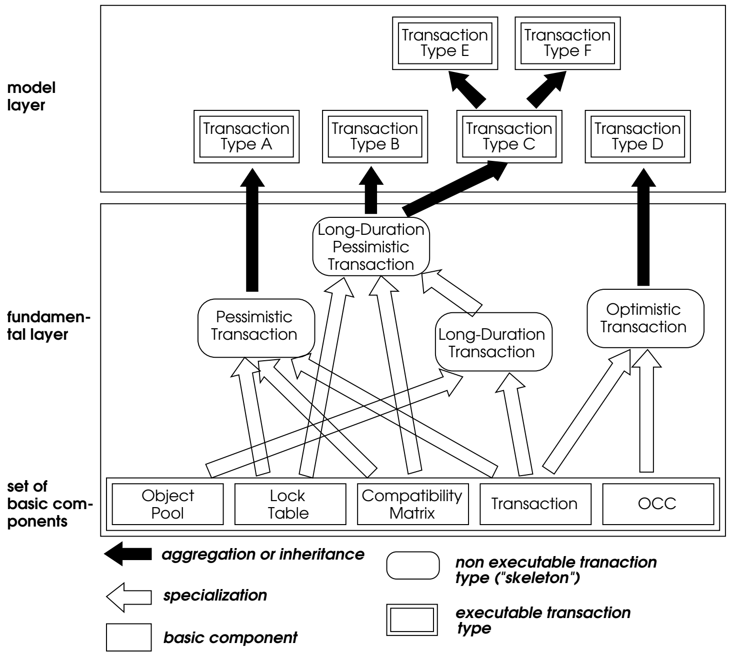

Abbildung 3.6: Kompatibilitätsmatrix[US92, S.440] und Aufbau[US92, S.455] des Tool Kits

Lesen (**S**), Ändern (**U**), Löschen (**X**) und neue Version Erzeugen (**D**). Jede Sperre gewährt dem Besitzer die Möglichkeit einige der obigen Operationen auszuführen (interner Effekt). Andere Benutzer dürfen nur mit der Sperre verträgliche Operationen auf dem Objekt ausführen (externer Effekt). Die sich ergebende Kompatibilitätsmatrix zeigt Abbildung 3.6.

Weitere Einsatzgebiete erschließen sich Sperren, die fest Objekten beziehungsweise Subjekten zugeordnet sind. Die Anbindung von Sperren an Objekte erlaubt es, die in Zukunft möglichen Operationen auf den Objekten zu begrenzen. Sinnvoll ist dies z.B. bei Standardbibliotheken oder historischen Daten, die nur gelesen werden dürfen, sowie bei alten Software–Versionen, von denen nur noch neue Versionen erzeugt werden dürfen. Subjekten (Applikationen, Benutzer(gruppen)) zugeordnete Sperren reservieren ein Objekt über die Laufzeit einer Transaktion hinaus für das Subjekt, z.B. um eine unfertige Prozedur zwischen verschiedenen Entwicklern hin und her zu reichen.
Eine angestrebte Erweiterung wird in [Unl91] präsentiert: Das Anhängen von Regeln an eine Sperre. Die Regeln ermöglichen eine wesentliche feinere Unterscheidung zwischen noch erlaubten Operationen anderer Transaktionen auf dem Objekt, als dies nur durch die verschiedenen Sperrmodi möglich ist.

NSE

Dieses von SUN ([Sun91]) für die Software–Entwicklung zur Verfügung gestellte Verfahren ist eines der wenigen realisierten optimistischen Konzepte. Im Gegensatz zu bekannten pessimistischen Software–Entwicklungs–Verfahren wie SCCS und RCS, die jeweils nur einem Programmierer den Zugriff auf ein Quellcodefragment mittels eines sogenannten 'Check Out' erlauben, können bei NSE beliebig viele Programmierer gleichzeitig Änderungen an einem Quelltext vornehmen. Beim Abschluß ihrer "Transaktion" müssen sie dann ihre Bearbeitung ggf. mit in der Zwischenzeit erfolgten Änderungen durch einen sogenannten 'Merge'–Prozeß in Einklang bringen.

Hierfür benutzt NSE eine hierarchische Struktur, die, ähnlich den Gruppenstrukturen, das gesamte zu bearbeitende Projekt in (sich u.U. überlappende) Teilbereiche untergliedert, so daß einzelne Entwickler schließlich nur den für sie wichtigen Teil sehen und verändern können. Änderungen finden auf den lokalen Kopien jedes Entwicklers statt, die dieser vorher aus seiner übergeordneten Gruppe "erworben" haben muß. Es dürfen aber beliebig viele Entwickler Kopien erwerben, ohne daß Konflikttest stattfinden. Erst falls eine Transaktion beendet werden soll (in NSE: die Änderungen werden in die nächst höhere Gruppe zurückgespielt) findet im Rahmen der Validierungsphase ein Test auf Konflikte statt. Diese führen aber nicht zum Abbruch der Transaktion, sondern zwingen den Benutzer seine Änderungen mit den Änderungen der in Konflikt stehenden Transaktionen in Einklang zu bringen, d.h. zu einer konsistenten Version zu vereinigen. Erst diese Version wird dann in die übergeordnete Gruppe übernommen.

Um Konfliktgefahren frühzeitig erkennen zu können, bietet NSE einen Notifikationsdienst (vgl. auch [BK91, S.306f]) an, von dem sich jeder Benutzer benachrichtigen lassen kann, wenn für ihn interessante Ereignisse aufgetreten sind. Solche Ereignisse können z.B. das Zurückspielen von Änderungen durch andere Entwickler, aber auch bereits das "Erwerben" von Daten, die sich bereits im Besitz des Benutzer befinden, durch andere sein. Diese Benachrichtigungen ermöglichen rechtzeitige Absprachen zwischen den Benutzern, beziehen also die "realen" Interaktionen zwischen ihnen in die Konfliktvermeidung ein.

3.3 Deduktive Datenbanken und Transaktionen

Die in den vorhergehenden Abschnitten aufgeführten Transaktionskonzepte sind fast alle für relationale oder objektorientierte Datenbanken entwickelt worden. Meine Literatursuche nach Transaktionssystemen für deduktive Datenbanken zeigte praktisch keinen Erfolg. Hierfür muß es Gründe geben. Vorstellbar sind zwei Richtungen:

1. Es besteht kein Bedarf.
2. Es ist unmöglich bzw. prohibitiv schwierig.

Beide sollen hier untersucht werden.

3.3.1 Bedarf an Transaktionssystemen für deduktive DB

Für deduktive Datenbanken gilt genau wie für konventionelle, daß ein Transaktionssystem benötigt wird, wenn der konkurrierenden Zugriff mehrerer Benutzer koordiniert werden soll. Der Anwendungbereich deduktiver DB liegt allerdings z.Z. noch hauptsächlich in der Forschung, so daß Eigenschaften wie „funktioniert überhaupt“ oder „optimiert besser“ ein höheres Gewicht als „läuft korrekt mit mehreren Benutzern“ besitzen.
Andererseits erschließen deduktive Datenbanken (wie z.B. *ConceptBase*) als Speicher- und Verwaltungsmedium Anwendungen im Gebiet der Gruppenarbeit, wo eine Koordination der Zugriffe durch die DB einer Koordination durch die Applikation vorzuziehen wäre. Zudem

führt jedes Buch über Betriebssysteme und viele Datenbankbücher die Erhöhung des Durchsatzes durch Ausnutzung der durch E/A-Operationen entstehenden Wartezeiten als entscheidenden Vorteil einer nebenläufigen Programmierung an.

Es besteht also ein Bedarf an Transaktion–Systemen für deduktive DB, wenn er auch anscheinend noch nicht dringend ist.

3.3.2 Probleme deduktiver Transaktionskonzepte

Deduktive DB unterscheiden sich von relationalen und objektorientierten besonders durch die Art der Anfrageauswertung und der Datenspeicherung. Hieraus ergeben sich Probleme für den Entwurf eines Transaktionskonzeptes insbesondere im Zusammenhang mit klassischen Ideen.

Pessimismus, Optimismus und Sperren

Wie bereits in Kapitel 3.1.3 beschrieben, wird bei einem pessimistischen Verfahren versucht, Konflikte zu verhinden, indem man sie beim Auftreten erkennt und den entsprechenden Zugriff verweigert (z.B. durch Sperren) bzw. eine der beteiligten Transaktionen abbricht (z.B. bei Zeitstempelverfahren). Betrachten wir im folgenden nur das Sperren; andere Verfahren zeigen ähnliche Ergebnisse[10]:

Beispiel 3.6 *In Beispiel 2.1 sei auf das Fakt* lebt_in(Gertrud,Berlin) *durch eine andere Transaktion eine Schreibsperre gesetzt. Es werde die Anfrage A=vorfahr_in(meinold,Y,berlin) gestellt.*

Falls diese Anfrage per Vorwärtsauswertung bearbeitet wird, muß bereits in der ersten Iteration solange gewartet werden, bis die Sperre aufgehoben wird, da in jeder Iteration alle Fakten gelesen werden. Auch ein semi-naiver Algorithmus trifft auf das gleiche Problem. Und alles, obwohl das Fakt gar nicht zur Beantwortung der Anfrage gebraucht wird.

Bei einem Rückwärtsauswerter stellt sich dieses Problem nur in geringerem Ausmaß. Er kann durch Benutzung eines Indexes – falls ein solcher vom zugrunde liegenden DBMS angeboten wird und die Zielprädikate entsprechend instanziiert sind – das gesperrte Fakt meist umgehen. In diesem Beispiel ist genau dies natürlich nicht der Fall, da durch die "depth–first/left–to–right"–Strategie der meisten Rückwärtsauswerter, die sich auch auf Kombinationsmethoden wie "Magic Sets" überträgt, nicht bekannt ist, daß lebt_in(Gertrud,Berlin) nicht benötigt wird.

Man kann natürlich argumentieren, daß auch in einem relationalen System Anfragen nicht beantwortet werden können, obwohl nur irrelevante Tupel durch eine andere Transaktion gesperrt sind (z.B.:$\sigma_{name=meinold}(lebt\_in \bowtie parent)$ in einer ähnlichen Datenbasis ohne Optimierung). Allerdings werden in deduktiven Datenbanken Regeln nicht nur in Anfragen vorkommen, sondern auch zur Berechnung der intensionalen Datenbank benötigt. Hierdurch sind Regelauswertungen häufiger und betreffen auch größere Datenmengen als Anfragen in klassischen DB, wodurch diese Probleme öfter zu erwarten sind.

Ein weiteres Problem stellt die Möglichkeit der Verwendung der Negation und die damit einhergehende Nichtmonotonie der intensionalen DB dar.

[10]So würde das Zeitstempelverfahren im folgenden Beispiel entweder ebenfalls warten oder sogar eine Transaktion abbrechen

Beispiel 3.7 *In obigem Beispiel sei zusätzlich die Regel*

vorfahr_nicht_in(X,Y,S):-NOT(wohnt(Y,S)),vorfahr(X,Y)

bekannt. Das Ergebnis einer Anfrage A=vorfahr_nicht_in(meinold,Y,Berlin) in Konkurrenz zu einer Update–Transaktion, die parent(rudi,herta) *und* wohnt(herta,berlin) *einfügt, zeigt, daß das bei relationalen DB relativ ungefährliche Phantom–Problem hier zum falschen Ergebnis 'herta'$\in Y$ führen kann.*

Zur Verhinderung dieses Problems müßten Sperren auf die Indizes aller benutzten Fakten- (und Regel-) mengen über große Teile der Laufzeit der Transaktion gehalten werden. Wie oben bereits erwähnt, können aber umfangreiche Gebiete der Datenmenge betroffen sein. Zudem wird in Kapitel 3.4 gezeigt werden, daß in *ConceptBase* die Laufzeit einer Transaktion die Dauer mehrerer Tage erreichen kann.

Desweiteren muß durch den mehrfach erwähnten 'Datenhunger' deduktiver Regeln und Anfragen bei einem pessimistischen Konzept mit einem hohen Verwaltungsaufwand für Sperren[11] gerechnet werden, sofern nicht die Granularität auf Faktenmengen–Ebene (also Relationenebene) angehoben wird.

Die durch die oben angeführten Problem"lösungen" hervorgerufenen negativen Auswirkungen auf die mögliche Parallelität liegen auf der Hand.

Es bleibt also festzuhalten, daß ein Konzept, das zur Sicherstellung der Serialisierbarkeit Lesezugriffe verweigert, bei deduktiven Datenbanken auf Probleme stößt. Falls also pessimistische Verfahren verwendet werden sollen, so müssen diese zumindest über eine Versionsverwaltung (wie z.B. Multiversion Timestamp–Ordering) verfügen, die jederzeitigen Lesezugriff zuläßt.

Regeln

Im Gegensatz zu relationalen DB liegen in deduktiven DB Regeln zu einem überwiegenden Teil in der Datenbasis vor, um die Berechnung der intensionalen DB zu ermöglichen. Dies ermöglicht es, die Ergebnisse von Regeln vorauszuberechnen und abzuspeichern, bzw. Regeln aufgrund der bekannten Daten einzuschränken, um eine effektivere Auswertung zu erreichen. So wünschenswert eine solche Möglichkeit für die Optimierung ist, so unbequem macht sie die Aufgabe einer Transaktionsverwaltung.

So können verschiedene Transaktionen verschiedene Regelmengen sehen. Sei es, weil sie eine Regel exklusiv gesperrt haben, sei es, weil es die im letzten Unterabschnitt geforderte Versionsverwaltung gibt oder weil die Transaktionen durch ein Gruppenkonzept verschieden eingeschränkte Sichten auf die Datenbasis haben. Die Vorausberechnungen/Einschränkungen müßten dann u.U. für jede Transaktion neu berechnet und abgespeichert werden. Der Speicherbedarf ist durch eine solche Versionierung abgeleiteter Information nicht mehr nur vom Umfang der DB, sondern auch von der Anzahl der laufenden Transaktionen abhängig, was besonders bei Hauptspeicherdatenbanken schnell zu einem Speicherüberlauf führen kann. Berücksichtigt man nun noch, daß diese Versionen auch verwaltet werden müßten, so läßt sich ein deutlicher Konflikt zwischen Transaktionsverwaltung und Optimierung erkennen.

[11] bzw. analogen Verwaltungsinformationen wie Timestamps

Integration

Viele deduktive Datenbanken bestehen aus zwei Schichten mit unterschiedlichen Paradigmen (vgl. z.B. [CGT90]). Während zur Datenspeicherung häufig auf ein relationales System zurückgegriffen wird, sind für die Auswertung deduktive Fähigkeiten vonnöten, wie sie z.B. durch ProLog bereitgestellt werden.

In [Kem91, Kap.1] wurde aber gezeigt, daß sich die Implementierung einer Transaktionsverwaltung nicht auf eine Schicht eines DBMS beschränken kann, sondern alle Bereiche von der Anfragebearbeitung bis zur Pufferverwaltung einschließen muß, um optimale Ergebnisse zu erreichen. Dies wird bei deduktiven DB durch die unterschiedlichen Programmierkonzepte der beiden Schichten erschwert, da nicht nur die Implementierung in verschiedenen Sprachen erfolgen muß, sondern auch ein Interface zwischen den zwei Teilen der Transaktionsverwaltung geschaffen werden muß.

Es wurden einige zusätzliche Probleme aufgezeigt, die beim Entwurf von Transaktionskonzepten für deduktive Datenbanken auftreten. Sie müssen zwar bei der Gestaltung berücksichtigt werden, können aber nicht als unüberwindbar angesehen werden. Da auch Bedarf nach entsprechenden Verfahren vorhanden ist, bleibt die geringe Forschungstätigkeit in diesem Bereich unverständlich.

3.4 *ConceptBase* und Transaktionen

Aus der Architektur und dem Aufgabenbereich des ConceptBase–Servers ergeben sich weitere Anforderungen an ein zu implementierendes Transaktionskonzept, die die Auswahl aus den in Abschnitt 1 und 2 vorgestellten weiter einschränken, andererseits aber auch eine völlig neue Herangehensweise ermöglichen.

3.4.1 Designumgebungen und lange Transaktionen

Ein bedeutender Anwendungsbereich von *ConceptBase* sind entwurfunterstützende Tools wie CoAUTHOR [Ehe93], GKBMS [Jar93], NATURE und andere [RL93, RJM92]. In Kapitel 3.2 wurde gezeigt, daß klassische Verfahren (also alle Verfahren aus Kapitel 3.1) ohne Abwandlungen nicht für Entwurfsanwendungen in Frage kommen.

Die Interaktivität der Transaktionen in Entwicklungsumgebungen hat auch zur Folge, daß von einigen Konzepten benötigte Informationen (wie z.B. Writesets) nicht im Vorhinein zur Verfügung stehen. Auch die für das Konzept von [GMS87] benötigten Kompensationsfunktionen lassen sich nur schlecht im Voraus bestimmen, wenn dem Benutzer die völlige Freiheit bei Anfragen und Änderungen gelassen werden soll. Andererseits können durch die Interaktivität dem Benutzer Koordinierungsaufgaben übertragen werden, die bei einem überwiegenden Batch–Betrieb das System erledigen müßte (falls es dazu überhaupt in der Lage wäre).

3.4.2 Constraints und Integrität

ConceptBase besitzt zur Überprüfung interner und benutzerdefinierter Konsistenzbedingungen einen Integritätstester. Während wir uns mit den Problemen, die die Konsistenzbedingungen zusätzlich zu den unter "Regeln" im letzten Kapitel erwähnten verursacht, erst in Kapitel 4.3.1 beschäftigen wollen, bleibt hier zu bemerken, daß dies ein Transaktionskonzept auf zwei Arten beeinflußt.

Erstens reicht es zur Aufrechterhaltung der Konsistenz nicht mehr, wie bei konventionellen Systemen, einfach für eine serialisierbare Ausführung korrekter Transaktionen zu sorgen: Bei jeder Änderung oder beim Commit einer Transaktion muß geprüft werden, ob die Konsistenzbedingungen verletzt sind.

Tests bei jeder Änderung wären sehr zeitaufwendig, verletzten die Idee, daß ein konsistenter Zustand erst zum Ende einer Transaktion hergestellt werden muß, ließen aber eine gezielte Verweigerung inkonsistenter Änderungen zu und realisierten damit eine pessimistische Einstellung.

Wird die Konsistenz erst zum Commit–Zeitpunkt sichergestellt, so dürfen Änderungen natürlich auch erst nach dem Commit global sichtbar werden. Außerdem kann nur noch der Abschluß der ganzen Transaktion verweigert werden (optimistisch): Entweder durch einen Abort oder mit einer Bitte nach Abhilfe an den Benutzer. Aborts kommen, wie im letzten Unterabschnitt gezeigt, nicht in Frage.

Zweitens enthalten die Integritätsbedingungen semantische Informationen, die sich ein Transaktionskonzept zunutze machen kann. Dies kann soweit gehen, daß auf Serialisierbarkeit als Garant der Konsistenz verzichtet wird und diese nur noch durch umfangreiche Konsistenzbedingungen und den Integritätstester sichergestellt wird.

3.4.3 Objektidentität und Konflikterkennung

In Kapitel 2.2.4 wurde die Abspeicherung der *ConceptBase*–Frames als Propositionen beschrieben. Ein wichtiger Teilaspekt dieser Tatsache ist, daß jede Proposition einen eindeutigen Identifikator hat. Zusätzlich lassen sich Propositionen nicht *ändern* sondern nur *einfügen* (TELL) oder *entfernen* (UNTELL). Hieraus wird klar, daß der Konflikttest wesentlich komplizierter ist, als in einer relationalen DB. Welche Konflikte können auf Propositionsebene überhaupt auftreten?[12]

W/W: Write–Write–Konflikte (also der Verlust von Änderungen) sind per Definition unmöglich, da Propositionen nur eingefügt werden können und jede neue Proposition auch eine neue ID erhält.

W/R: Ein Lesen nicht freigegebener Änderungen würde zur Gefahr kaskadierender Abbrüche führen. Dies kann umgangen werden, indem man den in *ConceptBase* vorhandenen temporären Speicher zur Aufnahme aller Änderungen nutzt, die erst beim Commit global sichtbar gemacht werden.

R/W: Ein solcher Konflikt kann nur auftreten, wenn eine gelesene Proposition von einer anderen Transaktion gelöscht werden soll. Falls hieraus eine referentielle Inkonsistenz entsteht, wird sie vom Integritätstester erkannt werden. Um Serialisierbarkeit zu erreichen, müßte diese Konfliktart verhindert werden.

[12] Die Konflikte W/W, W/R und R/W entsprechen 'lost update', 'dirty read' und 'no repeatable read'. (vgl. [GR93])

Wie man sieht, sind auf Propositionsebene nicht sehr viele Konflikte zu erkennen, bzw. ihre Behebung ist meist einfach. Wenn man hingegen bei der Konflikterkennung von der 'Individual'–Ebene ausgeht, sieht es schon ganz anders aus:

W/W: Durch die Länge der Transaktionen sind Schreibzugriffe verschiedener Transaktionen auf dasselbe Individual zu erwarten. Durch sie können wie bereits erwähnt keine Änderungen verloren gehen, aber das Einfügen widersprüchlicher Informationen kann – für den Integritätstester nicht erkennbare – Inkonsistenz erzeugen. Ein 'Employee', dem mehrere Benutzer gleichzeitig verschiedene Gehälter eintragen, ist ein solches Beispiel. Es läßt sich dadurch verhindern, das das Gehaltsattribute als "single" spezifiziert wird. Die Existenz von nicht so einfach behebbarer Beispiele ist unklar und wird in Kapitel 4.1.3 genauer behandelt.

W/R: Dieser Konflikt läßt sich, wie oben bereits beschrieben, durch Benutzung des temporären Speichers vermeiden.

R/W: Konflikte dieser Art müssen verhindert werden, wenn eine konsistente Analyse gewünscht wird:

Beispiel 3.8 *Eine Transaktion berechne die Steuern eines Angestellten, während gleichzeitig von anderen Transaktionen das Einkommen und die Zahl der Kinder des Angestellten verändert werden.*

Andererseits würde die Verhinderung von R/W–Konflikte während der Transaktionslaufzeit eine Benutzung von *ConceptBase* nahezu unmöglich machen, da viele Anwendungen als erstes den gesamten, sie interessierenden Datenbestand einlesen, um ihn dem Benutzer anzeigen zu können. Sobald aber zwei Transaktionen dies durchgeführt haben, kann keine von ihnen mehr ein Objekt ändern, ohne einen R/W–Konflikt auszulösen.

Es läßt sich also eine wesentlich größere Zahl möglicher Konflikte (insbesondere W/W) auf 'Individual'–Ebene erkennen. Dies ist jedoch mit höherem Aufwand verbunden und teilweise von fraglichem Nutzen. Ein zusätzliches Problem der Konflikterkennung auf Individualebene besteht darin, daß die Behandlung in den ProLog–Code eingesetzt werden müßte, während sich bei den Propositionen die sowieso entstehende Schnittstelle zwischen ProLog–Kern und Objektspeicher anbietet.

3.5 Bewertung der vorgestellten Konzepte

Wie schon erwähnt, können die in Abschnitt 3.1 vorgestellten Konzepte nicht auf die Aufgabenstellung angewendet werden, da sie die Isolation zu gewährleisten suchen. Zudem wurde in 3.3.2 gezeigt, daß **Sperrverfahren**, die Lesezugriffe verweigern, sich mit deduktiven Datenbanken schlecht vertragen. Auch **Escrow–Sperren** sind von ihrer Konzeption mit Intervallen zu sehr von der Notwendigkeit der Unterstützung vielfältiger Datentypen entfernt.

Im Gegensatz zu den Escrow–Sperren, ließen sich **'Field Calls'** auch in nicht–numerischen Datenbanken wie *ConceptBase* einsetzen. Allerdings ist die Angabe einer notwendigen Bedingung für die Ausführbarkeit der Änderungsoperation u.U. schwierig und ihre Auswertung bedeutet zusätzlichen Rechenaufwand: Zusätzlich nämlich zur ohnehin stattfindenden Auswertung der Integritätsbedingungen.

Das Konzept der **Multigranularität** ist auch für objektorientierte Datenbanken von Bedeutung. Die Instanziierungs- und Spezialisierungsbeziehungen formen hier einen gerichteten Graphen (siehe Bild 3.7), so daß Sperren auf verschiedenen Ebenen ebenfalls wünschenswert, wenn auch komplizierter, sind.

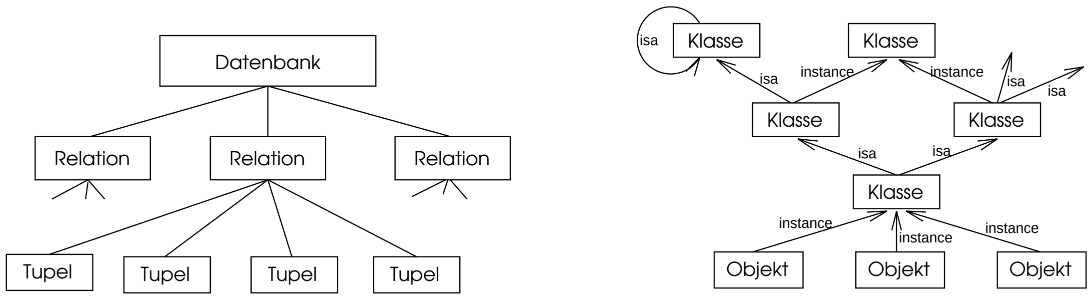

Abbildung 3.7: Datenstruktur in einer relationalen DB und mögliche Hierarchiestruktur bei einer objektorientierten DB

Beim **DAG–Sperren**, das ja für Graphen gedacht ist, ergeben sich Probleme aus den Tatsachen, daß *ConceptBase*s Objekthierarchie nicht zykelfrei ist und keine feste Struktur besitzt. Zudem läßt sich bei interaktiven Transaktionen nie vorhersagen, ob der Benutzer auf ein Datum später noch einmal zugreifen will. Dies wäre aber nach dieser Strategie unmöglich, wenn die Sperre auf dem Datum frühzeitig freigegeben wurde, da jedes Datum nur einmal gesperrt werden darf. Zur Anwendung in *ConceptBase* müßte dieses Verfahren also in das Strikte 2PL–Protokol degeneriert werden, da Sperren erst beim Transaktionsende freigegeben werden dürften.

Bei den **Zeitstempelverfahren** stört insbesondere die Idee, die Transaktionen unbedingt in ihrer Startreihenfolge zu serialisieren, wodurch sich unnötige Transaktionsabbrüche ergeben können. Wie in Abschnitt 3.3.2 gezeigt, vereinfacht aber die Verwaltung von Versionen, um Lesevorgänge ungestört ablaufen lassen zu können, die Zusammenarbeit von Transaktionssystemen und deduktiven Datenbanken, so daß Multiversionen–Verfahren interessant sind.

Geschachtelte Transaktionen ermöglichen das Zusammensetzen einer Transaktion aus (einfacheren) Teiltransaktionen. *ConceptBase* kennt allerdings intern keine Transaktionsprogrammiersprache, in der solche Transaktion geschrieben und zusammengestellt werden könnten. Vielmehr bleibt die "Programmierung" des Transaktionsablaufes allein dem Applikationsprogrammierer überlassen, der dem *ConceptBase*–Server lediglich Anfragen und Änderungsoperationen übermitteln kann. Eine Änderung dieser Struktur ist (unter Kosten–Nutzen–Gesichtspunkten) weder zweckmäßig noch möglich. An ähnlichen Problemen scheitern die **Mehrschichten-Transaktionen**, die sich bekanntlicherweise die internen Abstraktionsstufen der Datenbank zunutze machen. In *ConceptBase* existieren lediglich zwei dieser Stufen: die Propositionsebene und die für den Benutzer sichtbare 'Frame'ebene. Die Befehle der 'Frame'ebene zeichnen sich nicht durch eine reichhaltige Semantik aus, durch deren Ausnutzung Mehrschichten–Transaktionen erst einen Parallelitätsgewinn versprechen würden. Auch der wesentliche andere Grund für einen Performancegewinn mit Mehrschichten–Serialisierbarkeit, die Vergröberung der Sperrgranularität auf der untersten Ebene durch die Verwaltung von Sekundärspeicherseiten, existiert in *ConceptBase* (bisher) nicht.

Da die klassischen Konzepte also nur Denkansätze bieten, ist es natürlich interessant zu sehen, ob die fortgeschrittenen Systeme geeignetere Lösungsansätze bieten. Leider sieht man ihnen ihre Herkunft aus dem objektorientierten Bereich meist an. So ist die Ausnutzung der **Kommutativität** von Objektmethoden zwar eine gute Idee, sie stößt aber an ihre Grenzen, wenn die einzigen "Methoden", die es gibt, TELL, UNTELL und ASK sind, also schließlich doch nur Schreib-/Leseoperationen.

Die Verwendung von **Kompensationsfunktionen/-transaktionen** würde es erlauben, Sperren nur kurzfristig zu halten, und damit viele der in Kapitel 3.3.2 genannten Schwierigkeiten beim Zusammentreffen von Sperren und deduktiven Datenbanken lösen. Eine Definition von Kompensationsfunktionen für die von *ConceptBase* zur Verfügung gestellten Operationen (s.o.) scheitert aber an deren geringem semantischen Gehalt. Da *ConceptBase* zudem keine Programmiersprache besitzt, in der Transaktionen und ihre Kompensationsfunktionen in die Datenbank integriert werden könnten, müßte die Durchführung einer etwaigen Kompensation allein von dem Applikationsprogramm sichergestellt werden. Dies ist nicht nur fehlerträchtig – auch Applikationsprogrammierer sind vergeßlich –, sondern bürdet dem Entwerfer der Anwendung die Schwierigkeit auf, Kompensationstransaktionen zu entwerfen, die auch im Zusammenspiel mit ihm unbekannten anderen Anwendungen noch funktionieren sollen. Dies zumal Operationen in *ConceptBase* nur selten kommutativ sind.

Es lassen sich auch einfach Beispiele konstruieren, in denen eine Kompensation als unmöglich angesehen werden muß. Betrachtet man z.B. die Bemerkungen ('annotation') in der Anwendung "CoAuthor", so kann es Probleme geben, wenn eine einmal eingefügte Bemerkung wieder entfernt werden soll. Erstens können an diese Bemerkung andere Bemerkungen angehängt worden sein, was sich allerdings noch sinnvoll durch ein Mitlöschen dieser Bemerkungen lösen ließe. Zweitens kann eine negative Anmerkung eine Abstimmung über das zugehörige Objekt auslösen, die u.U. negativ ausgeht, worauf das Objekt entfernt wird. Soll und kann das Objekt wieder zurückgeholt werden, wenn die die Abstimmung auslösende Bemerkung wieder entfernt wird?

Einen für die angestrebte Verwendung zu niedrigen Parallelitätsgrad ermöglicht die **Restrukturierung** von Transaktionen, da weiterhin durch lange Sperren Lesezugriffe verhindert würden. Auch setzt die Spezifiktation der 'Read-' und 'Writesets' durch den Benutzer bei ihm ein Wissen über das zugrundeliegende Schema der Datenbank und den bisherigen Ablauf der Transaktion voraus, wenn die Bedingungen (3.1) bis (3.5) eingehalten werden sollen. Interessant erscheint allerdings die Idee, konsistente Teilergebnisse einer Transaktion freigeben zu können, ohne die ganze Transaktion beenden zu müssen.

Benutzerdefinierte Korrektheits- und Synchronisationskriterien stellen einerseits eine passende Erweiterung zu *ConceptBase* dar. Während *ConceptBase* durch seine Integritätsbedingungen die Datensemantik nutzt, beruht dieses Konzept auf Verwertung der den Transaktionen inhärenter Semantik. Auch das Abspeichern des zur Auswertung benötigten Wissens in Regeln/Grammatiken in der Datenbank ist beiden Konzepten gemein. Allerdings träten die bereits in der Beschreibung in Kapitel 3.2.2 angeführten Schwierigkeiten beim Entwurf der zur Beschreibung der erlaubten Verhaltensweisen benötigten Regeln in *ConceptBase* verstärkt auf. In eingen Modellierungsanwendungen werden Arbeitsablaufschemata dynamisch entwickelt, so daß eine Grammatik am Ende der Entwicklung stehen könnte, aber nicht als Voraussetzung.

Das **DDG-Verfahren** ist das einzige Verfahren, das gezielt für deduktive Datenbanken entwickelt wurde. Es versucht insbesondere das Problem, daß durch die Auswertungsmechanismen große Teile der Datenbank gesperrt werden müssen (vgl. Kap. 3.3.2), zu umschiffen, indem es ein nicht–zweiphasiges Verfahren verwendet. Der Freigabe von Sperren, die nicht mehr benötigt werden, steht nun nichts mehr entgegen. Obwohl aber *ConceptBase* eine deduktive Datenbank ist, sieht es mit der Anwendbarkeit schlecht aus. Die langen, interaktiven Transaktionen, die keine Voraussage der Lese- und Schreibmengen ermöglichen, führen bei diesem Verfahren dazu, daß eine Transaktion tatsächlich die ganze Datenbasis sperren müßte (Sperre auf den Wurzelknoten), um dem Benutzer alle Möglichkeiten offen zu halten.
Für das Setzen kurzfristiger Sperren (z.B. während der Auswertung einer einzigen Anfrage oder in Verbindung mit dem 'Sagas'–Konzept) böte sich dieses Verfahren allerdings an.

Da das **Gruppenkonzept** im Gegensatz zu den Geschachtelten Transaktionen Daten und nicht Transaktionen aufteilt, erscheint es für *ConceptBase* anwendbar. Insbesondere bei Benutzung eines *ConceptBase*–Servers durch verschiedenste Anwendergruppen ist eine Unterteilung in Gruppen mit unterschiedlicher Isolation nach innen und außen wünschenswert. Mittels einer Einteilung in Gruppen ließe sich auch eine Beschränkung der Änderungen der Benutzer auf ihnen zugeordnete Daten realisieren. Nicht vernachlässigt werden darf natürlich der durch die Verwaltung von Gruppen steigende Zugriffsaufwand auf Daten.

Eine Einschränkung der Brauchbarkeit des **Tool Kit Ansatzes** ergibt sich dadurch, daß er bisher hauptsächlich für eine Verwendung von Sperren ausgearbeitet wurde. Und während die Idee, jedem Applikationsentwickler die Entscheidung über das für ihn am sinnvollste Transaktionskonzept selbst zu überlassen, durchaus interessant erscheint, dürfte die Implementierung der hierfür notwendigen Grundbausteine und Kontrolleinheiten ziemlich arbeitsaufwendig sein.

Name	opt./ pes.	eingesetzt	lange TAs	Kooperation	interaktiv	Bemerkungen
2PL	pes.	überall	nein	nein	ja	konventionell
Mehrschicht-Transaktionen	-	DASDBS	begrenzt	nein	ja	
Kommutativität	pes.	-	begrenzt	begrenzt	ja	benötigt semantikreiche Operationen
Kompensations funktionen	pes.	-	ja	ja	begrenzt	Kompensation nur schwer zu definieren
Restrukturierung	pes.	-	etwas	ja	ja	nur geringe Erhöhung der Parallelitätsmöglichkeiten
Benutzerdefinierte Korrektheitskrit.	-	-	ja	ja	ja	umfangreiche Vorarbeiten zur Erstellung der Grammatiken
DDG–Verfahren	pes.	-	etwas	nein	nein	für deduktive DB entwickelt
Gruppenkonzept	-	-	-	ja	-	
Tool Kit	z.Z. pes.	Prototyp	wählbar	wählbar	ja	Transaktionstypen müssen aus Grundelementen zusammengesetzt werden
NSE	opt.	NSE	ja	begrenzt	ja	

Tabelle 3.5: Übersicht über die vorgestellten Transaktionskonzepte

Wenn auch **NSE** von SUN inzwischen wegen Problemen mit der Betriebssystemeinbindung eingestellt wurde, so bietet es doch interessante Ansätze. Insbesondere den Benutzer bei der Beseitigung von Kon flikten zu unterstützen (wie es ja auch [BK91, S.281] fordern), statt einfach die Transaktion abzubrechen, und ihm lediglich Änderungen in abgegrenzten Bereichen zu erlauben, könnte in ein Transaktionskonzept für *ConceptBase* integriert werden. Andererseits erscheint die Isolation zwischen den einzelnen Entwicklern auf unterster Stufe zwar für Programmieraufgaben angemessen, dürfte für interaktivere Anwendungen aber zu hoch sein.

Es bleibt also festzuhalten, daß sich nicht nur die klassischen Transaktionskonzepte, sondern auch viele fortgeschrittene Verfahren für *ConceptBase* schlecht eignen, da *ConceptBase* einige Abweichungen vom Standarderscheinungsbild neuerer Datenbanksysteme (insbesondere objektorientierter) zeigt. Zum einen bieten TELL, UNTELL und ASK nicht die reichhaltige Operationssemantik, die viele Konzepte auszunutzen versuchen. Aber auch die Tatsache, daß *ConceptBase* keinerlei Transaktionsprogrammierung gestattet, verringert für einige Konzepte (z.B. Kompensationsfunktionen) die Anwendbarkeit. Die Interaktivität der *ConceptBase*–Anwendungen läßt zudem wenig Vorhersagen über den Transaktionsverlauf zu, was z.B. dem DDG–Konzept, aber auch einigen nicht vorgestellten Verfahren, die die Read- und Writesets der Transaktionen benötigen, zu schaffen macht.

Kapitel 4

Das Transaktionssystem für ConceptBase

Nachdem im letzten Kapitel verschiedene Transaktionskonzepte vorgestellt und größtenteils als den Anforderungen nicht entsprechend bewertet wurden, stellt sich die Frage nach einer passenden Lösung für *ConceptBase*. Das im Rahmen dieser Diplomarbeit entworfene Konzept wird nun behandelt. Die darauffolgenden Abschnitte beschreiben dann die reale Implementierung und die dabei aufgetretenen Probleme.

4.1 Entwurf

Da kein vorhandenes Konzept übernommen werden konnte, habe ich versucht, die passendsten Ideen aus verschiedenen Ansätzen zu einem zu *ConceptBase* passenden Konzept zu integrieren. Hierfür mußten, wie der nächste Abschnitt zeigt, einige "liebgewonnene" klassische Transaktionseigenschaften aufgegeben werden. Dadurch war es dann allerdings möglich, auf der Basis einer Einteilung der Benutzer in Gruppen, ein optimistisches Konzept mit nach Bedarf anpaßbaren Isolationseigenschaften zu entwickeln. Ebenfalls vorgesehen wurde die Möglichkeit, die durch eine Gruppe änderbaren Objekte zu beschränken sowie die intelligente Erkennung von Konflikten auf 'Individual'ebene und ihre Behebung mittels 'Merging'.

4.1.1 Serialisierbarkeit wiederaufgegriffen – und fallengelassen

Wie in Kapitel 3.1 gezeigt, beruhen klassische Transaktionskonzepte darauf, die Aktionen der Transaktionen in eine serialisierbare Historie zu bringen. Die meisten fortgeschrittenen Systeme gewichten hingegen auf die eine oder andere Weise die Serialisierbarkeit geringer. Wofür wird diese Serialisierbarkeit überhaupt gebraucht?

Von den im oben erwähnten Kapitel aufgeführten vier klassischen Eigenschaften einer Transaktion (ACID) beruhen, wie bereits gezeigt wurde, in klassischen Konzepten zwei auf der Serialisierbarkeit: Konsistenz und Isolation. Isolation ist bekanntermaßen in Entwicklungsumgebungen nur eingeschränkt erwünscht, zur Erhaltung der Konsistenz bieten sich in *Concept Base* andere Möglichkeiten als die Serialisierbarkeit. Inwiefern diese Transaktionseigenschaften aufgeben werden, zeigen die nächsten beiden Unterkapitel. Vorher soll aber noch kurz darauf eingegangen werden, wie die beiden anderen Eigenschaften sichergestellt werden können.

Atomarität Bisher wird in *ConceptBase* die Atomarität von Transaktionen durch das Eintragen von Änderungen in einen lokalen temporären Speicher erreicht, dessen Inhalt erst nach erfolgreichem Abschluß der Operation in die Datenbasis übernommen wird. Es bietet sich an, diesen Mechanismus beizubehalten, da so das Eintragen inkonsistenter Informationen in die Datenbasis verhindert werden kann und das Rücksetzen von Transaktionen (Typ–1 Recovery) einfach durch das Löschen des temporären Speichers vorgenommen werden kann.

Dauerhaftigkeit Die Typ–2 bis Typ–4 Recovery soll wie gesagt in dieser Diplomarbeit nicht behandelt werden und ist bisher in *ConceptBase* auch nicht vorgesehen. Im Rahmen der Dauerhaftigkeit muß also nur beachtet werden, daß erfolgreich abgeschlossene Transaktionen nicht mehr zurückgesetzt werden dürfen. Zudem sollte das Transaktionskonzept die spätere Einführung einer Recovery–Komponente nicht von vorneherein unmöglich machen.

Isolation – Vielfalt statt Abgrenzung

Auf Seite 25 wurde ausgeführt, warum Isolation in Entwicklungsumgebungen nur eingeschränkt erwünscht ist: Entwickler wollen kooperativ zusammenarbeiten, was durch die Isolation verhindert wird. Andererseits ist eine gewisse Menge Isolation nötig, da nicht jeder unbedingt mit jedem kooperieren will. Anders gesagt: Wenn jede Transaktion die Zwischenergebnisse aller anderen sieht, läßt sich das Ergebnis der Transaktion kaum noch vorhersagen. Eine Festlegung auf ein bestimmtes Maß an Isolation, wie es in vielen anderen anwendungsnahen Konzepten gemacht wird, wirkte sich nachteilig aus, weil *ConceptBase* kein festgelegtes Anwendungsgebiet hat, auf das sich Isolationsanforderungen anpassen ließen.

Aus diesem Grund habe ich mich entschieden, die Wahl des Isolationsgrades zwischen den Transaktionen (vgl. Kap. 3.1.2) dem Benutzer bzw. dem Anwendungsprogrammierer freizustellen.

3° Isolation: Die stärkste Art der Isolation sollte hauptsächlich zwischen Projekten angewendet werden, die nichts miteinander zu tun haben. Die Datensicht eines Projekt bleibt dadurch während seiner ganzen Laufzeit gleich, und fremde Transaktionen können nicht durch Änderungen Verwirrung stiften.

Auch für alte Anwendungen, die bisher fälschlicherweise eine vollständige Isolation erwartet haben, eignet sich dieser Modus. Da nur unter 3° Isolation eine konsistente Analyse sichergestellt ist, kann die Ausführung von längeren Anfragen mit diesem Isolationsgrad ebenfalls sinnvoll sein.

2° Isolation: Diese Isolationsstufe entspricht am ehesten dem bisherigen Zustand in *ConceptBase*. Es dürfte die passendste Isolation für einen weiten Bereich von Anwendungen innerhalb eines Projekts sein. Benutzer können hierbei Änderungen sofort sehen, die andere Benutzer am Transaktionsende in die Datenbasis zurückgespielt haben[1]. Ihre

[1] Vor dem Transaktionsende werden Änderungen im temporären Speicher abgelegt, der, wie gesagt, erhalten werden soll.

eigenen Änderungen bleiben so lange unsichtbar, bis sie für den jeweiligen Benutzer zufriedenstellend und für das System konsistent sind. Damit die Änderungen anderer Benutzer nicht nur in der Datenbasis sichtbar werden, sondern auch dem Benutzer angezeigt werden, ist eine zügige Benachrichtigung der Applikationen sowie ein Neuaufbau ihrer Bildschirmanzeige notwendig.

Eine sinnvolle Erweiterung wäre es, konsistente Teilbereiche der Änderungen vor dem Transaktionsende allgemein zugänglich zu machen, wie es z.B. im Rahmen der Restrukturierung von Transaktionen (S.30) vorgeschlagen wurde.

0° Isolation: Diesen Isolationsgrad könnte man auch als "keine Isolation" bezeichnen. Jede Transaktion sieht sofort alle (nicht nur zurückgespielte) Änderungen anderer Transaktionen. Da dies ohne die Verwendung von Konzepten wie etwa "Kompensationsfunktionen" zu kaskadierendem Rücksetzen führen kann, emfiehlt es sich, alle beteiligten Transaktionen nur gemeinsam zu beenden oder zurückzusetzen. Genaugenommen arbeiten also alle Benutzer gemeinsam in einer Transaktion.

Geeignet erscheint dieser Isolationsgrad besonders für solche Kooperationsformen, bei denen mehrere Benutzer gleichzeitig ein Dokument oder andere gemeinsame, stark zusammenhängende Daten verändern. Es dürfte klar sein, daß die nicht vorhandene Isolation Maßnahmen der Applikationen erfordert, um die Zusammenarbeit der Entwickler zu koordinieren.

1° Isolation: Da dieser Grad der 0° Isolation ähnlich wäre und zudem Schwierigkeiten bei der Implementation bereitet, ist er im Konzept nicht vorgesehen. Er ließe sich ausgehend von 0° Isolation durch entsprechende Kommunikation der Applikationen oder durch das zusätzliche Einführen von Schreibsperren erreichen.

Wie kann das gleichzeitige Auftreten so unterschiedlicher Isolationsgrade in einer Datenbank verwaltet werden? Zum einen ist ein Aufbau vorstellbar, bei dem für jeden Isolationsgrad ein eigener Transaktionstyp zur Verfügung gestellt wird, mit dessen Hilfe alle Transaktionen direkt auf der Datenbasis arbeiten. Dies würde aber bedeuten, daß eine 0° isolierte Transaktion die Änderungen aller anderen Transaktionen sofort sehen würde, während sie lediglich Wert auf die von einigen bestimmten Transaktionen durchgeführten Änderungen legt. Die Einteilung der Transaktionen in verschiedene Gruppen[2] nach ihren "Interessengebieten" verspricht hier Abhilfe.

Kann innerhalb der Gruppen eine andere Isolation als zwischen ihnen gewählt werden, und werden die Gruppen hierarchisch angeordnet, so erhält man ein Konzept, das alle Wünsche an Isolationsvariationen erfüllen können sollte. Eine genauere Beschreibung der Gruppenhierachie findet sich in Kapitel 4.1.2.

Wegen des geplanten Zur–Verfügung–Stellen verschiedener Isolationsgrade muß, wie in vielen anderen Konzepten auch, auf die Serialisierbarkeit verzichtet werden. Ihre Ziele und Methoden widersprechen bekanntlich einem niedrigen Isolationsgrad.
Nachdem also die geplante Isolation den Verzicht auf Serialisierbarkeit erfordert, wird im nächsten Abschnitt untersucht, wie sich die Konsistenz ohne sie aufrechterhalten läßt.

[2]Wie es z.B. auch das Tool Kit Konzept vorsieht, das ebenfalls verschiedene Isolationsgrade ermöglicht.

Konsistenz – von verteilt nach zentral

Die meisten Datenbanksysteme versuchen die Konsistenz ihrer Daten dadurch zu erhalten, daß sie annehmen, daß es keine integritätsverletzenden Transaktionen gibt, wenn man sie kontrolliert (seriell) ausführt. Sämtliche Integritätsbedingungen stecken also im Code der Transaktionen. Dies nennt sich dezentraler oder auch präventiver Ansatz (vgl. [Jeu92, S.109f]).

ConceptBase hingegen besitzt in der Datenbasis abgespeicherte Integritätsbedingungen, so daß der umgekehrte Ansatz, zentral oder auch reaktiv genannt, verwirklichbar wäre. Hierbei führen die Transaktionen beliebige Manipulationen auf den Daten durch, wobei inkonsistente Änderungen aufgrund des in der Datenbank vorhandenen Wissens über mögliche Integritätsverletzungen zurückgewiesen werden. Die Ausführungs- und Verzahnungsreihenfolge der Transaktionen hat hier keine Auswirkungen auf die Konsistenz der Daten.

Transaktionscode:

```
lock(Flugzeug1);
frei:=Flugzeug1.freiePlätze;
if(frei>0)
   Flugzeug1.freiePlätze:=frei−1;
   Ausgabe("OK");
   unlock(Flugzeug1);
else
   Ausgabe("Alles voll");
   unlock(Flugzeug1);
endif
```

Datenmodell:

```
Class Flugzeug with
   attribute,single
      freiePlätze: Integer
   constraint
      Ladebeschränkung:
            forall f/Flugzeug i/Integer
            (f freiePlätze i)
            ==> i >= 0
end
```

Abbildung 4.1: Je ein Beispiel für präventive und für reaktive Integritätskontrolle

Die aktuelle Version von *ConceptBase* benutzt einen gemischten Ansatz, d.h. es gibt zwar Integritätsbedingungen in der Datenbasis, aber die Transaktionen versuchen selbständig, so wenig Inkonsistenz wie möglich zu erzeugen. Dies hat zur Folge, daß *ConceptBase* nicht alle zum Modell gehörigen Integritätsbedingungen bekannt sind, so daß weiterhin Wert auf eine serialisierbare[3] Ausführung der Transaktionen gelegt werden muß, wenn die Erhaltung der Konsistenz garantiert werden soll.

Diese doppelte Integritätskontrolle vereinfacht zwar die Arbeit für einen Anwendungsentwickler, da er nicht alle Bedingungen von vorneherein in sein Datenmodell integrieren muß, macht aber den Vorteil der zentralen Kontrolle zunichte, daß die Transaktionen nicht mehr serialisiert werden müssen. Im letzten Unterkapitel wurde jedoch gezeigt, daß wir auf die Serialisierbarkeit verzichten müssen, wenn die Isolation vermindert werden soll.

Damit das von mir vorgestellte Konzept funktionieren kann, müssen also alle Integritätsbedingungen in die Datenbasis aufgenommen werden. Hierdurch kann nicht nur auf die Serialisierbarkeit verzichtet werden, eine Konsistenzprüfung mittels Integritätsbedingungen erkennt auch Fehler, die durch "externe" Kooperation der Transaktionen (also z.B. durch informelle Gespräche der Entwickler bei einer Tasse Kaffee) entstehen, während einer präventiven Integritätskontrolle solche Abhängigkeiten verborgen blieben.

Der Verzicht auf die Serialisierbarkeit bringt allerdings auch einige Problemfälle mit sich, in denen die Konsistenz nach klassischer Auffassung nur ungenügend gesichert wird. Man

[3] *ConceptBase* implementiert bisher sogar eine serielle Ausführung.

betrachte dazu das folgende Beispiel:

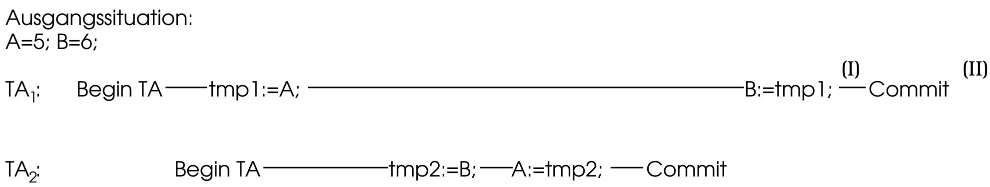

Abbildung 4.2: Nichtserialisierbarer Ablauf paralleler Transaktionen

Die Verwebung von Transaktion 1 und 2 kann nicht zu einem seriellen Ablauf aufgelöst werden. Eine Aussage über die Korrektheit wird dadurch aber noch nicht getroffen, da diese im vorgestellten Konzept über Integritätsbedingungen gesichert werden soll. Solange es keine Bedingung A=B gibt – und das ist aufgrund der Ausgangssituation unwahrscheinlich –, stellt die Abbildung eine konsistente Datenänderung dar.
In der Praxis stellt sich natürlich die Frage: Was passiert, wenn sich eine Transaktion darauf verläßt, daß A gleich B ist?

- Eine Nachfolgetransaktion TA_3 von TA_1 findet am Punkt (II) für sie ggf. überaschenderweise A≠B vor. Allerdings ist bereits die Annahme von TA_3 falsch, sich noch auf Ergebnisse aus TA_1 verlassen zu können. Schließlich können die Auswirkungen von TA_1 nach ihrer Beendigung durch andere Transaktionen geändert werden, unabhängig davon, wie dicht TA_3 auf TA_1 folgt. Abhilfe verspricht hier nur, TA_1 gar nicht erst zu beenden.

- Ob A=B innerhalb von TA_1 am Punkt (I) gilt, hängt davon ab, unter welchem Isolationsgrad TA_1 abläuft. Ist er gleich 3, so gilt A=B, sonst nicht. Dieses Ergebnis entspricht der Definition der Isolationsgrade und ist damit unabhängig vom verwendeten System zur Aufrechterhaltung der Konsistenz korrekt.

Werden diese durchaus mit der Transaktionstheorie in Einklang stehenden Abweichungen von der Intuition beachtet, so erhält das Verfahren die Konsistenz. Allerdings nur – und das soll hier noch einmal betont werden – wenn alle die Konsistenz beschreibenden Bedingungen in das verwendete Telos–Datenmodell aufgenommen wurden.

Auf die Serialisierbarkeit als Korrektheits- und Isolationskriterium kann und muß im vorgeschlagenen Transaktionskonzept also verzichtet werden.

4.1.2 Gruppen – jedem die seine

Im Abschnitt über Gruppenstrukturen (Kap. 3.2.3) wurde darauf hingewiesen, daß zwischen der Aufteilung von Projekten auf Entwickler einschließlich der unterschiedlichen Isolationsanforderungen und dem Aufbau eines Baumes (als Datenstruktur) eine natürliche Äquivalenz besteht. Dieser von vielen Konzepten ausgenutzte Umstand wird auch hier verwendet. Die Transaktionen werden auf eine Hierarchie von Gruppen verteilt, die den Aufbau des Projektes nachbildet. Im Gegensatz zu Konzepten wie z.B. den Geschachtelten Transaktionen besteht die Hierarchie aus zwei verschiedenen Komponenten:

1. Den Gruppen, die als passive Elemente hauptsächlich für die Verwaltung der Objekte und Zugriffsbeschränkungen (s.u.) zuständig sind. (Nach der Definition von [US92] sind sie also „service transactions“.)
2. Den Transaktionen, die als „operational transactions“ Anfragen und Änderungswünsche über ihre Gruppe an die Datenbank stellen.

Diese Zweiteilung ähnelt der von [NRZ92] verwendeten Einteilung in „transaction groups“ und „cooperative transactions“.

Auch die verschiedenen Isolationsbedürfnisse werden berücksichtigt. So besteht zwischen den Transaktionen innerhalb einer Gruppe nur eine 0° oder 2° Isolation während die Gruppen gegeneinander auf Grad 3 isoliert sind. Um dies zu erreichen, wird folgendermaßen vorgegangen: Jede Gruppe erhält bei ihrer Erzeugung eine logische Kopie aller Objekte aus ihrer Obergruppe. "Logische Kopie“ bedeutet, daß keine wirkliche Kopie der Daten angelegt wird, sich die sichtbaren Daten aber so verhalten, als wären sie kopiert worden. Die durch das Kopieren erhaltene Sicht wird weder durch Änderungen in der Obergruppe noch durch solche in etwaigen Untergruppen beeinflußt. Sichtbar werden lediglich die Endergebnisse der in der Gruppe ablaufenden Transaktionen sowie die von den Untergruppen explizit "hochgespielten“ Daten. Soll die Sicht einer Gruppe mit der der Obergruppe in Einklang gebracht werden (sollen also die in der Obergruppe sichtbar gewordenen Änderungen übernommen werden), so muß dies ebenfalls explizit angefordert werden. Zudem kann sowohl das "Hoch-“ als auch das "Runterspielen“ von Änderungen natürlich Konsistenzverletzungen auslösen, die vom Integritättester erkannt und vom Benutzer behoben werden müssen.

In der Gruppe bekannt gewordene Änderungen beliebigen Ursprungs werden für alle Transaktionen in der Gruppe sofort sichtbar, da diese höchstens Isolationsgrad 2 haben. Soll also eine Transaktion mit Isolationsgrad 3 ablaufen, so muß für sie eine eigene Gruppe erzeugt werden, die keine Untergruppen besitzt und während der Laufzeit der Transaktion keine Anpassung an den Zustand der Obergruppe vornehmen darf. Inkonsistenzen zwischen den Änderungen der 3° isolierten Transaktion und den übrigen werden allerdings erst beim "Hochspielen“ der Änderungen entdeckt.

Bei den 0° isolierten Transaktionen sind noch zwei Besonderheiten zu beachten. Zum einen ergeben sich, wie bereits weiter oben erwähnt, durch die geringe Isolation Probleme mit der Rücksetzbarkeit einzelner Transaktionen. Aus diesem Grund können alle 0° isolierten Transaktionen nur gemeinsam zurückgesetzt oder erfolgreich beendet werden. Zum anderen besitzen lediglich alle 0° isolierten Transaktionen gegeneinander diesen niedrigen Isolationsgrad. Gegenüber 2° isolierten Transaktionen beträgt ihr Isolationsgrad 2. Dieses der Definition etwas widersprechende Verhalten soll verhindern, daß zusätzlich zu den eben genannten Rücksetzproblemen noch Abhängigkeiten der 0° isolierten von den 2° isolierten Transaktionen auftreten. Die derzeitige Implementation geht sogar noch weiter und läßt nur entweder 0° oder 2° isolierte Transaktionen innerhalb einer Gruppe zu, der Isolationsgrad der Transaktionen ist also eine Eigenschaft der Gruppe.

Jeder Gruppe sind gewisse Attribute zugeordnet, die neben einem eindeutigen Namen, der Ober- und den Untergruppen auch die Möglichkeit bieten, den Zugriff auf diese Gruppen lediglich für bestimmte Personen zu gestatten. Zwar besitzt *ConceptBase* kein Authentisierungssystem, so daß von einem sicheren Zugriffsschutz keine Rede sein kann, aber wenigstens

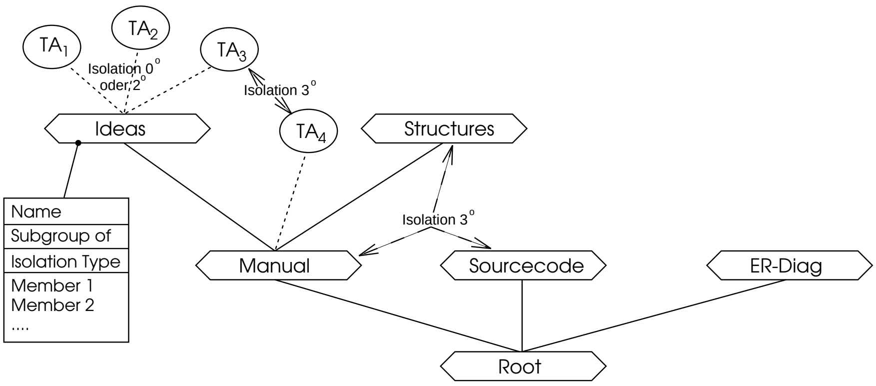

Abbildung 4.3: Struktur aus Gruppen und Transaktionen

in einer wohlwollenden Umgebung bietet sich – insbesondere in Zusammenhang mit den in Abschnitt 4.1.5 vorgestellten Verfahren – die Möglichkeit, den Datenzugriff zu beschränken.

Die Gruppenhierarchie muß natürlich auch dynamisch veränderbar sein und sich visualisieren lassen. Letzteres kann sehr einfach mit den bereits in *ConceptBase* vorhandenen Tools (Graphbrowser) erfolgen, wenn die grundlegenden Daten der Gruppenstruktur zur Verfügung stehen. Entweder blendet der Transaktionsmanager diese Daten virtuell in die Datenbank ein, ohne diese selbst zu verändern, oder er legt sie tatsächlich dort ab. Zwar hätte das zweite Verfahren den Vorteil, daß sich die Gruppenhierarchie durch 'TELL's und 'UNTELL's ändern ließe. Andererseits ergäben sich aber Probleme sowohl mit der Synchronisierung der vom Transaktionsmanager intern gehaltenen Strukturdaten mit den in der Datenbank vorhandenen, als auch mit der Sichtbarkeit und Zugriffsbeschränkung für diese Daten. Das Verschwinden einer bereits benutzten Gruppe durch das Rücksetzen der die Gruppe erzeugenden Transaktion stellt einen der möglichen GAUs dar.
Aus diesen Gründen ist es vorzuziehen, die Daten lediglich intern im Transaktionsmanager zu halten, der diese bei entsprechenden Anfragen in Propositionsform zur Verfügung stellt. Hierarchieänderungen lassen sich dann durch spezielle Befehle außerhalb einer Transaktion vornehmen.

4.1.3 Konflikte – erstmal finden

In früheren Kapiteln wurden drei Konfliktarten zwischen Schreib- und Leseoperationen genannt: R/W-, W/R- und W/W–Konflikte. Sie werden hauptsächlich bei der Bestimmung der Konflikt–Serialisierbarkeit von Transaktionen benötigt. Da aber dieses Konzept nicht mehr auf Serialisierbarkeit beruht, stellt sich die Frage, inwiefern die drei Konfliktarten noch aus anderen Gründen beachtet, erkannt und behoben werden müssen.

Für die R/W- und W/R–Konflikte läßt sich die Frage schnell negativ beantworten. Ein W/R–Konflikt entspricht einem 'dirty read', also dem Lesen noch nicht festgeschriebener Änderungen (vgl. Kap. 3.4.3, 3.1.2 und [GR93]). Dies kann im vorgestellten Konzept bei Transaktionen vom Isolationsgrad 2 nicht auftreten, da Änderungen grundsätzlich lokal gehalten werden und erst am Transaktionsende in die öffentliche Datenbasis eingefügt werden. Transaktionen in einer Isolationsgrad 0 gewährenden Gruppe sehen zwar sofort die Änderungen ihrer Mit-transaktionen, dies entspricht aber dem Isolationsgrad und braucht darum nicht verhindert zu werden.

Der R/W–Konflikt bedeutet, daß ein Datenelement bei einem späteren Lesezugriff einen anderen Wert aufweisen würde als bei einem früheren. Genau diese Eigenschaft läßt 2° Isolation – im Gegensatz zu 3° – aber zu. R/W–Konflikte stellen also in diesem Konzept keine Fehler, sondern eine erwünschte Verringerung der Isolation dar, so daß sie ebenfalls nicht verhindert werden müssen.

Einen Problemfall stellt der W/W–Konflikt dar. Seine Hauptberechtigung zieht er neben der Konfliktserialisierbarkeit aus der Transaktionseigenschaft "Dauerhaftigkeit". Bei einem W/W–Konflikt überschreibt nämlich eine Transaktion Änderungen einer anderen, möglicherweise ohne sie vorher zur Kenntnis genommen zu haben. Die Änderungen würden also verloren gehen, was der "Dauerhaftigkeit" widerspricht. In *ConceptBase* wird ein solcher Informationsverlust aber bereits durch die Verwaltung der Propositionen unmöglich gemacht. Jede neu eingefügte Proposition erhält einen eindeutigen Identifikator und Änderungen an einmal eingetragenen Propositionen sind unmöglich. Außerdem kann eine Proposition nur gelöscht werden, wenn ihr Identifikator bekannt ist, was erst der Fall ist, wenn die sie erzeugende Transaktion erfolgreich abgeschlossen wurde. Ergebnisse einer Transaktion können also erst vernichtet werden, wenn sie nach Abschluß dieser Transaktion von einer anderen gelesen wurden. Das ist aber keine Verletzung der "Dauerhaftigkeit".

Neben Verletzungen der "Dauerhaftigkeit" kann ein W/W–Konflikt auch bedeuten, daß zwei Benutzer gleichzeitig ein Objekt verändern, ohne voneinander zu wissen. Hieraus resultierende Konsistenzverletzungen (z.B. Löschen des Objekts bei gleichzeitigem Einfügen neuer Attributwerte) werden vom Integritätstester erkannt. Möglicherweise gibt es aber Fälle, bei denen parallele Änderungen zu einem nicht inkonsistenten aber dem Verständnis von Korrektheit widersprechenden Ergebnis führen.

Beispiel 4.1 *Die Instanz "max" der Klasse "Employee" besitzt noch keine eingetragenen Attribute. Während nun Benutzer A das Attribut* name : "Max Müller" *einträgt, löscht Benutzer B gleichzeitig die Instanzbeziehung von "max" zu "Employee" und trägt eine neue Beziehung zur Klasse "Plant" ein. Auch die "Plant"-Klasse besitzt ein Namensattribut, das allerdings normalerweise den Gattungsnamen angibt. Der Abschluß der beiden Transaktion würde also keine Konsistenzverletzung auslösen, aber eine neue Pflanzenart "Max Müller" kreieren.*

Die Argumentation, daß dies ebenso das Ergebnis zweier serieller Transaktionen sein könnte, ist falsch. Benutzer B hätte dann nämlich festgestellt, daß das Objekt einen für eine Pflanze ungewöhnlichen Namen hat und diesen entweder gelöscht oder die Uminstanziierung unterlassen.

Das Beispiel ist natürlich an den Haaren herbeigezogen, zumal Änderungen der Instanziierungsbeziehung in diesem Ausmaß nie vorkommen sollten. Andere aufgetauchte Beispiele, die die Instanziierung konstant lassen, bewirken allerdings nur das Eintragen redundanter Informationen, was zwar nicht erstrebenswert aber auch nicht falsch ist.

Der Mangel an Beispielen für krasse Fehler durch W/W-Konflikte führte mich im Laufe meiner Überlegungen zu der Überzeugung, daß der für das Erkennen dieser Konflikte notwendige Aufwand in keinem Verhältnis zum dadurch erhaltenen Gewinn steht. Da ich aber andererseits keinen Ansatzpunkt gefunden habe, zu beweisen, daß nicht doch Beispiele mit schwereren Fehlern existieren, wird im folgenden der ursprünglich für die Erkennung von W/W–Konflikten vorgesehene Mechanismus beschrieben, wenn er auch nicht implementiert wurde.

Weil, wie in Kapitel 3.4.3 gezeigt, W/W–Konflikte auf Propositionsebene nicht existieren, müssen sie auf Objektebene erkannt werden. Als Konflikt wird angesehen, wenn zwei parallele Transaktionen Änderungen am gleichen Objekt vornehmen; also, im Vorgriff auf Begriffe aus der Implementierung, wenn verschiedene Transaktionen gleichzeitig auslaufende Kanten in die Beziehungsmengen eines TOB (vgl. S.10 und S.63) einfügen oder aus ihnen löschen. Zur Erkennung merkt sich jede Transaktion den Zeitpunkt ihres frühesten Schreibzugriffs auf jedes Objekt und überprüft an ihrem Ende, ob zwischen diesem Zeitpunkt und dem Transaktionsende Änderungen an dem Objekt in die Hauptdatenbasis eingetragen wurden. Ein positives Ergebnis dieses Tests bedeutet einen W/W–Konflikt.

4.1.4 Optimismus – und wenn's schiefgeht?

Nach den guten Nachrichten des letzten Abschnitts stellt sich natürlich die Frage, ob Optimismus oder Pessimismus überhaupt noch benötigt wird, wenn es keinererlei Konflikte zwischen den Transaktionen gibt. Es darf aber nicht vergessen werden, daß "Konflikte" in diesem Konzept lediglich in "Verletzungen der Integritätsbedingungen" umgewandelt wurden. Zudem soll hier auch die ursprünglich geplante Behandlung von W/W–Konflikten erwähnt werden.

Warum optimistisch?

Die Nachteile eines pessimistischen Konzepts wurden bereits in verschiedenen Kapiteln dargelegt. Zum einen beruhen die meisten pessimistischen Synchronisationsverfahren darauf, die Konsistenz verletzende Zugriffe nicht zuzulassen. Das Abweisen von Lesezugriffen führt in deduktiven Datenbanken aber schnell zu einer streng seriellen Ausführung der Transaktionen oder zu 'Deadlocks'. Um die in *ConceptBase* vorhandenen Integritätsbedingungen pessimistisch zu wahren, müßte nach jedem Befehl einer Transaktion ihre Einhaltung geprüft werden. Effizient ist dies nur mit Konsistenzbedingungen möglich, wie sie in Kapitel 3.2.2 beschrieben wurden, nicht aber mit denen in *ConceptBase*. Neben dem hohen Zeitverbrauch für die Konsistenztests verhindert ein pessimistischer Integritätstester auch, innerhalb einer Transaktion inkonsistente Zustände zu erreichen, was möglicherweise aber gewünscht wird.

Optimistisch hingegen wird der Test auf Konsistenz lediglich einmal am Schluß der Transaktion ausgeführt, was die soeben beschriebenen Nachteile verhindert. Während der Laufzeit der Transaktion erhält der Konsistenztester lediglich die *ConceptBase* zugrunde liegenden Axiome (s. S.12) aufrecht, um den verschiedenen Auswertungsroutinen einen geordneten Betrieb zu ermöglichen.

Das zur Verhinderung von Konsistenzverletzungen einzusetzende Protokoll ist also klar: ein optimistisches. Aber welches erkennt und verhindert W/W–Konflikte (vgl. Abschnitt 4.1.3) am besten?
Da bereits ein Teil des Konzeptes Optimismus verwendet, bietet es sich natürlich an, auch W/W–Konflikte optimistisch zu erkennen. Das in Abschnitt 4.1.3 beschriebene Verfahren kommt dann zur Anwendung.

Zur Erkennung von W/W–Konflikten allein ließe sich auch ein pessimistisches Konzept verwenden. Wenn die Transaktionen vom zentralen Objektserver über ein Sperrprotokoll eine Berechtigung erwerben müssen, bevor sie ein Objekt ändern dürfen, sind keine Schreib–Schreib–Konflikte möglich. Nachteilig wirkt sich allerdings aus, daß nun pro Objekt nur noch ein Benutzer zugreifen darf, was eine hohe 'Deadlock'rate zur Folge haben kann. Desweiteren verzögern sich Schreibzugriffe, da sie einen Nachrichtenaustauch mit dem Objektserver erfordern.

Aus diesen Gründen sollte die ursprünglich vorgesehene Erkennung von W/W–Konflikten optimistisch erfolgen.

Reaktionen auf Fehler

Optimistische Verfahren verlangen nach einer fehlgeschlagenen Validierung am Transaktionsende ein Zurücksetzen der Transaktion, was, wie mehrfach erwähnt, für interaktive Entwurfsanwendungen unannehmbar ist. Das vorgestellte Konzept gibt darum dem Benutzer die Gelegenheit, aufgetretene Fehler durch weitere Befehle in seiner Transaktion zu beheben und danach erneut ein 'Commit' zu versuchen. Die notwendigen Reaktionen des Benutzers richten sich nach dem aufgetretenen Fehler.
Bei einer Verletzung von Konsistenzbedingungen muß der Benutzer diese anhand der Fehlermeldungen des *ConceptBase*–Kerns durch Einfügen oder Löschen weiterer Daten beheben. Sowohl die nur schwer zu verstehenden Fehlermeldungen als auch die Unkenntnis des Benutzers über das zugrundeliegende Datenmodell können die Fehlerbeseitigung erschweren. [ML91] schlagen darum ein Verfahren vor, das aufgrund der verletzten Konsistenzbedingungen die Ursachen für die Verletzung ermittelt und dem Benutzer anschließend mehrere Möglichkeiten zur Reparatur vorschlägt. Die Implementation eines solchen Konzeptes liegt natürlich weit außerhalb des Themengebiets dieser Diplomarbeit.

Aufgetretene W/W–Konflikte können durch die Ausführung zusätzlicher Befehle innerhalb der Transaktion nicht behoben werden. Hier muß dem Benutzer vielmehr die Möglichkeit gegeben werden, ähnlich wie in NSE (vgl. Kap. 3.2.3) die Ergebnisse der verschiedenen Schreibzugriffe auf das Objekt zu einem Gesamtzustand zusammenzufassen ('Merging'). Für dieses Verfahren gilt die schon eben erwähnte Einschränkung, daß ein Benutzer, dessen einzige Erfahrung mit *ConceptBase* die grafische Benutzeroberfläche einer Anwendung ist, durch Einblicke in die der Anwendung zugrundeliegende Datenstrukturen überfordert ist.

Auch in *ConceptBase* kann es Transaktionen geben, die nicht interaktiv sind. Da bei ihnen Konflikte nicht durch Benutzereingriffe behoben werden können, müssen sie abgebrochen werden, falls bei Transaktionsende ein Fehler erkannt wird. Damit der Transaktionsmanager die richtige Fehlerbehandlung anstoßen kann, muß ihm am Transaktionsanfang mitgeteilt werden, ob eine Transaktion interaktiv ist oder nicht.

4.1.5 Zugriffsbeschränkung – Finger weg von meinen Objekten!

Wie bereits in Kapitel 3.3.2 gezeigt, führt das Verhindern oder Hinauszögern von Lesezugriffen in deduktiven Datenbanken zu unannehmbaren Leistungseinbußen. Wenn in diesem Abschnitt also von Zugriffsbeschränkungen die Rede ist, so handelt es sich immer um die Beschränkung des schreibenden Zugriffs.

Das Gruppenkonzept (S.34) bezieht einige seiner Vorteile aus der Annahme, daß sich die Daten größtenteils disjunkt auf die verschiedenen Gruppen verteilen lassen. Da dies in *ConceptBase* für lesenden Zugriffe nicht durchführbar ist, bleibt nur noch die Möglichkeit übrig, Änderungsbefugnisse aufzuteilen. Dies ist auch sinnvoll, da die Zahl möglicher Konflikte zwischen den Gruppen abnimmt, je weniger Objekte von mehreren Gruppen aus geändert werden können.

Zur Verwirklichung dieser Idee wird ein dem aus SCCS bekannten 'Check Out' ähnlicher Mechanismus eingeführt. Bevor ein Objekt von einer Transaktion geändert werden kann, muß es zuerst von der entsprechenden Gruppe zum Schreiben angefordert worden sein. Diese vom eigentlichen Schreiben unabhängige Operation kann nur Erfolg haben, wenn das Objekt in der Obergruppe mit Schreiberlaubnis vorhanden ist. Ein automatisches mehrstufiges Anfordern solcher Berechtigungen – wie z.B. im Tool Kit Ansatz vorgesehen – findet nicht statt, da dies der Idee der Zugriffsbeschränkung widersprechen würde. Weil dieses Verfahren auch nicht zur Synchronisation genutzt wird, können im Gegensatz zum 'Check Out' von SCCS mehrere Untergruppen gleichzeitig eine Schreibberechtigung anfordern. Eine genauere Betrachtung, <u>wer</u> am sinnvollsten Schreibberechtigungen für eine Gruppe anfordern darf (alle Gruppenmitglieder, der Gruppenerzeuger, bestimmte Personen mit einem ausreichenden Rechtestatus) bleibt einer eventuellen Ausarbeitung über ein Authentisierungskonzept für *ConceptBase* überlassen.

Eine Erweiterung auf verschiedene Arten von Berechtigungen für Schreiben, Löschen, Einfügen von Instanzen und das Erzeugen ein- oder auslaufende Kanten, wie sie z.B. von [US92] vorgeschlagen werden, wäre eine passender Ausbau. Es bleibt auch zu überlegen, wie die Anforderung von Schreibberechtigungen über das Anfordern jeweils einzelner Objekte hinausgehend erweitert werden kann.

Von verschiedenen Personen wurden Zweifel geäußert, ob das optimistische Konzept in gewissen Situationen das Arbeiten nicht dadurch unmöglich macht, daß sämtliche Inkonsistenzen erst zum Transaktionsschluß geballt bekannt werden. Insbesondere treten diese Situationen bei gleichzeitigen Änderungen an einer Klasse und ihren Instanzen durch zwei verschiedene Transaktionen auf. Werden in der Klasse enthaltene Regeln geändert, so kann es u.U. vorkommen, daß die Ergebnisse der Transaktionen sich so stark widersprechen, daß eine der beiden Transaktionen zurückgesetzt werden muß, was natürlich unerwünscht ist.

Abhilfe versprächen "echte" Schreibsperren, die im Gegensatz zu obigen Schreibberechtigungen nur an jeweils einen Benutzer vergeben würden. Bei Änderungen an einer Klasse würden alle abhängigen Klassen und Instanzen gesperrt, so daß die Zahl unerwartet auftretender Inkonsistenzen verringert wird. Aufgrund der in Abbildung 3.7 sichtbaren, sowohl nach oben als auch nach unten verzweigenden Klassenhierarchie erscheint aber bereits das Sperren aller von einer Klasse abhängenden Unterklassen und Instanzen schwierig. Zudem können die Regeln und Konsistenzbedingungen einer Klasse auch Instanzen völlig anderer Klassen benutzen, so daß durch Änderungen an diesen Instanzen ebenfalls die oben erwähnten Inkonsistenzen entstehen können. Es läßt sich also kaum ermitteln, welche Objekte gesperrt werden müßten, um die unbeeinflußte Änderung einer Klasse zu garantieren.
Es erscheint darum sinnvoller, Änderungen am Schema, die solche Effekte hervorrufen könnten, extern zu koordinieren.

4.1.6 Zusammenfassung

Das vorgeschlagene Konzept beruht hauptsächlich darauf, statt der Konsistenzsicherung durch Serialisierbarkeit die in *ConceptBase* eingebaute reaktive Konsistenzkontrolle zu benutzen. Damit gestattet es einer Entwicklungsumgebung ein angemessenes höheres Maß an Kooperation, als es mit klassischen Verfahren möglich wäre. Durch die Aufteilung der Entwickler auf Gruppen wird die zwischen unabhängigen Arbeiten notwendige Isolation erreicht. Die Konflikterkennung und -behebung verläuft optimistisch, wobei ein Konflikt kein sofortiges Rücksetzen auslöst, sondern dem Benutzer die Möglichkeit zur Fehlerkorrektur gegeben wird.

Dieses Konzept erfordert ein gewisses Umdenken bei allen *ConceptBase*–Benutzern. Insbesondere das Aufnehmen aller Konsistenzbedingungen in das Datenmodell sowie die Behandlung der verschiedenen Isolationsgrade in den Applikationen sind bisher unbekannte Anforderungen bei der Applikationsentwicklung.

4.1.7 Korrektheit

Für viele Transaktionskonzepte existieren Korrektheitsbeweise ([Wei88, RMB+93, CHM92]). Hierbei wird meist gezeigt, daß die Verfahren eine Serialisierung erreichen bzw. die Datenkonsistenz erhalten. Das Prinzip der Serialisierbarkeit wurde für dieses Konzept jedoch fallengelassen und die Datenkonsistenz wird per Definition erhalten, da vor jedem Transaktionsende die Konsistenzbedingungen der Datenbank überprüft werden. Es erscheint darum fraglich, ob ein formaler Korrektheitsbeweis möglich oder auch nötig ist.

Ich möchte darum informell die wichtigsten Maßnahmen zur Erhaltung der ACID–Eigenschaften aufzählen:

Atomarität wird durch die Ansammlung aller Datenänderungen einer Transaktion in einem temporären Speicher erreicht, der erst nach erfolgreichem Konflikt- und Konsistenztest in die Hauptdatenbasis überspielt wird.

Konsistenz wird mit Hilfe der in das jeweilige Modell integrierten Konsistenzbedingungen sichergestellt, die vor jedem Transaktionsende von dem in *ConceptBase* enthaltenen Integritätstester überprüft werden.

Isolation wird in drei verschiedenen Ausprägungen zur Verfügung gestellt. Die 'No dirty read'-Eigenschaft wird durch den schon erwähnten temporären Speicher und eine Beschränkung der Zugriffe auf die Datenbank während der Rückspielzeit einer abgeschlossenen Transaktion gewährleistet. 'Lost updates' können in *ConceptBase*, wie bereits mehrfach erwähnt, nicht auftreten, und 'Repeatable reads' werden für 3° Isolation durch das Aufbewahren alter Propositionen zur Verfügung gestellt.

Dauerhaftigkeit umfaßt in diesem Konzept nur die Sicherstellung, daß Ergebnisse einer Transaktion nicht einfach verloren gehen dürfen. Erreicht wird dies durch die Eigenschaft von *ConceptBase*, jeder Proposition eine neue ID zuzuordnen, so daß sich das Problem der 'lost updates' gar nicht erst stellt.

4.2 Implementierung

In einem theoretischen Konzept lassen sich viele Eigenschaften versprechen und Anforderungen erfüllen. Eine entscheidende Frage ist aber immer, ob sich diese Ergebnisse auch in die Praxis umsetzen lassen. Der Versuch dieser Umsetzung und die dabei aufgetretenen Probleme werden im folgenden beschrieben.

4.2.1 Grobarchitektur

In Kapitel 2.2.5 wurde die bisherige Architektur von *ConceptBase* erläutert. Aus Benutzersicht ist *ConceptBase* ein Client–Server–System, wobei die Applikationen die Klienten sind und der *ConceptBase*–Kern der Server ist. Unter den auf Seite 14 aufgestellten Anforderungen findet sich die „Verbesserung der Antwortzeiten bei gleichzeitigen Befehlen mehrerer Benutzer“. Dies umschreibt den Wunsch, vom bisherigen Zustand, daß der Server die Befehle aller Benutzer streng seriell ausführt, zu einer Regelung zu gelangen, bei der die Anfrage eines Benutzers nicht mehr mehrere Minuten lang den Server für alle anderen Befehle blockiert.

Multi–threaded vs. ProLog

Den Grund für dieses Verhalten stellt die Programmierung des Servers in ProLog dar, die es nicht erlaubt, den Server mit einer 'Multi–thread' Architektur zu versehen, was eigentlich Stand der Technik wäre. 'Multi–thread' bedeutet, daß mehrere Prozeßflüsse (für jeden Benutzer ein eigener) gleichzeitig denselben Code auswerten (natürlich unter entsprechender Synchronisation mittels Mutex–Variablen oder Semaphoren). Damit dies möglich ist, muß der Code reentrant sein, das heißt, er darf nicht selbst-modifizierend sein und darf Änderungen globaler Variablen nur in einem ihm zugeordneten lokalen Speicher durchführen. Es ist aber nicht nur unklar, inwieweit der von BIM–ProLog implementierte ProLog-Auswerter – also insbesondere die 'Warren–Abstract–Machine' – reentrant ist, auch im eigentlichen ProLog–Programm des Servers werden häufig globale Variablen (durch “assert()“ oder “:- 'foo@bar'“–Konstruktionen) benutzt, so daß die in reinem Prolog–Code vorhandene Seiteneffektfreiheit verloren gegangen ist.

Da der gesamte ProLog–Kern also für eine 'Multi–threaded' Implementierung ausscheidet, müssen Alternativen untersucht werden: Im Gegensatz zum ProLog–Kern ließen sich die in C++ geschriebenen Programmteile – also Objektspeicher und Teile des CB–Interfaces – durchaus reentrant implementieren. Allerdings müßte der von ihnen eingeschachtelte ProLog–Teil als kritischer Abschnitt betrachtet werden, so daß sich jeweils nur ein Prozeß in ihm befinden dürfte. Da die Befehle also weiterhin seriell abgearbeitet werden müßten, würde die hierdurch ermöglichte Parallelität im Bereich der Kommunikation nach außen und im Bereich der grundlegenden Datenverwaltung kaum eine Verbesserung der Antwortzeiten mit sich bringen. Der Flaschenhals liegt im ProLog–Teil.

2–Ebenen–Client–Server–Architektur

Wenn das Problem darin besteht, daß ein Benutzer mit einem Befehl den ProLog–Kern des Servers “beliebig“ lange blockieren kann, warum geben wir dann nicht jedem Benutzer seinen

eigenen ProLog–Teil? In dieser Konfiguration wird ein Benutzer nur noch durch die Auswertung der eigenen Befehle gebremst, was normalerweise sowieso der Fall ist, da er auf ihre Ergebnisse wartet. Wenn das noch nicht ausreicht, wäre auch ein ProLog–Kern pro Transaktion vorstellbar.
Statt des normalen Aufbaus aus einem gemeinsamen Codebereich, in dem viele Threads mit jeweils begrenzten lokalen Daten ablaufen, wird also eine Struktur gewählt, in der in vielen Codebereichen jeweils ein eigener Prozeß mit jeweils einer größeren Menge lokaler Daten Operationen ausführt.

Für jeden Benutzer eine Kopie des *ConceptBase*–Kerns anzulegen, klingt nach „verteilter Datenbank". Eine verteilte Datenbank würde aber zusätzliche Probleme im Bereich der Synchronisation und Replikation erzeugen, die, da nicht im Aufgabenbereich dieser Diplomarbeit liegend, möglichst vermieden werden sollten. Aus diesem Grund wird ein zentraler Prozeß eingeführt, der allen *ConceptBase*–Kern–Kopien (im folgenden kurz "Kerne" genannt) als Datenserver dient. Dieser Prozeß (im folgenden "Objektserver") verwaltet alle bekannten Propositionen zusammen mit Informationen über den Aufbau der Gruppenhierarchie sowie der Sichtbarkeit der Propositionen in den Gruppen. Replikations- und Synchronisationsprobleme werden dadurch gelößt, daß der Objektserver als primärer Datenspeicher und als Koordinator dient. Da der Objektserver zudem noch völlig neu programmiert werden muß, kann eine Implementationssprache gewählt werden, die die Realisierung als 'Multi–threaded' Server zuläßt.
Die neue Architektur für *ConceptBase* besitzt also ein zweistufiges 'Client–Server'–System, bei dem die Klienten der ersten Stufe (die Kerne) die Server der zweiten Stufe (für die Applikationen) sind. Eine graphische Darstellung findet sich in Bild 4.4.

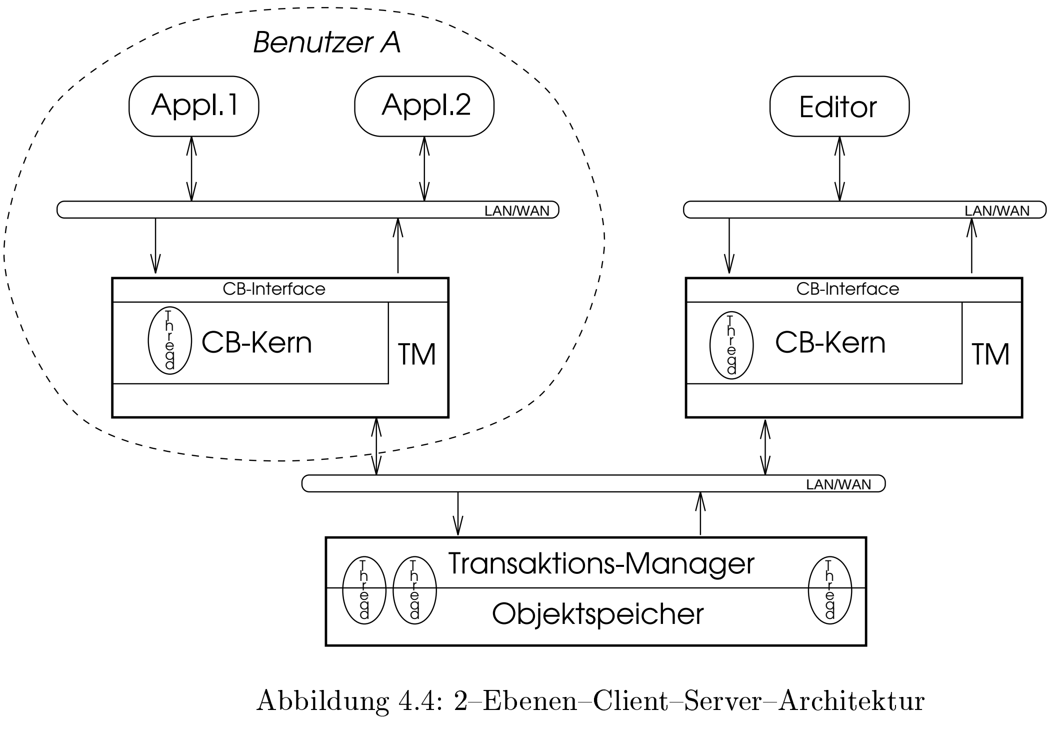

Abbildung 4.4: 2–Ebenen–Client–Server–Architektur

Die Vorteile dieses Ansatzes bestehen zum einen darin, daß er die parallele Auswertung von Benutzerbefehlen zuläßt, ohne einen völligen Neuentwurf des *ConceptBase*–Kerns zu erfordern. Außerdem kann der Objektserver bei Ausführung auf einem Mehrprozessorrechner aufgrund seiner Architektur mit mehreren 'Threads' sogar die Anforderungen mehrerer Kerne echt parallel bearbeiten. Die Nachteile sollen aber nicht verschwiegen werden:

- Wie aus Tabelle 2.1 zu ersehen ist, führt der *ConceptBase*–Kern bereits bei einfachen Aufgaben eine hohe Zahl an Leseoperationen auf den Propositionen durch. Bei einer Architektur wie in Abbildung 4.4 bedingt aber jede Leseanweisung einen Zugriff auf die im Objektserver gehaltenen Daten, also die Übertragung zweier Nachrichten zwischen Kern und Objektserver. Da sich die beiden Prozesse ggf. auf verschiedenen Rechnern befinden, benötigt ein Lesezugriff ein Vielfaches der Zeit, die ein Hauptspeicherzugriff[4] kosten würde. Im Vergleich zum augenblicklichen *ConceptBase*–System ergibt sich also ein deutlicher Leistungseinbruch. Andere Forscher ([RD91]) haben in Simulationen gezeigt, daß sich der Durchsatz in einem 'Client–Server'–System durch das Hinzufügen von Caches an die Klienten um eine Größenordnung verbessern läßt.

  Jeder Kern erhält also einen Cache, um einmal vom Objektserver erhaltene Propositionen für weitere Zugriffe bereit zu halten. Zusätzlich wird auch die Verwaltung der temporären Bereiche der Transaktionen, den Kernen übertragen, so daß der Nachrichtenverkehr zwischen Kernen und Objektspeicher deutlich verringert wird. Auf den Cache und Probleme der Anpassung an Änderungen im primären Speicher wird näher in Abschnitt 4.2.3 eingegangen.

- Dadurch, daß jeder Benutzer seinen eigenen Kern benutzt, ergibt sich der Vorteil, daß sich die Rechenlast dieser Kerne auf verschiedene Computer verteilen läßt. Ein Nachteil besteht aber darin, daß jeder dieser Kerne Speicher für den BIM–ProLog–Interpreter samt Code und Daten und für den oben erwähnten Propositionen–Cache verbraucht. Werden mehrere Kerne auf dem gleichen Rechner ausgeführt werden, stehen dem durch 'Paging' zu erwartenden Leistungseinbruch kaum noch Vorteile gegenüber.

- Wie noch gezeigt wird, führt die Zweiteilung in Kern mit angegliedertem Cache und Objektserver zu einem erhöhten Verwaltungsaufwand und zusätzlicher Komplexität der Implementierung, da verschiedene Datenkopien synchronisiert und verwaltet werden wollen. Hinzu kommt, daß die Gründe für die Aufteilung in Kern und Objektserver nur bedingt mit dem Transaktionskonzept und eher mit der Implementierung von *Concept Base* in ProLog zu tun haben.

Um die streng serielle Auswertung von Befehlen aufbrechen zu können, wird also das einstufige 'Client–Server'–System von *ConceptBase* zu einem zweistufigen erweitert, das durch die Zuweisung eines eigenen ProLog–Teils an jeden Benutzer und durch die 'Multi–threaded' Implementierung des zugrundeliegenden primären Speichers den Grad möglicher Parallelität beträchtlich erhöht.

4.2.2 Änderungen am Objektspeicher

Die Datenspeicherung des Transaktionskonzepts beruht auf dem von Hubertus Pützer und Rainer Gallerdörfer entwickelten Objektspeicher (vgl. Kap. 2.2.4). Sowohl an der Datenstruktur (dem TOB–Objekt) als auch an der eigentlichen Implementierung waren Abänderungen

[4] In der derzeitigen *ConceptBase*–Implementierung entspricht ein Lesezugriff auf eine Proposition einer gewissen Anzahl an Hauptspeicherzugriffen.

notwendig, die im folgenden erläutert werden sollen. Aufgrund dieser Änderungen war es nicht möglich, Verbesserungen, die Hr. Pützer nach der Trennung der Entwicklungen ca. Anfang September vornahm, zu integrieren.

Der weitestgehende Unterschied betrifft die Neugestaltung der TOB–Struktur. Wegen der Existenz mehrerer Gruppen und Transaktionen mit u.U. verschiedenen Sichten auf die Datenbasis, mußte eine Möglichkeit integriert werden, die Ergebnisse eines Beziehungsmengenzugriffs je nach zugreifender Gruppe/Transaktion zu variieren. Hierfür sind zwei Vorgehensweisen offensichtlich.

1. In einer Menge OID_{Grp}[5] werden die Identifikatoren aller einer Gruppe/Transaktion bekannten Objekte abgespeichert. Nach jedem Zugriff auf die Datenbasis wird die hierbei erhaltene Lösungmenge von Objektidentifikatoren mit OID_{Grp} geschnitten. Das Ergebnis stellt die "sichtbare" Lösung dar, also die Lösung, die wirklich nur die für die Gruppe/Transaktion sichtbaren Objekte enthält.

2. Die Beziehungsmengen eines Objekts werden für jede Gruppe/Transaktion getrennt gespeichert. Bei einem Zugriff werden nur die passenden Beziehungsmengen ausgewertet, so daß keine anschließende Schnittoperation nötig ist.

Die erste Methode verwendet Thomas List zur Modellierung von historischer Datenbank und temporärem Speicher in seiner Integration des Objektspeichers. Der unbestreitbare Vorteil ist die Einfachheit dieser Vorgehensweise und ihr geringerer Speicherverbrauch. Sie hat hingegen den Nachteil, bei jedem Zugriff einen Mengenschnitt zu erfordern. Die zum Schneiden zweier Mengen benötigte Zeit wächst aber mindestens linear mit der Größe der kleineren Menge, die in diesem Fall fast immer die Lösungsmenge des Zugriffs sein dürfte. Dadurch werden Anfragen, die große Ergebnismengen liefern, wie z.B. die Anfrage nach den Instanzen von "Class" oder "Individual", verlangsamt.

Das zweite Verfahren erfährt diese Verlangsamung natürlich nicht, benötigt dafür aber mit der Zahl der Gruppen und Transaktionen linear wachsenden Speicher. Einen zusätzlichen Vorteil stellt noch die Vereinfachung des Abbruchs von Transaktionen dar. Da jede Transaktion die von ihr beeinflußten Beziehungsmengen anderer Objekte neu anlegt, können diese Beziehungsmengen bei einem Rücksetzen der Transaktion einfach gelöscht werden.

Einen weiteren Grund für die notwendige Überarbeitung stellt die Verwendung von Zeigern in den Beziehungsmengen dar. Zwar wird der Zugriff auf Objekte dadurch beschleunigt, weil die Umsetzung von Identifikatoren in Speicherzeiger vermieden wird. In einer Datenbank, in der Objekte auf verschiedene Prozesse und Rechner verteilt sind, lassen sich Speicheraddressen aber nicht mehr zur Objektidentifikation verwenden. An vielen Stellen mußten darum Zeiger durch OIDs ersetzt werden. Um die hierdurch häufiger nötigen Dereferenzierungen der OIDs zu beschleunigen, wurde außerdem die OID-Klasse abgewandelt, um ein schnelleres Hashing zu ermöglichen. Weitere Abwandlungen betreffen die Vereinheitlichung der Benutzung von Zeigern und "realer" Strukturen in den verwendeten C++–Klassen sowie die Gliederung der sechs Beziehungsmengen des 'Bodys' in zwei Arrays, um die zugreifenden Funktionen generalisieren zu können.

[5] OID=Objektidentifikator

Die neuen TOB–Datenstrukturen

Die folgende Darstellung zeigt die drei Derivate der TOB–Struktur, die das Transaktionskonzept verwendet.

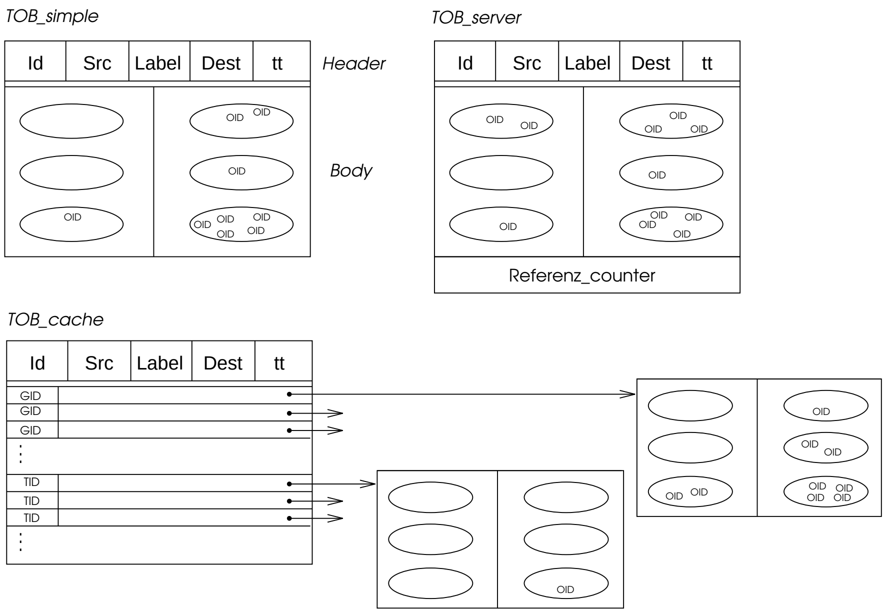

Abbildung 4.5: Die Datenstrukturen TOB_simple, TOB_Server und TOB_cache

Wie man sieht entspricht die TOB_simple–Struktur weitgehend der von Rainer Gallersdörfer entwickelten Struktur. Zusätzliche Verwaltungsinformation, die im 'Header' abgelegt wird, ist in der Grafik allerdings nicht enthalten. TOB_simple wird zur Speicherung der Propositionen im temporären Bereich verwendet. Da dieser einer Transaktion fest zugeordnet ist, die sich wiederum fest in einer Gruppe befindet, ist eine Unterteilung der Beziehungsmengen nicht notwendig.

TOB_server unterscheidet sich von TOB_simple nur durch einen zusätzlichen Referenzierungszähler. Diese Struktur wird im primären Speicher des Objektservers benutzt, wobei der Referenzierungszähler zur Feststellung dient, wann sich die durch die Struktur repräsentierte Proposition in keiner Gruppe mehr befindet, um sie dann ggf. auf einem langsameren Speichermedium abzulegen.

Es überrascht, daß es auch hier nur eine Variante der Beziehungsmengen gibt, obwohl alle Gruppen im Objektserver bekannt sind und sich die Beziehungsmengen von Gruppe zu Gruppe unterscheiden. Der Grund liegt darin, daß der Objektspeicher, um seinen Speicherbedarf zu erniedrigen, die Gruppensichten nach der oben unter 1. erwähnten Methode mittels OID_{Grp}–Mengen verwaltet. Da der Zugriff eines Kerns auf die im Objektserver liegenden Daten sowieso mehrere IPC–Nachrichten und mindestens einen Prozeßwechsel erfordert, fällt die vom Mengenschnitt benötigte Zeit weniger ins Gewicht. Dies um so mehr, als einmal angeforderte Daten von den Kernen ge'cache't werden, so daß der Schnittaufwand nicht bei jedem Zugriff anfällt.

In TOB_cache sieht man hingegen deutlich die Abspeicherung einer eigenen Beziehungsmengenstruktur ('Body' genannt) für jede Gruppe/Transaktion. An den 'Header' sind zwei Hashtabellen angehängt, in die nach Bedarf Zeiger auf Beziehungsmengenstrukturen für die verschiedenen Gruppen/Transaktionen eingetragen werden können. Die TOB_cache–Struktur wird, wie der Name schon andeutet, zur Datenspeicherung im Cache der Kerne verwendet. Da der Cache einen möglichst schnellen Zugriff gewähren soll, wird die schnellere, aber speicheraufwendigere Methode zur Unterscheidung der Beziehungsmengen gewählt. Hinzu kommt, daß im Gegensatz zum Objektserver in einem Kern meist nur eine begrenzte Zahl an Gruppen bekannt ist, so daß der Speicherbedarf begrenzt bleibt.

4.2.3 Überblick über die Feinarchitektur

Im folgenden wird die Architektur des Objektservers und des den Kernen zugeordneten Transaktionsmanagerteils anhand zweier Grafiken dargestellt und die Funktionsweise erläutert. Es dürfte klar sein, daß eine ausführliche Dokumentation den Rahmen dieser Ausarbeitung sprengen würde. Interessenten seien auf den entsprechenden Sourcecode verwiesen.

Der Abbildung 4.6 dargestellte Teil des Transaktionsmanagers befindet sich unter der Propositionsschnittstelle von *ConceptBase* (in Abbildung 2.3 also im Bereich A). Er kontrolliert dort den Zugriff auf die Propositionen und leitet Befehle des Benutzers an den Objektserver weiter.
Dem direkten Kontakt mit den ProLog–Routinen dienen die drei funktionalen Module Antwort, Prop und Commands. Das Erste realisiert die Auswertung von retrieve_prop–Anfragen mit verschiedenen Suchmustern, indem es die Ergebnis–OID–Mengen zurückgibt. Das zweite Modul enthält Funktionen, um Propositionen zu schreiben und zu löschen, während das dritte Funktionen für die Ausführung aller sonstigen Befehle (also z.B. Erzeugen einer Transaktion/Gruppe, Beenden einer Transaktion, Hochspielen von Änderungen in Gruppen, ...) enthält. Die Module erlangen anhand eines mitübergebenen Transaktionsidentifikators aus einer Hashtabelle den Zeiger auf das zugehörige Transaktionsobjekt (TA) und nutzen hauptsächlich die von diesem zur Verfügung gestellten Methoden.

Alle Speicherzugriffe von Instanzen der TA–Klasse laufen über den der jeweiligen Transaktion zugeordneten temporären Speicher TempMem. Angefragt werden kann ein TOB, das durch seine OID spezifiziert wird, oder die Menge aller OIDs von Propositionen mit einem bestimmten Label. Der temporäre Speicher prüft, ob er die gesuchte Information selbst hält, bzw. ob sie ihm als gelöscht bekannt ist. Andernfalls wird die Anfrage an den Cache weitergeleitet.

Der Cache ist im Gegensatz zu den temporären Speichern nur einmal pro Kern vorhanden. Dies erhöht zwar die Komplexität der Verwaltung, spart aber andererseits Speicher, da für eine Gruppe ge'cache'te Daten auch von anderen Gruppen genutzt werden können. Der Zugriff auf die Propositionen erfolgt über die CacheGroupData–Struktur, die für jede benötigte Gruppe existiert. Mit der in ihr enthaltenen Hashtabelle kann auf die im Cache enthaltenen Propositionen zugegriffen werden. Läßt sich eine gesuchte Proposition oder ein gesuchtes Label hier nicht finden und ist auch nicht als "im Objektserver nicht vorhanden" bekannt, so wird eine Anfrage an den Objektserver gestellt. Die geschieht mittels eines K_Message–Objekts, das zusammen mit einigen anderen die Benutzung der TLI–Funktionen (TLI=Transport Level Interface; vgl. [Sun93b]) kapselt.

Der Kern kann vom Objektserver zwei Arten von Nachrichten erhalten. Zum einen Antworten

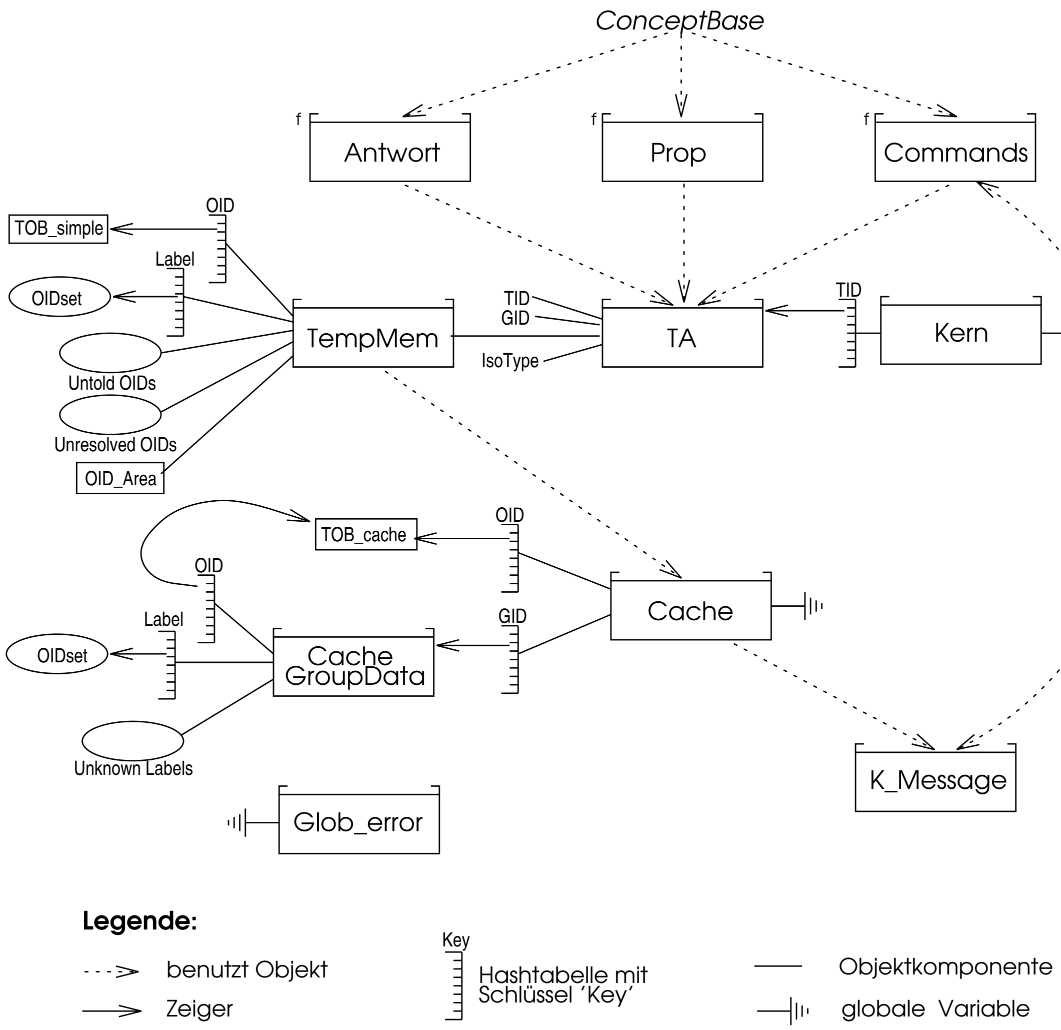

Abbildung 4.6: Architektur des dem ProLog-Kern zugeordneten Transaktionsmanagers

auf Anfragen des Caches, die dieser in seinen Speicher einordnet. Zum anderen Mitteilungen des Servers über Änderungen in einer der Gruppen, auf die vom Kern bisher zugegriffen wurde. Diese Änderungsmeldungen, die das Sichtbarwerden der von anderen Kernen durchgeführten Änderungen gemäß 2° Isolation bewirken, werden vom Modul Commands verarbeitet und in Änderungen im Cache und Mitteilungen an die betroffenen Transaktionen umgesetzt.

In Abbildung 4.7 wird der Aufbau des Objektservers aus C++–Objekten dargestellt. Im Gegensatz zum Kern laufen hier mehrere 'Thread's parallel ab. Der Hauptprozeßfluß befindet sich in der Mainloop des Server–Objekts. Bei Meldungen neuer Kerne am incoming–Port erzeugt er einen neues Client–Objekt mit zugehörigem 'Thread', das in Zukunft für alle Anfragen und Befehle des Kerns zuständig ist. Eingehende Nachrichten auf den im Portarray abgelegten Deskriptoren veranlassen den Server–'Thread' den zum Deskriptor gehörenden Client aufzuwecken, damit er auf die Nachricht reagieren kann.

Der in jedem Client–Objekt ablaufende 'Thread' kann über die im Server vorhandenen Tabellen auf die zur Bearbeitung eines Befehls benötigte Gruppe Group zugreifen. Die der Gruppe

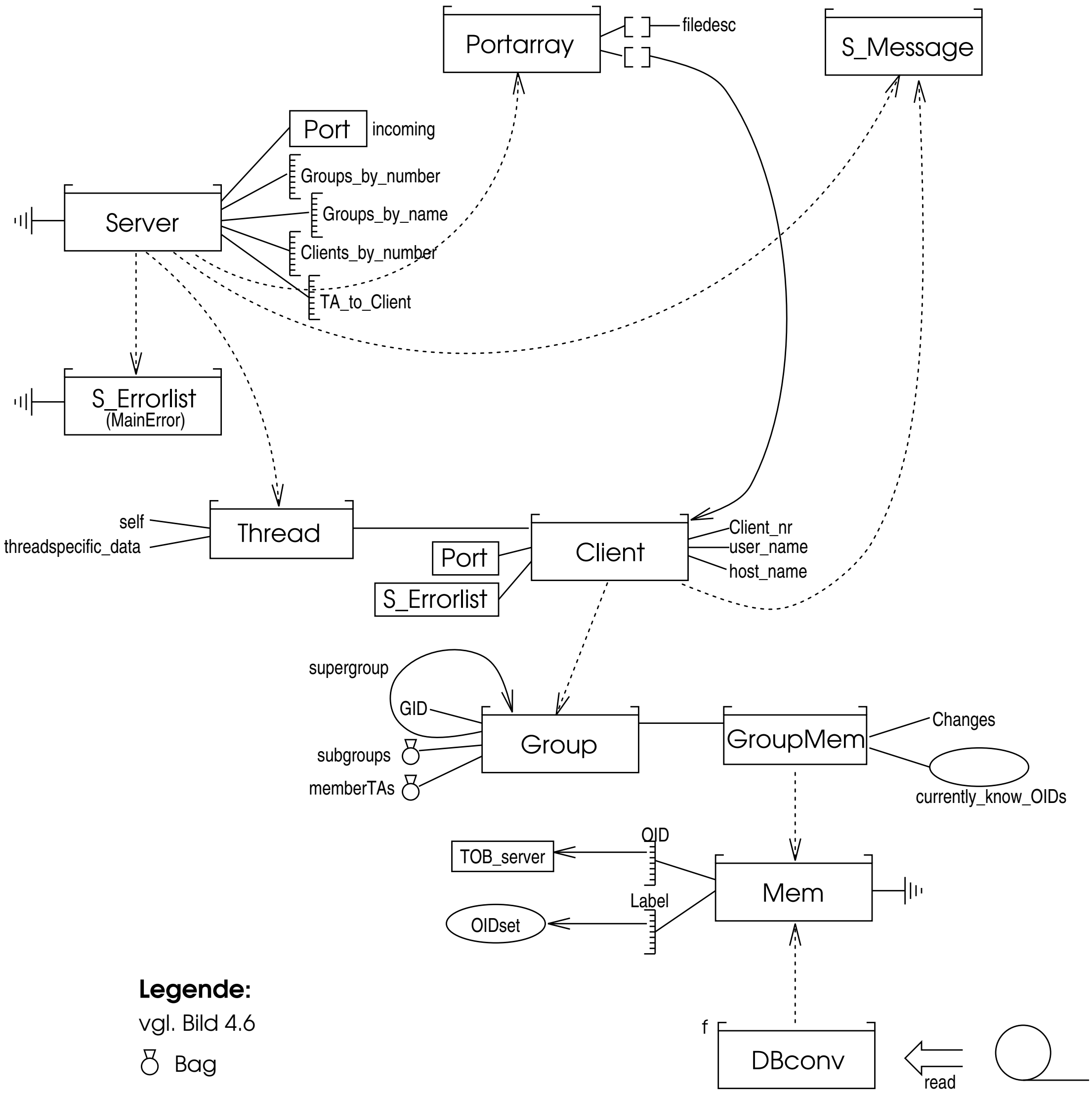

Abbildung 4.7: Architektur des Objektservers

bekannten Propositionen verwaltet GroupMem, das Leseanfragen unverändert an Mem weitergibt und das von dort gelieferte Ergebnis mit den in der Gruppe bekannten Propositionen (currently_known_OIDs) schneidet (vgl. S.4.2.2). Fügt ein Client neue Propositionen in eine Group ein, so muß er dies allen anderen Client–Objekten mitteilen, die einen Kern versorgen, der eine Transaktion in dieser Gruppe besitzt. Jedes Client–Objekt schickt daraufhin eine Nachricht mit den eingetretenen Änderungen an seinen zugeordneten Kern, damit dort der Cache angepaßt werden kann.
DBconv erlaubt es, den von einem unmodifizierten *ConceptBase*–Server abgespeicherten Propositionenbestand in den Hauptspeicher Mem aufzunehmen.

Im folgenden wird noch genauer auf einige Facetten der Implementation eingegangen, die nicht den Gewohnheiten entsprechen.

Index: Hashing oder Bäume?

In den verschiedenen Klassen des Transaktionsmanagers findet immer wieder der Zugriff auf bestimmte Propositionen statt. Um diese Zugriffe zu beschleunigen wird natürlich ein Index benötigt. In kommerziellen Datenbanken werden meistens Baumstrukturen als Indizes verwendet, der hier vorgestellte Transaktionsmanager benutzt Hash–Tabellen. Der Vorteil von Hash–Algorithmen liegt in der niedrigen Komplexität eines Zugriffs von O(1) wohingegen in Bäumen ein Zugriff die Komplexität O(log(n)) besitzt. Auch bei Einfüge-/Löschoperationen bleibt dieses Verhältnis im Normalfall bestehen. Lediglich bei einem Tabellenüberlauf benötigen die verwendeten GNU–Hashklassen O(n) Zeit, um die gesamte Tabelle zu kopieren. Da aber Überläufe selten sind und, wie in Tabelle 2.1 zu sehen, Lesezugriffe die Einfüge- und Löschoperationen bei weitem übersteigen, bleibt der Geschwindigkeitsvorteil der Hashmethode erhalten. Daß andere Systeme mit durchaus ähnlichem Änderungs-/Leseverhältnis Baumstrukturen benutzen, liegt in der Unfähigkeit der Hashverfahren begründet, die gespeicherten Daten gemäß der Sortierordnung zu durchlaufen. Während dies für kommerzielle Anwendungen essentiell ist, fehlen bei *ConceptBase* Anfragen der Art Liefere alle Einträge, deren Attribut X zwischen y und z liegt“. Dem Einsatz von Hashtabellen als Index steht also nichts entgegen.

Cache: unbegrenzt oder mit Ersetzungsalgorithmus?

Bei den den Kernen zugeordneten Caches war zu entscheiden, ob der ihnen zur Verfügung stehende Speicher begrenzt werden soll oder nicht. Jedes Verfahren hat seine spezifischen Nachteile:

begrenzter Speicher

- Es wird ein Ersetzungsalgorithmus benötigt, der bei vollem Cache entscheidet, welche Propositionen aus dem Cache entfernt werden sollen. Dies macht die Implementierung des Caches aufwendiger.
- Da bei einigen Propositionen zusätzliche Information (die Beziehungen zu den Propositionen im temporären Speicher) angehängt wird, müßte darauf geachtet werden, diese vom Ersetzungsalgorithmus auszunehmen.
- Die Abspeicherung im Cache erfolgt in Strukturen unterschiedlicher Länge und Anzahl. Neuerhaltene Information “paßt“ darum u.U. nicht in den freigewordenen Speicher. Eine Fragmentierung des Speichers wird so wahrscheinlich.

unbegrenzter Speicher

- Die Ersetzung nicht mehr benötigter Information wird durch das Betriebssystem übernommen (’Paging’), was keine Kontrolle über die verwendeten Algorithmen und Zustandsinformationen zuläßt.
- Informationszugriffe über Hashtabellen haben eine geringe Lokalität und verringern dadurch die Effektivität des Betriebssystem-’Paging’s.

Insbesondere aus Gründen des Aufwandes, wurde die Variante mit unbegrenztem Cache gewählt. Ein Effekt dieser Entscheidung sollte sein, daß die Zugriffsgeschwindigkeit nach Füllung des Caches mit den benötigten Daten ähnlich zu der des Objektspeichers von Hubertus Pützer sein sollte, der ebenfalls alle Daten im Hauptspeicher hält.

Synchronisation

Da im Objektserver mehrere 'Threads' gleichzeitig ablaufen, müssen ihre Zugriffe auf gemeinsame Daten synchronisiert werden. Gemeinsam benutzte Daten sind zum einen die verschiedenen Hashtabellen, zum anderen die eigentlichen Daten in Form der TOB_server- und OIDset–Objekte. In beiden Fällen kann die Synchronisation sowohl mittels wechselseitigen Ausschlusses ('mutual exclusion' oder kurz 'mutex') als auch mit Schreib–Lese–Sperren durchgeführt werden. Wechselseitiger Auschluß benötigt nur einen geringen Aufwand, erlaubt aber auch nur einem einzelnen 'Thread' Zugriff auf die geschützten Daten. Die Synchronisierung als Schreiber–Leser–Problem erhöht neben dem Verwaltungsaufwand auch die mögliche Parallelität, da nun mehrere Klienten gleichzeitig lesend zugreifen können. Aus diesen Gründen werden häufig benötigte Daten wie z.B. Indizes durch Schreib–Lese-Sperren geschützt, während die eher selten zugegriffenen aber häufig vorhandenen Daten (z.B. die TOB_server–Objekte) durch wechselseitigen Ausschluß gesichert werden.

Nicht nur Schreib- und Leseoperationen müssen synchronisiert werden, sondern auch die Aktionen innerhalb einer Gruppe während der Validierungs- und Schreibphase einer Transaktion. Daß während der Validierung einer Transaktion eine stabile Sicht benötigt wird und sich darum keine andere Transaktion in ihrer Schreibphase befinden darf, ist klar. Aber auch das gleichzeitige Validieren zweier Transaktionen muß verhindert werden, da sonst ihre Änderungen nicht auf gegenseitige Integrität überprüft werden.
Während der Schreibphase einer Transaktion kann die Gruppe vorübergehend einen inkonsistenten Zustand annehmen. Ein solcher Zustand sollte für die Transaktionen nicht sichtbar werden. Aus diesem Grund werden für die Dauer der Schreibphase jegliche Zugriffe auf die Daten der Gruppe untersagt.

4.2.4 Schnittstelle zu *ConceptBase*

Der Kontakt zwischen dem Transaktionsmanager und *ConceptBase* findet über zwei Schnittstellen statt. Das Lesen und Schreiben von Propositionen geschieht über von Hubertus Pützer realisierten C++–Funktionen, die das auf ProLog–Seite durchgeführte Backtracking mit Hilfe eines Stacks unterstützen. Während nämlich der Transaktionsmanager alle auf ein Suchmuster passenden Propositionen auf einmal ermittelt, erwartet der *ConceptBase*–Kern die Übergabe einer einzelnen Proposition pro Aufruf. Alle Befehle des Benutzers, die lediglich für den Transaktionsmanager von Belang sind (z.B. Gruppenverwaltung), werden vom CB–Interface am Kern vorbei direkt dem Transaktionsmanager zugeleitet.

Einige Besonderheiten der Schnittstelle bezüglich der Atomarität und den OIDs sind noch zu beachten. So stellt der Transaktionsmanager lediglich die Atomarität eines einzelnen Propositionszugriffs sicher. Zwischen zwei Zugriffen kann sich also der Zustand der Datenbank (dem Isolationsgrad 2 entsprechend) geändert haben.
Bei den OIDs der Propositionen hat sich nicht nur die Struktur geändert – der Kern sieht nun

einen 10-stelligen String. Die OID einer Proposition wird nun erst beim Hochfahren des Servers festgelegt und eine Konstanz zwischen verschiedenen Serverläufen wird nicht garantiert. Dies hängt mit der Struktur der OIDs zusammen, die für jede Transaktion die Eindeutigkeit der erzeugten OIDs sicherstellt. Damit aber der für die Unterscheidung der Transaktionen vorgesehene Zahlenbereich nicht erschöpft, werden beim Starten des Objektservers alle Propositionen mit OIDs aus dem Bereich 1, der keiner Transaktion zugeordnet ist, erzeugt.

4.3 Ergebnisse der Implementierung

4.3.1 Probleme

Im Rahmen der Implementierung des Transaktionskonzeptes ergaben sich einige Probleme aus der Struktur des *ConceptBase*–Servers, die insbesondere den Sinn eines Integrationversuches zweifelhaft werden ließen.

Bei der Besprechung der Architektur von *ConceptBase* (s. S.11f) wurde zwar dargestellt, daß aus den Regeln und 'Constraints' abgeleitete Information außerhalb der Propositionendatenbasis abgelegt werden. Der Aufbau unterhalb des Moduls AssertionCompiler wurde allerdings wegen seiner Komplexität ignoriert. Die dort stattfindenden Operationen stören einen Transaktionsmanager jedoch erheblich.
Durch die Module LTAssertionCompiler, LTAssertionTranslator und LTcodeGenerator werden nämlich die in Objekten enthaltenen Regeln, soweit sie keine 'MagicSet'–Regeln sind, in ausführbaren ProLog–Code umgewandelt und durch das Module PROLOGruleProcessor schließlich mittels assert() in das *ConceptBase* implementierende ProLog-Programm eingefügt.

Diese dynamischen Änderungen am Programmcode beenden nicht nur jede Reentranz (vgl. Kap.4.2.1). Der so außerhalb der Propositionendatenbasis und damit außerhalb der Reichweite des Transaktionsmanagers abgespeicherte Code wird auch beständig zur Auswertung von Anfragen und Regeln benutzt. Aber auch bei einer Ausdehnung des Transaktionsmanagers ließe sich der Zugriff nicht beschränken, da der in das ProLog–Programm eingefügte Code direkt durch den ProLog–Interpreter ausgewertet wird. Ein unbeschränkter Zugriff führt aber dazu, daß für die Transaktionen aller Regelcode sichtbar ist, auch der Code solcher Regeln, deren Propositionsdarstellung für sie unsichtbar ist. Die Transaktionen werten also Anfragen mit Hilfe von Regeln aus, die sie gar nicht kennen dürften.

Dieses Problem stellt sich unabhängig vom verwendeten Transaktionskonzept sobald eine Transaktion A mehr Regeln "sieht" als eine andere. Um die korrekte Ausführung von A sicherzustellen, muß nämlich der Code für die entsprechenden Regeln eingefügt werden und ist damit für alle sichtbar. Unterschiede in der Sichtbarkeit treten aber beim Einfügen neuer Daten in allen Konzepten auf.

Zur Lösung sind drei Ansätze möglich:

- Damit die in den ProLog–Kernen abgespeicherte Information jeweils nur für die richtige Transaktion sichtbar ist, erhält nicht jeder Benutzer sondern jede Transaktion einen eigenen Kern zugeordnet.

  Diese Vorgehensweise hat allerdings einige Nachteile. Zum einen verbraucht das Erzeugen und Löschen eines kompletten Kerns am Anfang und Ende jeder Transaktion ein großes Maß Rechenzeit und Speicher. Zum anderen löst es einige der weiter unten stehenden Probleme (wie z.B. das Abspeichern des Regelcodes in einer eigenen Datei) nicht, sondern verschärft sie noch.

- Vorstellbar ist auch, in den ProLog–Code der Regeln Konjunktionen einzufügen, die die Ausführung des Codes nur während der Auswertung von Anfragen bestimmter Transaktionen zulassen. In ProLog sähe dies etwa folgendermaßen aus:

  Beispiel 4.2 *Die Regel*

  `Kopf(Vars) :- Rumpf(Vars).`

  wird durch

  `Kopf(Vars) :- Transaktion(142), Rumpf(Vars).`

  ersetzt wobei das globale Fakt Transaktion/1 *stets die Nummer der gerade ausgewerteten Transaktion enthält. Obiger Code dürfte also nur zur Beantwortung von Anfragen der Transaktion 142 herangezogen werden.*

  Komplizierter wird dieses Verfahren allerdings, sobald der Code mehr als einer – aber nicht allen – Transaktionen zur Verfügung stehen soll. Um das mehrfache Einfügen des Codes zu vermeiden, müßten entweder statt "Transaktion(142)" kompliziertere Ausdrücke eingfügt werden oder es müßte auf noch einzuführende "Transaktionsgruppen" getestet werden. Alle Transaktionen innerhalb einer Transaktionsgruppe zeichneten sich dann dadruch aus, daß für sie die selben Regeln sichtbar sind. Da bereits das Einfügen einer neuen Regel in eine Klasse die Regelsicht einer Transaktion verändert, stimmten Transaktionsgruppen nicht mit den Gruppen des Transaktionskonzeptes überein.

  Wie man sieht, kann die Komplexität der durchzuführenden Umwandlung erheblich sein. Außerdem verlangsamen die zusätzlich eingefügten Audrücke die Regelauswertung. Da zudem auch dieser Ansatz einige der folgenden Probleme ungelöst läßt und Änderungen in den obskureren Regionen von *ConceptBase* erfordert, erscheint auch dieser Ansatz als zumindest sehr aufwendig zu implementieren.

- Als letzte Möglichkeit bietet sich an, von *ConceptBase* grundsätzlich das Abspeichern aller Informationen in der Propositionsdatenbasis (oder einer noch zu erstellenden parallelen Regeldatenbasis) zu verlangen. Vor der Auswertung müßte der Code dann jeweils beim Transaktionsmanager angefordert werden, was die volle Kontrolle durch letzteren gewährleisten würde.

  Die Auslagerung des Codes verhindert aber die Fortführung der bisherigen Strategie, die Auswertung aus Geschwindigkeitsgründen dem ProLog–Interpreter zu überlassen. Da der Code nicht mehr in das ProLog–Programm eingefügt werden darf, ist er für den Interpreter nicht mehr sichtbar. Die Regelauswertung müßte also von einem Meta–Auswerter übernommen werden, was die Leistung deutlich verringern würde.

Alle aufgeführten Lösungsansätze erhöhen also den Rechenzeitbedarf des Kerns und erforden – bis auf den ersten – beträchtlichen Implementierungsaufwand. Von der Einarbeitung eines Ansatzes im Rahmen der Diplomarbeit wurde deshalb abgesehen.

Ein weiteres Problem stellen die von *ConceptBase* zur Beschleunigung von Konsistenztest und Regelauswertung benutzten Trigger dar. Die Trigger lösen bei einer Änderung der Daten die anhängenden Regeln und Konsistenzchecks aus. *ConceptBase* bemerkt allerdings Änderungen nur, wenn sie verschiedenen ProLog–Module durchlaufen, wenn sie sozusagen die Kernarchitektur „von oben“ erreichen. Beim vorgestellten Konzept werden Änderungen in den Gruppen allerdings zwischen den einzelnen auf Propositionsebene arbeitenden Transaktionsmanager ausgetauscht, kommen also „von unten“. Um die Triggerauslösung zu gewährleisten müßte also der an den Kern gekoppelte Teil des Transaktionsmanagers diesem ankommende Änderungen mitteilen. Aufgrund dieser Mitteilungen könnten noch zu schreibende Routinen im Kern überprüfen, welche Trigger beeinflußt wurden.

Die Regelübersetzungen ließen sich theoretisch bei jeden Hochfahren des Servers aus den in der Propositionendatenbasis vorhandenen Informationen neu erzeugen. Um den hierfür notwendigen Aufwand zu sparen, speichert die derzeitige *ConceptBase*–Implementation den Regelcode ab. Allerdings geschieht dies völlig getrennt von den Propositionen in einer “.rule“–Datei. Schwierigkeiten ergeben dadurch, daß es im vorgestellten Konzept mehrere Kerne gibt, die jeweils eine “.rule“–Datei lesen und schreiben. Selbst wenn das gegenseitige Überschreiben verhindert werden kann, ist dieses Vorgehen weder speicheffizient, noch existiert eine primäre, bei Widersprüchen ausschlaggebende Version der Datei. Die einfachste Abhilfe für dieses Problem stellt die oben erwähnte Abspeicherung des ProLog–Codes in der Propositionenbasis dar.

Selbst bei Lösung aller dieser Probleme wäre die Integration noch durch einen bei den Integrationsversuchen des Objektspeichers von Hubertus Pützer sichtbar gewordenen Fehlers in BIM–Prolog behindert worden. Mit Abhilfe wird erst in der nächsten Version gerechnet.

Zusätzliche Schwierigkeiten bereitet die Einbeziehung der bisher in *ConceptBase* vorhandenen historischen Datenbank, da der Objektspeicher von Hubertus Pützer hierfür bisher keine Unterstützung bietet. Zwar ließe sich die von Thomas List gewählte Implementierung der historischen Datenbank durch OID–Mengen übernehmen. Aber auch sie enthält keine Maßnahmen zur Auslagerung historischer Information auf langsamere Speichermedien.

4.3.2 Nichtimplementierte Konzeptteile

Die bisherige Implementation des Konzepts beinhaltet keineswegs alle im Entwurfsteil vorgestellten Teile. Die schwerwiegenste Lücke entsteht wohl dadurch, daß die Verwaltungsroutinen für die Gruppen nicht vorhanden sind. Wie bereits in der Beschreibung der Feinarchitektur durchschimmerte, funktionieren die Gruppen im Rahmen der Propositionszugriffe bereits vollständig, aber alle die Erzeugung, Löschung oder sonstige Verwaltung von Gruppen betreffenden Methoden existieren noch nicht. Daher gibt es bisher nur eine einzige Gruppe in der sich alle Daten und Transaktionen befinden. Ohne Gruppen kann natürlich auch die geplante Pflicht, vor einem Schreibzugriff erst eine Berechtigung von der Obergruppe anfordern zu müssen, nicht sinnvoll implementiert werden.

Auf die Implementierung der Erkennung von W/W–Konflikte wurde aus in Kapitel 4.1.3 genannten Gründen verzichtet. Die zur Unterstützung der Konfliktbeseitigung wünschenswerte ’Merge’–Komponente wäre selbst, falls die Erkennung von W/W–Konfliken integriert worden wäre, aus Zeitgründen wahrscheinlich unimplementiert geblieben.

Die Persistenz wird im Objektserver durch das Speichern aller bekannten Propositionen in einer Datei erreicht. ’Log’–Dateien oder ähnliche Maßnahmen zur regelmäßigen Abspeicherung von Änderungen existieren nicht. Zwar war die Bereitstellung einer ’Recovery’–Komponente nicht Aufgabe dieser Diplomarbeit, die zur Verfügung gestellte Persistenz stellt aber einen deutlichen Rückschritt gegenüber dem derzeitigen Leistungsstand von *ConceptBase* dar.

4.3.3 Messungen

Um die Leistung des Transaktionsmanagers mit der des Objektspeicher zu vergleichen, wurden die in Tabelle 2.2 aufgeführten Tests mit ihm wiederholt. Die in der folgenden Tabelle aufgeführten Zeiten wurden durch zehntausendfache Ausführung derselben Zugriffe gewonnen. Hiervon wurden jeweils 9999 Zugriffe aus dem 'Cache' befriedigt, so daß die Übertragungszeit zwischen Objektserver und Kern kaum ins Gewicht fällt. Die Übertragungszeit selbst ließ sich nicht feststellen, die Messung der für das Füllen des 'Caches' benötigten Zeit nicht durch mehrmaliges Wiederholen der gleichen Operationen ermittelt werden kann. Eine einmalige Ausführung benötigt hingegen so wenig Zeit, daß die zur Verfügung stehenden Meßmethoden zu ungenau sind. Eine höhere Genauigkeit ließe sich durch sogenanntes 'Profiling' erreichen, der verwendete C++–Compiler ist aber noch nicht fähig, die dazu benötigte Information zu erzeugen.

Suchmuster	Zeit [ms]			Verhältnis
	Pützer	OID	Namen	Spalte 3 : Spalte 2
1	0.275	0.634	0.657	2.3
2	0.037	0.221	0.242	6.0
3	0.343	0.416	0.534	1.2
4	0.150	0.370	0.462	2.5
5	0.755	0.927	1.198	1.2
8	0.128	0.017	1.623	0.13

Tabelle 4.1: Zeiten für einen Zugriff bei Objektspeicher und TA–Manager

Die Bedeutung der Suchmusternummern wurde im Zusammenhang mit Tabelle 2.2 erklärt, die Daten der zweiten Spalte aus ihr übernommen. Die Zeiten der dritten und vierten Spalte wurden durch Messungen an der Implementierung des Transaktionsmanagers gewonnen. Die beiden Spalten unterscheiden sich durch die Vorverarbeitung des Anfragedaten. Die von Herrn Pützer übernommenen Suchmuster enthalten bereits OIDs, was im vorgestellten Konzept aufgrund der Veränderlichkeit derselben zwischen verschiedenen Objektserverausführungen gefährlich ist. Für die Messungen von Spalte vier wurde darum eine zusätzliche Stufe eingeschaltet, die den 'Label' einer Proposition in die zugehörige OID umwandelt (entspricht ungefähr einer zusätzlichen Suche mit Suchmuster 2). Die Zeiten in Spalte 3 enthalten diese Stufe nicht und lassen sich darum besser mit den beim Objektspeicher ermittelten Werten vergleichen.

Insgesamt überrascht, daß das die Implementierung des Konzepts teilweise erheblich langsamer als der Objektspeicher ist, obwohl beide ähnliche Datenstrukturen und Algorithmen verwenden. Erklären läßt sich dies zum einen durch den für Gruppen– und Cacheverwaltung zusätzlich notwendigen Aufwand. Zum anderen wurde bereits in Kapitel 4.2.2 erwähnt, daß in den von Herrn Pützer übernommenen Klassen Auswertungsroutinen die Verwendung von Zeigern vereinheitlicht wurde. Dies hat möglicherweise zu einem verstärkten Kopieren von Objekten geführt und damit den Zeitbedarf erhöht.

Für die Richtigkeit dieser These sprechen die für Suchmuster 8 gemessenen Werte. Der Eintrag in Spalte drei ist keineswegs ein Druckfehler, sondern zeigt den Leistungsverlust durch Kopieren. Suchmuster 8 entspricht dem Suchen einer Proposition mittels ihrer OID. Da eine solche Anfrage maximal ein Ergebnis haben kann, liefert die Suchroutine des Transaktionsmanagers eine Referenz auf diese Proposition. Der Objektspeicher baut hingegen noch eine Struktur auf, die das Backtracking unterstützt. Allein die zur Alloziierung und Initialisiierung notwendigen Operationen verlangsamen die Suche um das siebenfache.

Genauere Untersuchungen darüber, wo am meisten Zeit verloren geht, ließen sich nicht vornehmen, da das hierfür notwendige 'Profiling' nicht durchführbar war.

4.3.4 Kontrolle der Anforderungserfüllung

Auf Seite 14 wurden Anforderungen an das zu entwickelnde Transaktionskonzept aufgestellt, deren Erfüllung nun überprüft werden soll. Hierbei wird zwischen dem Konzept und dem derzeitigen Stand der Implementierung unterschieden.

Von den **ACID**–Eigenschaften werden die ersten drei wie verlangt durch das Konzept sichergestellt (siehe Kapitel 4.1.7), wenn auch teilweise auf etwas ungewöhnliche Weise mittels Integritätstester und verschiedenen Isolationsgraden. Die Implementation muß leichte Abstriche im Bereich Isolation hinehmen, da durch die nicht vorhandenen Gruppen die 3° Isolation nicht zur Verfügung steht.

Durch den Verzicht auf Serialisierbarkeit umgeht das Konzept viele Probleme, die im Zusammenhang mit **langen Transaktionen** auftreten können, so daß ihrerBenutzung nichts im Wege steht. Lediglich die ohne Maßnahmen wie frei setzbare Sicherungspunkte mangelhafte Unterstützung eines teilweisen Rücksetzens benötigt noch einige Verbesserungen. Unterschiede zwischen Konzept und Implentation bestehen hier keine.

Die **Kooperation** zwischen den Benutzern wird durch die verschiedenen zur Wahl stehenden Isolationsgrade ermöglicht. Durch die zusätzliche Integration eines der Transaktionsrestrukturierung (vgl. Kapitel 3.2.2) ähnlichen Konzepts ließe die Kooperation sich allerdings noch erleichtern. Der in der Implementation fehlende Isolationsgrad 3 beeinträchtigt die Kooperation nur in geringem Maße, da für sie eher die niedrigeren Isolationsgrade benötigt werden.

Die Vermeidung der **Beschränkung auf bestimmte Anwendungsgebiete** konnte nicht vollständig erfüllt werden. Selbst wenn man von den von *ConceptBase* vorgegebenen Einschränkungen (keine Rechenoperationen, ...) absieht, erfordert dieses Konzept eine wahrscheinlich nicht immer erfüllbare Eigenschaft: Alle im realen Problem vorhandenen Konsistenzbedingungen müssen sich als *ConceptBase*–'Constraints' ausdrücken lassen.

Das **Anwortverhalten bei mehreren gleichzeitigen Benutzern** konnte durch die in der Implementation verwendete Architektur aus mehreren Kernen und einem zentralen Objektserver mit mehreren 'Threads' deutlich verbessert werden. Die sich ergebenden **Leistungseinbußen** des Konzepts werden zum einen durch die daher notwendige Übertragung von Informationen zwischen Objektserver und Kernen bedingt. Zum anderen hat anscheinend auch der Übergang auf eine homogenere Verwendung von Zeigern Gerschwindigkeitseinbußen verursacht.

Im Proposal wurden verschiedene Vorschläge gemacht, wie die **Kompatibilität zu alten Applikationen** erhalten werden kann. Diese wurden nicht weiter verfolgt, da die Anpassung alter Applikationen nur geringe Schwierigkeiten bereitet. Wird für sie ein Gruppe mit 2° Isolation erzeugt (in der derzeitigen Implementation nur begrenzt möglich) und werden die einzelnen Befehle und Anfragen der Anwendungen an *ConceptBase* jeweils mit einem "Transaktionsanfang"–"Transaktionsende"–Paar eingeklammert, sollten sie weiterhin funktionieren. Die notwendigen Änderungen am Quelltext können dabei gleich zu Überlegungen genutzt werden, wie man sich die Transaktionsf"higkeiten von *ConceptBase* besser zunutze machen kann.

Das vorgestellte Konzept erfüllt also die meisten der gestellten Anforderungen voll und die anderen zumindest teilweise. Allerdings erfüllt die Implementation eine im zweiten Kapitel nicht gestellte aber implizit vorausgesetzte Anforderung nicht: Sie läßt sich nicht in die derzeitige Version von *ConceptBase* integrieren.

Kapitel 5

Zusammenfassung

Das Ziel dieser Diplomarbeit war die Entwicklung eines Transaktionskonzepts für die Objektbank *ConceptBase*. Das Studium klassischer Konzepte ergab wenig überaschend, daß sie für eine Datenbank zur Entwurfsunterstützung wie *ConceptBase* wenig brauchbar sind. Insbesondere die hohe Isolation der Benutzer widerspricht der Notwendigkeit zu Kommunikation und Kooperation zwischen den Entwicklern. Als Nebeneffekt der Untersuchungen wurde festgestellt, daß Verfahren, die Lesezugriffe über einen längeren Zeitraum verweigern, für deduktive Datenbanken ungeeignet sind, und es wurde versucht aufzuzeigen, welche Probleme die Verbindung deduktiver Datenbanken mit Transaktionskonzepten bisher behindert haben.

Aber auch sogenannte fortgeschrittene Verfahren, die zur Verwendung in ähnlichen Anwendungsbereichen wie *ConceptBase* entworfen wurden, erwiesen sich als nicht anwendbar. Verursacht wurde dies durch komplizierte Vorausleistungen der Benutzer (Kompensationsfunktionen, Spezifikation von LR(0)–Grammatiken), durch mangelnde allgemeine Verwendbarkeit (NSE) sowie durch das Basieren auf objektorientierten Eigenschaften (Kommutativität von Objektmethoden).

Aus diesem Grund wurde von den klassischen ACID–Eigenschaften und den Vorteilen einiger fortgeschrittener Konzepte ausgehend ein neues optimistisches Konzept entwickelt, das bei Konsistenzsicherung und Isolation ungewöhnliche Wege geht. Der Wahl des Isolationsgrad zwischen den Transaktionen bleibt dem Anwendungsentwickler überlassen, wenn er in seiner Freiheit auch etwas durch die notwendige Aufteilung der Transaktionen in Gruppen eingeschränkt wird. Der in *ConceptBase* vorhandene Integritätstester übernimmt alleinverantwortlich die Aufrechterhaltung der Konsistenz, wobei ihm Benutzer ihre Korrektheitsvorstellungen durch die ebenfalls schon vorhandenen 'Constraints' mitteilen können. So von der Pflicht zur Sicherstellung der Serialisierbarkeit entbunden, konnte ein optimistisches Konzept verwirklicht werden, das im Gegensatz zu vielen anderen optimistischen Verfahren möglicherweise praktisch anwendbar ist.

Dem so erhaltenen Konzept fehlt die Eleganz eines einheitlichen Ansatzes oder einer zugrundeliegenden theoretischen Abhandlung, wie sie die meisten Konzepte in der Literatur inzwischen besitzen. Es erfüllt jedoch die gestellten Anforderungen und läßt sich auf einem vorhandenen Betriebsystem realisieren. Durch die in der Implementation teilweise erreichte Fähigkeit, verschiedene Benutzer von mehreren 'Threads' gleichzeitig bedienen zu lassen, kann die Mehrprozessorarchitektur der neueren Sun–Computer ausgenutzt werden.

Eine erfolgreiche Integration in *ConceptBase* wurde durch konzeptunabhängige Schwierigkeiten verhindert, so daß sich auch dieses Konzept in die Reihe der interessanten aber niemals angewendeten Transaktionsverfahren eingliedert. Die Integrationsschwierigkeiten führen allerdings zu dem Schluß, daß jedwedes Transaktionskonzept für *ConceptBase* auf nur schwer zu überwindende Probleme treffen wird.

Literaturverzeichnis

[ACL85] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Models for studying concurrency control performance: Alternatives and implications. In *Proceedings of the ACM SIGMOD International Conference on the Managment of Data*, pages 108–119. ACM Press, 1985.

[AE92] D. Agrawal and A. El Abbadi. Transaction management in database systems. In Elmagarmid [Elm92], chapter 1.

[And91] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison–Wesley, Reading, Mass., 1986.
Das sogenannte 'Dragonbook'.

[BBH+89] Michael L. Brodie, Francois Bancilhon, Craig Harris, Michael Kifer, Yoshifumi Masunaga, Earl D. Sacerdoti, and Katsumi Tanaka. Next generation database managment systems technology. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *Deductive and Object–Oriented Databases; Procceedings of the First International Conference on Deductive and Object–Oriented Databases*, pages 335–345. North–Holland, 1989.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Mass., 1987.

[BK91] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[BKK85] Francois Bancilhon, Won Kim, and Henry F. Korth. A model of cad transactions. In *Proceedings of the International Conference on Very Large Data Bases*, pages 25–33, Stockholm, August 1985. Morgan Kaufmann.

[BSW88] C. Beeri, H.-J. Schek, and G. Weikum. Multi–level transaction managment: Theoretical art or practical need? In *International Conference on Extending Database Technology*, pages 134–154, 1988.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer–Verlag, Berlin Heidelberg, 1990.

[CHM92] Vinay K. Chaudhri, Vassos Hadzilacos, and John Mylopoulos. Concurrency control for knowledge bases. In *Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, Mass., 1992. Morgan Kaufmann Publishers.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Comunications of the ACM*, 13(6), June 1970.

[CR92] Panos K. Chrysanthis and Krithi Ramamritham. ACTA: The SAGA continues. In Elmagarmid [Elm92], chapter 10.

[Dat90] C. J. Date. *An Introduction to Database Systems*, volume I. Addison–Wesley, Reading, Massachusetts, fünfte edition, 1990.

[EG89] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD International Conference on the Managment of Data*, pages 399–407. ACM Press, 1989.

[Ehe93] Stefan Eherer. *Eine Software–Umgebung für die kooperative Erstellung von Hypertexten.* Dissertation, RWTH Aachen, Dezember 1993.

[Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, San Mateo, California, 1992.

[ELMB92] Ahmed K. Elmagarmid, Yungho Leu, James G. Mullen, and Omran Bukhres. Introduction to advanced transaction models. In Elmagarmid [Elm92], chapter 2.

[FO89] A. A. Farrag and M. T. Ozso. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, 1989.

[Gal90] Rainer Gallersdörfer. Realisierung einer deduktiven Objektbank durch abstrakte Datentypen. Diplomarbeit, Universität Passau, Dezember 1990.

[Geb92] Michael Gebhardt. Transaktionen in objektorientierten Datenbanken. In *Seminar Transaktionskonzepte*. Lehrstuhl für Informatik V, RWTH Aachen, SS 1992.

[GM78] H. Gallaire and J. Minker. *Logic and Databases.* Plenum Press, New York, 1978.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on the Managment of Data.* ACM Press, May 1987.

[Gos91] A. Goscinski. *Distributed Operating Systems: The Logical Design.* Addison–Wesley, 1991.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts & Techniques.* Morgan Kaufmann, San Mateo,Cal., 1993.

[Gra81] J. N. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the International Conference on Very Large Data Bases*, pages 144–154. Morgan Kaufmann, 1981.

[Her90] Maurice Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, March 1990.

[HR83] T. Härder and A. Reuter. Principles of transaction–oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[Jar91] Matthias Jarke. Skript zur Vorlesung „Anfrageauswertung in Datenbanken". Lehrstuhl Informatik V, RWTH–Aachen, WS 1991.

[Jar92] Matthias Jarke, Ed. ConceptBase V3.1 User Manual. Aachener Informatik Berichte 92-17, Fachgruppe Informatik, RWTH Aachen, 1992.

[Jar93] Matthias Jarke, editor. *Database application engineering with DAIDA*. Springer–Verlag, 1993.

[JEG+93] Matthias Jarke, Stefan Eherer, Rainer Gallersdörfer, Manfred A. Jeusfeld, and Martin Staudt. ConceptBase – a deductive object base manager. Aachener Informatik–Berichte 93-13, Fachgruppe Informatik, RWTH Aachen, 52056 Aachen, Deutschland, October 1993.

[Jeu92] Manfred A. Jeusfeld. *Änderungskontrolle in deduktiven Objektbanken*. Infix–Verlag, St. Augustin, Deutschland, 1992.

[JS93] Matthias Jarke and Martin Staudt. An application perspective to deductive object bases. In *Proceedings of the Workshop on Combining Declarative and Object–Oriebted Databases*. ACM Press, May 1993.

[Kem91] Alfons Kemper. Skript zur Vorlesung „Implementierung von Datenbanksystemen". Lehrstuhl Informatik III, RWTH–Aachen, WS 1991.

[KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the International Conference on Very Large Data Bases*, pages 95–106, Brisbane, Australia, 1990. Morgan Kaufmann.

[KM92] Alfons Kemper and Guido Moerkotte. *Object–Oriented Information Management for Engineering Applications*. Prentice–Hall, 1992.

[KMSJ88] M. Koubarakis, J. Mylopoulos, M. Stanley, and M. Jarke. Telos: A knowledge representation language for requirements modeling". Technical report, University of Toronto, Canada, 1988.

[KP92] Gail E. Kaiser and Calton Pu. Dynamic restructuring of transactions. In Elmagarmid [Elm92], chapter 8.

[KS88] H. F. Korth and G. Speegle. Formal model of correctness without serializability. In *Proceedings of the ACM SIGMOD International Conference on the Managment of Data*, pages 379–388, Chicago, June 1988.

[LS87] P. C. Lockmann and J. W. Schmidt, editors. *Datenbank–Handbuch*. Springer, Berlin, Hamburg, 1987.

[MBJK90] John Mylopoulos, Alex Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing knowledge about information systems. *ACM Transaction on Information Systems*, 8(4):325–362, October 1990.

[Mey88] Bertrand Meyer, editor. *Object–Oriented Software Construction.* Prentice–Hall, 1988.

[ML91] Guido Moerkotte and Peter C. Lockemann. Reactive consistency control in deductive databases. *ACM Transactions on Database Systems*, 16(4):670–702, December 1991.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Cambridge, Mass., 1985.

[NRZ92] Marian H. Nodine, Sridhar Ramaswany, and Stanley B. Zdonik. A cooperative transaction model for design databases. In Elmagarmid [Elm92], chapter 3.

[O'N86] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, 1986.

[Pap86] Christos Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[Püt92] Hubertus Pützer. Realisierung eines Objektspeichers für die Wissensrepräsentationssprache Telos. Diplomarbeitsproposal, 1992.

[Püt93] Hubertus Pützer. Entwurf und Realisierung eines Objektspeichers für die Wissensrepräsentationssprache Telos. Diplomarbeit, RWTH–Aachen, 1993.

[RD91] Nick Roussopoulos and Alexis Delis. Modern Client-Server DBMS Architectures. *SIGMOD Record*, 20(3):52–61, September 1991.

[Ree78] D. Reed. Naming and synchronization in a decentralized computer system. Ph.D. thesis; Tech report 205, MIT Laboratory of Computer Science, 1978.

[RJM92] T. Rose, M. Jarke, and J. Mylopoulos. Organizing software repositories — modeling requirements and implementation experiences. In *Proc. 16th Intl. Computer Software & Applications Conference*, Chicago, September 1992.

[RL93] B. Ramesh and Luqi. Process knowledge based rapid prototyping for requirements engineering. In *Proceedings of IEEE Intl. Symposium on Requirements Engineering*, San Diego, Ca., January 1993.

[RMB+93] Rajeev Rastogi, Sharad Mehrotra, Yuri Breitbart, Henry F. Korth, and Avi Silberschatz. On correctness of non–serializable executions. In *Proceedings of the Principles of Database Systems Conference*, pages 97–108. ACM Press, 1993.

[Rös91] Karl Rösch. Entwurf eines Kooperationsmodells und eines darauf basierenden Korrektheitskriteriums für kooperative Transaktionssysteme im Datenbankbereich. Diplomarbeit, Universität Karlsruhe; Lehrstuhl P. C. Lockemann, September 1991.

[SK80] Abraham Silberschatz and Zvi Kedem. Consistency in hierarchical database systems. *Journal of the ACM*, 27(1):72–80, 1980.

[Ska89] Andrea H. Skarra. Concurrency control for cooperating transactions in an object-oriented database. *SIGPLAN Notices*, 24(4), April 1989.

[Sun91] Sun Microsystems, Inc., Mountain View, CA, USA. *Network Software Environment 1.3: Overview and Tutorial*, 1991.

[Sun93a] SunSoft, Mountain View, CA, USA. *SunOS 5.2: Guide to Multi–Thread Programming*, 1st edition, May 1993.

[Sun93b] SunSoft, Mountain View, CA, USA. *SunOS 5.2: Network Interfaces Programmer's Guide*, 1st edition, May 1993.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice–Hall, Englewood Cliffs, New Jersey, 1992.

[Thö92] H. J. Thönnissen. Entwurf und Implementierung einer Objektalgebra für ein deduktives Objektbanksystem. Diplomarbeit, RWTH Aachen, 1992.

[Ull91] Jeffrey D. Ullman. A comparison between deductive and object–oriented database systems. In C. Delobel, M. Kifer, and Y. Masuraga, editors, *Deductive and Object–Oriented Databases; Procceedings of the Second International Conference on Deductive and Object–Oriented Databases*, number 566 in LNCS, pages 263–277. Springer Verlag, 1991.

[Unl91] Rainer Unland. Topaz: A tool kit for the construction of application specific transaction managers. Technischer Bericht MIP-9113, Universität Passau, Fakultät für Mathematik und Informatik, October 1991.

[US92] Rainer Unland and Gunter Schlageter. A transaction manager development facility for non standard database systems. In Elmagarmid [Elm92], chapter 11.
Mehr oder weniger eine gekürzte Fassung von [Unl91].

[VGH93] Gottfried Vossen and Magret Groß-Hardt. *Grundlagen der Transaktionsverarbeitung.* Addison–Wesley, Bonn, 1993.

[WA93] Man Hon Wong and Sivyakant Agrawal. Context–based synchronization: An approach beyond semantics for concurrency control. In *Proceedings of the Principles of Database Systems Conference*, pages 276–287. ACM Press, 1993.

[Wei88] Gerhard Weikum. *Transaktionen in Datenbanksystemen.* Addison–Wesley, Bonn; Reading, Mass., 1988.

[Wol92] Peter Wolks. Lange Transaktionen. In *Seminar Transaktionskonzepte*. Lehrstuhl für Informatik V, RWTH Aachen, SS 1992.

[WR92] Helmut Wächter and Andreas Reuter. The ConTract model. In Elmagarmid [Elm92], chapter 7.

[WS92] Gerhard Weikum and Hans-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In Elmagarmid [Elm92], chapter 13.

[Yan82] M. Yannakakis. A theory of safe locking policies in database systems. *Journal of the ACM*, 29(3):718–740, 1982.