Aktive Mechanismen zur Konsistenzsicherung in Föderationen heterogener und autonomer Datenbanken

Dem Fachbereich Informatik
der Technischen Hochschule Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
vorgelegte

Dissertation

von
Dipl.-Ing.

Thomas Kudraß

aus Weimar

Referent: Prof. Dr. A.P. Buchmann
Korreferent: Prof. Dr. E.J. Neuhold

Tag der Einreichung: 14.03.1997
Tag der mündlichen Prüfung: 21.04.1997

Darmstadt 1997
D 17
Darmstädter Dissertation

So eine Arbeit wird eigentlich nie fertig,
man muß sie für fertig erklären, wenn man nach Zeit und Umständen das Möglichste gethan hat.
Johann Wolfgang von Goethe*

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Fachgebiet Datenverwaltungssysteme 1 (DVS1) des Fachbereichs Informatik der Technischen Hochschule Darmstadt. Sie kann einerseits als "H-Teil" des von Prof. Buchmann ins Leben gerufenen REACH-Projekts (Realtime Active Heterogeneous System) gesehen werden, ist aber andererseits auch in sich abgeschlossen.

Mein erster Dank gebührt Herrn Prof. Dr. Alejandro Buchmann für die Empfehlung, mich mit diesem interessanten Thema zu beschäftigen, und die wissenschaftliche Betreuung der Arbeit. Seine zahlreichen Anregungen und wertvollen Kommentare haben wesentlich zur Verbesserung der Arbeit beigetragen. Von Nutzen waren dabei auch viele internationale Kontakte, die durch ihn ermöglicht wurden. Prof. Dr. Erich Neuhold danke ich für die Übernahme des Korreferats.

Die kostenlose Überlassung des Systems Persistence für die Durchführung der Arbeit verdanke ich Prof. Dr. Arthur Keller von der Stanford University und der Firma Persistence Software.

Danken möchte ich meinen beiden langjährigen Weggefährten und ehemaligen "Mitbewohnern" bei DVS1, Holger Branding und Jürgen Zimmermann, für fachliche und menschliche Unterstützung sowie zahlreiche anregende und nützliche Diskussionen während unserer gemeinsamen Zeit. Besonders hervorheben möchte ich dabei die große Hilfsbereitschaft und das Engagement von Jürgen.

Zu Dank verpflichtet bin ich zahlreichen Studenten, die im Rahmen von Studien- und Diplomarbeiten an der Implementierung des Prototypsystems beteiligt waren. Sie haben mit ihrem Einsatz und ihren Ideen einen wichtigen Beitrag zum Gelingen der Arbeit geleistet. Es sind dies: Thomas Bittmann, Reiner Bruttger, Rainer Jaspert, Klaus und Kurt Haberhauer, Reinhold Krayer, Andreas Loew, Miro Mrsic, Michael Plocki und Jochen Reckziegel.

Nennen möchte ich auch unseren Neuzugang Christoph Liebig sowie die Studenten Michael Hurler und Thomas Schmitz, die eine Bereicherung für unsere Gruppe waren und mir in der Endphase der Arbeit den Rücken stärkten.

Für die nötige Auflockerung zum Feierabend sorgte José Coelho Morais, der darüber hinaus mit seinem Service am Arbeitsplatz mir manchen Weg ersparte.

Bei dieser Gelegenheit möchte ich auch diejenigen Menschen erwähnen, die mein wissenschaftliches Interesse für Datenbanken geweckt bzw. später mich hierin gefördert haben, namentlich in chronologischer Folge: Dr. Karl-Heinz Herwig, Prof. Dr. Dietrich Schubert, Prof. Dr. Peter Hupfer und Dr. Günter von Bültzingsloewen.

Schließlich gilt mein ganz persönlicher Dank meinen Eltern für ihre Zuversicht und ihr Vertrauen. Sie haben mir mit guten Ratschlägen aus eigener Lebenserfahrung heraus den notwendigen Rückhalt für eine lange und oft beschwerliche Wegstrecke gegeben.

Darmstadt, im März 1997

---

* "Italienische Reise II", Caserta, den 16. März 1787 (Weimarer Ausgabe, Abt. 1, Band 31)

Zusammenfassung

Heterogene Informationssysteme sind heutzutage aus historischen und technischen Gründen unvermeidlich. Daten werden typischerweise auf heterogenen Hardware- und Softwareplattformen verwaltet, wobei eine Migration der Legacy-Systeme zu einer homogenen Umgebung aufgrund hoher Kosten und des Verlustes vorhandener Funktionalität oft nicht praktikabel erscheint. Zwischen diesen existierenden Datenbeständen existieren globale Integritätsbedingungen, bei deren Einhaltung auch die syntaktische und semantische Heterogenität der Daten berücksichtigt werden muß.

Zur globalen Integritätskontrolle wird ein aktiver Datenbank-Ansatz vorgeschlagen, wobei ein Datenbankverbund als eine Sammlung aktiver Objekte modelliert wird. Die aktiven Objekte werden durch ein objektorientiertes Vermittlersystem verwaltet. Das Vermittlersystem hat die Aufgabe, ein homogenes Interface zu heterogenen Komponenten bereitzustellen und die globale Konsistenz zwischen ihnen zu überwachen. Bei dieser Lösung können die externen Datenspeicher unverändert existieren, allerdings entstehen zusätzliche Anforderungen an die Funktionalität, die von den lokalen Systemen bereitgestellt werden muß. Damit ist zugleich ein Grundkonflikt angesprochen, der in dieser Arbeit eine wichtige Rolle spielt: die Einschränkung von lokaler Autonomie als Preis für die Einhaltung globaler Konsistenzbedingungen. Anhand einer Reihe von Beispielen, die in ein Klassifikationsschema eingeordnet werden, wird gezeigt, welche Entwurfsentscheidungen bei der Entwicklung eines Vermittlersystems bzw. bei der Anpassung der lokalen Systeme zu treffen sind. Dabei zeigt sich, daß traditionelle Konsistenzbegriffe homogener Datenbanksysteme oft nicht mehr ausreichend sind und zeitliche und extensionale Abschwächungen in kontrollierter Weise toleriert werden müssen.

Das globale Objektmodell des hier vorgestellten Vermittlers ist angelehnt an ein Standardmodell (ODMG-93/C^{++}) mit Erweiterungen, die die Voraussetzung bieten, komplexe Abhängigkeiten durch ECA-Regeln (Event Condition Action) zu beschreiben. ECA-Regeln bilden in dieser Arbeit somit das grundlegende Modellierungskonstrukt zur Spezifikation globaler Integritätsbedingungen, die oft auch modellinhärent in kanonischen Datenmodellen enthalten sind. Aufgrund ihrer Ausdrucksmächtigkeit sind sie auch für abgeschwächte Konsistenzkriterien in Multidatenbankumgebungen geeignet.

Die vorliegende Arbeit beschreibt die Umsetzung der Konzepte aktiver Konsistenzsicherung am Beispiel eines Verbundes heterogener relationaler Datenbanksysteme. Das entwickelte Vermittlersystem umfaßt Komponenten zur Erkennung, Signalisierung und Protokollierung lokaler Datenbankereignisse. Diese werden auf globaler Seite interpretiert und durch eine Regelverarbeitungskomponente behandelt. Angewandt werden diese Techniken auf die Kontrolle von replizierten Datenbeständen. Auch die Spezifikation komplexerer Integritätsbedingungen ist mit Hilfe eines Eingabewerkzeuges möglich. Eine Besonderheit stellt die Behandlung lokaler Schemaereignisse dar, die die Konsistenz zwischen Schemata beeinflussen können. Es wird die Gesamtarchitektur des Vermittlersystems und das Zusammenspiel seiner Komponenten vorgestellt.

Abstract

Heterogeneous information systems are today inevitable for historical and technical reasons. Data is typically maintained on heterogeneous hardware and software platforms and a migration of the legacy systems to a homogeneous environment is not feasible due to high costs and the loss of available functionality. Between existing databases there are global integrity constraints that have to consider both syntactic and semantic data heterogeneity.

An active database approach is proposed for the problem of global integrity control. Database federations are modelled as collections of active objects. The active objects are managed by an object-oriented mediator system. The mediator system has to provide a homogeneous interface to heterogeneous components and the functionality to monitor the global consistency among them. This solution allows the external data stores to exist without significant changes but requires additional functionality that has to be provided by the local systems. This addresses a basic principle discussed in this thesis: the conflict between local autonomy and global consistency. The cost of maintaing global constraints is the necessary restriction of the local autonomy. Through specific examples a general schema of trade-offs is derived and it is shown what design decisions are needed for the development of a mediator system and the adaptation of the component systems. It becomes apparent that traditional definitions of consistency in homogeneous database systems are often insufficient, therefore temporal relaxations and partial consistency have to be tolerated in a controlled manner.

The global object model of the mediator developed in this thesis is based on a standard data model (ODMG-93/C^{++}) and extensions through which complex dependencies can be described by the use of ECA rules (Event Condition Action). Hence, ECA rules represent our fundamental modeling construct to specify global integrity constraints that are often model-inherent constraints of canonical data models. Due to their expressiveness they are also well suited for weaker consistency criteria, such as temporally relaxed consistency.

The submitted thesis describes the implementation of active consistency control mechanisms in a federation of heterogenous relational database systems. The developed mediator system comprises components for the detection, signalling and logging of local database events. These events are interpreted at the global level and processed by a rule management component. The proposed active techniques are applied to the control of autonomous replicated data stores. The specification of complex global integrity constraints is facilitated by an editor tool and can be mapped to ECA rules of the active mediator system. Local schema events may affect the schema consistency and are treated separately. The overall architecture of the mediator system and the interplay of its components are presented.

Inhaltsverzeichnis

Inhaltsverzeichnis . . . . . **vii**

Abbildungsverzeichnis . . . . . **xiii**

Tabellenverzeichnis . . . . . **xv**

Abkürzungsverzeichnis . . . . . **xvii**

1 Einleitung **1**

1.1 Motivation 1
1.2 Einordnung in das REACH-Projekt 2
1.3 Überblick 4

2 Der Konsistenzbegriff in homogenen und heterogenen Systemen **7**

2.1 Der Konsistenzbegriff 8
2.2 Eine Formalisierung des Konsistenzbegriffes 10
2.2.1 Grundbegriffe eines Objektmodells 10
2.2.2 Integrität in einem Objektdatenmodell 13
2.3 Eine Taxonomie für Integritätsbedingungen 15
2.3.1 Klassen-Regeln 15
2.3.1.1 Intra-Instanzenregeln 15
2.3.1.2 Inter-Instanzenregeln 16
2.3.2 Klassenübergreifende Regeln 16
2.4 Konsistenz in heterogenen Datenbanken 18
2.4.1 Allgemeines 18
2.4.2 Heterogenität als Hauptfaktor globaler Integritätsbedingungen 19
2.4.2.1 Semantische Heterogenität auf Klassenebene 20
2.4.2.2 Semantische Heterogenität auf Instanzenebene 23
2.4.3 Globale Integrität in heterogenen Datenbanken 24
2.5 Abschwächung von Konsistenzbedingungen 27
2.5.1 Zeitliche Dimension der Abschwächung 27
2.5.1.1 Konsistenzwahrung an Aktivitätsgrenzen 28
2.5.1.2 Zustandsbasierte Abschwächung 28

2.5.1.3 Anwendbarkeit zeitlicher Abschwächungen . . . 29
2.5.1.4 Der Zeitbegriff . . . 29
2.5.2 Extensionale Dimension der Abschwächung . . . 30
2.6 Kontrollabhängigkeiten . . . 30
2.6.1 Abgleichstrategien für redundante Daten . . . 31
2.6.2 Verhältnis lokaler und globaler Constraints . . . 33
2.7 Weitere Konsistenzbedingungen in Multidatenbanken . . . 35
2.7.1 Beschreibung der Metadaten . . . 35
2.7.2 Beschreibung der Abhängigkeiten . . . 36
2.8 Verwandte Arbeiten: Modellierung interdependenter Daten . . . 37

3 Konzepte aktiver Datenbanksysteme . . . 43

3.1 Grundbegriffe . . . 43
3.2 Charakterisierung aktiver Datenbanksysteme . . . 45
3.3 Globale Integritätskontrolle durch aktive Systeme . . . 49

4 Lokale Autonomie in Multidatenbanken . . . 51

4.1 Grundbegriffe und Klassifizierung von Multidatenbanksystemen . . . 52
4.2 Architektur von Multidatenbanksystemen . . . 54
4.2.1 Begriffe . . . 54
4.2.2 Vergleich und Bewertung . . . 55
4.3 Der Autonomiebegriff in Multidatenbanksystemen . . . 56
4.3.1 Autonomie in homogenen verteilten Systemen . . . 57
4.3.2 Autonomie in Multidatenbanksystemen . . . 58
4.3.3 Verallgemeinerung des Begriffs Autonomie in interoperablen Systemen . . . 62
4.4 Der Zusammenhang von Autonomie und globaler Konsistenz in Legacy-Systemen 66
4.4.1 Beschreibung von Autonomie-Verletzung . . . 67
4.4.2 Der Trade-Off zwischen lokaler Autonomie und globaler Konsistenz . . . 69
4.4.2.1 Anforderungen aus Sicht der Konsistenzwahrung . . . 69
4.4.2.2 Anforderungen gegenüber der Autonomie . . . 71
4.4.3 Quantitative Betrachtung von Autonomie vs. Konsistenz . . . 72
4.4.4 Zusammenfassung und Fazit . . . 73

5 Aktive Objekte zur Konsistenzkontrolle in heterogenen Systemen . . . 75

5.1 Lösungsansatz: Aktive Objekte . . . 76
5.2 Ein ECA-Regelmodell für aktive Objekte in Multidatenbanken . . . 77
5.2.1 Eventmodell . . . 77
5.2.1.1 Eventbegriff . . . 77
5.2.1.2 Events in Multidatenbanken . . . 78
5.2.2 Regelmodell . . . 80
5.3 Ausführungsmodell für aktive Objekte in Multidatenbanken . . . 82
5.3.1 Direkte Verarbeitung lokaler Events . . . 82
5.3.1.1 Ablaufmodell . . . 82

5.3.1.2 Auswirkungen auf lokale Autonomie 83
5.3.1.3 Semantische Aspekte 84
5.3.2 Indirekte Verarbeitung lokaler Events 85
5.3.2.1 Ablaufmodell 85
5.3.2.2 Auswirkungen auf lokale Autonomie 87
5.3.2.3 Semantische Aspekte 87
5.4 Datenmodell für aktive Objekte in Multidatenbanken 88
5.4.1 Semantische Beziehungen in kanonischen Datenmodellen 88
5.4.2 Modellierung modellinhärenter Bedingungen durch ECA-Regeln 89
5.4.3 Modellierung expliziter Integritätsbedingungen durch ECA-Regeln 90
5.5 ODMG-93 als globales Objektdatenmodell 92
5.5.1 Die Bestandteile des Objektmodells 93
5.5.2 Die Beziehung ODMG-Modell / C^{++} 94

6 Umsetzung 97

6.1 Ein aktives Vermittlersystem 98
6.1.1 Einführung in Vermittler 98
6.1.2 Ein Vermittlersystem zur Konsistenzkontrolle 98
6.2 Zu lösende Probleme 100
6.3 Detektion lokaler Events in relationalen Systemen 102
6.3.1 Arten von Events 102
6.3.2 Lokale Detektionsmechanismen 103
6.3.2.1 Wrapping 103
6.3.2.2 Aktive Eigenschaften von relationalen Datenbanksystemen - Das Trigger-Konzept 104
6.3.2.3 Auditing 105
6.4 Die Protokollierung von Datenänderungen 106
6.5 Eventsignalisierung durch Middleware 107
6.5.1 Database Middleware 107
6.5.2 Remote Procedure Calls 107
6.5.3 Object Request Broker 108
6.5.4 Message Oriented Middleware 108
6.5.5 Auswahl der Kommunikationsplattform 109
6.6 Objektorientierte Integration von relationalen Datenbanken 109
6.6.1 Existierende objekt-relationale Ansätze 109
6.6.2 Leistungsumfang einer objektorientierten Zugriffsschicht auf relationalen Datenbanken 110
6.6.3 Abbildung zwischen objektorientiertem und relationalem Modell 111
6.6.3.1 Strukturelle Abbildung 111
6.6.3.2 Operationale Abbildung 112
6.6.4 Kommerzielle objekt-relationale Produkte 112
6.6.4.1 UniSQL/M 113
6.6.4.2 Subtleware 113

6.6.4.3 Persistence . . . 113
6.6.4.4 Weitere Systeme . . . 114
6.6.5 Systemauswahl . . . 115
6.7 Integration der Regelverarbeitung . . . 115

7 Detektion und Signalisierung lokaler Events . . . 117

7.1 Implementierung eines Datenbank-Gateways . . . 117
7.1.1 Plattform: Sybase Open Server . . . 117
7.1.2 Realisierung der Event-Detektion . . . 119
7.1.3 Die Architektur des Gateways . . . 120
7.1.4 SQL-Syntaxanalyse . . . 121
7.1.5 Realisierung der Protokollierung von DML-Operationen . . . 121
7.1.5.1 DML-Kommandos . . . 122
7.1.5.2 Transaktionskommandos . . . 124
7.1.5.3 Protokollverdichtung . . . 125
7.1.5.4 Protokollierung im Überblick . . . 127
7.1.6 Behandlung von DDL-Befehlen . . . 127
7.1.6.1 Extraktion relevanter Informationen . . . 127
7.1.6.2 Detektion “verborgener” Events . . . 128
7.1.6.3 Ausführung detektierter Operationen . . . 129
7.1.7 Weitere Komponenten des Gateways . . . 130
7.1.8 Sonstige Aspekte . . . 131
7.1.9 Vergleich mit anderen Plattformen . . . 131
7.2 Integration der Eventsignalisierung in die Systemumgebung . . . 132
7.2.1 Remote Procedure Call im Überblick . . . 132
7.2.2 ExecDML: Die RPC-Schnittstelle zum DML-Vermittlerprozeß . . . 133
7.2.3 ExecDDL: Signalisierung lokaler Schemaveränderungen . . . 134
7.2.4 Generierung eines RPC-Servers . . . 136

8 Implementierung eines aktiven heterogenen Systems . . . 137

8.1 Das Produkt Persistence als Plattform . . . 138
8.1.1 Das Objektmodell von Persistence . . . 138
8.1.2 Dictionary Reader . . . 139
8.1.3 Relational Interface Generator . . . 141
8.1.4 Relational Object Manager . . . 143
8.1.5 Gleichzeitiger Zugriff auf mehrere Datenbanken . . . 145
8.2 DML-Vermittler . . . 147
8.2.1 Abbildung von SQL-Anweisungen auf die Persistence-Schnittstelle . . . 147
8.2.2 Einbettung in die Systemumgebung . . . 149
8.3 Vermittler-Generatoren . . . 150
8.3.1 Das Konzept des “Vermittler-Generators” . . . 150
8.3.2 Die Implementierung des Vermittler-Generators . . . 150
8.3.2.1 Architektur und Module des Vermittler-Generators . . . 150

8.3.2.2 Der Objektschema-Analysator 153
8.3.2.3 Der Präprozessor des Vermittler-Generators 154
8.4 Die aktive Komponente des Vermittlers 154
8.4.1 Aktive Verarbeitung mit Hooks in Persistence 154
8.4.2 Behandlung von Events 155
8.4.2.1 Integration von Datenbank-Events 155
8.4.2.2 Detektion von Transaktions- und Connect-Events 157
8.4.2.3 Time-Events 158
8.4.3 Repräsentation der Regeln 158
8.4.4 Architektur und Funktionalität von MERU 160
8.4.5 Funktionsweise und Architektur des TemporalEventManagers 162
8.4.6 Besondere Implementierungsaspekte 166
8.4.7 Bewertung 167
8.5 Implementierung der asynchronen Replikationskomponente 170
8.5.1 Ablaufmodell 170
8.5.2 Spezifikation der Replikate und Algorithmen 171
8.5.3 Bewertung der Replikationskomponente 175
8.6 Ein Werkzeugkasten zur Eingabe globaler Integritätsbedingungen 176
8.6.1 Spezifikation globaler Integritätsbedingungen (GISpeL) 176
8.6.1.1 Sprachdefinition 176
8.6.1.2 Parser 178
8.6.2 Editor (GISpeL-GUIDe) 180
8.6.3 Automatische Ableitung von Regeln (RuDe) 182
8.7 Der DDL-Vermittler 185

9 Verwandte Arbeiten 189

9.1 Kontrolle interdependenter Daten in heterogenen Systemen 190
9.2 Protokolle für globale Integritätskontrolle 192
9.3 Erweiterung von Multidatenbanksystemen um ECA-Regeln 194
9.4 Verteilte Aktive Objekte 196
9.5 Mediators 197
9.6 Verteilte Künstliche Intelligenz 199

10 Zusammenfassung und Ausblick 201

10.1 Zusammenfassung der Ergebnisse 201
10.2 Ausblick auf künftige Arbeiten 204

A Grammatik von GISpeL 207

B Anwendungsbeispiel 209

Literaturverzeichnis 217

Abbildungsverzeichnis

2.1 Modellierung eines Realitätsausschnittes 8
2.2 Konzepte des Objektmodells 12
2.3 Modellierung eines Weltausschnittes in zwei Datenbanken 19
2.4 Beziehungen zwischen Klassenextensionen 21

3.1 Aktive Datenbanksysteme 44

4.1 Multidatenbanksystem-Architekturen 53
4.2 Fünf-Ebenen-Architektur eines Multidatenbanksystems [SL90] 55
4.3 Interaktionen in einer autonomen Umgebung 63
4.4 Struktur eines lokalen Legacy-Systems 66
4.5 Kategorien von Autonomie 69
4.6 Dimensionen globaler Konsistenz 70
4.7 Quantitative Darstellung von Autonomie 73

5.1 Eventdetektion in mehreren Ebenen 78
5.2 Event-Hierarchie in einer Multidatenbankumgebung 79
5.3 Direkte Verarbeitung lokaler Events in einem Multidatenbanksystem 83
5.4 Indirekte Verarbeitung lokaler Events in einem Multidatenbanksystem 86
5.5 Existenzabhängigkeiten (strict vs. relaxed) 90
5.6 Das ODMG-93-Metamodell 94

6.1 Schichtenarchitektur für Vermittlerprozesse 99
6.2 Modell des aktiven Vermittlersystems 100
6.3 Aufbau der Testumgebung des Vermittlersystems 101
6.4 Wrapping-Mechanismen 104
6.5 CORBA Event Services 108
6.6 DB-Zugriffsschicht zwischen OO Anwendung und relationaler Datenbank 111
6.7 Systemarchitektur von Persistence im Überblick 114

7.1 Open Server-Anwendung als Gateway 118
7.2 Ablauf der Behandlung eines Language Events 119
7.3 Architektur des Datenbank-Gateways 120
7.4 Zustandsübergangsdiagramm der lexikalischen Analyse 122
7.5 Zusammenhang von Protokollierungsaktionen und -tabellen 127
7.6 Modell eines Remote Procedure Call [Ste90] 132
7.7 Struktur der RPC-Parameter von `ExecDMLParm` 134
7.8 Transformation von DDL-Statements in RPC-Datenstrukturen (Beispiel) 135
7.9 Dateien bei der Generierung eines Sun-RPC-Programms (Beispiel) 136

8.1 Schema und Ausschnitt aus zugehöriger `.persist`-Datei 139
8.2 Klassen in Persistence 141
8.3 Konzept des Semantic Key Swizzling 143
8.4 Konzept der Smart Pointer 144
8.5 Gleichzeitiger Zugriff auf mehrere Datenbanken in Persistence 145
8.6 Datenbankverbindungen in Persistence 146
8.7 Kommunikation zwischen Gateway, Vermittlerprozeß und lokalem DB-Server 150
8.8 Generierung eines DML-Vermittlers (Prinzipdarstellung) 151
8.9 Vermittler-Generierungsprozeß im Überblick 152
8.10 Klassenhierarchie für Zeitereignisse 158
8.11 Beziehungen zwischen Regeln und DB-Events 160
8.12 Architektur von MERU 162
8.13 Thread-Architektur des TemporalEventManagers 163
8.14 Beziehungen zwischen Regeln und Time-Events 163
8.15 Verarbeitungsmodell für Replikationskontrolle im Vermittlersystem 171
8.16 Hierarchie der Replikationsklassen 172
8.17 Hauptfenster des GISpeL-GUIDe nach dem Laden einer GISpeL-Datei 181
8.18 Regelableitungsprozeß 182
8.19 Architektur des DDL-Vermittlers 186

B.1 Anwendungsbeispiel in OMT-Darstellung 210

Tabellenverzeichnis

2.1 Domänen-Unterschiede (durch Korrespondenzfunktionen ausgedrückt) 25
2.2 Integritätsbedingungen in heterogenen Datenbanken 26
2.3 Darstellung der verschiedenen Einbringstrategien 33
2.4 Charakterisierung der Beispieldaten 33
2.5 Konsistenzregeln in einer Multidatenbankumgebung 37
2.6 Bewertung von Ansätzen zur Modellierung von globalen Konsistenzbedingungen . . . 41

3.1 Kombinationen von Kopplungsmodi 45
3.2 Bewertung aktiver Systeme 45
3.3 Eigenschaften aktiver Systeme zur globalen Integritätssicherung. 49

4.1 Vergleich von Multidatenbanksystem-Architekturen 56
4.2 Überblick über Arten lokaler Autonomie 65
4.3 Autonomie vs. Konsistenz 71

5.1 Typen von ECA-Regeln in Multidatenbanken. 80
5.2 Einteilung aktiver Multidatenbanken. 81
5.3 Direkte Eventverarbeitung vs. Autonomie 84
5.4 Indirekte Eventverarbeitung vs. Autonomie 87
5.5 Effekte bei konsistenzverletzenden Aktionen 91
5.6 Verhältnis von C^{++} und ODMG. 95

6.1 Auswirkungen von SQL-Datenbankereignissen auf globale Konsistenz 102

7.1 Intensionales Protokoll: `statement_log` 122
7.2 Log-Tabelle zur Protokollierung von modifizierten Tupeln 122
7.3 Protokollierung von DML-Kommandos 123
7.4 Protokolltabelle für Transaktionskommandos `transaction_log` 124
7.5 Protokollierung von Transaktionskommandos 125
7.6 Regeln zur Verdichtung des Protokolls 125
7.7 Zeitpunkte bei der Protokollverdichtung 126
7.8 DDL-Operationscodes 128

8.1 Methoden der Klasse <Class> (Auswahl) 142
8.2 Methoden der Klasse `PS_Connection` 145
8.3 Abbildung der Operationstypen auf Persistence-Methodenaufrufe. 148
8.4 Globale Variablen im Vermittler-Generator. 153
8.5 Datenbank-Events in Persistence. 155
8.6 Parameter bei der Signalisierung von Datenbank-Events 156
8.7 Charakterisierung des Regelmanagement-Systems MERU. 169
8.8 Replikationsbeziehung zwischen 3 Tabellen (Beispiel) 172
8.9 Zusätzliche Tabellen für die Replikation. 173
8.10 Datenbankereignisse in GISpeL 177
8.11 Direktiven in Template-Dateien für den ECA-Regelcode. 183

Abkürzungsverzeichnis

ACID	**A**tomarity **C**onsistency **I**solation **D**urability
API	**A**pplication **P**rogramm **I**nterface
BLOOM	**B**arce**lo**na **O**bject **M**odel
CAD	**C**omputer **A**ided **D**esign
CDM	**C**anonical **D**ata **M**odel
CORBA	**C**ommon **O**bject **R**equest **B**roker **A**rchitecture
CRUD	**C**reate **R**ead **U**pdate **D**elete
DB	**D**ata**b**ase
DBMS	**D**ata**b**ase **M**anagement **S**ystem
DBS	**D**ata**b**ase **S**ystem
DDL	**D**ata **D**efinition **L**anguage
DML	**D**ata **M**anipulation **L**anguage
DOM	**D**istributed **O**bject **M**odel
DTP	**D**istributed **T**ransaction **P**rocessing
ECA	**E**vent **C**ondition **A**ction
EER	**E**xtended **E**ntity **R**elationship
ER	**E**ntity **R**elationship
FDBS	**F**ederated **D**ata**b**ase **S**ystem
FG	**F**ach**g**ebiet
GAI	**G**lobal **A**pplication **I**nterface
GISpeL	**G**lobal **I**ntegrity **Spe**cification **L**anguage
GT	**G**lobal **T**ransaction
HiPAC	**Hi**gh **P**erformance **Ac**tive System
IDL	**I**nterface **D**efinition **L**anguage
LAI	**L**ocal **A**pplication **I**nterface
LAS	**L**ocal **A**pplication **S**ystem
LDB	**L**ocal **D**ata**b**ase
LDBI	**L**ocal **D**ata**b**ase **I**nterface
LDBMS	**L**ocal **D**ata**b**ase **M**anagement **S**ystem
LDBS	**L**ocal **D**ata**b**ase **S**ystem
LT	**L**ocal **T**ransaction
MDB	**M**ulti**d**ata**b**ase
MDBS	**M**ulti**d**ata**b**ase **S**ystem

MERU	**Me**diator's **Ru**le System
MOM	**M**essage **O**riented **M**iddleware
ODMG	**O**bject **D**atabase **M**anagement **G**roup
OID	**O**bject **Id**entifier
OLE	**O**bject **L**inking and **E**mbedding
OMG	**O**bject **M**anagement **G**roup
OMT	**O**bject **M**odelling **T**echnique
OO	**O**bject-**O**riented
OODBMS	**O**bject-**O**riented **D**ata**b**ase **M**anagement **S**ystem
OQL	**O**bject **Q**uery **L**anguage
ORB	**O**bject **R**equest **B**roker
RDB	**R**elational **D**ata**b**ase
RDBMS	**R**elational **D**atabase **M**anagement **S**ystem
REACH	**Re**altime **Ac**tive **H**eterogeneous System
RIG	**R**elational **I**nterface **G**enerator
ROM	**R**elational **O**bject **M**anager
RPC	**R**emote **P**rocedure **C**all
SQL	**S**tructured **Q**uery **L**anguage
VLDB	**V**ery **L**arge **D**ata**b**ases
XDR	E**x**ternal **D**ata **R**epresentation
2PC	**2**-**P**hase **C**ommit

Kapitel 1

Einleitung

1.1 Motivation

Seit der Entstehung der elektronischen Datenverarbeitung ist mit dem Fortschreiten der Informationstechnologie unaufhörlich der Bestand an Daten gewachsen, der auf unterschiedlichste Art in den Unternehmen verwaltet wird. Die Daten können in einem von mittlerweile über 200 verfügbaren Datenbankmanagementsystemen (DBMS), aber auch in vielen anderen Formen wie z.B. in Dateisystemen oder kommerziellen Applikationssystemen mit eigener Datenhaltung vorliegen. Die Vision einer unternehmensweit einheitlichen Datenverwaltung erwies sich als Illusion, die auch nicht durch die Entwicklung verteilter Datenbanksysteme gelöst wurde. Der top-down Entwurf verteilter Datenbanken stellte hohe technische Anforderungen an die vorhandenen lokalen Systeme, die nur teilweise datenbankbasiert waren. Die Bedeutung der lokalen Autonomie sowie die Notwendigkeit, existierende Datenspeicher (Repositories) und existierende Applikationen aus historischen und technischen Gründen beizubehalten, wurden in wachsendem Maße erkannt. Somit erwiesen sich heterogene Systeme als unvermeidlich, sowohl in der Hard- als auch in der Software. Bereits erbrachte hohe Investitionen in die Systeme als auch der Wert der Daten machten es unmöglich, auf homogene und damit monolythische System zu wechseln.

Seit Beginn der 80er Jahre ist der Bedarf an Integrationslösungen bedeutend gewachsen. Hierfür wurde das Konzept der Multidatenbanksysteme (MDBS) entwickelt, ein Ansatz, der die Idee eines Datenbanksystems auf eine global vereinheitlichte Umgebung überträgt und Heterogenität und Autonomie der lokalen Datenbanken berücksichtigt. Dieses Konzept wurde in einer Vielzahl von Forschungs-Prototypen umgesetzt, allerdings wurden kaum entsprechende Produkte der kommerziellen DBMS-Anbieter auf dem Markt angeboten. Hierfür gibt es eine Reihe von Gründen, insbesondere noch ungelöste technische Fragestellungen. Dazu zählen z.B. Anforderungen an erweiterte Transaktionsmodelle oder die Auflösung semantischer Konflikte bei der Schemaintegration.

Die Entwicklung der letzten Jahre zeigt, daß die Systemintegration eine Schlüsselanforderung geworden ist, begleitet von Schlagworten wie Re-Engineering, Interoperabilität, Data Warehousing, Objektorientierung. Dabei wurde die Objekttechnologie zur Basis vieler neuer Entwicklungen, was sich durch die Fülle der angebotenen Produkte als auch durch die wachsenden Zahl objektorientierter Standards belegen läßt. Hierfür seien stellvertretend genannt: CORBA (Common Object Request Broker Architecture) als Grundlage zahlreicher Produkte,

Microsofts OLE (Object Linking and Embedding), objektorientierte und objekt-relationale Datenbanksysteme. Zusammen mit der Objekttechnologie wird die Verteilung zum bestimmenden Trend der nächsten Jahre. Dabei werden Datenbanken auch künftig den Kern der Informationssysteme der nächsten Generation bilden müssen.

In dieser Entwicklung ergeben sich gegenüber zentralisierten Datenbanksystemen neue Fragestellungen, wie z.B. die nach dem Konsistenzbegriff für Daten in heterogenen Datenspeichern. Herkömmliche Konsistenzkriterien sind oft ungenügend, denn es müssen auch kontrollierte Abweichungen erlaubt werden, soweit es die Autonomie der lokalen Systeme erfordert. In der Praxis anzutreffende Applikationssysteme bestehen sehr oft aus Programmen, die jeweils auf Daten zugreifen, die mehrfach in separaten Systemen gespeichert sind (möglicherweise in unterschiedlichen Repräsentationen) oder in bestimmten Beziehungen zueinander stehen (z.B. als abgeleitete Daten). Den Abgleich zwischen diesen Daten nehmen spezielle Programme vor, die besonders in Zeiten laufen, wenn kein Benutzerbetrieb stattfindet.

Es wurden hierfür im Rahmen der Arbeit eine Reihe von Beispielanwendungen untersucht, die auf diesem Prinzip beruhen:
Beim Pharma-Unternehmen Rhône-Poulenc Rorer werden in einem Management-Informationssystem (MIS) Daten aus den unterschiedlichsten Anwendungssystemen vereint. Dabei auftretende Abweichungen zwischen Materialstammdaten im Produktionssystem und dem MIS werden durch regelmäßige Batch-Jobs korrigiert [KLB96a].
Bei der Deutschen Lufthansa AG werden Tarifinformationen aller Fluggesellschaften aus einer externen Datenbank importiert, inklusive der Lufthansatarife. Darüber hinaus werden aber selbst Daten über die eigenen Tarife gehalten, die als Grundlage für weitere Preisberechnungen dienen, was Quelle für Inkonsistenz sein kann.

In einem herkömmlichen DBMS-basierten System erfolgt die Konsistenzkontrolle außerhalb der Anwendungen durch das Datenbanksystem. Zunehmend kommen dabei auch aktive Konzepte zum Einsatz, insbesondere Trigger in relationalen Systemen. Was in vielen dieser Systeme mittlerweile weit ausgereift ist, die Trennung der Konsistenzkontrolle von den Applikationsprogrammen, ist noch eine Vision in heterogenen Systemen. Wünschenswert wäre ein zwischen den existierenden Applikationen vermittelndes System, das das Wissen über die globalen Konsistenzanforderungen in sich vereint und in entsprechende Mechanismen umsetzt. Die zur Konsistenzkontrolle erforderlichen Eigenschaften lassen sich gut in einem aktiven System realisieren: Beobachtung kritischer Situationen und Auslösung der notwendigen Reaktionen. Im Unterschied zu einem zentralisierten System kommt jedoch das Problem der Wahrung lokaler Autonomie hinzu, das einen Gegensatz zum Bestreben nach globaler Konsistenz bildet. Die lokale Autonomie, wie sie auch im Kontext von Multidatenbanksystemen behandelt wurde, soll hier insbesondere in ihrem Verhältnis zur globalen Integrität betrachtet werden.

Um die verschiedenen Aspekte der Konsistenzsicherung in heterogenen Systemen besser zu verstehen, wurde ein solches aktives objektorientiertes Vermittlersystem entwickelt und in einer heterogenen relationalen Datenbankumgebung erprobt.

1.2 Einordnung in das REACH-Projekt

Die vorliegende Dissertation ist, obwohl in sich abgeschlossen, als Teil des Forschungsvorhabens REACH (**RE**altime **AC**tive **H**eterogenenous System) am FG Datenverwaltungssysteme 1 der TH Darmstadt zu sehen [BBKZ92]. Im Rahmen des Projekts REACH wird an der Erfor-

schung von Grundlagen und der Entwicklung von Systemen gearbeitet, die aktive und Echtzeiteigenschaften in heterogenen Umgebungen bieten. Dabei sollen diese Eigenschaften nicht isoliert voneinander betrachtet, sondern auch ihre Wechselwirkungen in Kombination untersucht werden. Es lassen sich verschiedene Arten von Interaktion finden: Das Verhalten heterogener Systeme läßt sich durch Regeln beschreiben. Anwendungen sind z.B. die Spezifikation von Integritätsbedingungen und Zugriffsrechten in einem heterogenen System durch Regeln. Denkbar wäre auch, die Korrektheitskriterien und die Ausführungsstruktur des zugrundeliegenden Transaktionsmodells in einer Datenbankföderation durch Regeln auszudrücken. Bei der Integritätssicherung in heterogenen und verteilten Umgebungen spielt ebenso der Zeitaspekt eine wichtige Rolle.

Ein echtzeitfähiges System wäre in der Lage, zeitliche Garantien für die Einhaltung von Integritätsbedingungen zu geben, was jedoch eine starke Einschränkung der lokalen Autonomie bedeuten würde. Der in REACH verfolgte Ansatz, in einem Scheduler eine flexible Überlastbehandlung auf der Basis von Wertfunktionen zu realisieren und bei Nichteinhaltung von Zeitschranken alternative Prozesse zu starten, kann auch für die Kontrolle abgeschwächter Integritätsbedingungen in einer Datenbankföderation von Interesse sein. Dies kann durch Konzepte unterstützt werden wie *Milestones* (zur Beobachtung des Fortgangs einer Aktion) oder *Contingency Actions*, wie sie im Rahmen des Projektes vorgeschlagen wurden [BBKZ93].

Parallel zur vorliegenden Arbeit entstand der Prototyp eines aktiven objektorientierten DBMS [Zim97] als ein funktionsfähiges persistentes C^{++} System, basierend auf dem objektorientierten Datenbanksystem Open OODB von Texas Instruments. Hierbei wird der Schwerpunkt auf ein großes Ereignisrepertoire, eine mächtige Ereignisalgebra und die Bereitstellung zahlreicher Kopplungsmodi zwischen Ereignissen und Regelauswertung gelegt, wie sie auch in heterogenen Systemen von Nutzen sein können. Daten in einer Open OODB Datenbank lassen sich somit als aktive Objekte definieren.

Das zu entwickelnde Vermittlersystem sollte zwei Rollen haben: einerseits muß es als Plattform für den homogenen Zugriff zu externen existierenden Datenbeständen dienen, andererseits soll es auch als Plattform für die Neuentwicklung von Anwendungssystemen dienen, um die Migration in eine homogenere (objektorientierte) Welt zu erleichtern. Objektorientierte Modelle haben den Vorteil der leichten Erweiterbarkeit und sind sehr flexibel im Hinblick auf die Darstellung von Objekten unterschiedlicher Komplexität auf verschiedenen Abstraktionsebenen. Der Schwerpunkt der vorliegenden Arbeit wird auf die Umsetzung globaler Integritätsregeln gelegt. Erfahrungen aus der Realisierung des aktiven OODBMS waren zugleich wertvoll für die Entwicklung des aktiven Vermittlersystems. Dieses muß allerdings über ein “herkömmliches” aktives System hinaus das Problem behandeln, wie ein bestehendes lokales System überhaupt in die globale Integritätssicherung einbezogen werden kann ohne Einschränkung seiner Autonomie. Hierfür wurden zwei Anforderungen zugrundegelegt: Weder die existierenden lokalen Daten noch die dazugehörigen Applikationsprogramme sollten dabei verändert werden. Da das Vermittlersystem auch eine globale Zugriffsschnittstelle bieten soll, wird damit eine zweite Ebene eingeführt, auf der Ereignisse auftreten können, die durch Ausführung entsprechender Regeln behandelt werden müssen. Der Zeitaspekt wird im Vermittler dadurch berücksichtigt, daß Integritätsregeln durch Zeitereignisse getriggert werden können. Allerdings sind keine Garantien für die Beendigung einer Aktion möglich, weil im Rahmen dieser Arbeit der Realzeit-Aspekt keine Berücksichtigung finden konnte.

Somit stellt die hier vorliegende Arbeit im Sinne des REACH-Projektes anhand eines konkret realisierten Vermittlersystems dar, welche Konsequenzen sich aus der Kombination von “heterogen” und “aktiv” ergeben. Was kann man überhaupt in aktiver Weise kontrollieren, welche

Anforderungen dafür sind an den Entwurf eines solchen Vermittlers zu stellen, wie sind die lokalen Systeme dabei zu beeinflussen ?

1.3 Überblick

Diese Dissertation ist wie folgt gegliedert:

Nach der Einleitung befassen sich die nachfolgenden Kapitel 2 bis 4 mit theoretischen Grundlagen der Integritätssicherung in heterogen Datenbanken:

Im 2. Kapitel wird der Konsistenzbegriff definiert sowie eine Klassifikation von Integritätsbedingungen gegeben. Diese Klassifikation wird auf Integritätsbedingungen in heterogenen Datenbanken angewandt. Konsistenz wird in drei Dimensionen charakterisiert. Dementsprechend umfaßt sie Bedingungen, die Integrität und Kontrollabhängigkeiten sowie mögliche Abschwächungen beschreiben. Der Konsistenzbegriff wird dahingehend erweitert, daß auch das Verhältnis der lokalen Datenbanken zu globalen Metainformationen betrachtet wird.

Kapitel 3 gibt einen Überblick über Grundbegriffe und Leistungsmerkmale aktiver Datenbanksysteme als ein Werkzeug zur Konsistenzkontrolle. Diese Eigenschaften werden zu Anforderungen der globalen Konsistenzsicherung in Beziehung gesetzt.

Kapitel 4 gibt einen Überblick über die Architektur von Multidatenbanksystemen und bewertet diese im Hinblick auf die Wahrung globaler Integrität. Dabei wird der Begriff der Autonomie herausgearbeitet und im Verhältnis zu Konsistenz betrachtet.

In Kapitel 5 wird die Idee diskutiert, zur globalen Konsistenzwahrung einen heterogenen Datenverbund durch einen Raum aktiver Objekte zu modellieren. Um diesen Ansatz zu realisieren, muß ein Regel- und Ausführungsmodell sowie ein geeignetes Objektmodell definiert werden. Der Begriff des Ereignisses (Events) wird in Multidatenbankumgebungen erweitert sowie der grundlegende Ablauf der Regelverarbeitung in einem heterogenen System von der lokalen Eventdetektion bis zur Regelausführung beschrieben. Es wird gezeigt, wie durch ECA-Regeln sowohl modellinhärente als auch explizite semantische Integritätsbedingungen ausgedrückt werden können.

Die praktische Umsetzung der Idee, heterogene Datenbanken als aktive Objekte zu behandeln, erfolgt durch die Realisierung eines aktiven objektorientierten Vermittlersystems. In Kapitel 6 werden die für die Implementierung des Vermittlersystems notwendigen Plattformentscheidungen diskutiert, wobei die globale Integritätskontrolle am Beispiel heterogener relationaler Systeme behandelt wird.

In Kapitel 7 erläutern wir die Implementierung eines lokalen Gateways für ein relationales DBMS zur Protokollierung und Signalisierung lokaler Ereignisse als integralem Bestandteil des Vermittlersystems und Voraussetzung für die Anwendung aktiver Datenbankkonzepte auf globaler Ebene.

In Kapitel 8 werden die Komponenten des Vermittlersystems präsentiert und ihre Implementierung im einzelnen beschrieben: Dazu zählen ein SQL/C^{++}-Übersetzer für die Interpretation lokaler Ereignisse, ein Regelmanager und eine temporale Komponente. Ebenso werden Tools wie ein Generator zur Erzeugung von Vermittlern und Regeln sowie ein Constraint-Editor präsentiert. Es wird gezeigt, wie asynchrone Replikationsstrategien im System integriert werden können. Im letzten Abschnitt dieses Kapitels wird ein Vermittler vorgestellt, der Schemaveränderungen in lokalen Datenbanken behandelt.

Anschließend gibt Kapitel 9 einen Überblick über Arbeiten, die eine Verwandtschaft mit dem hier vorgestellten Ansatz eines aktiven Vermittlersystems aufweisen.

Kapitel 10 enthält eine Zusammenfassung der wichtigsten Erkenntnisse über das Verhältnis von Autonomie und Konsistenz, die im Rahmen der Prototypentwicklung gesammelt wurden. Die Arbeit endet mit einem Ausblick auf mögliche weiterführende Arbeiten.

Im Anhang demonstriert ein zusammenfassendes Beispiel Funktionalität und Anwendungsmöglichkeiten des aktiven Vermittlersystems.

Kapitel 2

Der Konsistenzbegriff in homogenen und heterogenen Systemen

Der Begriff der Konsistenz, der seit vielen Jahren in der Literatur etabliert ist, wird in diesem Kapitel aufgegriffen, wobei der Schwerpunkt auf Datenkonsistenz gelegt wird. Für die formale Definition von Konsistenz wird ein Objektdatenmodell zugrunde gelegt, das übertragbar ist auf andere Modelle (Abschnitt 2.2). Im nachfolgenden Abschnitt 2.3 werden statische Integritätsbedingungen in homogenen Datenbanksystemen ausführlich behandelt und nach ihrem Geltungsbereich mit Hilfe von Beispielen klassifiziert.

Wir modellieren Konsistenz in heterogenen und autonomen Datenbanksystemen in drei Dimensionen: Dabei umfaßt Konsistenz eine Menge von datenbankübergreifenden Abhängigkeiten (Integritätsbedingungen), eine Menge von Kontrollabhängigkeiten zwischen den DBMS sowie eine Menge von Bedingungen zur Abschwächung der Konsistenzkriterien.

Es wird gezeigt, daß die Einteilung von Integritätsbedingungen, so wie sie für homogene Datenbanken gegeben wurde, sich in geeigneter Weise auf heterogene und autonome Datenbanken übertragen läßt. Dabei ist allerdings zu beachten, daß die Semantik der Integrität zusätzlich dadurch bestimmt wird, ob disjunkte oder nichtdisjunkte Weltausschnitte modelliert werden. Dementsprechend können bei einer multiplen Repräsentation eines Realweltausschnittes weitere schemaübergreifende (und damit globale) Integritätsbedingungen hinzukommen. Bei globalen Integritätsbedingungen müssen allerdings die Besonderheiten heterogener Datenbanken beachtet werden. Dabei werden die möglichen semantischen Differenzen ausführlich illustriert. Abschnitt 2.4 endet mit einer Zusammenstellung von Integritätsbedingungen, die alle als Existenz- und Wertabhängigkeiten in heterogenen Datenbanken zu wahren sind.

Die Dimension der Konsistenzabschwächung gewinnt besondere Bedeutung in einem Verbund autonomer Datenbanken ohne zentrale Kontrolle. Konsistenzabschwächung kann zeitlich oder extensional ausgedrückt werden, was in Abschnitt 2.5 diskutiert wird.

In Abschnitt 2.6 erörtern wir die Dimension Kontrollabhängigkeiten und illustrieren diese am Beispiel von redundanten Daten, die in autonomen Kontrollbereichen verwaltet werden. Ein anderer Aspekt von Kontrollabhängigkeiten in bezug auf Konsistenz betrifft das Verhältnis von lokalen Constraints aus unterschiedlichen Datenbanken. Dabei kann auch ein Konflikt entstehen zwischen lokalen Integritätsbedingungen, die bereits in einem DBS definiert sind und von diesem kontrolliert werden, und globalen Bedingungen. Die Fälle, die dabei auftreten können, werden besprochen.

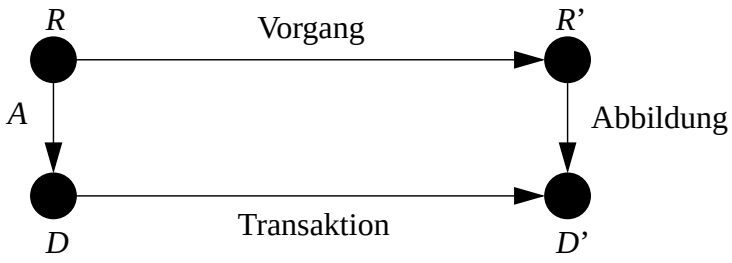
Außer den bereits diskutierten Wert- und Existenzabhängigkeiten gehen wir in Abschnitt 2.7 auch auf globale Struktur- und Verhaltensabhängigkeiten als weitere Kategorien globaler Integritätsbedingungen ein. Diese sind insbesondere dann interessant, wenn eine Menge von Datenbanken in einer Multidatenbank zusammen mit Metadaten verwaltet wird. Konsistenz bedeutet dann auch, daß die Metadaten die Semantik der zugrundeliegenden lokalen Objekte korrekt widerspiegeln.

Wir beenden dieses Kapitel mit einem kleinen Literaturüberblick über bekannte Vorschläge für die Modellierung globaler Integritätsbedingungen zwischen Daten, die auch als interdependente Daten bezeichnet werden (Abschnitt 2.8).

2.1 Der Konsistenzbegriff

Eine der Hauptaufgaben eines Datenbanksystems ist die Wahrung der Konsistenz der zu verwaltenden Daten, wobei die Kontrolle zentralisiert durch eine Komponente des Datenbanksystems erfolgt. Bis zur Einführung von Datenbanksystemen oblag die Kontrolle der Korrektheit der Daten den jeweiligen Anwendungsprogrammen, so daß jede Anwendung eigene Daten definierte und speicherte sowie deren Korrektheit überprüfte. Daraus resultierten als Hauptproblem eine redundante Speicherung von Daten und eine fehlende zentrale Kontrollinstanz, die eine Entscheidung über den korrekten Zustand aller Daten treffen konnte.

Um den Begriff der Konsistenz von seinem Wesen her zu erfassen, ist es sinnvoll, das Prinzip der Abbildung der realen Welt (*Universe of Discourse*) auf ein Datenmodell zu betrachten. Ein Objekt der Realwelt (aus einem Weltausschnitt) wird abgebildet auf ein Objekt eines Datenmodells, das durch ein Datenbankschema beschrieben ist. Ebenso lassen sich Vorgänge, die das Objekt verändern, auf entsprechende Transaktionen auf den korrespondierenden Datenbankobjekten abbilden (vgl. Abbildung 2.1).



R: Realitätsausschnitt (Miniwelt)
D: Datenbasis bezüglich einer Modellierung von *R*
A: Abbildung aller wichtigen Objekte und Beziehungen

Abbildung 2.1: Modellierung eines Realitätsausschnittes

Die Konsistenzanforderungen lassen sich ausdrücken durch eine Menge von Integritätsbedingungen im Modell. Durch sie ist eine möglichst gute Übereinstimmung von Objekten der Miniwelt und des Modells zu erzielen. In [Rei96] wird diese abstrakte Forderung als Kongruenz zwischen einer Ausprägung der Datenbasis *D* und einer Situation *S* in der Miniwelt *R* bezüglich einer Modellierung *M*(*S*) bezeichnet, die durch logikbasierte Ansätze erfaßt werden kann.

In [Wed89] wird die Problematik der Formulierung von Integritätsbedingungen aus dem Übersetzen von anwendungsspezifischen intensionalen Beschreibungstermini in extensionale und somit rechnergestützt nachprüfbare Eigenschaften von Objekten diskutiert. Die intensionale

Betrachtungsweise bezieht sich auf Eigenschaften (Struktur) von Begriffen, die extensionale auf deren Umfang, d.h. die Klasse der diese Eigenschaften aufweisenden Gegenstände. Durch die extensionale Beschreibung ist entscheidbar, ob ein Objekt einem Begriff zuzuordnen ist oder nicht. Die Datenbasis muß alle gültigen Situationen erfassen, was aber dazu führt, daß ungültige Situationen und die entsprechende Ausprägung in der Datenbasis durch die Definition von Integritätsbedingungen ausgeschlossen werden müssen.

In der Literatur taucht eine Vielzahl von Konsistenzbegriffen auf (siehe z.B. [BBC80, DKM85], zu deren Erklärung einige Grundbegriffe eingeführt werden sollen. Eine Einteilung von Konsistenzarten läßt sich nach dem Abstraktionsgrad bzw. der Systemebene vornehmen. Betrachtet wird dabei zunächst die Konsistenz in einem homogenen Datenbanksystem.

1. Ebene, auf der Konsistenz definiert wird
2. Zustandsorientierte vs. ablauforientierte Konsistenz
3. Geltungsbereich der Konsistenzbedingung

Ebenen der Konsistenz

Es werden drei Ebenen unterschieden, auf denen sich Integritätsbedingungen formulieren lassen. Dazu zählen die physische Ebene, die Datenmodell-Ebene und die semantische Ebene.

Die physische Datenintegrität bezeichnet die korrekte Abbildung von Datenstrukturen auf Einheiten physikalisch tieferer Ebenen, z.B. Sätze, Segmente und Dateien (Speicherkonsistenz). Diese ist fest an ein Datenmodell und eine DBMS-Implementierung gebunden und braucht deswegen nie vom Benutzer spezifiziert zu werden. Aspekte der physischen Integrität sollen deshalb an dieser Stelle nicht näher diskutiert werden.

Die Datenmodell-Ebene umfaßt alle modellinhärenten Integritätsbedingungen, d.h. Bedingungen, die sich mit den Mitteln des Datenmodells, z.B. der *Data Definition Language* (DDL), ausdrücken lassen. Somit lassen sich diese Bedingungen auch durch das verwendete DBS garantieren. Im folgenden sollen statische Eigenschaften betrachtet werden. Darunter befinden sich zwei Konsistenzeigenschaften, die von jedem Datenbanksystem eingehalten werden müssen: die Typkonsistenz und die referentielle Integrität. Unter Typkonsistenz versteht man, daß die Zugehörigkeit aller Objekte bzw. Werte zu einem Typ (im Sinne eines Wertebereichs) garantiert ist. Dabei hängen Art und Komplexität der Typen vom zugrundeliegenden Datenmodell ab. Referentielle Integrität umfaßt Bedingungen, die auf Beziehungen formuliert werden, die durch das Datenmodell definiert werden können (siehe auch Abschnitt 2.3.2). Zum dynamischen Teil der Integrität gehört die Einhaltung der Korrektheitskriterien des zugrundeliegenden Transaktionsmodells. Im einfachsten Fall könnte dies die Definition flacher Transaktionen als *Unit of Consistency* sein [Gra81].

Zur semantischen Ebene zählen Integritätsbedingungen (Synonym: *Constraints*), die vom Benutzer formuliert werden und sich aus den Erfordernissen der Anwendung ergeben. Diese Bedingungen erfassen die Korrektheit der modellierten Miniwelt im Modell aus Anwendungssicht. In [EN94] wird eine Einteilung in implizite und explizite Bedingungen angegeben, was davon abhängt, inwieweit die jeweilige DDL spezifische Schlüsselwörter für einzelne Constraints anbietet bzw. eine Bedingung als Prädikat oder Trigger beschrieben werden muß. Semantische Integritätsbedingungen lassen sich auch durch Datenbanksysteme kontrollieren, soweit geeignete Mechanismen zur Verfügung stehen (z.B. Trigger, ASSERT-Klauseln in relationalen Datenbanksystemen). Alternativ können sie in einer eigenständigen Ebene zwischen Applikation und Datenbank behandelt werden (vgl. hierzu Abschnitt 6.1.2 auf Seite 98).

Zustandsorientierte vs. ablauforientierte Konsistenz

Die zustandsorientierte (statische) Konsistenz umfaßt alle Integritätsbedingungen, die die Menge der möglichen Datenbankzustände einschränkt. Zu jedem Objekt der Datenbank kann zu einem beliebigen Zeitpunkt eine Aussage gemacht werden, ob die Menge der definierten Constraints erfüllt ist.

Die ablauforientierte (dynamische) Konsistenz umfaßt alle Integritätsbedingungen, die die möglichen Zustandsübergänge zu einem Zeitpunkt einschränken. In [Vos94] wird dabei unterschieden zwischen transitionalen oder halb-dynamischen Bedingungen sowie dynamischen Bedingungen (in [GA93] auch als temporale Bedingungen bezeichnet). Transitionale Bedingungen schränken Paare aufeinanderfolgender Zustände ein, dynamische Integritätsbedingungen können als Verallgemeinerung transitionaler Bedingungen auf beliebigen Zustandsfolgen ausgedrückt werden.

Geltungsbereich von Konsistenzbedingungen

Die folgende Unterteilung erfolgt unabhängig vom Datenmodell. Der Begriff Instanz findet seine Entsprechung z.B. als Tupel im Relationenmodell, als Record im Netzwerkmodell oder als Objekt in einem objektorientierten System (vgl. Abschnitt 2.3).

1. Attribut- oder Wertbereichsbedingungen
   Einschränkungen der Werte, die ein Attribut einer Instanz annehmen kann
2. Bedingungen, die mehrere Attribute einer Instanz verknüpfen
3. Bedingungen, die die Menge aller Instanzen eines Typs betreffen
4. Bedingungen, die Instanzen verschiedener Typen umfassen
5. Bedingungen, die sich über Instanzen mehrerer Datenbanken erstrecken

2.2 Eine Formalisierung des Konsistenzbegriffes

Wie bereits erwähnt, wird die Integrität einer Datenbank durch eine Menge von Integritätsregeln ausgedrückt. Dieser Abschnitt enthält eine formale Beschreibung der Integritätsbedingungen in einem objektorientierten Schema. Die gegebene Klassifikation erlaubt eine Abbildung auf andere Datenmodelle, wie z.B. das Relationenmodell.

2.2.1 Grundbegriffe eines Objektmodells

Ein Objektmodell basiert auf Klassen und Typen. Typen dienen der Beschreibung von Mengen komplexer Werte. Aus einer Menge von Basistypen lassen sich komplexe Typen bilden. Klassen beschreiben Mengen von Objekten, die eine eigenständige Identität besitzen und Struktur und Verhalten kapseln.

Zur strukturellen Beschreibung der Klassen eines Objektmodells ist ein Typsystem erforderlich, das Basistypen umfaßt sowie die Bildung komplexer Typen durch Anwendung von Typkonstruktoren gestattet. Eine rekursive Definition findet sich in [Vos94], die das Prinzip der Konstruktion eines solchen Typsystems veranschaulicht, wobei die Menge der Basistypen oder die Arten der zulässigen Typkonstruktoren in einzelnen Objektmodellen variiert werden können.

Definition 2.1 (Objekt-Typsystem *T*)
Ein Objekt-Typsystem T ist eine endliche Menge von Typen und wird rekursiv definiert durch:

1. Basistypen: `integer`, `string`, `float`, `boolean` $\subseteq T$
2. Tupel-Typ: $[A_1{:}t_1, \ldots, A_n{:}t_n] \in T$ für Attribute A_i und Typen $t_i \in T$, $1 \leq i \leq n$
3. Mengen-Typ: $\{ t \}$ für $t \in T$
4. Listen-Typ: $< t >$ für $t \in T$
5. C sei eine endliche Menge von Klassennamen, für die gilt: $C \subseteq T$

Die Menge aller Attribute, aus denen sich ein Typ t_i zusammensetzt, sei attr(t_i) = $\{A_1{:}t_1, \ldots, A_n{:}t_n\}$ mit Attributen A_j und Typen $t_j \in T$, $1 \leq j \leq n$.

Definition 2.2 (Domäne)
Die Domäne (*Domain*) ist der Wertebereich eines Typs $t_i \in T$, dom(t_i), und ist definiert als die Menge aller möglichen Werte des Typs.
Wenn angenommen wird, daß Objekte eine (systemverwaltete) Identität haben, durch die sie ausschließlich von anderen unterschieden werden können, ist die Einführung eines gesonderten Wertebereichs (Objekt-Domäne) notwendig. Dieser bezeichnet Mengen von Objekt-Identifikatoren (einschließlich eines nullwertigen Identifikators) *OID* und wird definiert als:

dom(c) = *OID* für jede Klasse $c \in C$

Definition 2.3 (Objektorientiertes Strukturschema)
Wir beschreiben ein objektorientiertes Datenbankschema strukturell nach [Vos94] wie folgt. Ein Strukturschema hat die Form S_{struc} = (C, T, type, isa) mit:

1. C ist eine endliche Menge von Klassennamen.
2. T ist eine endliche Menge von Typen.
3. type: $C \rightarrow T$ assoziiert einen Typ mit jedem Klassennamen.
4. isa kennzeichnet eine Vererbungsbeziehung auf Klassenebene, läßt sich auf eine Subtyp-Beziehung zurückführen.
   c isa $c' \Rightarrow$ type(c) $\leq$ type(c')
   Grundannahme der Relation type(c) $\leq$ type(c') ist, daß alle Attribute von c auch in c' vorkommen, die Typen gleichbenannter Attribute in c und c' Subtypen voneinander sein können und c neue Attribute enthalten kann, die in c' nicht enthalten sind [Vos94].

Hierbei werden eine Reihe von Aspekten außer Betracht gelassen, z.B. multiple Vererbung, die Unterscheidung in Instanz- und Klassenattribute und Sichtbarkeit von Attributen. Die Beziehungen zwischen Klassen und ihren Objekten sind hier auf Aggregation und Vererbung beschränkt.

Definition 2.4 (Verhaltensschema)
Ein Verhaltensschema hat die Form S_{behav} = (C, M, P, messg, impl) mit:

1. C ist eine endliche Menge von Klassennamen.
2. M ist eine endliche Menge von Message-Namen, zu jedem $m \in M$ gehört eine Menge sign(m) = $\{s_1, \ldots s_l\}$, $l \geq 1$, von Signaturen; jedes s_i ($1 \leq i \leq l$) hat die Form
   $$s_i\colon c \times t_1 \times \ldots \times t_p \rightarrow t \text{ (mit } c \in C, t_1 \ldots t_p \in T)$$
3. P ist eine endliche Menge von Methoden oder Programmen.
4. messg: $C \rightarrow 2^M$ ordnet jedem Klassennamen eine Menge von Methodennamen zu mit
   $(\forall\, c \in C)\ (\forall\, m \in \text{messg}(c))\ (\exists\, s \in \text{sign}(m))\ s[1] = c$

5. impl: $\{(m,c) \mid m \in \text{messg}(c)\} \rightarrow P$
   ist eine partielle Funktion, die Methoden eine Implementierung (als Programm) im Kontext von Klassen zuordnet.

Definition 2.5 (Schema einer Objektdatenbank)
Ein Schema einer Objektdatenbank setzt sich aus dem strukturellen Schema und dem Verhaltensschema zusammen: $S = (S_{struc}, S_{behav})$

Mit diesem Formalismus läßt sich nun der Begriff der Instanz einer Datenbank präzisieren.

Definition 2.6 (Objektdatenbank)
Eine Objekt-Datenbank über dem Schema S ist eine Menge von Klasseninstanzen

$d(S) = (O, \text{inst}, \text{val})$ mit:

1. $O \subseteq OID$: endliche Menge von Objekt-Identifikatoren
2. inst: $C \rightarrow 2^O$, bestimmt Zugehörigkeit von Objekten zu Klassen. Dabei gilt:
   a) $\text{inst}(c) \cap \text{inst}(c') = \emptyset$ (falls c und c' Wurzeln disjunkter Teilbäume der Vererbungshierarchie)
   b) für c isa c' gilt: $\text{inst}(c) \subseteq \text{inst}(c')$
3. val: $O \rightarrow V$, ordnet jedem Objekt einen Wert aus V zu:
   $(\forall\, c \in C)\ (\forall o \in \text{inst}(c))\ \text{val}(o) \in \text{dom}(\text{type}(c))$
   Die Menge V aller durch das Typsystem T festgelegten Werte ist wie folgt definiert:
   $V = \text{dom}(T) = \text{dom}(t_1) \cup \text{dom}(t_2) \cup \ldots \cup \text{dom}(t_n)$
   Eine Klasse c umfaßt somit eine Menge von Instanzen inst(c), die auch als *Extension* bezeichnet wird. Objekte einer Klasse c können durch einen Schlüssel key(c) eine benutzerdefinierte Identität aufweisen wobei gilt: $\text{attr}(\text{key}(c)) \subseteq \text{attr}(\text{type}(c))$.

Die Menge aller möglichen Datenbank-Instanzen eines Schemas S wird als Datenbank-Universum U_S bezeichnet.

Ein Datenbankzustand $d(S)$ ordnet einer Datenbank eines Schemas S einen Zeitpunkt θ zu, für den dieser Zustand Gültigkeit besitzt. Abbildung 2.2 faßt noch einmal die wesentlichen Bestandteile des eingeführten Objektmodells zusammen (nach [Vos94]).

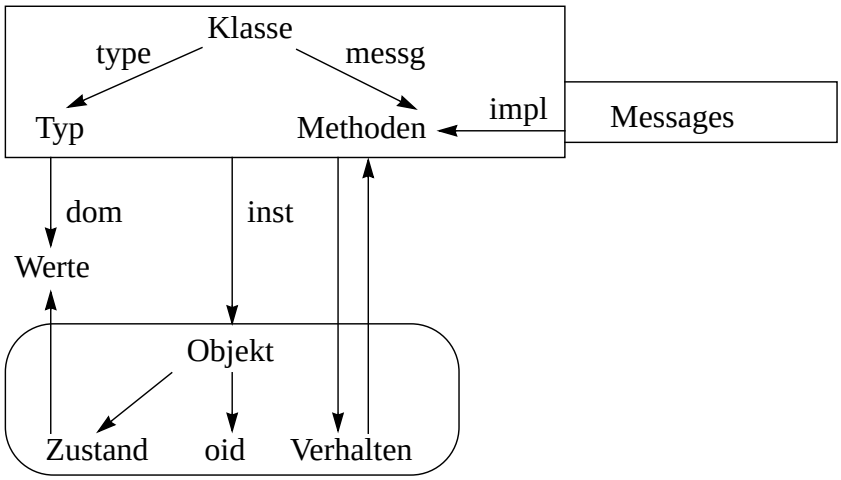


Abbildung 2.2: Konzepte des Objektmodells

Anwendbarkeit auf das Relationenmodell

Der eingeführte Formalismus soll von seiner Allgemeingültigkeit her auf andere Datenmodelle übertragbar sein, wie z.B. das Relationenmodell, das sich als Spezialfall eines Objekt-Typsystems ausdrücken läßt. Das relationale Typsystem umfaßt eine Menge von Basistypen, aus denen sich Relationen durch Anwendung des Tupel- und Mengen-Konstruktors konstruieren lassen.

1. Ein Typ t_i kann ein Basistyp sein: `integer`, `string`, `float`, `boolean`, oder:
2. t_i ist ein Relationenschema R, bestehend aus einem Namen und einer Attributliste der Form $[A_1{:}t_1, \ldots, A_n{:}t_n]$.
3. Es existieren keine Objekt-Domänen, deren Werte als Objekt-Identifikatoren dienen. Die Identifikation erfolgt über vom Benutzer ausgewählte Attribute bzw. Attributkombinationen.
4. Ein relationales Datenbankschema D ist demzufolge eine Menge von Relationenschemata $\{R_1, \ldots, R_n\}$. Im Unterschied zu Definition 2.3 ist keine Unterscheidung in Klassen und Typen sowie die Definition von Vererbungsbeziehungen vorgesehen.
5. Eine relationale Datenbank d ist eine Menge von Relationeninstanzen $\{r_1, \ldots, r_n\}$, wobei eine Relationeninstanz r_i einer Menge von Tupeln entspricht (analog zur extensionalen Sichtweise von Klassen im objektorientierten Modell).

2.2.2 Integrität in einem Objektdatenmodell

Basierend auf dem strukturellen Objektschema lassen sich Integritätsbedingungen formulieren, die Allgemeingültigkeit für alle Datenmodelle besitzen. In nachfolgenden Definitionen wird zwischen statischen und dynamischen Constraints unterschieden, die jeweils Aussagen über korrekte Datenbankzustände bzw. Datenbanktransitionen enthalten.

Definition 2.7 (Datenbank-Transition)
Eine Datenbank-Transition ist ein geordnetes Paar von Datenbankzuständen $<d_1,d_2>$, wobei 1 bzw. 2 als logische Zeitpunkte θ aufgefaßt werden, d.h. zwei aufeinanderfolgende Zustände der Datenbank. Ein Zustandsübergang wird durch ein Datenbankereignis E ausgelöst.

Bei einem Zustandsübergang könnnen Integritätsbedingungen verletzt werden. Änderungen in der Datenbank können verstanden werden als eine Abbildung der Menge aller Datenbankzustände in sich selbst, notiert als: $U_S \rightarrow U_S$.

Der nachfolgend beschriebene Formalismus für Objektdatenbanken führt drei Basisoperationen ein, nämlich Einfügen, Löschen und Ändern, die insgesamt die Menge der Datenbank-Ereignisse bilden. Diese Basis-Ereignisse sind generisch definiert und können somit auf konkrete logische Datenmodelle (z.B. Relationenmodell, ODMG-93) abgebildet werden.

Definition 2.8 (Datenbankereignisse)
Sei S = (C, T, M, P, type, isa, messg, impl) ein objektorientiertes Datenbankschema, auf dem eine Objekt-Datenbank $d(S)$ definiert ist. Folgende Datenbankereignisse E können zu Datenbanktransitionen führen:

a) Eine Einfügung (insert c) über S fügt eine Instanz $o \in$ inst(c) ein.

b) Eine Löschung (delete c) über S entfernt eine Instanz $o \in$ inst(c).

c) Eine Änderung (update *c*) über *S* verändert ein oder mehrere Attributwerte A_i einer Instanz $o \in \text{inst}(c)$.[1]

Definition 2.9 (Statische Integritätsbedingung)
Gegeben ist ein Datenbankschema *S*. Eine statische Integritätsbedingung (Constraint) *IS* ist eine boolesche Funktion, die auf einem Datenbankzustand *d* aus dem Datenbank-Universum U_S ausgewertet wird.

$$IS\colon U_S \rightarrow \text{bool}$$

Definition 2.10 (Korrekter Datenbankzustand)
Ein korrekter Datenbankzustand $d \in U_S$ erfüllt jedes Constraint aus einer Menge von statischen Integritätsbedingungen $IS = \{IS_1, \ldots, IS_m\}$, definiert auf *S*. Die Menge korrekter Datenbankzustände auf dem Schema *S* und der Constraint-Menge *IS* wird definiert durch:

$$U_S{}^{IS} = \left\{ d \in U_S \;\middle|\; \bigwedge_{i=1}^{i=m} IS_i(d) \right\}$$

Hierdurch werden die statischen Eigenschaften einer Datenbank spezifiziert, die zu irgendeinem Zeitpunkt erfüllt sein müssen.

Definition 2.11 (Dynamische Integritätsbedingung)
Gegeben ist ein Datenbankschema *S*. Eine dynamische Integritätsbedingung *IT* ist eine boolesche Funktion, die auf einer Datenbank-Transition, d.h. einem Paar von Datenbank-Zuständen $<d_1,d_2>$ ausgewertet wird.

$$IT\colon U_S \times U_S \rightarrow \text{bool}$$

Definition 2.12 (Korrekte Datenbanktransition)
Eine korrekte Datenbanktransition $<d_1,d_2>$ erfüllt alle Constraints aus einer Menge von dynamischen Integritätsbedingungen $IT = \{IT_1, \ldots, IT_n\}$, definiert auf *S*. Die Menge korrekter Datenbanktransitionen auf dem Schema *S* und der Constraint-Menge *IT* wird definiert durch:

$$V_S{}^{IT} = \left\{ <d_1,d_2> \in U_S \times U_S \;\middle|\; \bigwedge_{i=1}^{i=n} IT_i(d_1,d_2) \right\}$$

Die dynamischen Eigenschaften einer Datenbank werden also durch Transitionsconstraints beschrieben.

Die Integrität einer Datenbank, wie sie durch eine Menge von Integritätsbedingungen definiert wurde, beeinflußt auch die Transaktionen *T*, die auf der Datenbank *d* ausgeführt werden können und zu einem korrekten Datenbankzustand führen müssen, d.h. alle statischen Constraints müssen am Ende der Transaktion (im Zustand *T*(*d*)) wieder erfüllt sein. Ebenso darf die Ausführung einer Transaktion kein transitionales Constraint verletzen

$$IS(d) \Rightarrow IS(T(d)) \wedge IT(d) \Rightarrow IT(d, T(d))$$

Definition 2.13 (Korrekte Datenbanktransaktion)
Eine Transaktion *T* ist *korrekt* in Bezug auf einen korrekten Datenbankzustand *d* und eine Menge von Integritätsbedingungen genau dann, wenn eine beendete (committed) Ausführung von *T* auf *d* keine Datenbanktransition bewirkt, die irgendein Transitionsconstraint in *IT* ver-

1 Hierbei gelte die Annahme $\text{dom}(A_i) \neq \text{OID}$ (d.h. kein identifizierendes Attribut)

letzt, sowie der Datenbankzustand nach dem Ende von *T*, *T*(*d*), alle statischen Constraints *IS* erfüllt.

2.3 Eine Taxonomie für Integritätsbedingungen

Die vorgestellte Taxonomie teilt Integritätsbedingungen nach ihrem Geltungsbereich ein, wobei nicht nach statischen oder dynamischen Constraints unterschieden wird und nur homogene Datenbanken betrachtet werden. Constraint-Klassifikationen, wie sie für das Relationenmodell vorgeschlagen wurden [GA93], sind ein Teil dieser Taxonomie, die in einem objektorientierten Kontext formuliert wurde. Dabei ist zweierlei zu beachten: Bestimmte Konsistenzbedingungen brauchen in einem objektorientierten Modell nicht mehr explizit modelliert zu werden, da diese modellinhärent sind (z.B. Inklusionsbeziehungen zwischen Instanzen von Super-/Subtypen). Andererseits werden Objekte in einem Objektmodell bei der Abbildung auf das Relationenmodell häufig durch Normalisierung auf mehrere Tabellen aufgeteilt.

2.3.1 Klassen-Regeln

Nachfolgend werden Constraints betrachtet, die auf Instanzen einer Klasse beschränkt sind und auch als Objekt-Constraints bezeichnet werden können. Diese Constraints lassen sich noch danach unterscheiden, ob sie innerhalb einer Instanz gelten sollen (Intra-Instanz-Constraints) oder instanzenübergreifend (Inter-Instanzen-Constraints) [AQFG95, BCV86].

2.3.1.1 Intra-Instanzenregeln

- Domänen- oder Attributbedingungen
  Diese Bedingungen schränken die Werte ein, die ein Attribut in einer Instanz der Datenbank annehmen kann.[2] Folgende Typen lassen sich hierbei unterscheiden:
  a) Festlegung von Ober- bzw. Untergrenzen für die Werte eines numerischen Attributes, als Beispiel gelte die Bedingung: $(\forall\ e \in \text{Employee})\ 18 < e.age < 120$.
  b) Festlegung bestimmter Werte (im Sinne einer Aufzählung) für ein alphanumerisches Attribut, beispielsweise gilt: $(\forall\ e \in \text{Employee})\ e.sex \text{ in } \{m,f\}$.
  c) Not-Null-Bedingungen schließen die Verwendung von Null-Werten bei bestimmten Attributen aus: $(\forall\ e \in \text{Employee})\ e.empnr \text{ not null}$.
- Implikationsregeln
  Durch Implikationsregeln werden Abhängigkeiten zwischen Attributwerten eines Objektes spezifiziert, d.h. der Wert eines Attributs hat einen direkten Einfluß auf den Wert eines anderen Attributs in derselben Instanz. Funktionale Abhängigkeiten in Relationen lassen sich dem zuordnen.
  Ein Beispiel wäre: Alle Programmierer arbeiten in Darmstadt.
  $(\forall\ e \in \text{Employee})\ e.occupation = \text{“Programmer”} \rightarrow e.office = \text{“Darmstadt”}$.
  Implikationsregeln lassen sich verallgemeinert für beliebige Instanzen aus einer Klasse oder verschiedenen Klassen formulieren. Wenn zwei Instanzen o und o’ in einem Attribut X übereinstimmen, so gilt für deren Werte o.Y und o’.Z: $o.Y \rightarrow o'.Z$.
- Inter-Attributbedingungen

2 Von manchen Autoren, z.B. [Gal81] werden diese Bedingungen auch als Typ-Constraints bezeichnet.

Durch Inter-Attributbedingungen werden Constraints zwischen zwei oder mehr Attributen ausgedrückt.
Zum Beispiel: Das Gehalt eines Angestellten ist größer/gleich seinem Anfangsgehalt.
$(\forall\ e \in \text{Employee})\ \text{Employee.salary} \geq \text{Employee.initial\_salary}$

2.3.1.2 Inter-Instanzenregeln

Im Unterschied zur vorigen Kategorie muß bei diesen Regeln bei einem Update auf einem Objekt einer bestimmten Klasse eine Prüfung mehrerer oder sogar aller Instanzen der Klasse auf Einhaltung der Bedingung erfolgen.

- Berechenbare Bedingungen (Aggregatbedingungen)
  Die Überprüfung dieser Bedingungen erfordert die Berechnung von Werten aus allen Instanzen einer Klasse (auch als Aggregat-Bedingungen bezeichnet, da die Berechnung auf einer Aggregierung von Einzelwerten beruht).
  Ein typisches Beispiel: Die Summe der Gehälter der Angestellten einer Abteilung darf eine bestimmte Summe nicht überschreiten.
  $(\forall\ e \in \text{Employee})\ \text{sum(e.salary)} \leq 500.000$

- Nicht-Berechenbare Bedingungen (Rekursive Bedingungen)
  Hierbei handelt es sich um Constraints, die mehrere Instanzen einer Klasse betreffen. Dabei ist eine Unterscheidung zwischen direkt rekursiven Beziehungen und transitiv rekursiven Beziehungen möglich.
  Ein typisches Beispiel direkt rekursiver Beziehungen sind referentielle Integritätsbedingungen zwischen Objekten derselben Klasse. Der Manager eines Angestellten ist auch ein Angestellter - unter der Annahme, daß Top-Manager als Angestellte modelliert sind, die ihre eigenen Vorgesetzten sind.
  $(\forall\ e \in \text{Employee})\ ((\exists\ em \in \text{Employee})\ \text{e.manager} = \text{em.name})$
  Transitiv rekursive Constraints drücken den rekursiven Abschluß einer Beziehung aus. Ein charakteristisches Beispiel läßt sich in einer Datenbank für Zugverbindungen finden: Jeder Knoten des Netzwerks ist von jedem anderen aus erreichbar.

- Schlüsselbedingungen (Uniqueness)
  Diese Constraints beschreiben die Schlüsseleigenschaft für ein oder mehrere Attribute, d.h. sie müssen in einer Klasse einzigartige und somit eindeutige Werte aufweisen.
  Dies ist z.B. bei Personalnummern von Angestellten der Fall, für die gelte:
  $(\forall\ e1 \in \text{Employee})\ ((\neg\exists\ e2 \in \text{Employee})\ e1 \neq e2 \wedge \text{e1.empnr} = \text{e2.empnr})$

2.3.2 Klassenübergreifende Regeln

Diese Kategorie von Regeln bezieht Instanzen von mehreren Klassen ein, was insbesondere für die Modellierung von Beziehungen zwischen Klassen relevant ist (unabhängig davon ob im relationalen oder einem objektorientierten Modell). In objektorientierten Datenmodellen gehören auch Constraints auf Attributen, die Beziehungen zu anderen Klassen herstellen, zu klassenübergreifenden Integritätsbedingungen.

- Referentielle Integritätsbedingungen
  Referentielle Integritätsbedingungen drücken aus, daß ein Objekt, das durch ein anderes referenziert wird, auch tatsächlich existiert. Die Referenz kann hergestellt werden durch vom System verwaltete Objekt-Identifikatoren (in objektorientierten DBMS) oder durch Fremdschlüssel (in relationalen Systemen). Ein Hauptproblem bei der Wahrung referentieller Integrität besteht beim Löschen eines Objekts, auf das möglicherweise noch Ver-

weise bestehen. Für die Delete-Behandlung gibt es mehrere Optionen, die auch Eingang in den SQL92-Standard gefunden haben. Die Löschung eines referenzierten Objektes kann durch ein Blockieren der verletzenden Aktion verhindert werden (RESTRICT) oder zu den abhängigen Objekten propagiert werden (CASCADE). Denkbar sind auch das Zuweisen eines Default- oder Null-Wertes (NULLIFY) an das referenzierende Attribut.
Im Beispiel wird gefordert, daß zu jedem Beschäftigten eine Abteilung existieren muß.
$(\forall\ e \in \text{Employee})\ ((\exists\ d \in \text{Department})\ e.department = d.name)$

- Spezialfall: Inverse Beziehungen (Relationale Integrität)
Binäre Beziehungen lassen sich in objektorientierten Modellen bidirektional modellieren, d.h. beide Richtungen dieser Beziehung werden spezifiziert. Relationale Integrität läßt sich für zwei beliebige Objekte *a* und *b*, die in einer Beziehung *R* stehen, ausdrücken [JQ92]: $a\ R\ b \rightarrow a\ R^{-1}\ b$ (R^{-1} inverse Beziehung). Dafür existiert eine *Inverse*-Klausel, die in einer Reihe von Systemen (z.B. ObjectStore) Verwendung findet und auch Eingang in den ODMG-93-Standard gefunden hat. Hierdurch kann ein Traversierungspfad durch Angabe von Klasse und Attribut definiert werden.
Die bidirektionale Modellierung zwischen einem Angestellten und seiner Abteilung könnte z.B. so aussehen:
Class Employee: employed:Department inverse Department::employs
Class Department: employs:set<Employee> inverse Employee::employed
Die Schreibweise als prädikatenlogischer Ausdruck:
$(\forall\ e \in \text{Employee})\ ((\exists\ d \in \text{Department})\ e.employed = d \wedge e\ in\ d.employs)$

- Komplexitätsrestriktionen
Beziehungen zwischen zwei Klassen können durch Komplexitätsbedingungen (in manchen Modellen auch als Multiplizität bezeichnet [RBP+91]) näher spezifiziert werden, d.h. es werden Restriktionen definiert, wie oft ein Objekt mindestens bzw. höchstens an einer Beziehung zu Objekten anderer Klassen teilnehmen kann. In einem objektorientierten Datenmodell kann in 1:n- und m:n-Beziehungen die Anzahl der Elemente des mengenwertigen Attributes, das die Beziehung realisiert, eingeschränkt werden.
Zum Beispiel ist gefordert, daß jeder Angestellte an mindestens drei Projekten arbeitet.
$(\forall\ e \in \text{Employee})\ count(e.projects) \geq 3$
In einem relationalen Modell werden Beziehungen durch eigenständige Tabellen realisiert. Das obige Beispiel könnte dann folgendermaßen ausgedrückt werden:
$(\forall\ e1 \in \text{Emp})\ ((\exists\ e2 \in \text{EmpProj})\ e1.empnr = e2.empnr\ and\ count\ (e2) \geq 3)$

- Navigationale Integritätsbedingungen
Solche Bedingungen treten in komplexen Objekten auf, wo die Auswertung über eine Verknüpfung zu anderen Klassen ermöglicht wird. In einer Aggregationshierarchie eines objektorientierten Datenmodells ist die Navigation entlang des Referenzpfades der Komponentenobjekte erforderlich, im relationalen Modell ist ein Join-Prädikat auszuwerten.
Zum Beispiel soll gelten: Das Gehalt des Präsidenten eines Unternehmens soll nicht 500.000.- übersteigen.
Schreibweise mit geschachtelten Attributen:
$(\forall\ c \in \text{Company})\ (c.president.salary < 500\ 000)$
Schreibweise im Relationenmodell:
$(\forall\ c \in \text{Company})\ ((\exists\ e \in \text{Employee})\ c.president = e.name \wedge e.salary < 500\ 000)$

- Strukturelle Constraints
Strukturelle Constraints legen die Struktur komplexer Objekte fest. Dazu zählen Typ und Anzahl der Komponenten sowie Festlegungen der Topologie des Aggregats [BCV86].

Diese lassen sich in OO Datenmodellen durch die Schemadefinition festlegen, im relationalen Modell müssen Aggregatbedingungen und benutzerdefinierte Werte zur Festlegung einer Ordnung eingeführt werden.

Die gegebene Klassifizierung, die nach wachsender Reichweite der Bedingungen geordnet wurde, berücksichtigt somit, daß geschachtelte oder mengenwertige Attribute, die Verweise auf Objekte anderer Klassen beinhalten, den klassenübergreifenden Constraints zugeordnet werden müssen, vgl. auch [KLS92].

Für die eingeführte Klassifikation ist zu beachten, daß keineswegs immer eine 1:1-Abbildung zwischen den Objekten des konzeptionellen und des logischen Datenmodells gilt. Bei Abbildung eines strukturell objektorientierten Schemas (z.B. eines erweiterten ER-Diagramms) auf ein relationales Schema können aus einer Klasse durch Normalisierung mehrere Tabellen entstehen, so daß aus Intra-Instanzenbedingungen interrelationale Constraints werden können.

Die hier vorgestellten Aggregat- und Uniqueness-Constraints sind auch anwendbar auf Objekte verschiedener Klassen, z.B. wenn datenbankweit eindeutige Schlüssel vergeben werden sollen. Darauf wird in Abschnitt 2.4.3 noch einmal eingegangen, wenn Erweiterungen der Klassifikation für heterogene Datenbanken diskutiert werden.

2.4 Konsistenz in heterogenen Datenbanken

2.4.1 Allgemeines

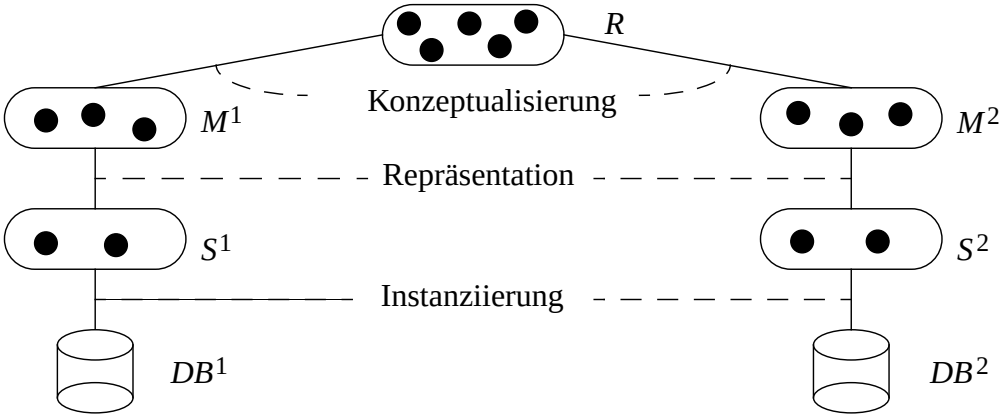
Wie bereits zu Beginn dieses Kapitels hervorgehoben wurde, kann eine Datenbank als ein Abbild der realen Welt angesehen werden, wobei durch Konsistenz eine widerspruchsfreie und korrekte Repräsentation der Realwelt-Objekte bezeichnet wird. Bei der Betrachtung mehrerer Datenbanken besteht oft das Problem, daß jeweils Ausschnitte der realen Welt modelliert werden, die sich teilweise überlappen können. Diese Überlappungen betreffen einen wesentlichen Aspekt der Konsistenz in heterogenen Datenbanksystemen. Wird ein und dasselbe Realwelt-Objekt durch mehrere Datenbanken modelliert, so liegen sie dort als “semantische Replikate” vor, zwischen denen die Konsistenz unter Berücksichtigung der heterogenen Repräsentation zu wahren ist. Erstmalig wurde ein Modellierungsansatz dafür durch Einführung sogenannter Identity Connections in [WQ87] vorgestellt. Redundante Daten, d.h. identische Kopien von Datenelementen in unterschiedlichen Datenbanken, können als ein Spezialfall semantischer Replikate betrachtet werden.

Natürlich sind auch Situationen möglich, bei denen semantische Replikate in einer zentralen homogenen Datenbank verwaltet werden, beispielsweise in einer Entwurfsdatenbank, in der das gleiche Objekt in zwei unterschiedlichen Formen modelliert wird (z.B. Pixeldarstellung vs. Begrenzungsdarstellung). Da die Konsistenzkontrolle hier jedoch dem zugrundeliegenden DBMS obliegt und ein Konflikt mit lokaler Autonomie nicht auftritt, werden solche Fälle an dieser Stelle hier nicht weiter untersucht.

Darüber hinaus existieren auch Beziehungen zwischen Objekten unterschiedlicher Datenbanken, die in der Realwelt voneinander verschieden sind. Abhängigkeiten zwischen Daten in heterogenen und autonomen Datenbanksystemen lassen sich auf der semantischen Konsistenzebene durch globale Integritätsbedingungen beschreiben, für die die eben eingeführte Taxonomie angewandt werden kann. Nicht betrachtet werden nachfolgend Konsistenzbedingungen, wie sie typischerweise für horizontal und vertikal fragmentierten Daten in homogenen verteilten Datenbanksystemen auftreten

2.4.2 Heterogenität als Hauptfaktor globaler Integritätsbedingungen

Um datenbankübergreifende Konsistenzbedingungen zu spezifizieren, sollen zunächst einige Grundbegriffe eingeführt werden, um unterschiedliche Repräsentationen der Daten zu erfassen und somit auch semantische bzw. syntaktische Heterogenität beschreiben zu können.



R: Realwelt = { Realwelt-Objekt1, Realwelt-Objekt2, . . . }
M: Modellwelt = { Konzept1, Konzept2, . . . }
S: Datenbankschema = { Klasse1, Klasse2, . . . }
DB: Datenbank = { Objekt1, Objekt2, . . . }

Abbildung 2.3: Modellierung eines Weltausschnittes in zwei Datenbanken

Ein Weltausschnitt kann auf verschiedene Weise modelliert und damit in einer Datenbank gespeichert sein. Dies ist z.B. der Fall bei CAD-Datenbanken, wo dasselbe Realwelt-Objekt unterschiedlich modelliert und repräsentiert sein kann (z.B. graphische vs. analytische Darstellung). Abbildung 2.3 zeigt, wie derselbe Weltausschnitt in unterschiedlichen (voneinander unabhängigen) Datenbanken gespeichert sein kann, was typisch ist für historisch gewachsene Informations-Infrastrukturen (Legacy-Systeme).

Ein Objekt der Realwelt wird auf ein *Konzept* (d.h. ein Objekt in einem konzeptuellen Datenmodell) abgebildet, das beim Entwurf des logischen Datenmodells durch ein Datenbankobjekt repräsentiert wird. In unserer gewählten objektorientierten Darstellung sind die Datenbankobjekte Klassen bzw. Attributen gleichzusetzen, deren *Semantik* (oder *Intension*) durch das zugrundeliegende Konzept ausgedrückt wird. Die Extension einer Klasse entspricht der Menge aller Instanzen dieser Klasse. Zwei Klassen aus verschiedenen Datenbanken können in einer semantischen Beziehung stehen, und zwar dann, wenn sie dasselbe Konzept vollständig oder teilweise repräsentieren. Beide Klassen bezeichnet man als korrespondierende Objekte, die Beziehung nennen wir *semantische Äquivalenz*. Dabei sind mehrere Fälle zu unterscheiden:

a) Semantisch äquivalente Klassen können unterschiedliche Extensionen aufweisen und somit auch verschiedene Realwelt-Objekte repräsentieren. Ein Beispiel dafür ist eine Klasse Employees zur Speicherung von Angestellten. Denkbar ist, daß eine solche Klasse in zwei Datenbanken jeweils unterschiedliche Angestellte umfaßt, obwohl dasselbe Konzept repräsentiert ist. Beide Klassen können sich aber im Kontext[3] unterschieden, d.h. die zugrundeliegenden Realwelt-Ausschnitte unterscheiden sich (z.B. alle Angestellten der Konzernzentrale, alle Angestellten im Tochterunternehmen).

3 Der Begriff “Kontext” wird von manchen Autoren unterschiedlich verwendet, hier nach [GSC95].

b) Wenn überlappende Weltausschnitte in unterschiedlichen Datenbanken modelliert sind, so stimmen auch die Extensionen von semantisch äquivalenten Klassen überein. Dabei bezeichnen wir die mehrfach verwalteten Objekte dieser Klassen auch als *semantische Replikate*. Der Zusatz semantisch ist hierbei wichtig, da im Unterschied zu herkömmlichen Replikaten auf der syntaktischen Ebene (vgl. Abschnitt 8.5) durchaus Unterschiede auftreten können. Ein Beispiel könnte eine Klasse Student sein, die in zwei verschiedenen Datenbanken an der Universität gepflegt wird, wobei jeder Student in beiden Datenbanken erfaßt sein muß und gemeinsame und differierende Attribute in jeder Datenbank hat.

c) Umgekehrt sind Situationen denkbar, in denen zwei Klassen semantisch nicht äquivalent sind und trotzdem in einer Beziehung durch ihre Extensionen stehen. In unserem Beispiel, das aus einer Klasse Professor und einer Klasse Employees_Wearing_Eyeglasses besteht, könnten diese Extensionen sogar übereinstimmen.

Darüber hinaus unterscheiden Sheth und Kashyap neben semantischer Äquivalenz weitere Formen semantischer Verwandtschaft (semantische Beziehung, semantische Relevanz, semantische Ähnlichkeit). Zugrunde liegt hierbei eine Klassifizierung von Abstraktionsbeziehungen zwischen Domänen bzw. Klassen, von denen die totale 1:1-Abbildung, in [VA96] auch als *Identity* bezeichnet, die Grundlage semantischer Äquivalenz bildet. Nachfolgend werden wir speziell diesen Fall weiter betrachten, da er in vielen Applikationsszenarien häufig auftritt, z.B. bei redundant gespeicherten Daten mit einer Replikationsbeziehung.

Um den Einfluß der Heterogenität auf die Wahrung globaler Konsistenz genauer beschreiben zu können, soll in den nächsten Abschnitten erst einmal betrachtet werden, welche Fälle von Heterogenität dabei für uns relevant sind. Für die Beschreibung der einzelnen Fälle verwenden wir wiederum die objektorientierte Terminologie im Hinblick auf die folgende Umsetzung in einen objektorientierten Vermittler (vgl. Abschnitt 5.4). Eine sehr gute Darstellung dieser Problematik in heterogenen Datenbanken findet sich u.a. in [Mad95], [GSC95] und [SK93].

Die Klassifizierung der Heterogenität kann auf zwei Ebenen erfolgen: auf Schemaebene und auf Ausprägungsebene.

2.4.2.1 Semantische Heterogenität auf Klassenebene

Wir gehen davon aus, daß zwei Klassen miteinander korrespondieren, d.h. sie haben die gleiche Bedeutung. Trotzdem können sich Unterschiede in ihren Extensionen ergeben, wie bereits festgestellt wurde. Mitunter ist es sehr schwierig, semantisch äquivalente Klassen in einer heterogenen Umgebung zu identifizieren, diese Problematik ist z.B. vertieft in [Cas93]. Unterschiede, die sich aus heterogenen Namen, Strukturen und Methoden in den einzelnen Datenbanken ergeben, sollen aus diesem Grunde hier auch nicht diskutiert werden.

1. Unterschiedliche Klassenextensionen
Es wird davon ausgegangen, daß auf die Extensionen semantisch äquivalenter Klassen Mengenoperatoren angewendet werden können, um deren Verhältnis zu beschreiben. In [GSC95] findet sich dazu eine Darstellung der möglichen Beziehungen zwischen den Extensionen (siehe Abbildung 2.4). Extensionale Äquivalenz zwischen zwei Extensionen bedeutet, daß jede Instanz der Extension von c_1 auch in der Extension von c_2 enthalten sein muß und umgekehrt. Strikte Inklusion zwischen c_1 und c_2 entspricht einer echten Teilmengenbeziehung $\text{inst}(c_1) \subseteq \text{inst}(c_2)$. Für die Überlappungsbeziehung gilt: $\text{inst}(c_1) \cap \text{inst}(c_2) \neq \emptyset$; entsprechend gilt für disjunkte Extensionen, daß deren Schnittmenge leer ist.

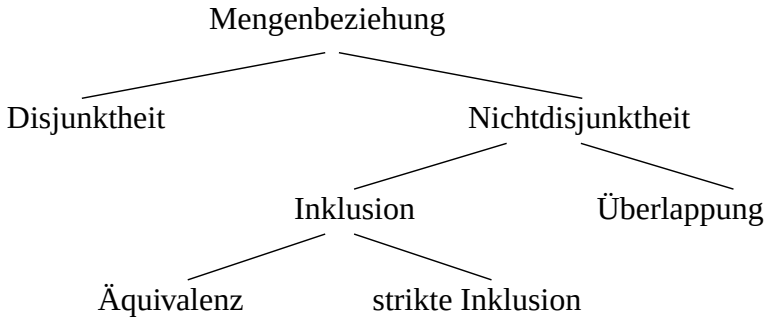


Abbildung 2.4: Beziehungen zwischen Klassenextensionen

2. Unterschiede in den Attributen
Zwei semantisch äquivalente Klassen können Attribute aufweisen, die zwar gleiche Wertebereiche besitzen, aber unterschiedliche Konzepte repräsentieren, d.h. sich in ihrer Semantik unterscheiden. Wir betrachten hierzu instanzspezifische Attribute. Typische Fälle, die auftreten können, sind:

- Unterschiedliche Existenz von Attributen
  Ein Attribut existiert nur in einer von zwei Klassen, die miteinander korrespondieren.
- Unterschiedliche Attributsemantik
  - Unterschiedliche temporale Semantik
    Beispielsweise hat in zwei Angestellten-Datenbanken die Gehaltsangabe jeweils zu unterschiedlichen Zeitpunkten Gültigkeit.
    DB1.Employee.salary aktueller Wert des Gehaltes
    DB2.Employee.salary Gehalt, gültig zum 1.1. 1995
  - Unterschiedliche Interpretation der Attribute
    Ein typisches Beispiel sind Gehaltsangaben, die brutto oder netto gespeichert sein können oder auch Informationen über Preise, die mit oder ohne Mehrwertsteuer angegeben werden können, z.B. die Preisangabe in DB1 stimmt mit der in DB2 überein zuzüglich Mehrwertsteuer: DB1.Article.price = DB2.Article.price * 1.15
- Unterschiede in den Attributconstraints
  - Nullwerte erlaubt $\leftrightarrow$ nicht erlaubt
  - Uniqueness-Constraint (Einzigartigkeit) durch Attribut erfüllt $\leftrightarrow$ nicht erfüllt
  - Attribut ist einwertig $\leftrightarrow$ mehrwertig

3. Unterschiede in den Domänen
Die Domänen lassen sich nach semantischen und syntaktischen Domänen unterscheiden. Eine semantische Domäne umfaßt die Werte eines Typs im konzeptionellen Modell. In Analogie zur semantischen Äquivalenz zwischen Klassen läßt sich auch eine semantische Korrespondenz zwischen Domänen angeben, wenn sie dasselbe Konzept repräsentieren (vgl. SK93]). Für diesen Zweck ist eine Abbildungsfunktion notwendig, die eine Zuordnung jedes Wertes des einen Wertebereichs zu einem Wert des anderen Wertebereichs vornimmt. Da die Schemata auf heterogenen Datenbanken mit unterschiedlichen Typsystemen implementiert sein können, können sich Domänen syntaktisch unterscheiden, auch wenn sie von der Semantik her äquivalent sind. Auf der Datenbanksystemebene ist eine Unterscheidung zwischen semantischen und syntaktischen Domänen allerdings nicht unbedingt möglich.

a) Semantische Differenzen zwischen Domänen

- Unterschiedliche Instanzen-Identifikation
  Eine sehr häufig auftretende Schwierigkeit besteht darin, daß Instanzen in mehreren Datenbanken auf unterschiedliche Weise identifiziert werden (in [WM89] als Instanzenidentifikationsproblem beschrieben). Wir können dieses Problem auf zwei Ebenen, der semantischen und der Datenmodellebene, betrachten. Auf der semantischen Ebene werden Instanzen benutzerabhängig identifiziert, d.h. die Wertebereiche der Schlüsselattribute unterscheiden sich voneinander.
  DB1.Company.compno = 3842
  DB2.Company.code = “OPEL”
  Das Beispiel zeigt eine Identifikation von Herstellern, einmal durch einen Namen, zum anderen durch einen numerischen Schlüssel. Ein Problem bei der Verwendung von Codes entsteht dadurch, daß die Codes in verschiedenen Datenbanken differieren können (durch Verwendung unterschiedlicher Verschlüsselungssysteme).
  Beide Datenbankobjekte sind semantisch äquivalent und korrespondieren mit demselben Realwelt-Objekt. Dafür ist in Analogie zur Wertabbildung eine Objekt-Abbildungsfunktion auf Objekt-Identifikatoren notwendig, die jedes Objekt einer Datenbank einem Objekt einer anderen Datenbank zuordnet.
  Unterschiede auf der Datenmodell-Ebene sind typisch bei der Verwendung einer relationalen und einer objektorientierten Datenbank. In der relationalen Datenbank werden die Instanzen durch vom Benutzer definierte Identifikatoren (Schlüssel), identifiziert, wohingegen in objektorientierten DBMS interne System-Identifikatoren (OID’s / Surrogate) Verwendung finden.

- Korrespondenz zwischen Wertebereichen von beschreibenden Attributen
  Es lassen sich über die Korrespondenz zwischen Objekt-Domänen hinaus auch andere Beziehungen zwischen Wertebereichen von Nichtschlüssel-Attributen finden. Wenn es sich um Wertebereiche handelt, die nur Literale umfassen (d.h. keine eigenständig identifizierbaren Objekte) so lassen sich bei nichtnumerischen Werten oft direkte Beziehungen angeben. Zum Beispiel kann eine Datenbank in verschiedenen Ländern eingesetzt sein, wobei die Werte der Landessprache angepaßt sind. Beispiel:
  DB1.Employee.rating = {“hervorragend”, “befriedigend”, “unbefriedigend”}
  DB2.Employee.rating = {“outstanding”,”satisfactory”,”unsatisfactory”}

- Unterschiedliche Wertgranularität
  Ein Attribut, das semantisch äquivalente Sachverhalte ausdrückt, kann in mehreren Datenbanken eine unterschiedliche Wertgranularität aufweisen. Das Problem tritt vor allem dann auf, wenn kontinuierliche Wertebereiche in diskrete Wertebereiche umgewandelt werden und dabei verschiedene (oft subjektiv bestimmte) Kriterien angewandt werden. In [GSC95] findet sich dazu ein ausführliches Beispiel für die heterogene Modellierung eines Attributes Farbe. Wir demonstrieren die Differenz von Granularitäten wiederum am Beispiel der Einstufung eines Angestellten.
  DB1.Employee.rating = {“gut”,”nicht gut”}
  DB2.Employee.rating = {1, 2, 3, 4, 5}

- Unterschiede in Dimension, Maßeinheit und Skalierung (nur bei numerischen Domänen)
  Ein Beispiel für unterschiedliche Dimension ist die Quantifizierung einer Menge von Äpfeln, entweder nach Gewicht, Stückzahl oder Preis.
  Unterschiedliche Maßeinheiten könnten z.B. bei Entfernungsangaben auftreten, die entweder in Kilometern oder in Meilen erfolgen können.

Werte, die durch Dimension und Maßeinheit charakterisiert sind, können durch einen Skalierungsfaktor an die Bedürfnisse einer lesbaren Darstellung (wie Vermeidung großer Zahlen) angepaßt werden, was allerdings auch als Maßeinheit interpretiert werden könnte.

- Zeitlich veränderliche Beziehungen (bei Währungseinheiten)
  Unterschiedliche Währungseinheiten können zwar auf differierende Maßeinheiten zurückgeführt werden, wobei als Besonderheit der Umrechnungsfaktor zeitlich veränderlich ist (dynamische Wertkorrespondenz). Dieser Faktor kann selbst wieder unterschiedliche Granularität aufweisen, z.B. als täglicher Faktor für den Zahlungsverkehr oder als jährlicher Faktor für Steuerabrechnungen. Im Beispiel gilt: Preise in DB1 sind in DM, in DB2 in US-Dollar angegeben, deren Verhältnis wird ausgedrückt durch den cvt_factor.
  DB1.Article.price = DB2.Article.price * cvt_factor

b) Syntaktische Differenzen zwischen Domänen
   Selbst bei gleicher semantischer Domäne können sich Unterschiede ergeben, die abhängig sind vom Typsystem des verwendeten Datenbanksystems. Diese werden auch als Repräsentationsdifferenzen bezeichnet. Zu den Typdifferenzen zählen: unterschiedliche Surrogattypen (bei verschiedenen OODBMS), Längenunterschiede (z.B. bei Zeichenketten), Unterschiede bei numerischen Typen (mögliche Repräsentation durch Integer-, Festpunkt- oder Gleitkommazahlen), Genauigkeitsdifferenzen (bei Gleitkommazahlen einfache oder doppelte Genauigkeit, bei Festkommazahlen Anzahl der Nachkommastellen), unterschiedliche Integer-Formate (short oder long). Eine ausführliche Beschreibung findet man in [GSC95]. Einen Spezialfall stellen Formatunterschiede bei der Datumsdarstellung dar. Beispiel: Ein Datum der Form “03/01/02” erlaubt zumindest drei unterschiedliche Interpretationen: (3. Januar 2002, 1. März 2002, 2. Januar 2003) und bedarf zusätzlicher Metainformation, wie z.B. eine Angabe darüber, ob es sich um deutsches oder US-amerikanisches Datumsformat handelt. Hinzu kommt in diesem Beispiel noch die Information über das Jahrhundert (1902 oder 2002).

4. Constraint-Differenzen
Constraints, so wie sie in Abschnitt 2.3 klassifiziert wurden, können in Datenbanken, die semantisch äquivalente Weltausschnitte modellieren, im Gegensatz zueinander stehen. Im einfachsten Falle ist ein Constraint nur in einer Datenbank definiert. Auf mögliche Strategien bei der Integration von Constraints wird in Abschnitt noch einmal eingegangen.

2.4.2.2 Semantische Heterogenität auf Instanzenebene

Zwei Instanzen semantisch äquivalenter Klassen mit korrespondierenden Attributen können sich in ihrer Ausprägung voneinander unterscheiden. Allgemein läßt sich eine Wertdiskrepanz folgendermaßen formulieren: Objekt o_1 der Klasse c_1 in Datenbank DB^1 besitzt im Attribut A_1 einen Wert v_1, das mit ihm korrespondierende Objekt o_2 der Klasse c_2 in Datenbank DB^2 besitzt im korrespondierenden Attribut den Wert v_2, wobei gilt $v_1 \neq v_2$. Dazu lassen sich einige Spezialfälle angeben. Wenn es sich bei dem betrachteten Attribut um ein mehrwertiges Attribut handelt, besteht auch eine Diskrepanz zwischen v_1 und v_2, wenn die Anzahl der Werte nicht übereinstimmt. Wenn im Attribut Nullwerte erlaubt sind, besteht auch eine Abweichung wenn in A_1 ein Nullwert steht, in A_2 hingegen nicht. Die beschriebenen Abweichungen werden auch als Dateninkonsistenz [SK93] oder Datenheterogenität [KS91] bezeichnet.

2.4.3 Globale Integrität in heterogenen Datenbanken

In Auswertung der genannnten Beispiele ergibt sich, daß es notwendig ist, Korrespondenzfunktionen einzuführen, um mit unterschiedlichen Wertebereichen operieren und globale Integritätsbedingungen formulieren zu können. Diese Funktionen wollen wir auf Objekte lokaler Datenbanken anwenden. Dazu führen wir folgende Notation ein, die an die Begriffsdefinition in Abschnitt 2.2 angelehnt und für eine Menge von Datenbanken $DB = \{DB^1, DB^2, ..., DB^n\}$ verallgemeinert wurde.

$S^i =$ Schema der Datenbank DB^i

$T^i =$ $\{t_{i1}, t_{i2}, \ldots, t_{in}\}$ Typsystem der Datenbank DB^i, umfaßt alle Typen t_{ij} $(1 \le j \le n)$ gemäß Definition 2.1

$C^i =$ $\{c^i_1, c^i_2, \ldots, c^i_n\}$ Menge von Klassennamen aus dem Schema S^i

$O^i =$ Menge von Klasseninstanzen in der Datenbank $d(S^i)$ mit $O^i \subseteq OID^i$

Definition 2.14 (Wertkorrespondenzfunktion)
Zwei Werte v und v' ($v \in \text{dom}(t_{ik})$, $v' \in \text{dom}(t_{jl})$, $i \neq j$), die in zwei verschiedenen Datenbanken DB^i und DB^j auftreten, korrespondieren miteinander, wenn sie dasselbe Konzept repräsentieren. Diese Korrespondenz kann durch eine Wertkorrespondenzfunktion, **eqv** (*equivalent value*), beschrieben werden, die jedem Wert aus $\text{dom}(t_{ik})$ einen Wert aus $\text{dom}(t_{jl})$ zuordnet und umgekehrt (totale 1:1-Abbildung).
eqv: $\text{dom}(T^i) \rightarrow \text{dom}(T^j)$ (Kurzschreibweise: $\cong^v$)

Definition 2.15 (Objektkorrespondenzfunktion)
Zwei Instanzen o und o' ($o \in \text{inst}(c_{ik})$, $o' \in \text{inst}(c_{jl})$) aus zwei verschiedenen Datenbanken DB^i und DB^j korrespondieren miteinander, wenn sie dasselbe Realwelt-Objekt repräsentieren (d.h. die Klassen c_{ik} und c_{jl} müssen nicht notwendig semantisch äquivalent sein). Diese Korrespondenz kann durch eine Objektkorrespondenzfunktion, **eqo** (*equivalent object*), beschrieben werden, die jedem Objekt aus $\text{inst}(c_{ik})$ ein Objekt aus $\text{inst}(c_{jl})$ zuordnet und umgekehrt (totale 1:1-Abbildung zwischen semantischen Replikaten).
eqo: $O^i \rightarrow O^j$ (Kurzschreibweise: $\cong^o$)

Die Objektkorrespondenz beruht auf der Wertkorrespondenz zwischen Attributen der in Beziehung stehenden Instanzen, in den meisten Fällen sind dies auch die jeweiligen Schlüsselattribute. Die Identifizierungsfunktion **id** verknüpft die jeweiligen Attribute und vergleicht deren Werte auf Wertkorrespondenz. Somit läßt sich definieren:

$o\ (o \in \text{inst}(c_{ik})) \cong^o o'\ (o' \in \text{inst}(c_{jl})) \Leftrightarrow$
$(\exists\ ((A_1, A_2, \ldots, A_n) \in \text{attr}(\text{type}(c_{ik}))\ \exists\ ((A'_1, A'_2, \ldots, A'_m) \in \text{attr}(\text{type}(c_{jl}))$
$\text{id}((A_1, A_2, \ldots, A_n), (A'_1, A'_2, \ldots, A'_m)) = \text{true}$

Die Wertkorrespondenz kann für nichtnumerische Domänen durch eine diskrete Funktion beschrieben werden (z.B. bei Aufzählungsdatentypen), für numerische Domänen durch eine kontinuierliche Funktion. Dabei nennen wir eine Domäne numerisch, wenn arithmetische Operationen auf ihr anwendbar sind. Für diskrete Funktionen sind einfache Lookup-Tabellen denkbar. Korrespondierende Objekte werden zwar auf der Basis von Attributwerten bestimmt, könnten aber bei systemverwalteter Objektidentität auch unter Verwendung der internen OID's

(Surrogate) gepflegt werden. Wir wollen nun die eingeführten Korrespondenzfunktionen auf die auf Seite 21 beschriebenen Domänen-Differenzen anwenden.

Domänen-Unterschied	Wertkorrespondenz	Objektkorrespondenz
a) semantische Differenzen		
Instanzen-Identifikation	-	wertbasiert / surrogatbasiert
Wertebereiche	diskret / kontinuierlich	-
Wertgranularität	diskret	-
Dimension. Maßeinheit, Skalierung	diskret / kontinuierlich	-
zeitliche veränderliche Beziehungen	kontinuierlich (Funktion zeitabhängig)	-
b) syntaktische Differenzen	diskret / kontinuierlich	-

Tabelle 2.1: Domänen-Unterschiede (durch Korrespondenzfunktionen ausgedrückt)

Wie sich aus Tabelle 2.1 ergibt, lassen sich Domänen-Differenzen immer durch eine Korrespondenzfunktion ausdrücken.

Unter Anwendung dieser Funktionen wollen wir nun erneut eine Klassifikation von Integritätsbedingungen in heterogenen MDBS nach ihrem Geltungsbereich vornehmen. Hierbei ist eine Unterscheidung zwischen der Ebene des konzeptionellen und des logischen Datenmodells vorzunehmen.

Im Falle homogener Datenbanken läßt sich das konzeptionelle Datenmodell auf ein logisches Datenmodell eines einzigen DBMS abbilden. Aufgrund dieser einheitlichen Repräsentation entsprechen sich die Constraints, so wie sie in Abschnitt 2.3 klassifiziert wurden, im konzeptionellen und im logischen Datenmodell.

Bei heterogenen Datenbanken müssen wir unterscheiden, ob dasselbe konzeptionelle Datenmodell auf unterschiedliche Weise repräsentiert ist bzw. inwieweit unterschiedliche Weltausschnitte in einzelnen Datenbanken gespeichert sind. Im einen Fall sprechen wir von semantisch äquivalenten Klassen, im anderen Fall von semantisch verschiedenen Klassen. Bei semantisch äquivalenten Klassen ist außerdem das Verhältnis ihrer Extensionen von Bedeutung (vgl. Abschnitt 2.4.2): Entweder es gibt disjunkte Klassen oder Klassen mit gemeinsamen Instanzen (semantische Replikate). In einem objektorientierten konzeptionellen Datenmodell kann eine Klasse somit mehrere semantisch äquivalente Klassen des logischen Datenmodells umfassen, d.h. die Angabe des Geltungsbereichs bezieht sich auf Objekte des konzeptionellen Datenmodells.

Allen Constraints ist gemeinsam, daß sie auf Ausprägungen von Datenbanken ausgedrückt werden können. Sie sind durch Beziehungen zwischen Werten oder Objekten aus unterschiedlichen Datenbanken gekennzeichnet. Dementsprechend können wir zwei Kategorien von Integritätsbedingungen, Existenz- und Wertbedingungen, unterscheiden:

Definition 2.16 (Existenzbedingung)
Eine Existenzbedingung enthält eine Aussage über die Existenz eines Objektes o_1 in DB^1. Wenn die Existenz des Objektes o_1 mit der Existenz eines anderen Objektes o_2 aus DB^2 verknüpft ist, so sprechen wir von einer Existenzabhängigkeit E (*Existence Dependency*).

Definition 2.17 (Wertbedingung)
Eine Wertbedingung enthält eine Aussage über den Zustand (bzw. den Wert eines Attributes) eines Objekt o_1 in DB^1. Wenn der Zustand des Objektes o_1 vom Zustand eines anderen Objektes o_2 aus DB^2 abhängt, so sprechen wir von einer Wertabhängigkeit V (*Value Dependency*).

In Tabelle 2.2 werden die bisher eingeführten Arten von Constraints auf heterogene Datenbanken angewandt. Dabei wird jeweils nach der semantischen Beziehung zwischen den Klassen unterschieden, auf denen sie definiert werden (X = Bedingung anwendbar). Durch die Kategorie wird die Zuordnung zu einer Existenz- oder Wertabhängigkeit bestimmt.

Constraint	Kat.	semantisch äquivalente Klassen		semantisch verschiedene Klassen
		disjunkte Extensionen	nichtdisjunkte Extensionen	
Intra-Instanzenbedingungen				
Domänen-/Attributbedingungen	-	-	-	-
Implikationsbedingungen	V	-	X	-
Inter-Attributbedingungen	V	-	X	-
Inter-Instanzenbedingungen				
Berechenbare Bedingungen	V	X	X	-
Nicht-Berechenbare Bedingungen	E	X	X	-
Schlüsselbedingungen (Uniqueness)	E	X	X	-
Inter-Klassenbedingungen				
Referentielle Integritätsbedingungen	E	X	X	X
Komplexitätsrestriktionen	V	-	-	X
Navigationale Integritätsbedingungen - arithmetische Constraints - berechenbare Bedingungen	V	-	-	X

Tabelle 2.2: Integritätsbedingungen in heterogenen Datenbanken

Attributbedingungen können in unserem Modell keine globalen Constraints sein, da sie als Intra-Instanzenregeln sich immer nur auf ein Attribut einer Instanz beziehen. Im Gegensatz dazu stehen Bedingungen zwischen mehreren Attributen derselben Instanz, die als Ausprägung eines Konzeptes vertikal auf mehrere Datenbankobjekte partitioniert sein kann.

Inter-Instanzenbedingungen lassen sich auf Objekte von semantisch äquivalenten Klassen so anwenden, als wenn diese Objekte alle zu einer gemeinsamen Klasse gehören würden, wie es für den homogenen Fall beschrieben wurde. Betrachten wir die zwei Klassen DB1.Employee (alle Angestellten der Mutterfirma) und DB2.Angestellter (alle Angestellten des deutschen Tochterunternehmens). Auf ihnen läßt sich gleichfalls eine Uniqueness-Bedingung (z.B. “die Beschäftigten-Nr. ist global eindeutig”) oder eine berechenbare Bedingung (z.B. “Summe aller Gehälter < 500.000 DM”) ausdrücken. Problematisch ist der Fall, wenn eine solche Inter-Instanzenbedingung auf semantisch äquivalente Klassen angewandt wird, deren Extensionen sich überlappen, da eine Unterscheidung zwischen semantischen Replikaten und semantisch ungleichen Objekten getroffen werden muß.

Klassenübergreifende Bedingungen können auch in heterogenen Datenbanken Gültigkeit haben unter möglicher Einbeziehung von Wertkorrespondenzfunktionen. Referentielle Integritätsbedingungen können sowohl zwischen semantisch äquivalenten Klassen als auch zwischen

verschiedenen Klassen formuliert werden. Angewandt auf äquivalente Klassen, lassen sich durch referentielle Bedingungen Mengenbeziehungen zwischen deren Extensionen kontrollieren (vgl. Abbildung 2.4 auf Seite 21).

2.5 Abschwächung von Konsistenzbedingungen

Bisher haben wir die Annahme zugrunde gelegt, daß die Datenbank zu jedem Zeitpunkt in einem konsistenten Zustand sein muß, so wie es das Transaktionskonzept vorsieht (vgl. Definition 2.13). Dies wird auch als Ubiquitätsprinzip bezeichnet. Es kann jedoch in der Weise abgeschwächt werden, daß die Menge der Zeitpunkte, an denen die Integritätsbedingungen erfüllt sein müssen, eingeschränkt wird. Dabei läßt sich die Konsistenz entlang einer zeitlichen Dimension und entlang einer extensionalen Dimension abschwächen, Kombinationen zwischen beiden sind denkbar [RC92]. Entlang der zeitlichen Dimension läßt sich die Abschwächung durch absolute Zeitpunkte, durch Ereignisse oder Zustandsveränderungen der Datenbank beschreiben, die jeweils auf einen Zeitpunkt abgebildet werden können. Die extensionale Dimension umfaßt in unserem Sprachgebrauch den Raum von Objekten, die von einer Konsistenzbedingung erfaßt werden.

2.5.1 Zeitliche Dimension der Abschwächung

Der Gedanke, den zeitlichen Aspekt in die Definition der Konsistenz einzubeziehen, wurde bereits in [WQ87] diskutiert und im Rahmen des Konzepts interdependenter Daten in [SR90] vertieft. Dabei wurden folgende Begriffe eingeführt:

Eventuelle Konsistenz (*Eventual Consistency*) beschreibt, daß Konsistenzbedingungen nur zu bestimmten Zeitpunkten erfüllt sein müssen, wobei in den dazwischenliegenden Intervallen Verletzungen geduldet werden. Zur Spezifikation der Zeiten, an denen die Integrität gewahrt sein soll, lassen sich verschiedene Methoden anwenden: a) Es werden Zeitpunkte spezifiziert, an denen z.B. redundante Kopien konsistent sein sollen, beispielsweise täglich 17.00 Uhr. Die Konsistenzbedingung wird also mit einem periodischen Zeitereignis verknüpft. b) Es lassen sich Zeitintervalle definieren, z.B. soll die Konsistenz zwischen redundanten Kopien täglich zwischen 9 und 12 Uhr eingehalten sein. c) Die Wiederherstellung der Konsistenz wird verknüpft mit dem Auftreten eines bestimmten Ereignisses. Die Konsistenz darf somit auch über längere Zeiträume verletzt sein, solange das entsprechende Ereignis nicht eintritt. In diese Kategorie läßt sich z.B. der Begriff der *Lazy Consistency* einordnen, wie er in Postgres [SHP88] Verwendung findet. Erst bei lesendem Zugriff auf einem Datenbestand sind eventuell vorhandene Inkonsistenzen zu beseitigen.

Verzögerte Konsistenz (*Lagging Consistency*) ist typischerweise anwendbar auf Daten, die redundant als Kopien gespeichert werden, wobei eine Kopie immer aktuell gehalten wird und andere Kopien davon abweichen dürfen. Während bei eventueller Konsistenz zumindest zu bestimmten Zeitpunkten die Konsistenz über dem gesamten Datenbestand erfüllt ist, kann es bei verzögerter Konsistenz immer sein, daß einzelne Daten noch nicht gemäß den Integritätsbedingungen aktualisiert sind. Einzelne Kopien sind jedoch konsistent. Diese Abschwächung von Konsistenz ist in verschiedenen Fällen sinnvoll: a) Eine Applikation greift zu einem Zeitpunkt nur auf eine Kopie zu. b) Anfragen werden immer an die aktuellste Kopie weitergeleitet. c) Die Applikation benötigt nicht die aktuellsten Daten.

2.5.1.1 Konsistenzwahrung an Aktivitätsgrenzen

Die Einhaltung der Konsistenz läßt sich an Ausführung bzw. Beendigung bestimmter Aktivitäten binden. Dabei gilt: Die Konsistenzabschwächung ist umso stärker, je größer die Abstände zwischen diesen Ereignissen sein können.

- nach Ausführung einer Operation
  Eine Datenbank muß konsistent sein unmittelbar nach Ausführung jeder Operation. Die Menge der Operationen soll charakterisiert sein durch die Basis-Ereignisse, die in Abschnitt 2.2.2 eingeführt wurden. Für die unmittelbare Konsistenzwahrung ist z.B. in SQL das Schlüsselwort SET CONSTRAINTS ... IMMEDIATE vorgesehen.

- nach Ausführung einer Menge von Operationen
  Nach Beendigung einer Folge von Operationen innerhalb einer Transaktion muß die Konsistenz wieder erfüllt sein. Ein Beispiele hierfür ist gegeben bei semantischer Atomarität [Gar83].

- bei Beendigung einer Transaktion
  Nach der letzten Operation, vor dem Beenden der Transaktion (Commit), wird die geforderte Konsistenz überprüft. Dieser Konsistenzwahrungmodus wird mit DEFERRED bezeichnet. Durch eine solche Anforderung wird auch der Atomaritätseigenschaft flacher Transaktionen Rechnung getragen.

- nach Beendigung einer Menge von Transaktionen (Verletzung der Ubiquität)
  Ein typisches Beispiel für eine derartige Abschwächung kann im Bereich CAD und Entwurfstransaktionen liegen, wobei Zwischenzustände toleriert werden, an denen mitunter sehr komplexe Konsistenzanforderungen nicht vollständig erfüllt sein können. Für derartige Anwendungen sind auch erweiterte Transaktionsmodelle erforderlich, z.B. das kooperative Transaktionsmodell [KS88].

2.5.1.2 Zustandsbasierte Abschwächung

Eine Abschwächung von Konsistenz läßt sich auch zustandsbasiert ausdrücken, wie es z.B. für Cache-Konsistenz gezeigt wurde [AB89]. Wechselseitige Konsistenzbedingungen lassen sich entlang einer Dimension spezifizieren, die auf dem Verhältnis der Zustände basiert, d.h. es werden zulässige Abweichungen der Daten untereinander quantifiziert. Solche Einschränkungen können umfassen:

- Anzahl der geänderten Datenelemente
  Es lassen sich Schwellwerte dafür angeben, wieviele Datenelemente verändert werden dürfen, bevor die Konsistenz wiederhergestellt werden muß. Ein Beispiel dafür ist gegeben bei Aggregatbedingungen, die erneut überprüft werden sollen, sobald ein bestimmter Prozentsatz der Tupel verändert wurde.

- Maximale Veränderung eines Datenwertes
  Eine Konsistenzbedingung wird solange nicht überprüft, wie Veränderungen einzelner Datenwerte ein bestimmtes Maximum nicht überschreiten. In [SR90] ist das Beispiel einer Lagerverwaltung genannt, bei der Verkäufe, die jeweils eine bestimmte Anzahl nicht überschreiten, nicht notwendig eine Kontrolle des gesamten Lagerbestandes nach sich ziehen. Das Maximum läßt sich als absoluter Wert spezifizieren oder auch als eine relative Angabe in Bezug zum existierenden Wert.

- Maximalanzahl der zulässigen Operationen
  Durch Begrenzung der Anzahl der Operationen auf einem Datenbestand läßt sich spezifizieren, wann eine Konsistenzbedingung überprüft bzw. wiederhergestellt werden muß. Bezogen auf das Beispiel in [SR90] könnte die Zahl der Verkaufstransaktionen beschränkt sein, nach der spätestens die Konsistenz mit den Lagerbestandsdaten wieder hergestellt sein muß.

Die drei oben genannten Kriterien können auch logisch miteinander verknüpft werden. So läßt sich z.B. eine Bedingung formulieren, die besagt, daß eine Konsistenzprüfung ausgelöst werden soll, wenn bei einer Änderungsoperation ein Maximalwert oder aber die maximale Anzahl von zulässigen Operationen überschritten wird.

2.5.1.3 Anwendbarkeit zeitlicher Abschwächungen

Bei den genannten Beispielen sind wir davon ausgegangen, daß zu einem Zeitpunkt, der nach der Konsistenzverletzung liegt, eine Integritätsbedingung überprüft und gegebenenfalls wiederhergestellt wird. D.h. wir müssen eigentlich zwei “Phasen” der Konsistenzkontrolle und ihre zeitliche Beziehung zum konsistenzverletzenden Ereignis unterscheiden

Es gibt Constraints, bei denen eine verzögerte Bedingungsprüfung es nicht erlaubt, die bei einer Konsistenzverletzung notwendige Wiederherstellungsaktion zu bestimmen. Hierzu zählen arithmetische Constraints und Schlüsselbedingungen (Uniqueness). In einem System autonomer und heterogener Datenbanken, in dem keine sofortige Überprüfung möglich ist, besteht aber die Möglichkeit, globale Bedingungen durch lokale Constraints zu ersetzen, die mit diesen kompatibel sind (eine Idee, die z.B. im *Demarcation Protocol* verwirklicht ist [BG94]). Damit kann sofort lokal reagiert werden. Eine Zurücksetzung der konsistenzverletzenden Aktion zu einem späteren Zeitpunkt schafft Probleme durch möglicherweise auftretende kaskadierende Aborts.

Eine sofortige Bedingungsprüfung kann mit einer verzögerten Aktionsausführung dann kombiniert werden, wenn eine Veränderung des Datenbankzustandes in der Zeit bis zur Aktionsausführung keine Auswirkung auf diese hat.

Bei einer zeitlichen Konsistenzabschwächung kann bei vielen Arten von Constraints die Bedingungsprüfung mit anschließender Aktionsausführung verzögert werden. Dies ist hauptsächlich für alle Arten von Wertabhängigkeiten möglich, bei denen die Propagierungsrichtung für zu ändernde Werte eindeutig feststeht (z.B. bei abgeleiteten Daten). Auf den damit verbundenen Aspekt der Kontrollabhängigkeiten wird nachfolgend in Abschnitt 2.6 eingegangen.

2.5.1.4 Der Zeitbegriff

Die Einbeziehung der Zeit zur Beschreibung abgeschwächter Konsistenz setzt voraus, daß die lokalen Zeiten autonomer Teilnehmersysteme synchronisiert werden können. Somit lassen sich zeitliche Abschwächungen über einer gemeinsamen globalen Zeit ausdrücken. Eine ausführliche Behandlung dieses Problems im Kontext der Detektion von Zeitereignissen in verteilten Systemen gibt Schwiderski in [Sch96], deshalb soll an dieser Stelle nicht weiter darauf eingegangen werden. Unsere Grundannahme beruht darauf, daß die Granularität einer globalen Uhr größer ist als die maximale Zeitdifferenz zwischen beliebigen zwei lokalen Uhren. Formal ausgedrückt heißt das: Es sei $clock_z(clock_k(i))$ der Zeitpunkt des Auftretens des i-ten Ticks auf dem Knoten k, der durch die Referenzuhr z gemessen werde. Die Präzision Π ist definiert als:

$$\Pi = \text{Max} \{ \forall i \, \forall j \, \forall k \mid clock_z(clock_k(i)) - clock_z(clock_l(j)) \}$$

Die Granularitätsbedingung besagt, daß die Granularität der globalen Zeit g_g nicht kleiner sein darf als die Präzision Π ($g_g > \Pi$). Bei autonomen Legacy-Systemen, die in dieser Arbeit betrachtet werden, gilt, daß nur mit einer relativ groben globalen Zeitgranularität gerechnet wird, auf deren Grundlage datenbankübergreifende Konsistenzbedingungen zeitabhängig definiert werden.

2.5.2 Extensionale Dimension der Abschwächung

Die in Abschnitt 2.3 diskutierte Taxonomie für Integritätsbedingungen erfaßt Constraints, die auf Extensionen einer Klasse (Inter-Instanzenregeln) oder mehrerer Klassen (klassenübergreifende Regeln) ausgedrückt werden. Dabei wird implizit immer gefordert, daß alle Instanzen der jeweiligen Extension die Regel erfüllen müssen. Eine extensionale Abschwächung schränkt die Menge der Objekte nach verschiedenen Kriterien ein. Denkbar sind:

- Definition eines Prädikats
  Es werden die Objekte beschrieben, für die eine Konsistenzbedingung gelten soll (z.B. alle Instanzen, die zu einem bestimmten Projekt ‘X’ gehören, müssen Constraints nach der DIN-Norm erfüllen).
- Definition der “Orte”
  Es werden die Orte spezifiziert, für die gelten soll, daß sich alle Daten, die an diesem Ort gespeichert sind, in einem konsistenten Zustand befinden (z.B. alle Daten, die sich auf einem bestimmten Rechner befinden).

2.6 Kontrollabhängigkeiten

Wie in [Mor84] gezeigt wurde, lassen sich semantische Integritätsbedingungen auf Gleichheitsbedingungen zurückführen und durch *Constraint Equations*, d.h. algebraische Gleichungen, beschreiben. Damit verbunden sind Kontrollabhängigkeiten, die zumeist implizit ausgedrückt werden, z.B. durch die Festlegung, daß die linke Seite einer Gleichung der rechten angeglichen werden muß. Kontrollabhängigkeiten sagen also etwas über die Dominanz einzelner Bestandteile von Integritätsbedingungen aus. Diese Dominanz kann *statisch*, d.h. laufzeitunveränderlich, definiert sein oder sich *dynamisch* entsprechend bestimmter Festlegungen verändern. Wir unterscheiden folgende Fälle von Kontrollabhängigkeiten:

- Abgeleitete Daten
  Hierbei läßt sich eine Unterscheidung treffen zwischen Quelldaten und von ihnen abgeleiteten Daten, die sich in anderen Datenbanken befinden können (im Sinne einer materialisierten Sicht). Auf den abgeleiteten Daten können direkt keine Änderungen vorgenommen werden, sondern diese müssen von den Quelldaten dorthin propagiert werden. Ein typisches Beispiel für abgeleitete Daten sind Aggregationen der Quelldaten:
  Department.Total-Salaries = sum(Employee.Salary)
  In dieser Anwendung ist die Gesamtsumme der Gehälter, die in einer Abteilung gezahlt werden, neu zu berechnen, wenn sich das Gehalt eines Angestellten ändert.
- Primär- und Sekundärkopien
  Daten, die in einer Primärdatenbank gespeichert sind, sind immer in einem konsistenten Zustand. Änderungen auf ihnen müssen in Sekundärdatenbanken nachvollzogen werden, die erst bei Übereinstimmung mit den Primärdatenbanken wieder einen konsistenten Zustand erreichen.

- Dynamische Kontrollabhängigkeiten
  Während in den ersten Fällen immer davon ausgegangen wurde, daß die Kontrollbeziehung zwischen Datenbanken konstant ist, ist es auch denkbar, daß die Frage nach der Dominanz einer Datenbank dynamisch entschieden wird, z.B. in Abhängigkeit vom Alter der letzten Änderungen (siehe dazu auch Seite 32).
- Unabhängige Datenbanken
  Alle Datenbanken sind gleichberechtigt und speichern Daten, die miteinander in Beziehung stehen. Es gibt somit keine feststehende Richtung, in der Änderungen aus einer Datenbank in eine andere propagiert werden müssen.

In autonomen Systemen können bei zeitlich und extensional abgeschwächten Konsistenzanforderungen unabhängige Modifikationen an Daten vorgenommen werden, die miteinander durch globale Integritätsbedingungen verbunden sind. Es gibt kein einheitliches DBMS, das eine zentrale Kontrolle über die globale Konsistenz ausüben kann.

Die zeitgleiche Existenz verschiedener Ausprägungen eines Datums ist insbesondere für heterogene Systeme typisch, die autonome Kontrollbereiche darstellen. Ein Kontrollbereich kann als Abstraktion eines lokalen DBS gesehen werden mit autonomer Kontrolle über Transaktionsausführung und Konsistenz. In einer lose gekoppelten Föderation ist es oft erstrebenswerter, nicht unmittelbar auf lokale Ereignisse zu reagieren, z.B. kann die Propagierung von Änderungen bis zu einem späteren (benutzerdefinierten) Zeitpunkt verzögert werden. So kann es passieren, daß Daten die redundant gespeichert sind, in verschiedenen Versionen auftreten, die irgendwann einmal wieder zusammengeführt werden müssen.

2.6.1 Abgleichstrategien für redundante Daten

Wenn man die unterschiedlichen Ausprägungen miteinander korrespondierender Datenobjekte zu irgendeinem Zeitpunkt wieder zusammenführen will, bedarf es bestimmter Abgleichstrategien. Die Aufgabe der Abgleichstrategien besteht darin, konfligierende Informationen über Datenmodifikationen beim Zusammenführen anwendungsspezifisch zu verarbeiten. In einem Entscheidungsprozeß werden anhand von Kriterien die Modifikationsinformationen gewichtet und eine resultierende Datenmenge als Ergebnis eines Mischvorganges gebildet. Nach [Jab90] lassen sich solche Entscheidungskriterien in vier Klassen einteilen:

- Zeit (d.h. Zeitstempel einer Modifikation)
  Bevorzugung der jüngsten oder der ältesten Änderung
- Organisatorische Maßnahmen
  Bestimmung von Rangfolgen zwischen Anwendungssystemen / Datenbanksystemen
- Heuristiken
- Entscheidung des Anwenders

In [Jab90] sind vier sogenannte Einbringstrategien für das Zusammenführen von replizierten Daten bei asynchroner Verarbeitung beschrieben. Diese lassen sich auch auf redundante Daten bzw. korrespondierende Objekte anwenden.

- vollständige Dominanz
- partielle Dominanz
- asymmetrische zeitliche Konkurrenz
- symmetrische zeitliche Konkurrenz

Die ersten drei sind asymmetrische Verarbeitungsstrategien, d.h. die Kommunikation ist zwischen zwei Kontrollbereichen gerichtet. Die symmetrische Form gilt für eine dynamische Kontrollabhängigkeit und wird auch als *peer-to-peer* Replikation bezeichnet. Sie ist die kritischste, für die gegenwärtig kaum kommerzielle Lösungen angeboten werden [Soe96].

Vollständige Dominanz

Der Kontrollbereich i ist dem Kontrollbereich j komplett übergeordnet. Beim Abgleichvorgang werden alle Datensätze aus dem Datenbestand i ohne Änderungen in den Datenbestand j übernommen, der dadurch vollständig ersetzt wird. Zwischen zwei Abgleichzeitpunkten können sich so die Datenbestände unabhängig entwickeln, sind aber nach Durchführung der Abgleichstrategie vollkommen identisch.

Partielle Dominanz

Auch bei der partiellen Dominanz ist der Kontrollbereich i dem Kontrollbereich j übergeordnet. Es werden aber nur alle Datensätze aus dem Datenbestand i in den Datenbestand j übernommen, die seit dem letzten Abgleich modifiziert wurden. Alle anderen Datensätze des Datenbestandes j bleiben unverändert. Ist zwischen zwei Abgleichzeitpunkten der gleiche Datensatz in beiden Datenbeständen geändert worden, dann wird der Datensatz aus dem Datenbestand i in den Datenbestand j übernommen. Die Modifikationen im Datenbestand j sind für den Kontrollbereich i nicht von Bedeutung. Nach Durchführung der Einbringstrategie gleichen sich somit die Datensätze in beiden Beständen, die zwischen den Abgleichzeitpunkten im Bereich i geändert wurden.

Asymmetrische zeitliche Konkurrenz

Bei dieser Einbringstrategie werden alle Datensätze aus dem Kontrollbereich i in den Kontrollbereich j übertragen, die seit dem letzten Abgleich modifiziert wurden. In Konfliktfällen, d.h. bei Modifikation desselben logischen Objekts, wird aber die Modifikation in den Datenbestand j aufgenommen, die ein jüngeres Datum besitzt.[4]

Symmetrische zeitliche Konkurrenz

Nach Ausführung dieser Strategie enthalten die Datenbestände beider Kontrollbereiche i und j die aktuellsten Ausprägungen der Datensätze, die seit dem letzten Abgleich modifiziert wurden. Die Kommunikation zwischen den Kontrollbereichen ist bei dieser Einbringstrategie symmetrisch, da kein Kontrollbereich gegenüber einem anderen dominant ist.

In Tabelle 2.3 werden die Auswirkungen der verschiedenen Einbringstrategien auf die jeweiligen Datenbestände exemplarisch gezeigt. In dem Beispiel werden die Datensätze zu drei verschiedenen Zeitpunkten betrachtet: Ausgehend von den Ausgangsdatenbeständen werden die Datenbestände nach lokalen Datenmodifikationen in den jeweiligen Kontrollbereichen gezeigt. Zuletzt sind die Daten im Zustand nach Ausführung der jeweiligen Einbringstrategie dargestellt. Tabelle 2.4 enthält alle möglichen Kombinationen von Datenänderungen, die zur Darstellung der verschiedenen Einbringstrategien gebraucht werden.

4 Es sind auch Strategien denkbar, die ältere Modifikationen bevorzugen, z.B. bei Buchungssystemen.

Ausgangsdatenbestände der Kontrollbereiche	KB i	a	b	c	d	e	f	g		
	KB j	a	b	c	d	e			h	i
lokale durchgeführte Datenmodifikationen	KB i	a’	b’	c’	d	e	f’	g		
	KB j	a”	b”	c	d”	e			h”	i
Ergebnisdatenbestände der Kontrollbereiche										
vollständige Dominanz	KB i	a’	b’	c’	d	e	f’	g		
	KB j	a’	b’	c’	d	e	f’	g		
partielle Dominanz	KB i	a’	b’	c’	d	e	f’	g		
	KB j	a’	b’	c’	d”	e	f’		h”	i
asymmetrische zeitliche Konkurrenz	KB i	a’	b’	c’	d	e	f’	g		
	KB j	a”	b’	c’	d”	e	f’		h”	i
symmetrische zeitliche Konkurrenz	KB i	a”	b’	c’	d”	e	f’	g	h”	
	KB j	a”	b’	c’	d”	e	f’		h”	i

KB = Kontrollbereich; ’ bzw. ” = modifizierter Datensatz in KB i oder j

Tabelle 2.3: Darstellung der verschiedenen Einbringstrategien

Datum	existiert in KB i	existiert in KB j	modifiziert in KB i	modifiziert in KB j	Reihenfolge der Modifikationen
a	ja	ja	ja	ja	$t_i < t_j$
b	ja	ja	ja	ja	$t_i < t_j$
c	ja	ja	ja	nein	
d	ja	ja	nein	ja	
e	ja	ja	nein	nein	
f	ja	nein	ja		
g	ja	nein	nein		
h	nein	ja		ja	
i	nein	ja		nein	

Tabelle 2.4: Charakterisierung der Beispieldaten

2.6.2 Verhältnis lokaler und globaler Constraints

Ein typisches Problem bei der Integration besteht darin, daß separate Integritätsbedingungen aus autonomen Subsystemen zusammengeführt werden müssen. Wir drücken das Verhältnis zweier lokaler statischer Integritätsbedingungen dadurch aus, indem wir jeweils die Menge der durch sie bestimmten konsistenten lokalen Datenbankzustände betrachten. Dabei werden paarweise zwei Integritätsbedingungen auf ihr Verhältnis hin untersucht unter Ausschluß anderer Constraints.[5] Da wir von heterogenen Datenbanken ausgehen, basiert der Vergleich auf dem vorher eingeführten Begriff der Objekt- und Wertkorrespondenz.

Zwei Integritätsbedingungen $IS_i \in IS^1$ in DB^1 und $IS_j \in IS^2$ in DB^2 sind **äquivalent**, wenn für die Mengen der in Bezug auf IS_i bzw. IS_j korrekten Datenbankzustände in beiden Datenbanken ${U_{S1}}^{ISi}$ und ${U_{S2}}^{ISj}$ gilt: ${U_{S1}}^{ISi} = {U_{S2}}^{ISj}$.

5 Diese Annahme wurde vereinfachend getroffen, d.h. eine einzelne Integritätsbedingung kann für ein Datenbankobjekt nicht alle korrekten Zustände bestimmen (vgl. auch Definition 2.10 auf Seite 14).

Zwei Integritätsbedingungen $IS_i \in IS^1$ in DB^1 und $IS_j \in IS^2$ in DB^2 sind **überlappend**, wenn die Mengen der korrekten lokalen Datenbankzustände sich überlappen, d.h. es gilt:
$U_{S1}{}^{ISi} \cap U_{S2}{}^{ISj} \neq \emptyset$.

Eine Integritätsbedingung $IS_i \in IS^1$ in DB^1 ist in $IS_j \in IS^2$ in DB^2 **enthalten**, wenn die durch sie repräsentierten korrekten Datenbankzustände eine Teilmenge der durch IS_j repräsentierten korrekten Zustände darstellen:
$U_{S1}{}^{ISi} \subset U_{S2}{}^{ISj}$.

Zwei Integritätsbedingungen $IS_i \in IS^1$ in DB^1 und $IS_j \in IS^2$ in DB^2 sind **inkompatibel**, wenn sie jeweils disjunkte Mengen korrekter lokaler Datenbankzustände spezifizieren:
$U_{S1}{}^{ISi} \cap U_{S2}{}^{ISj} = \emptyset$.

Durch globale Constraints können Bedingungen auf einem lokalen Datenbankzustand formuliert werden, so daß dieser, obwohl lokal korrekt, im Widerspruch zu einer globalen Integritätsbedingung stehen kann. Ein Beispiel wäre durch eine globale Wertabhängigkeit gegeben, d.h. die Existenz eines Wertes in einer Datenbank DB^1 impliziert die Existenz eines Wertes in der korrespondierenden Datenbank DB^2. Zugleich wird jedoch dieser Wert durch ein lokales Constraint in DB^2 verboten. In Analogie zum Verhältnis zwischen lokalen Bedingungen läßt sich auch das Verhältnis eines globalen zu einem lokalen Constraint durch den Vergleich der jeweils korrekten Datenbankzustände beschreiben. Hierzu bezeichnen wir die Menge korrekter lokaler Datenbankzustände von DB^1, bezogen auf ein lokales Constraint $IS_i \in IS^1$, mit $U_{S1}{}^{ISi}$. Durch eine globale Bedingung GIS_j wird eine Menge global korrekter Datenbankzustände in DB^1, $U_{S1}{}^{GISj}$, spezifiziert. Eine globale Integritätsbedingung kann mit einer lokalen Bedingung äquivalent sein, mit dieser überlappen, eine Unter- oder Obermenge von korrekten lokalen Zuständen ausdrücken oder vollständig inkompatibel sein. Die für statische Constraints gemachten Ausführungen gelten ebenso für dynamische Bedingungen, die an dieser Stelle jedoch nicht vertieft werden sollen.

Bei der Betrachtung des Verhältnisses von lokalen und globalen Constraints ist der Aspekt der Dominanz interessant, insbesondere vor dem Hintergrund, daß die lokalen Datenbanken bei einer losen Kopplung unter Wahrung ihrer Autonomie erhalten bleiben sollen. Hierbei können wir verschiedene Fälle unterscheiden, die, geordnet nach wachsendem Grad an Autonomie, nachfolgend skizziert werden:

1. Globale Constraints sind immer einzuhalten ohne Berücksichtigung bereits definierter lokaler Constraints. Dieser Fall bedeutet die stärkste Einschränkung der lokalen Autonomie in Bezug auf die Kontrolle lokaler Constraints.

2. Lokale Constraints werden durch das lokale DBMS eingehalten. Zusätzlich werden dabei globale Constraints beachtet, soweit sie auch lokale Constraints betreffen. Bezogen auf das Verhältnis zwischen einem lokalen und einem globalen Constraint bedeutet das, daß die Schnittmenge $U_{S1}{}^{ISi} \cap U_{S1}{}^{GISj}$ der durch sie beschriebenen Mengen korrekter Datenbankzustände in DB^1 Gültigkeit hat. Die lokale Integritätskontrolle muß zu diesem Zweck angepaßt werden mit einem Verlust an Autonomie als Konsequenz.

3. Die Einhaltung globaler Constraints wird durch eine globale Komponente überwacht, wobei allerdings lokale Constraints berücksichtigt werden. Die resultierende Menge korrekter Datenbankzustände in DB^1 ergibt sich wie im Fall 2 wieder durch Schnittmengenbildung. Der Unterschied besteht jedoch darin, daß keine Modifikation des lokalen Constraint-Managements notwendig ist, da im Konfliktfall diese Vorrang haben. Somit ergibt sich ein Gewinn an lokaler Autonomie.

4. Lokale Constraints müssen unbedingt gewahrt bleiben; dabei werden globale Constraints, die möglicherweise verletzt werden, ignoriert. Hierdurch ist größtmögliche lokale Autonomie gegeben. Es ist keine Kontrolle globaler Integrität möglich, ausgenommen, die globalen Constraints werden so formuliert, daß sie keine Einschränkung der Menge lokal korrekter Datenbankzustände bedeuten.

Die Gewinnung globaler Constraints (Interdatenbankabhängigkeiten) ist nicht Gegenstand dieser Arbeit, wohl aber ein aktuelles Forschungsthema. Hierzu sei verwiesen auf die Arbeit von Klusch, der ein sogenanntes föderatives Zellsystem, FCSI, zur Unterstützung einer kontextbasierten Erkennung plausibler Interdatenbankabhängigkeiten beschreibt. Hierbei werden insbesondere Ansätze aus der Verteilten Künstlichen Intelligenz bzw. terminologischen Wissensrepäsentation genutzt [Klu94]. In anderen Ansätzen wird die Integration lokaler Constraints als ein Teilaspekt der Schemaintegration behandelt, wobei globale Constraints automatisch oder semiautomatisch abgeleitet werden [AQFG95].

2.7 Weitere Konsistenzbedingungen in Multidatenbanken

Neben der Modellierung globaler Konsistenzbedingungen, die nur zwischen lokalen Datenbanken bestehen, müssen auch Daten einbezogen werden, die auf einer globalen Ebene definiert sind. Diese Daten fungieren als Metadaten für die lokalen Daten (Metadata Repository [RS94]). Dementsprechend erweitern wir die Definition einer Multidatenbank.

Definition 2.18 (Multidatenbank)
Eine Multidatenbank *MDB* wird definiert als eine Menge von lokalen Datenbanken und einer globalen Metadatenbank *GM*, $MDB = \{LDB^1, \mathrm{LDB}^2, \ldots, LDB^n, GM\}$. Die globale Metadatenbank besteht aus einer Menge von Metaobjekten, die Struktur, Verhalten sowie Konsistenzbedingungen auf den lokalen Daten mit globaler Reichweite, aber auch implizit die Semantik der Daten ausdrücken können.

2.7.1 Beschreibung der Metadaten

Nachfolgend werden drei Arten von Metadaten unterschieden: Proxy-Objekte (GP), Interdatenbank-Objekte (ID), semantische Objekte (S).

Struktur und Verhalten lokaler Objekte werden durch **globale Proxy-Objekte (GP)** beschrieben, die als Stellvertreter für die lokalen Daten agieren. Sie können durch eine Sicht auf den lokalen Basisdaten realisiert werden. Die Struktur korrespondiert dabei mit der Struktur des äquivalenten lokalen Objektes, das durch die Sicht beschrieben wird. Das Verhalten ist gekennzeichnet durch eine Menge von Operationen, die dem Benutzer an einer globalen Applikationsschnittstelle zur Verfügung stehen. Es bildet die Funktionalität der Schnittstelle des zugrundeliegenden lokalen Systems ab.

Interdatenbank-Objekte (ID) beschreiben Beziehungen zwischen Objekten der lokalen Datenbanken, die auf der globalen Ebene gepflegt werden können. Die Interdatenbank-Abhängigkeiten werden in drei Kategorien eingeteilt:

- Wertkorrespondenzfunktionen (vgl. Definition 2.14 auf Seite 24)
- Objektkorrespondenzfunktionen (vgl. Definition 2.15 auf Seite 24)
- Beziehungen zwischen semantisch verschiedenen lokalen Objekten

Die Repräsentation der Funktionen und Beziehungen kann auf verschiedene Art und Weise erfolgen, z.B. durch eine Menge von Paaren von Werten oder Objektidentifikatoren. Diese können als Bestandteil globaler Integritätsbedingungen (Existenz- oder Wertabhängigkeiten) in Regeln ausgewertet werden.

Semantische Objekte (S) repräsentieren Wissen über die lokalen Objekte, das in einem lokalen Schema nicht explizit dargestellt ist. Mögliche Realisierungen sind z.B. semantische Werte, die mit einem Kontext assoziiert sind (vgl. hierzu [SSR94, Mad95]). Dazu zählen u.a. Informationen über Skalierung (Genauigkeit), Bedeutungsdefinitionen von Attributen, Informationen über Maßeinheiten (z.B. Währungen, Längeneinheiten usw.), Datenformat-Infor-mationen (z.B. europäisch vs. amerikanisch). Die Semantik eines lokalen Objektes ist typischerweise in den Applikationsprogrammen verborgen, durch die dessen Verhalten repräsentiert ist. Änderungen lassen sich hier nur schwer erkennen

2.7.2 Beschreibung der Abhängigkeiten

Die Einbeziehung von Metadaten in einen Verbund heterogener Datenbanken führt zu weiteren Kategorien von globalen Konsistenzbedingungen, da auch Struktur und Verhalten der lokalen Datenbankobjekte in die Betrachtung einbezogen werden. Dementsprechend definieren wir Struktur- und Verhaltensabhängigkeiten wie folgt:

Definition 2.19 (Strukturabhängigkeit)
Eine Strukturabhängigkeit S ist dann gegeben, wenn die Struktur eines Objektes o_1 aus DB^1 von der Struktur eines anderen Objektes o_2 aus DB^2 abhängt, d.h. Änderungen in der Struktur von o_2 haben Einfluß auf die Struktur von o_1.

Definition 2.20 (Verhaltensabhängigkeit)
Eine Verhaltensabhängigkeit B ist dann gegeben, wenn das Verhalten eines Objektes o_1 aus DB^1 vom Verhalten eines anderen Objektes o_2 aus DB^2 abhängt, d.h. Änderungen in der Signatur oder der Implementierung der Methoden von Objekt o_2 haben Einfluß auf das Verhalten von Objekt o_1.

In einer Multidatenbank können folgende Abhängigkeiten in bezug auf die Metadaten gelten:

a) Existenz- und Wertabhängigkeiten zwischen lokalen Datenbankobjekten LDB-LDB
(siehe hierzu Tabelle 2.2 auf Seite 26)
Zur Spezifikation globaler Integritätsbedingungen in heterogenen Datenbanken werden Metadaten benötigt, die die unterschiedliche Semantik der lokalen Objekte explizit ausdrücken und somit Bestandteil des Konsistenzprädikates sein können:
   - Einbeziehung von semantischen Objekten S
   - Einbeziehung von Interdatenbank-Objekten ID zur Bestimmung korrespondierender Werte oder Objekte

b) Struktur- und Verhaltensabhängigkeiten LDB-GP
Da die globalen Proxy-Objekte eine Sicht auf die lokalen Objekte und damit auch deren Struktur repräsentieren, müssen lokale Schemaveränderungen in der Definition des Proxy-Objekts reflektiert werden (in [VPR95] als *local-global-relativism* bezeichnet).

Die Wiederherstellung der Konsistenz kann einerseits (laufzeit)dynamisch erfolgen, typischerweise nach Zustandsveränderungen bei der Kontrolle von Existenz- und Wertabhängigkeiten. Die Wahrung von Struktur- und Verhaltensabhängigkeiten kann demgegenüber nur off-line erfolgen in dem Sinne, daß Veränderungen im Schema oder in bestimmten Methoden eine Neu-

übersetzung davon betroffener Applikationen erfordern. Tabelle 2.5 zeigt noch einmal die typischen Fälle, wie globale Konsistenz verletzt und wiederhergestellt werden kann.

Kategorie	Lokales Ereignis	Wiederherstellungsaktion
S,B	explizite Änderung der Semantik (d.h. lokale Schemamodifikation)	Aktualisiere in GP
S,B	implizite Änderung der Semantik des lokalen Schemas (d.h. Modifikation der lokalen Applikationen)	Aktualisiere in S
E,V	Zustandsveränderung in LDB[1]	Aktualisiere in LDB[2]

Tabelle 2.5: Konsistenzregeln in einer Multidatenbankumgebung

2.8 Verwandte Arbeiten: Modellierung interdependenter Daten

In der Literatur finden sich zahlreiche Ansätze, die eine Klassifikation und Modellierung von Multidatenbank-Constraints (auch als *Interdependencies* bezeichnet) beschreiben, von denen einige in diesem Kapitel skizziert werden.

Wiederhold und Qian führen in [WQ87] den Begriff der *Identity Connection* als zusätzlichen Beziehungstyp in einem strukturellen Modell (formalisiertes ER-Modell) neben Owner-Membership- und Teilmengenbeziehungen ein. Ziel ist, replizierte Daten modellieren zu können, die in einer verteilten Umgebung asynchron geändert werden. Eine Identity Connection besagt, daß alle Tupel an der Beziehung beteiligt sind und daß für jedes Tupel die Zielfelder mit den Quellfeldern übereinstimmen. Die symbolische Darstellung erfolgt durch eine Linie mit Gleichheitszeichen an beiden Enden (=—=) oder nur an einem (—=), je nachdem ob es sich um eine bidirektionale Beziehung handelt oder nicht. Die Kardinalität von Identity Connections ist im allgemeinen 1:1 und kann zeitweilig partiell sein. Die Autoren empfehlen, mehrere Replikationsbeziehungen jeweils durch eine (binäre) Identity Connection zu modellieren. Denkbar ist bei komplexeren Constraints auch eine Kombination von Identity Connections mit anderen Beziehungstypen, beispielsweise bei partieller Replikation. Bei der Spezifikation von Identity Connections können auch Zeitconstraints modelliert werden (eventuelle Konsistenz), wodurch das Verhalten verteilter autonomer Systeme Berücksichtigung finden kann. Dabei gibt es drei Arten von Zeitconstraints:

Δ Zeitintervall, z.B. weniger als Δ = 12 h
@ Zeitpunkt, z.B. @5pm
! Zeitpunkt eines triggernden Ereignisses, das mit der Identity Connection verbunden ist

Das Modellierungskonstrukt ist flexibel im Hinblick auf die bereits diskutierten globalen Constraints soweit sie Wertabhängigkeiten betreffen. Ein Beispiel soll die Ausdrucksmöglichkeiten der Identity Connections demonstrieren.

Beispiel 2.1: (Identity Connection) [WQ87]
In einer Produktionsumgebung wird der Bestand durch einen lokalen Computer im Lager der Fabrik gewartet. Für den Verkauf werden die Bestandsdaten in tagesaktueller Form benötigt. Wegen der ständig stattfindenden Bestandsveränderungen im Lager kann der Verkaufscomputer nicht immer auf dem aktuellen Stand gehalten werden. Durch die Connection wird definiert, daß die Bestandsdaten in der Verkaufs-Datenbank täglich um 19 Uhr an die Lagerbe-

standsdaten angepaßt werden.
factory-stock: identity, factory-inv.{p-no, made} —= sales-inv.{p-id, avail}, @7pm;

Durch Anwendung von Aggregationsfunktionen lassen sich ebenso komplexere berechenbare Bedingungen ausdrücken, so daß der Gebrauch von Identity Connections auch für abgeleitete Daten möglich ist.

Sheth und Rusinkiewicz führen das Konzept der interdependenten Daten ein, d.h. Daten, die durch Konsistenzbedingungen miteinander verbunden sind, aber durch heterogene und autonome Systeme verwaltet werden. In [RSK91] werden *Data Dependency Descriptors* vorgestellt, durch die Interdependenzen zwischen Daten, die sich eventuell auch strukturell und semantisch stark unterscheiden, spezifiziert werden können. Die Deskriptoren umfassen nicht nur ein Abhängigkeitsprädikat, sondern auch den Grad der zulässigen Inkonsistenz und Methoden zur Wiederherstellung der Konsistenzbedingung, die vormals getrennt behandelt wurden. Ein Data Dependency Descriptor (D^3) besteht aus fünf Teilen:

$$D^3 = < S, U, P, C, A >$$

- *S* ist die Menge der Quelldatenobjekte (*Source*).
- *U* ist das Zieldatenobjekt (*Target*).
- *P* ist ein Abhängigkeitsprädikat (*Dependency Predicate*), durch das eine Beziehung zwischen Quell- und Zieldaten definiert wird und das bei Erfülltsein auf TRUE gesetzt ist.
- *C* ist das Konsistenzprädikat, durch das die Konsistenzanforderungen spezifiziert werden, d.h. insbesondere, wann das Prädikat *P* erfüllt sein muß. *C* kann in zwei Dimensionen definiert werden: einer Datenzustandsdimension *s* und einer temporalen Dimension *t*. *C* kann aus mehreren Termen c_i zusammengesetzt sein.
- *A* ist die Aktionskomponente und enthält Informationen darüber, wie die Konsistenz zwischen Quell- und Zieldaten wiederhergestellt werden kann. Hier wird typischerweise der Name einer Methode oder eines Programms angegeben. Zusätzlich kann ein Ausführungsmodus angegeben werden zur Kontrolle der zu triggernden Transaktion, innerhalb der *A* stattfindet: *coupled* / *decoupled* sowie *vital* / *non-vital*.

Temporale Konsistenzterme identifizieren den Zeitpunkt bzw. Zeitraum, wann die Bedingung *P* gelten muß. Die Symbolik basiert auf [WQ87], zusätzlich gibt es einen periodischen Operator, der z.B. so angewendet wird: ε(day @ 17:00) bedeutet “täglich 17.00 Uhr”.

Zustandsbasierte Konsistenzbedingungen bestimmen die zulässige Divergenz zwischen verbundenen Daten. Dieses Limit kann absolut oder relativ ausgedrückt werden, oder (falls Datenversionen unterstützt werden) durch eine Versionenanzahl, um die sich die Daten höchstens unterscheiden dürfen.

Die Konsistenz kann aber auch auf Operationen bezogen werden. So können wir die maximale Anzahl zulässiger Operationen auf einem bestimmten Objekt beschränken. Der zugehörige Konsistenzterm sieht z.B. so aus: 10 `updates on` R (R ∈ *S*). Durch dieses Push-Constraint wird ausgedrückt, daß nach 10 Update-Operationen auf der Relation R die Konsistenz wiederhergestellt sein muß. In gleicher Weise ließe sich auch ein Pull-Constraint formulieren (in [SHP88] als *lazy consistency* bezeichnet): `read on` U. Erst bei einem Zugriff auf die Zieldaten muß die Konsistenz wieder erfüllt sein.

Abschließend sei ein Beispiel für die Definition eines Dependency Descriptors [RSK91] gegeben, die in ihrer Gesamtheit ein sogenanntes *Interdatabase Dependency Schema* (IDS) bilden:

Beispiel 2.2: (Data Dependency Descriptor [RSK91])

S: D1.EMP	*S*: D3.EMP_COPY
U: D3.EMP_COPY	*U*: D1.EMP
P: EMP = EMP_COPY	*P*: EMP = EMP_COPY
C: ε(`day`)	*C*: 1 `update` **`on`** S
A: Duplicate_EMP	*A*: Propagate_Update_To_EMP **`as`** `coupled` **`and`** `vital`

Die Deskriptoren beschreiben eine bidirektionale Beziehung zwischen zwei Datenbankobjekten (hier Tabellen von Angestellten), die repliziert sein sollen, so daß das Prädikat *P* in beiden Deskriptoren übereinstimmt. Ein Update in der EMP_COPY Tabelle muß sofort in die EMP Tabelle propagiert werden, wohingegen eine Abgleichung der EMP_COPY Tabelle nur einmal täglich notwendig ist und zwar unabhängig vom Umfang der Änderungen, die in der Zwischenzeit in EMP eingebracht werden. Dies birgt allerdings auch die Gefahr in sich, daß Updates in D1.EMP verlorengehen können.

Sheth und Rusinkiewicz weisen darauf hin, daß die von ihnen verwendeten Deskriptoren (insbesondere die *C*- und *A*-Komponente) auch durch ECA-Regeln in aktiven Datenbanken ausgedrückt werden können (siehe Kapitel 3). Die D^3-Spezifikation ist nicht deklarativ, da die Wiederherstellungsprozeduren benutzerdefiniert sind.

Ceri und Widom unterscheiden in ihrem Ansatz zwei Kategorien von Abhängigkeiten: Wert- und Existenzabhängigkeiten [CW93]. *Existenzabhängigkeiten* drücken aus, daß das Vorhandensein eines Objekts in einer Datenbank die Existenz eines damit in Beziehung stehenden Objekts in einer anderen Datenbank impliziert. *Wertabhängigkeiten* besagen, daß der Wert eines Objekts in einer Datenbank auf irgendeine Weise mit dem Wert eines korrespondierenden Objekts in einer anderen Datenbank in Beziehung steht. Beide Arten von Abhängigkeiten lassen sich differenzieren in gerichtete und ungerichtete Beziehungen.

Allgemein werden gerichtete Existenzabhängigkeiten folgendermaßen definiert:

$T.(C_1, \ldots, C_n) \leftarrow$ **SELECT** $X_1, \ldots, X_m$ **FROM** $T_1, \ldots, T_k$ **WHERE** P $(m = n)$

Für jedes Tupel, das durch das SELECT-Statement auf der rechten Seite geliefert wird, existiert ein Tupel in der Tabelle *T*, projiziert auf die Spalten $C_1, \ldots, C_n$. Voraussetzung ist eine Übereinstimmung in den Domänen von C_i und X_i $(1 \le i \le n)$.

Beispiel 2.3: (Gerichtete Existenzabhängigkeiten [CW93])
Zur Illustration dient ein Beispiel zweier Datenbanken, die zwei Kraftwerksnetze beschreiben: Wenn es einen Knoten in der Datenbank IN (für Gesamt-Italien) gibt mit den Attributen region = 'Milano' und function = 'plant', so muß dieses Kraftwerk auch in der Datenbank MN (für Region Mailand) existieren:

```
mn.plant.plant-id   ←select id from in.node
                      where region = 'Milano' and function = 'plant'
```

Basierend auf der Spezifikation [CW93] beschreiben Castellanos, Kudrass u.a. [CKSG94] eine Klassifizierung der operationalen Aspekte am Beispiel der Existenzabhängigkeiten. Konsistenzverletzende Aktionen und mögliche Reaktionen werden kategorisiert und in einem Metaklassenansatz im BLOOM-Modell [CSG92] weitergeführt (vgl. auch Abschnitt 5.4.3).

Li und McLeod identifizieren vier Arten von möglichen Interdependenzen [LM93] zwischen zwei Objekten o_i und o_j aus zwei verschiedenen Datenbanken als Grundlage ihres Objektmo-

dells für Datenbankföderationen. Existenz- und Wertabhängigkeiten (ED bzw. VD) sind in gleicher Weise definiert wie bei [CW93]. Darüber hinaus werden strukturelle und verhaltensmäßige Abhängigkeiten eingeführt. Eine *strukturelle Abhängigkeit (SD)* besagt, daß strukturelle Veränderungen an einem Objekt einer Datenbank (z.B. Hinzufügen, Löschen oder Redefinition eines Attributs) die Struktur eines anderen Objekts einer anderen Datenbank beeinflussen. Eine *verhaltensmäßige Abhängigkeit (BD)* zwischen o_i und o_j liegt dann vor, wenn Änderungen am Verhalten von Objekt o_i (z.B. Modifikation einer Methodensignatur oder einer Implementierung) Änderungen des Verhaltens von o_j nach sich ziehen. Basierend auf dieser Taxonomie werden alle 15 möglichen Varianten (4 + 6 + 4 + 1) diskutiert, die von einfachen Abhängigkeiten bis hin zur vollständigen Kombinationen aller vier Arten reichen können. Ein Trivialfall wäre das Verhältnis von replizierten Daten, die alle vier Abhängigkeitsklassen erfüllen.

Beispiel 2.4: (BD + VD [LM93])
Objekt o_i ist von o_j in Verhalten und Wert abhängig (kurz: BV-abhängig). Das wäre der Fall, wenn Objekt o_j in DB1 eine Fernbedienung für das Fernsehgerät o_i in DB2 wäre. Verhalten und Zustand der Fernbedienung haben Auswirkung auf Verhalten und Zustand des Fernsehgeräts, können aber selbstverständlich nicht dessen Struktur oder Existenz beeinflussen.

Bewertung der Modelle

Für die Überwachung globaler Integritätsbedingungen in einem heterogenen System sollten mindestens folgende Anforderungen erfüllt sein:

- Modellierung globaler statischer Constraints (vgl. Abschnitt 2.4.3):
  Es muß unterschieden werden können, ob Bedingungen zwischen semantisch äquivalenten oder verschiedenen Daten ausgedrückt werden sollen. Es handelt sich entweder um Bedingungen, die sich auf Werte von Objekten (z.B. Implikationsbedingungen, Aggregatbedingungen) oder auf die Existenz von Objekten (z.B. Uniqueness, referentielle Integrität) beziehen.

- Darstellung prozeduraler Aspekte / Ausführungsspezifikation:
  Während es bei der Constraint-Modellierung nur um das WAS geht, sollte es auch möglich sein, das WIE zu beschreiben, insbesondere im Hinblick auf Aktionen zur Wiederherstellung der Konsistenz. Hier ist auch eine Menge von Default-Aktionen denkbar, die groß genug ist, alle bedeutenden Situationen abzudecken.

- Abschwächung von Konsistenzbedingungen nach zeitlichen oder extensionalen Aspekten:
  Zugrunde liegt die Annahme, daß aufgrund des autonomen Verhaltens der beteiligten Systeme nicht zu jedem Zeitpunkt und an jedem Ort alle Constraints erfüllt sein können und somit der Konsistenzbegriff weiter gefaßt werden muß.

- Lokale Datenmodellunabhängigkeit:
  Die Ausdrucksmittel zur Beschreibung der globalen Integritätsbedingungen sollten unabhängig von den Datenmodellen der beteiligten lokalen Datenbanken sein bzw. sich leicht auf deren Datenmodelle abbilden lassen.

- Unterstützung unterschiedlicher Kontrollabhängigkeiten in einem verteilten System:
  Das Verhältnis von Datenbanken (möglicherweise auch von einzelnen Teilen) zu anderen wird bestimmt durch deren lokale Autonomie, d.h. je autonomer ein System, umso geringerer Kontrolle ist es unterworfen.

Weitere, als optional anzusehende, Kriterien umfassen:

- Modellierung globaler struktureller und verhaltensmäßiger Constraints.
- Modellierung globaler dynamischer Integritätsbedingungen (transitional, temporal).
- Berücksichtigung syntaktischer Heterogenität zwischen Domänen.

Gemäß den als notwendig erkannten Kriterien soll in Tabelle 2.6 eine Einordnung und Bewertung der vier soeben skizzierten Modellierungsansätze vorgenommen werden:

Kriterium	Identity Connections [WQ87]	D^3 und IDS [RSK91]	Produktions-regeln [CW93]	Abhän-gigkeiten [LM93]
Modellierung globaler semantischer Constraints:				
- Wert- / Existenzabhängigkeiten	+	o	+	+
- Unterscheidung semantisch äquivalente vs. verschiedene Daten	+	-	-	o
Prozedurale Aspekte	-	+	-	-
Abschwächung von Konsistenzbedingungen	+	+	-	-
Datenmodellunabhängigkeit	+	+	-	-
Unterstützung globaler Kontrollabhängigkeiten	-	-	-	-

+ gut unterstützt / O teilweise unterstützt / - nicht unterstützt

Tabelle 2.6: Bewertung von Ansätzen zur Modellierung von globalen Konsistenzbedingungen

Daraus ergibt sich, daß speziell Probleme, die aus der lokalen Autonomie der teilnehmenden Datenbanksysteme resultieren, bereits bei der Modellierung von globalen Integritätsbedingungen verstärkt Beachtung finden müssen. Die möglichen Wechselwirkungen und Zielkonflikte zwischen unterschiedlichen Graden lokaler Autonomie und global ausgeübter Kontrolle müssen eingehender untersucht werden, hierzu sei insbesondere auf Abschnitt 4.4 auf Seite 66 verwiesen.

Kapitel 3

Konzepte aktiver Datenbanksysteme

In diesem Kapitel gehen wir auf das ECA-Regelkonzept ein, das aktiven Datenbanksystemen zugrunde liegt und ein ausdrucksmächtiges Mittel zur Spezifikation von Integritätsbedingungen darstellt. Anders als bei logikbasierten Formalismen, die auf Deklarationen beruhen (z.B. in deduktiven Datenbanken), können Bedingungen und Aktionen von ECA-Regeln mit den Mitteln des DBMS oder einer Wirtssprache formuliert werden. Die Grundidee von ECA-Regeln zur Integritätssicherung beruht darauf, die Verletzung der Datenbankkonsistenz zu erkennen und entsprechende Aktionen zur Wiederherstellung eines konsistenten Zustandes einzuleiten.

In Abschnitt 3.2 wird eine Charakterisierung aktiver Datenbanksysteme vorgenommen, die sich an drei Aspekten orientiert. Zum einen kann man das Regelmodell beschreiben, indem man die Bestandteile von ECA-Regeln (Event, Condition, Action) sowie deren Beziehungen analysiert, um die Ausdrucksmöglichkeiten der Regeln zu bewerten. Wann und wie die Regeln abgearbeitet werden, ist durch das Ausführungsmodell eines aktiven DBS festgelegt. Der dritte Bewertungsaspekt umfaßt die Möglichkeiten, die ein System zum Regelmanagement bietet.

Ausgehend von dem Ziel der Sicherung globaler Integrität, werden die genannten Eigenschaften im Hinblick auf ihre Bedeutung für die globale Konsistenz betrachtet, wobei nach dem Grad der angestrebten Konsistenz unterschieden werden muß.

3.1 Grundbegriffe

Aktive Datenbanksysteme sind Datenbanksysteme, die bestimmte Situationen beobachten und bei ihrem Eintreten in definierter Weise reagieren. Ihr Verhalten wird durch Event-Condition-Action-Regeln (ECA-Regeln) beschrieben, die in der Datenbank definiert und gespeichert werden. Ihr Prinzip ist in Abbildung 3.1 veranschaulicht. Im Gegensatz dazu führen konventionelle passive Datenbanksysteme nur Anfragen oder Transaktionen aus, die explizit von einem Benutzer oder Applikationsprogramm gestartet wurden.

Aktive Datenbanksysteme sind für die Wahrung von Integritätsbedingungen geeignet, indem sie durch die Definition von ECA-Regeln eine einheitliche und ausdrucksmächtige Spezifikation der zu kontrollierenden Konsistenz ermöglichen. Ein aktives DBMS ist verantwortlich für die Detektion der Ereignisse und deren Signalisierung an die Regelausführungskomponente. Ein Ereignis kann ein oder mehrere Regeln triggern. Eine Regel wird ausgeführt, indem der

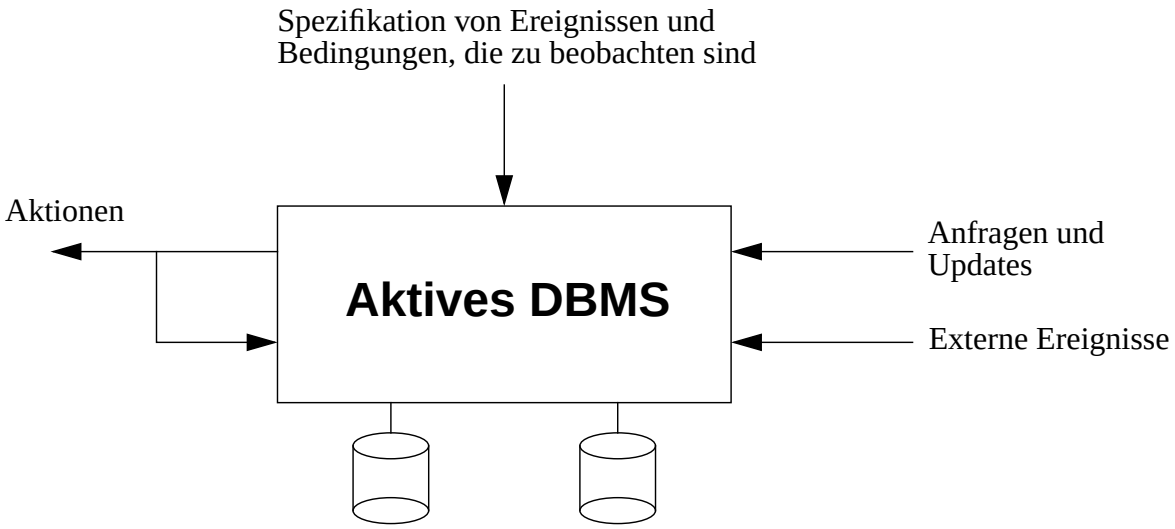


Abbildung 3.1: Aktive Datenbanksysteme

Bedingungsteil ausgewertet wird und bei Erfülltsein der Bedingung die Aktion ausgeführt wird. Syntaktisch wird eine ECA-Regel in folgender Form notiert:

ON <event> **IF** <condition> **DO** <action>

Die Bestandteile einer ECA-Regel können durch sogenannte Kopplungsmodi verbunden sein. Mit der Kopplung bezeichnet man die zeitliche Beziehung zwischen einem Ereignis und der von ihm aktivierten Aktion. Dabei wird unterschieden zwischen der Kopplung Event-Condition (EC) sowie der Kopplung Condition-Action (CA). Eine klare Trennung zwischen Ereignis und Bedingung wurde erstmals im Rahmen des HiPAC-Projekts gefordert [DBM88].

Die erlaubten Kombinationen, wie sie in HiPAC definiert wurden [HLM88], sind in Tabelle 3.1 dargestellt.[6] Der Kopplungsmodus DETACHED kann auf zweierlei Weise interpretiert werden:

INDEPENDENT: Angestoßene Aktionen werden in einer eigenständigen Transaktion ausgeführt, deren Verlauf unabhängig vom Ergebnis der ursprünglichen Transaktion ist.

CAUSALLY DEPENDENT: Die getriggerte Aktion wird in Abhängigkeit vom Ergebnis der triggernden Transaktion ausgeführt. In aktiven Datenbanken können auch Ereignisse auftreten, die nicht dem Transaktionsprinzip unterliegen, d.h. nicht zurückgesetzt werden können. Aus diesem Grunde wurden in [BBKZ93, Buc94] weitere Kopplungsmodi zusätzlich eingeführt. Dazu zählen SEQUENTIAL (Beginn der Transaktion nach dem Commit der triggernden Transaktion), PARALLEL (Beginn der getriggerten Transaktion parallel zur triggernden Transaktion) und EXCLUSIVE (Commit der getriggerten Transaktion nur bei Abbruch der triggernden Transaktion).

6 Die Kopplungsmodi *detached* und *decoupled* sind synonym.

	CA-Modus		
EC-Modus	IMMEDIATE	DEFERRED	DETACHED
IMMEDIATE	Condition und Action unmittelbar nach Event in der gleichen Transaktion	Condition unmittelbar nach Event, Action am Ende der Transaktion	Condition unmittelbar nach Event, Action in separater Transaktion
DEFERRED	nicht erlaubt	Condition und Action am Ende der Transaktion	Condition am Ende der Transaktion, Action in separater Transaktion
DETACHED	Condition und Action in separater Transaktion	nicht erlaubt	Condition in separater Transaktion, Action in einer weiteren Transaktion

Tabelle 3.1: Kombinationen von Kopplungsmodi

In [DGG95] sind die wesentlichen Anforderungen an aktive Datenbanksysteme in einem Manifesto zusammengefaßt. Einen vergleichenden Überblick über aktive Datenbanksysteme findet man u.a. in [HW93, PDW+93, WC96].

3.2 Charakterisierung aktiver Datenbanksysteme

Nach einer Vielzahl von Prototyp-Entwicklungen aktiver Datenbanksysteme enstand die Notwendigkeit, diese qualitativ zu vergleichen und gleichzeitig auch die allen gemeinsamen Merkmale zu klassifizieren (*Feature Benchmark*) [CB95]. Die Diskussion der Eigenschaften soll dazu dienen, die Merkmale zu charakterisieren, die eine Voraussetzung für die Integritätskontrolle bilden, bzw. den Zusammenhang zwischen den vorhandenen Eigenschaften und dem Grad der zu wahrenden Integrität herzustellen.

Der Benchmark erfolgt auf fünf Ebenen: Event, Condition, Action, Ausführungsmodell und Management-Fähigkeiten (bezogen auf das Management der Regeln).

Event	
E-1	Role $\in$ {mandatory, optional, none}
E-2	Source $\subset$ {db-operation, method invocation, transaction, clock, error, external}
E-3	Type $\subset$ {primitive, composite}
E-4	Scope $\subset$ {instance, collection}
E-5	WhenRaised $\subset$ {before, after}
Condition	
C-1	Role $\in$ {mandatory, optional}
C-2	Mode $\subset$ {immediate, deferred, detached}
C-3	Scope $\subset$ {instance, target, database}
C-4	Available State $\subset$ {transaction, event, condition}

Tabelle 3.2: Bewertung aktiver Systeme

Action	
A-1	Options $\subset$ {update-db, update-rules, inform, abort, do-instead}
A-2	Mode $\subset$ {immediate, deferred, detached dependent, detached independent}
A-3	Scope $\subset$ {instance, target, database}
A-4	Available State $\subset$ {transaction, event, action}
Execution Model	
X-1	Transition Granularity $\in$ {instance, set}
X-2	Binding Model $\subset$ {instance, set, prior}
X-3	Constraints $\subset$ {timing, alternatives}
X-4	Scheduling $\in$ {all fired, some fired}
X-5	Priorities $\in$ {numerical, relative, none}
X-6	Conflict Resolution {one, all sequential, all parallel, unspecified}
X-7	Run-Time Depth Limit $\in$ {none, limited, unspecified}
X-8	Nested Transaction $\in$ {none, yes}
Management	
M-1	Data Model $\in$ {relational, extended relational, deductive, object-oriented,...}
M-2	Description $\subset$ {programming language, extended query language, objects}
M-3	Operations $\subset$ {activate, deactivate}
M-4	Adaptability $\in$ {compile time, run time}
M-5	Authorization $\in$ {none, yes}
M-6	Modularization $\in$ {none, data model, yes}

Tabelle 3.2: Bewertung aktiver Systeme (Forts.)

Event

E-1: *Role*
Hierdurch wird bestimmt, ob Events bei einer Regel immer angegeben werden müssen (*mandatory*) oder nicht zwingend erforderlich sind (*optional*, *none*). Im ersten Fall werden nur ECA-Regeln unterstützt, ansonsten CA-Regeln (Condition-Action).

E-2: *Source*
Ein Event kann ausgelöst werden durch eine Datenbank-Operation (*db-operation*) oder durch den Aufruf einer benutzerdefinierten Methode oder Operation (*method invocation*). Ein Event kann aber auch zu bestimmten Zeitpunkten in einer Transaktion ausgelöst werden (*transaction*), z.B. bei Beginn oder bei Abbruch der Transaktion. Denkbare Events sind auch Zeit-Events (*clock*). Weitere Möglichkeiten, die zur Auslösung eines Events führen können, sind Fehlersituationen (*error*) oder externe Einflüsse (*external*).

E-3: *Type*
Ein Event kann einfach (*primitive*) oder zusammengesetzt (*composite*) sein. Ein kompositer Event entsteht durch eine Kombination von primitiven oder kompositen Events durch Anwendung von Operatoren, die in einer Event-Algebra zur Verfügung stehen [DBM88, CM91].

E-4: *Scope*
Der *Scope* (in [PDW+93] als *Level* bezeichnet) gibt an, ob ein Event für jedes Objekt einer Menge, d.h. eine Klasse oder Relation, definiert ist (*collection*) oder für spezifische Instanzen (*instance*), z.B. zur Zugriffskontrolle.

E-5: *WhenRaised*
Diese Eigenschaft wird nur für Datenbank- oder Transaktions-Events betrachtet und besagt, ob

der Signalisierungszeitpunkt des Events vor (*before*) oder nach (*after*) der Ausführung der zum Event gehörenden Operation stattfindet.

Condition

C-1: *Role*
Eine Condition kann vorgeschrieben (*mandatory*) oder *optional* sein. Bei Fehlen einer Bedingung wird die Aktion in jedem Fall ausgeführt, auch als EA-Regeln bezeichnet (Event-Action).

C-2: *Mode*
Der Kopplungsmodus *Mode* beschreibt die Ausführung der Condition relativ zum auslösenden Event. Hierzu sei auf Tabelle 3.1 verwiesen.

C-3: *Scope*
Durch den *Scope* wird die Reichweite der Objekte definiert, die durch die Condition angesprochen werden können. Dies ist im allgemeinen immer die gesamte Datenbank (*database*), wie in [CB95] untersucht wurde, könnte aber in Einzelfällen auch eine einzelne Instanz (*instance*) oder eine Tabelle bzw. Klasse (*target*) sein.

C-4: *Available State*
In diesem Parameter wird festgelegt, welcher Datenbankzustand für die Condition zum Ausführungszeitpunkt sichtbar ist. Folgende Möglichkeiten sind denkbar: der Zustand zu Beginn der eventauslösenden Transaction (*transaction*), der Zustand zum Auftrittszeitpunkt des Events (*event*) oder der aktuelle Zustand, wie er für die Condition bei deren Ausführung sichtbar ist (*condition*). Dieser Parameter wird in [CB95] weiter verfeinert und als *Past State Reference* und *Past State Scope* jeweils für Condition und Action beschrieben.

Action

A-1: *Options*
Die möglichen Operationen, die durch die Action ausgeführt werden, sind im Parameter *Options* beschrieben. Typische Reaktionen sind: ein Update auf der Datenbank (*update-db*), die Ausgabe einer Nachricht (*inform*), das Zurücksetzen der eventauslösenden Aktion durch Abbruch der Transaktion oder Ausführung einer kompensierenden Operation (*abort*). Denkbare Möglichkeiten sind auch ein dynamisches Ändern von Regeln (*update-rules*) oder die Ausführung einer alternativen Aktion an Stelle der ursprünglichen eventauslösenden Aktion (*do-instead*).

A-2: *Mode* / A3: *Scope* / A4: *Available State*
Hierfür gelten dieselben Aussagen wie bei der Condition (siehe C-2 bis C-4).

Ausführungsmodell

X-1: *Transition Granularity*
Die Granularität der eventauslösenden Zustandsübergänge kann ein einzelnes Objekt (*instance*) oder eine Menge von geänderten Objekten (*set*) sein. Letzteres ist dann möglich, wenn eine mengenorientierte DML wie SQL verwendet wird.

X-2: *Binding Model*
Im Binding Model wird beschrieben, welche Informationen vom Event an die Condition oder Action weitergereicht werden. Dabei kann das vom Event betroffene Objekt (*instance*) oder

eine Menge von Objekten (*set*) übergeben werden (bei mengenwertiger Transitionsgranularität). Außerdem kann der Zustand eines Wertes vor dem Event (*prior*) verfügbar sein.

X-3: *Constraints*
Mit dem Parameter Constraints wird im Ausführungsmodell beschrieben, ob zeitliche Beschränkungen für die Ausführung der Regel definiert werden können (*timing*) oder auch Alternativen möglich sind, wenn das Constraint verletzt ist (*alternative*), in HiPAC als *Contingency Action* bezeichnet [DBM88]. Eine Garantie für die Einhaltung eines zeitlichen Constraints könnte nur ein echtzeitfähiges System geben [BB95].

X-4: *Scheduling*
In *Scheduling* ist definiert, ob beim gleichzeitigen Triggern mehrerer Regeln alle (*all fired*) oder nur einige (*some fired*) Regeln ausgeführt werden.

X-5: *Priorities*
Die Regelauswahl kann durch Prioritäten gesteuert werden, absolut (*numerical*), in Beziehung zu anderen Regeln (*relative*), oder es sind keine Prioritäten vorgesehen (*none*).

X-6: *Conflict Resolution*
In [CB95] sind verschiedene Möglichkeiten der Konfliktauflösung beschrieben.

X-7: *Run-Time Depth Limit*
Dieser Parameter bestimmt die Aufruftiefe kaskadierender Regeln.

X-8: *Nested Transactions*
Hiermit wird ausgesagt, ob geschachtelte Transaktionen unterstützt werden oder nicht.

Management der Regeln

M-1: *Data Model*
Das Datenmodell, auf dem ein Regelsystem basiert, hat Einfluß auf den Charakter des Gesamtsystems (z.B. Art der Events, Art der Objekte).

M2: *Description*
Regeln können ausgedrückt werden mit den Mitteln einer Programmiersprache (*programming language*), einer um Regelkonstrukte erweiterten Anfragesprache (*extended query language*) oder als Objekte (*objects*). In [CB95] erfolgt noch eine Unterscheidung danach, wie Condition und Action ausgedrückt werden können, z.B. als DML-Anweisung, als Prozeduraufruf oder als Prädikat (nur bei Condition).

M-3: *Operations*
Aktive Systeme sollten die Möglichkeiten bieten, Regeln ein- und auszuschalten. Eine ausgeschaltete Regel reagiert somit nicht mehr auf ein Event. Allerdings sollten die Gefahren bei Konsistenzregeln bedacht werden.

M-4: *Adaptability*
Durch den Parameter *Adaptability* wird ausgedrückt, wie flexibel Änderungen in Regeln möglich sind, entweder nur statisch zum Übersetzungszeitpunkt (*compile time*) oder auch dynamisch zur Laufzeit (*run time*).

M-5: *Authorization*
Hiermit wird ausgesagt, ob das System eine Autorisierungskontrolle für die Regeln unterstützt oder nicht.

M-6: *Modularization*
Dieses Kriterium beschreibt, ob das aktive System die Möglichkeit bietet, Regeln zu gruppieren, um sie gemeinsam zu behandeln. Dabei können auch die Fähigkeiten des Datenmodells ausgenutzt werden.

3.3 Globale Integritätskontrolle durch aktive Systeme

Aktive Datenbanksysteme sollen auch bei der Kontrolle in heterogenen und autonomen Informationssystemen Anwendung finden, was deren Fähigkeit voraussetzt, mehrere (möglicherweise) lokale Komponentensysteme zu integrieren. Eine wesentliche Voraussetzung für aktives Verhalten ist die Fähigkeit zur Eventdetektion, die in einem heterogenen System für alle Komponenten gegeben sein muß. Dabei sind möglicherweise sehr verschiedene lokale Event-Detektoren beteiligt. Im Unterschied zu einem homogenen aktiven DBMS gibt es nicht nur einen einzigen Transaktionsmanager, so daß die Ausführung der zu triggernden Aktionen unter der Kontrolle der jeweiligen lokalen Systeme erfolgt. Weitere Ausführungen hierzu findet man in [DGG95].

Die Integritätsbedingungen in heterogenen Datenbanken wurden bereits in Abschnitt 2.4 auf Seite 18 charakterisiert. Die Anforderungsanalyse aus [PDW+93] wird um eine globale Reichweite ergänzt, die alle beteiligten Datenbanken umfaßt (*all databases*), sowie eine Differenzierung nach dem erlaubten Konsistenzgrad vorgenommen. Dabei wird unterschieden, ob ein global korrekter Datenbankzustand immer gelten muß (strenge Konsistenz) oder Abweichungen entlang der Zeitdimension oder der Reichweite erlaubt sind (abgeschwächte Konsistenz, vgl. Abschnitt 2.5). Je nach der zulässigen extensionalen Abschwächung ergeben sich Anforderungen an die *Scope* Eigenschaften. Tabelle 3.3 stellt die relevanten Anforderungen in beiden Fällen gegenüber.

Param.	Strenge Konsistenz	Zeitlich und extensional abgeschwächte Konsistenz
E-1	Role $\in$ {mandatory}	Role $\in$ {mandatory}
E-2	Source $\subset$ {db-operation}	Source $\subset$ {db-operation, clock, external}
E-3	Type $\subset$ {primitive}	Type $\subset$ {primitive}
E-4	Scope $\subset$ {collection}	Scope $\subset$ {collection}
E-5	WhenRaised $\subset$ {after}	WhenRaised $\subset$ {after}
C-1	Role $\in$ {mandatory}	Role $\in$ {mandatory}
C-2	Mode $\subset$ {immediate, deferred}	Mode $\subset$ {detached dependent, detached independent}
C-3	Scope $\subset$ {all databases}	Scope $\subset$ {instance, target database}
C-4	Available State $\subset$ {event, condition}	Available State $\subset$ {event, condition}
A-1	Options ={update-db, inform, abort}	Options ={update-db, inform, abort}
A-2	Mode $\subset$ {immediate, deferred}	Mode $\subset$ {detached dependent, detached independent}
A-3	Scope $\subset$ {all databases}	Scope $\subset$ {instance, target, database}
A-4	Available State $\subset$ {event, action}	Available State $\subset$ {event, action}
X-1	Transition Granularity $\in$ {instance, set}	Transition Granularity $\in$ {instance, set}

Tabelle 3.3: Eigenschaften aktiver Systeme zur globalen Integritätssicherung

Param.	Strenge Konsistenz	Zeitlich und extensional abgeschwächte Konsistenz
X-2	Binding Model $\subset$ {instance, set, prior}	Binding Model $\subset$ {instance, set, prior}
X-3	Constraints $\subset$ {}	Constraints $\subset$ {timing, alternative}
X-4	Scheduling $\subset$ {some fired}	Scheduling $\subset$ {some fired}
M-3	Operations $\subset$ {activate, deactivate}	Operations $\subset$ {activate, deactivate}
M-4	Adaptability $\in$ {run time}	Adaptability $\in$ {run time}

Tabelle 3.3: Eigenschaften aktiver Systeme zur globalen Integritätssicherung (Forts.)

Wesentlich ist, daß Datenbank-Events detektiert werden können, bei zeitbezogener Konsistenz auch temporale Events. Events müssen klassenbezogen definierbar sein. Zur Erkennung von Konsistenzverletzungen genügen *after*-Events, so daß der bereits veränderte Zustand ausgewertet werden kann. Unterschiede zwischen strenger und abgeschwächter Konsistenz gibt es bei den Kopplungsmodi, weil im Fall der strengen Konsistenz davon ausgegangen wird, daß die Constraint-Verletzung immer innerhalb der triggernden Transaktion behandelt wird (spätestens beim Commit). Dem liegt aber die Annahme zugrunde, daß die Transaktion die *Unit of Consistency* (Konsistenzeinheit) ist, wie es typischerweise für flache oder geschlossen geschachtelte Transaktionen zutrifft. Wenn Transaktionen global konsistenzverletzend sein können, muß eine Korrektur auch außerhalb der Transaktion möglich sein, entweder als *detached* Transaktion oder zu einem benutzerdefinierten Zeitpunkt. Geeignete Transaktionsmodelle für verteilte aktive Objektsysteme sind z.B. das DOM-Transaktionsmodell [BÖH+92] oder das Polytransaktionsmodell [SRK92]. Die auszuführenden Aktionen können Datenbankoperationen sein, aber auch (wenn keine automatische Wiederherstellung möglich ist) eine Benachrichtigung der Benutzer oder der Abbruch der konsistenzverletzenden Operation.

Das Ausführungsmodell sollte eine instanz- oder mengenwertige Granularität unterstützen. Entsprechend gilt, daß das *Binding Model* Instanzen oder Mengen vorsehen muß. Für dynamische Constraints werden auch die alten Attributwerte (*prior*) benötigt. Bei abgeschwächter Konsistenz sollten Timeout Constraints und ggf. Alternativen möglich sein. Zur Regelauswahl genügt meist die *some fired* Strategie, wenn bei einer Konsistenzverletzung mehrere Regeln mit derselben Wirkung gefeuert werden sollen.

Das Anlegen von Regeln zur Laufzeit ist zwar nicht notwendig, sollte aber unabhängig von den Applikationen und dem Erzeugen des (globalen) Datenbankschemas geschehen. Dynamisches Ein- und Ausschalten der Regeln ist für zeitlich wechselnde Konsistenzanforderungen oder bei Nichtverfügbarkeit bestimmter Komponenten-Datenbanken nützlich. Allerdings bedeutet die Aktivierung einer Regel nicht, daß Constraints, die durch diese Regel beschrieben werden, sofort geprüft werden. Nachträgliches Einfügen bzw. Aktivieren von Regeln verursacht das Problem, daß ehemals konsistente Daten plötzlich als inkonsistent gelten. Idealerweise sollte ein solcher Fall vermieden werden, um zu garantieren, daß ein konsistenter Zustand solange gilt, wie die entsprechende ECA-Regel aktiv ist. Ansonsten sollte das Aktivieren bzw. Anlegen von Regeln als ein besonderes Event auf einer Meta-Ebene betrachtet werden, das eine nachträgliche Konsistenzprüfung auslöst. Die Reaktionen hängen von der zeitlichen Abschwächung des Konsistenzkriteriums ab. Sie können reichen von einer Benachrichtigung (*inform*), bis hin zu einer nachträglichen Korrektur der Datenbank (*update-db*). Der Aspekt dynamischer Regeländerung im Kontext von CAD-Datenbanken wird u.a. in [BD88] diskutiert.

Kapitel 4

Lokale Autonomie in Multidatenbanken

Der Begriff der Autonomie erlangte Bedeutung im Zusammenhang mit der Entwicklung der Konzepte von föderierten Datenbanksystemen, wobei bei der Integration lokaler Datenbanksysteme zu entscheiden ist, was diese an Rechten und Freiheitsgraden aufzugeben haben. Dieses Kapitel gliedert sich wie folgt:

In Abschnitt 4.1 erfolgt zunächst eine Einführung der gängigen Grundbegriffe, die in Multidatenbanksystemen Verwendung finden, woraus sich auch eine Klassifizierung verschiedener Architekturen ergibt. Nachfolgend (Abschnitt 4.2) wird die Architektur von Multidatenbanksystemen anhand der bekannten Fünf-Ebenen-Referenzarchitektur besprochen. Dabei wird ein Vergleich angestellt, inwieweit unterschiedliche Multidatenbank-Architekturen sich für die Wahrung globaler Constraints eignen.

Einen Schwerpunkt bildet Abschnitt 4.3, der einen Überblick enthält, welche Autonomiedefinitionen bereits Eingang in die Literatur gefunden haben. Die “Geschichte” der Autonomie beginnt mit der Entwicklung verteilter Systeme und wird um neue Gesichtspunkte bereichert, wenn heterogene Datenbanksysteme betrachtet werden. Autonomie wurde zunächst hauptsächlich im Zusammenhang mit lokalen Transaktionen und der Notwendigkeit für unveränderten Zugriff der lokalen Anwendungen auf ihre Daten benutzt. Prinzipiell sind zwei Aspekte von Autonomie erkennbar, diejenigen, die bei der Integration lokaler Komponenten eine Rolle spielen, und jene, deren Auswirkungen zur Laufzeit des föderierten Systems sichtbar werden. Der Begriff der Autonomie läßt sich auch verallgemeinern, wenn interagierende Einheiten betrachtet werden. Dabei ist die Feststellung interessant, daß die Autonomie lokaler Systeme ein Ausdruck der Autonomie der (Teil-)Organisationen ist, die diese Systeme kontrollieren und benutzen.

Bisher wurden die Begriffe Konsistenz und Autonomie - sowohl in dieser Arbeit als auch in der existierenden Literatur - überwiegend getrennt diskutiert. Der Gegensatz von globaler Konsistenz einerseits und lokaler Autonomie andererseits wurde schon von mehreren Autoren angesprochen. Abschnitt 4.4 arbeitet anhand des Problems der globalen Integritätssicherung heraus, welche Trade-Offs zwischen beiden Zielen auftreten können. Dazu definieren wir Grade lokaler Autonomie von Legacy-Systemen. Ausgehend von dem bereits in Kapitel 2 eingeführten Konsistenzbegriff, der in drei Dimensionen definiert wird, leiten wir aus diesem eine prozedurale Sichtweise ab. Unsere Voraussetzung ist, daß die Integritätskontrolle auf aktiven Mechanismen beruht, deren Anwendung die Kooperationsbereitschaft der lokalen Teilnehmer erfordert, verbunden mit einer (teilweisen) Aufgabe von deren Autonomie. In einer Übersicht

stellen wir die Einschränkung einzelner Autonomiegrade den damit erreichbaren Konsistenztypen gegenüber. Neben dieser qualitativen Sichtweise skizzieren wir auch einen Ansatz zur quantitativen Bewertung von Autonomie.

Die Untersuchung der Konsequenzen lokaler Autonomie hat eine zentrale Bedeutung für die Integration einzelner Komponenten, zwischen denen datenbankübergreifend Konsistenz gewahrt werden soll.

4.1 Grundbegriffe und Klassifizierung von Multidatenbanksystemen

Zwei gegenläufige Trends im Einsatz von Datenbanksystemen bestimmen die gegenwärtige Entwicklung. Nachdem bis Ende der 70er Jahre zentralisierte Datenbanksysteme dominierten, zeichnet sich nun eine Entwicklung zu Dezentralisierung und Verteilung ab, die durch technologische Fortschritte (Netzwerktechnologie, Ablösung von Mainframe-Systemen durch PCs und Workstations) forciert wird. Zugleich werden Technologien benötigt, die auch in einer veränderten, weitgehend dezentralen DV-Landschaft, einen integrierten Zugriff auf die Datenbestände des Unternehmens ermöglichen. Eine mögliche Antwort auf diese Probleme kann durch die Entwicklung und den Einsatz von sogenannten *Multidatenbanksystemen* gegeben werden, die somit auch den Investitionsschutz für Hardware und Software gewährleisten. Eine andere Lösung beim integrierten Zugriff auf heterogene, historisch gewachsene Datenbestände eines Unternehmens, bieten sogenannte *Data Warehouses* [Inm96].

Ein Multidatenbanksystem (in [HM85] auch als *föderiertes Datenbanksystem* bezeichnet) besteht aus einer Anzahl von kooperierenden aber autonomen Komponentendatenbanksystemen. Die Komponentendatenbanksysteme können in unterschiedlichem Ausmaß integriert sein, wodurch sich auch eine Klassifizierung der Multidatenbanksysteme ergibt. Multidatenbanksysteme erlauben den integrierten Zugriff auf heterogene, bereits existierende Datenbanken in einem verteilten System. Sie werden entworfen im bottom-up-Verfahren, d.h. ausgehend von vorhandenen Datenbanken, die möglicherweise unterschiedliche Weltausschnitte modellieren oder dasselbe Modell in unterschiedlichen Schemata repräsentieren. Die am Verbund teilnehmenden Datenbanken werden wir nachfolgend als *Komponentendatenbanken* bezeichnen. Ein wesentliches Merkmal von Multidatenbanksystemen ist, daß jeder teilnehmende Knoten die lokale Kontrolle über seine Ressourcen und Verarbeitung beibehält. Die globale Kontrolle beruht auf einer Abstimmung zwischen den Teilnehmern über den Grad der Kooperation, was mögliche Einschränkungen der Autonomie einschließt.

Multidatenbanksysteme lassen sich charakterisieren in drei zueinander orthogonalen Dimensionen: Verteilung, Heterogenität, Autonomie. Verteilung umfaßt die physische und geographische Verteilung der Daten auf den Rechnersystemen, die auf unterschiedliche Weise erfolgen kann. Typisch hierfür sind Replikation von Daten, vertikale und horizontale Partitionierung. Heterogenität kann auf unterschiedlichen Abstraktionsebenen betrachtet werden. Sie reicht von Unterschieden in der Hardware, den Betriebs- und Datenbanksystemen bis hin zu syntaktischen und semantischen Differenzen zwischen lokalen Datenbankschemata. Eine ausführliche Darstellung dazu geben Kim und Seo in [KS91]. Die dritte Dimension, Autonomie, kennzeichnet die Fähigkeit von Komponentendatenbanksystemen, Entscheidungen über ihre Struktur und ihr Verhalten zu treffen ohne externe Einflußnahme, die insbesondere in Datenbankföderationen notwendig ist. So hat beispielsweise die Notwendigkeit, ein unverändertes Funktionieren bestehender Programme zu gewährleisten, eine eingeschränkte Kooperationsbereitschaft

lose gekoppelter Organisationen zur Folge. Eine Darstellung der Autonomie-Problematik im Kontext der Wahrung globaler Integritätsbedingungen gibt Abschnitt 4.4.

Hauptalternativen für die Architektur von Multidatenbanksystemen sind in Abbildung 4.1 dargestellt. Gemeinsam ist allen, daß das Problem der Datenmodellheterogenität (bzw. Schemaheterogenität) durch die Einführung eines kanonischen Datenmodells (CDM) behandelt wird. Die Benutzer der Föderation greifen auf die Komponentendatenbanken über eine Menge von Sichten (Views) zu, so daß Anfragen auf diesen Sichten in Anweisungen der jeweiligen lokalen Query-Sprache übersetzt werden müssen. Der prinzipielle Unterschied zwischen den Architekturen liegt darin, welche Sichten verfügbar sind und wer für deren Bereitstellung und Wartung verantwortlich ist.

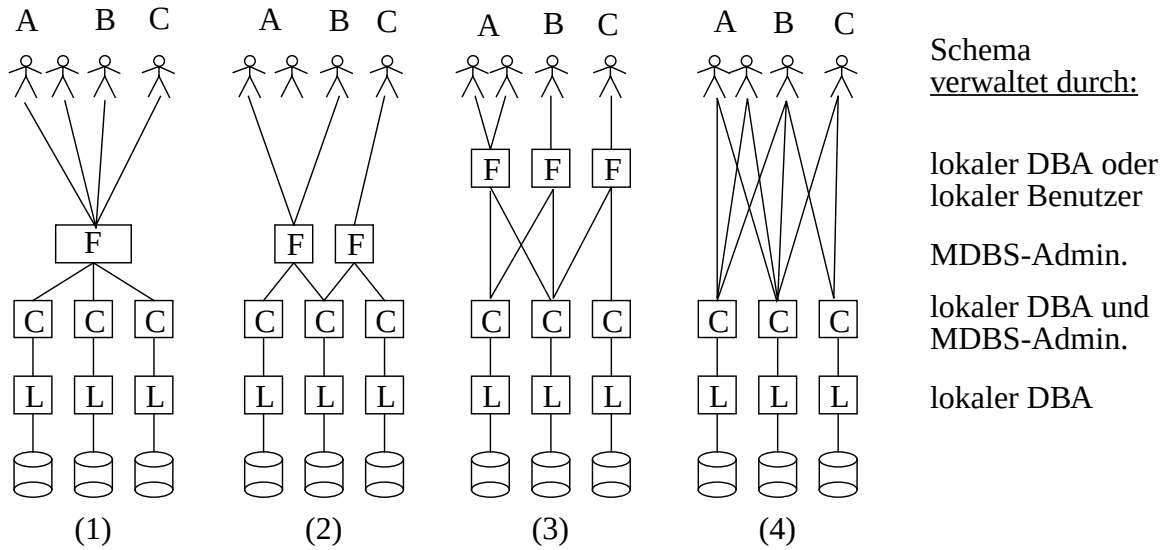


Abbildung 4.1: Multidatenbanksystem-Architekturen

In der Verwendung der Terminologie folgen wir Sheth und Larson [SL90]. Demnach können Multidatenbanksysteme in föderierte Datenbanksysteme (FDBS) und nicht-föderierte Systeme unterschieden werden.

Nicht-föderierte Systeme verhalten sich wie verteilte Datenbanksysteme, verbunden mit einer völligen Aufgabe der lokalen Autonomie der Komponentendatenbanksysteme. Somit ist auch keine Unterscheidung zwischen lokalen und globalen Benutzern mehr möglich. Aus diesen Gründen werden sie auch nicht weiter betrachtet.

Föderierte Datenbanksysteme stellen einen Kompromiß zwischen totaler Integration mit vollständiger Aufgabe der Autonomie der Teilsysteme und dem Verzicht auf Integration unter Beibehaltung der lokalen Autonomie dar, indem sowohl globale Benutzer unterstützt werden als auch lokale Benutzer auf gewohnte Weise ihre Anfragen stellen können. D.h. sowohl lokale als auch globale Operationen können im FDBS ausgeführt werden. Globale Operationen erlauben den Datenzugriff mit Hilfe des FDBS auf einer Anzahl von Komponentendatenbanken. Lokale Operationen werden direkt über das lokale Interface auf dem Komponenten-DBS ausgeführt.

Föderierte Datenbanksysteme können eingeteilt werden in lose gekoppelte oder eng gekoppelte entsprechend dem Grad der Integration und der Verteilung der Verantwortlichkeiten für die Wartung der Schemata.

Ein *eng gekoppeltes* FDBS ist dadurch gekennzeichnet, daß die Föderation (bzw. deren Administrator) für die Pflege der Föderation und die Kontrolle des Zugriffs zu den Komponenten-DBS verantwortlich ist. Bei der Entwicklung eines föderierten Datenbanksystems werden ein oder mehrere föderierte Schemata erzeugt, auf denen den globalen Benutzern jeweils eine Menge von Operationen angeboten werden.

Ein eng gekoppeltes FDBS kann ein oder mehrere föderierte Schemata aufweisen. FDBS mit einem *einzigen föderierten Schema* entsprechen nach [BHP92] dem globalen Schema-Ansatz, der Ähnlichkeit zu verteilten Datenbanksystemen aufweist, indem alle lokalen Datenbanken integriert sind. Nachdem das integrierte Schema einmal definiert ist, ist der Zugriff für die Benutzer sehr einfach, die mit den Daten arbeiten können, als ob sie aus einer einzigen Datenbank stammen. Strukturelle und semantische Konflikte müssen einmal bei der Erzeugung des globalen Schemas aufgelöst werden. Eng gekoppelte Systeme mit *mehrfachen Föderationen* erlauben die Erzeugung mehrerer föderierter Schemata aus jeweils unterschiedlichen lokalen Schemata bzw. Teilen davon. Die Benutzer greifen auf die Komponenten-Datenbanken über eines der föderierten Schemata zu. Der Nachteil dieses Ansatzes besteht jedoch darin, daß sich Constraints über mehrere Datenbanken nur sehr schwer kontrollieren lassen.

Ein FDBS ist *lose gekoppelt*, wenn der Benutzer selbst für die Erzeugung und Pflege der Föderation verantwortlich ist, und keine Kontrolle durch das föderierte System ausgeübt wird. Interdatabase Dependencies werden somit auch nicht unterstützt. Diese Architektur wurde bereits von Heimbigner und McLeod in [HM85] als föderiertes Datenbanksystem beschrieben, wir verwenden diesen Begriff hier aber in einem weiteren Sinne (vgl. Seite 53). Bright, Hurson und Pakzad [BHP92] fassen lose gekoppelte Architekturen, die auf ein integriertes Schema verzichten, unter dem Begriff *Multidatabase Language*-Ansatz zusammen. Er ist dadurch gekennzeichnet, daß der Zugriff auf die Komponentendatenbanken über eine Multidatenbanksprache erfolgt, besondere Beiträge hierfür wurden durch Litwin geleistet, z.B. im Rahmen des MRDSM-Projektes und später durch Entwicklung einer Multidatenbanksprache für objektorientierte Komponentendatenbanksysteme [Lit92].

4.2 Architektur von Multidatenbanksystemen

4.2.1 Begriffe

Sheth und Larson führen in [SL90] eine 5-Ebenen-Referenzarchitektur für Multidatenbanksysteme ein, die zugleich die Grundlage für die nachfolgende Darstellung bildet.

Ein *lokales Schema* ist das konzeptuelle Schema eines Komponenten-DBS. Weil die Komponenten-DBS verschiedene Datenmodelle nutzen können, können die lokalen Schemata in unterschiedlichen Datenmodellen dargestellt werden.

Für jedes lokale Schema gibt es ein korrespondierendes *Komponentenschema*. Das Komponentenschema repräsentiert die gleiche Information wie das lokale Schema, ist aber stattdessen im kanonischen Datenmodell (CDM) ausgedrückt. Anfragen auf dem Komponentenschema müssen somit in Anfragen des zugrundeliegenden lokalen Schemas übersetzt werden.

Für jedes Komponentenschema können ein oder mehrere *Exportschemata* definiert werden. Ein Exportschema stellt eine Teilmenge des Komponentenschemas dar und definiert, welcher

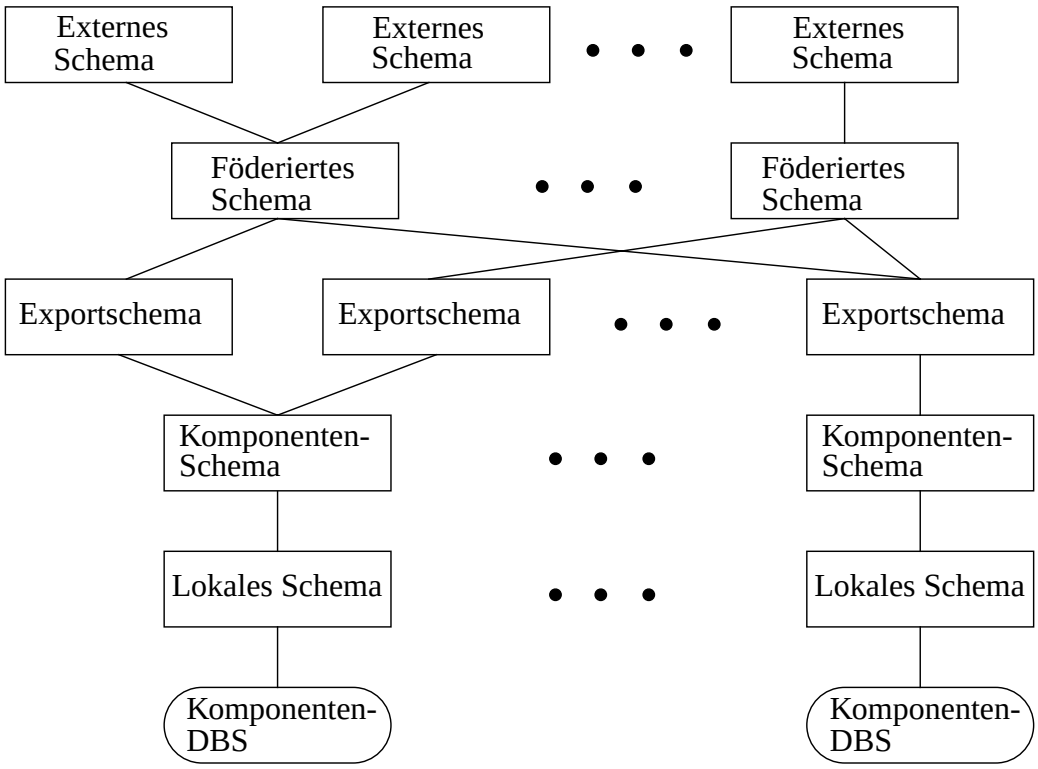


Abbildung 4.2: Fünf-Ebenen-Architektur eines Multidatenbanksystems [SL90]

Teil des Komponentenschemas für eine bestimmte Benutzergruppe verfügbar gemacht werden soll.

Ein *föderiertes Schema* ist eine Integration von mehreren Exportschemata. Es erlaubt den Zugriff auf Daten aus mehreren Datenbanken in der Weise, als ob diese in einer einzigen Datenbank gehalten würden. Eine Query auf einem föderierten Schema wird in Queries der zugrundeliegenden Exportschemata übersetzt, deren Resultate zu einer Ergebnismenge der global gestellten Anfrage verarbeitet werden. Alle föderierten Schemata werden im CDM ausgedrückt.

Für jedes föderierte Schema können ein oder mehrere *externe Schemata* definiert werden. Ein externes Schema stellt eine Teilmenge des föderierten Schemas dar, das in der Weise transformiert werden kann, um die Anforderungen unterschiedlicher Klassen von globalen Benutzern zu erfüllen. Die Rolle des externen Schemas entspricht der des externen Schemas in der 3-Ebenen-ANSI/SPARC-Schemaarchitektur [ANS75]. Ein externes Schema kann auch in einem anderen Datenmodell als dem kanonischen beschrieben sein.

Mit Hilfe dieses Referenzmodells läßt sich die Architektur der in Abschnitt 4.1 skizzierten Klassen von Multidatenbanksystemen ausdrücken (siehe Abbildung 4.2). Der Einfachheit halber verzichten wir auf die Darstellung der Export- und externen Schemata, da sie nur Teilmengen der jeweils zugrundeliegenden Schemata repräsentieren.

4.2.2 Vergleich und Bewertung

Globale Constraints lassen sich in eng gekoppelten Multidatenbanken mit einer Föderation sehr leicht ausdrücken, indem diese mit den Mitteln des kanonischen Datenmodells entweder explizit oder implizit formuliert werden. Allerdings lassen sie sich auf den lokalen Datenbanken nur erzwingen, wenn diese ihre Autonomie weitgehend aufgeben. Der globale Schema-Ansatz klingt zwar attraktiv, ist in der Praxis aber eher unrealistisch und eigentlich nur für Sy-

steme kleiner Größe denkbar, die eine enge Kopplung bevorzugen. In Architekturen, die mehrere föderierte Schemata unterstützen, ist die Kontrolle von datenbankübergreifenden Constraints dadurch erschwert, daß es kein zentrales Schema zu deren Beschreibung gibt, auch wenn es eine globale Administration gibt. In lose gekoppelten Systemen ist es ebenfalls unmöglich, globale Constraints zu garantieren, zumal nun auch die Kontrolle für die Erstellung der föderierten Schemata bei lokalen Benutzern und DB-Administratoren liegt. Litwin diskutiert das Problem in lose gekoppelten Systemen mit Multidatenbank-Sprache, daß aufgrund des Fehlens eines globalen Schemas die Sprache Ausdrucksmittel anbieten muß, um die Spezifikation und Kontrolle von globalen Abhängigkeiten zwischen lokalen Datenbanken, die nicht über ein globales Schema integriert sind, zu unterstützen. Ein Beispiel dafür, wie solche Änderungsabhängigkeiten (*Update Dependencies*) in einem Verbund relationaler Komponenten-DBS ausgedrückt werden, ist in [LMR90] beschrieben unter Verwendung eines logikbasierten nicht-prozeduralen Formalismus.

Extensional abgeschwächte Konsistenzwahrung, so wie sie in Abschnitt 2.5.2 auf Seite 30 eingeführt wurde, läßt sich auch in Multidatenbanksystemen unterschiedlicher Architekturen realisieren. Eine Möglichkeit dazu bietet die Definition von Exportschemata.

Tabelle 4.1 faßt die Bewertung der verschiedenen Architektur-Ansätze durch Hervorhebung ihrer Stärken und Schwächen noch einmal zusammen. Kein Ansatz allein kann gleichzeitig alle Wünsche berücksichtigen. Stattdessen ist die Wahl einer Architektur vor dem Hintergrund der funktionalen Anforderungen an das Gesamtsystem, der personellen und technischen Ressourcen und möglichen Anpassungen / Modifikationen der lokalen Systeme abhängig.

Eigenschaft	Eng gekoppeltes FDBS		Lose gekoppeltes FDBS	
	eine Föderation	mehrere Föderationen		mit Multi-DB-Sprache
Einfacher Zugriff über integriertes Schema	+	+	+	-
Konsistenzwahrung	+	o	o	-
Eignung für große Systeme	-	+	+	+
Konfliktauflösung bei Integration	+	+	+	-
Aufwand für globale Administration	-	-	+	+
Flexibilität	-	o	o	+

\+ gut unterstützt / o teilweise unterstützt / - nicht unterstützt

Tabelle 4.1: Vergleich von Multidatenbanksystem-Architekturen

4.3 Der Autonomiebegriff in Multidatenbanksystemen

Wie sich bereits angedeutet hat, ist das Problem der Autonomie zu einer wichtigen Fragestellung in verteilten und heterogenen Systemen geworden, wobei immer wieder das richtige Maß zwischen einer Einschränkung der lokalen Autonomie und dem notwendigen Grad an Kooperation gesucht wird.

4.3.1 Autonomie in homogenen verteilten Systemen

Die Frage nach dem Wesen der Autonomie gewann erstmals Interesse, beginnend mit der Entwicklung verteilter Systeme, Mitte der 70er Jahre. Ein früher Ansatz wurde 1978 von Enslow beschrieben [Ens78]. Dort ist eine Definition eines verteilten Systems gegeben, die auf der Einteilung in drei Dimensionen beruht. Diese drei Dimensionen beschreiben die Dezentralisierung in der Hardware-Organisation, in der Kontrollpunktorganisation und in der Datenbankorganisation. Die Dezentralisierung der Kontrolle kennzeichnet die Dimension der Autonomie in verteilten Systemen, wie sie in späteren Publikationen dargestellt wurde, z.B. [ÖV91]. Enslow gibt einen Überblick über die Dezentralisierung der Kontrolle, sortiert nach deren Grad:

- Einziger fester Kontrollpunkt
- Feste Master-Slave-Beziehung (Client-Server)
- Dynamische Master-Slave-Beziehung (durch Software modifizierbar)
- Mehrere autonom operierende Kontrollpunkte (möglicherweise repliziert)
- Mehrere Kontrollpunkte, die bei Tasks kooperieren, die in Subtasks zerlegt wurden
- Replizierte, identische Kontrollpunkte, die bei der Ausführung von Tasks kooperieren
- Mehrere Kontrollpunkte, die bei der Ausführung von Tasks voll kooperieren

Enslow behauptete bereits, daß verteilte Systeme für ein hohes Maß an Autonomie der Operationen aller Komponenten und Ressourcen (logisch und physisch) sorgen müssen. Dies kann bewirkt werden durch gemeinsame Protokolle auf logischer als auch auf physischer Ebene, um ein gewisses Maß an Kooperation zwischen Sender und Empfänger von Nachrichten bei gemeinsamen Aktionen zu gewährleisten. Enslow betonte, daß jede Ressource in der Lage sein muß, eine Anforderung (Request) eines Service abzuweisen, sogar nach Annahme der Message, weil es keine Kontrollhierarchie innerhalb des verteilten Systems gibt. Dieser Operationsmodus wird von ihm *Cooperative Autonomy* genannt. Ein hohes Maß an Autonomie in einem verteilten System ist durch mehrere voll kooperierende Kontrollpunkte charakterisiert. Die Autonomie eines Teilnehmers in einem verteilten System ist aber nicht allein durch eine Dezentralisierung der Kontrolle charakterisiert, sondern auch durch Abhängigkeiten, die durch die Datenverteilung zwischen ihnen verursacht werden.

Lindsay und Selinger untersuchen lokale Autonomie (*Site Autonomy*) als ein grundlegendes Entwurfsprinzip in verteilten Datenbanken im Rahmen des R*-Projektes [LS80].

Garcia-Molina und Kogan [GK88] geben eine ausführliche Klassifikation von Knotenautonomie (*Node Autonomy*) in verteilten homogenen Systemen. Dabei wird die Knotenautonomie positiv bewertet als eine natürliche Erweiterung der Autonomie der Organisationen und Abteilungen, die die Computersysteme nutzen. Verschiedene Aspekte von Autonomie werden hierbei diskutiert.

Darunter befindet sich die Namensautonomie, die weiter verfeinert wird: Objekte in einem verteilten System haben Namen, die zur Laufzeit in Adressen übersetzt werden. Die Namenserzeugungsautonomie (*Name Creation Autonomy)* ist dadurch bestimmt, ob ein Knoten die Zustimmung eines anderen Knotens beim Erzeugen eines Namens benötigt. Dabei lassen sich zwei Teilaspekte unterscheiden: die Auswahlautonomie (*Name Selection Autonomy*) und die Registrierungsautonomie (*Name Registration Autonomy*). Die Auswahlautonomie ist charakterisiert durch die Freiheit eines Knotens, eine beliebige Zeichenkette als Namen auszuwählen. Sie wird beschränkt durch syntaktische Konventionen (z.B. alle Dateinamen sollen mit dem Identifikator der lokalen Maschine beginnen). Die Registrierungsautonomie beschreibt, ob ein lokaler Knoten einen Name Server konsultieren muß, um die Gültigkeit des Namens zu überprüfen oder auf Duplikate zu testen. Eine hohe Registrierungsautonomie bedeutet, daß ein

Knoten einen Namen lokal validieren und registrieren kann. Übersetzungsautonomie (*Translation Autonomy)* ist die Fähigkeit eines Knotens, einen Namen in die korrespondierende physikalische Adresse unabhängig von anderen Knoten umzurechnen. Die Arten der Namensautonomie haben wechselseitigen Einfuß aufeinander. Ein Beispiel: Wir könnten uns ein System vorstellen, wo Namen beliebig ausgewählt und lokal registriert werden (mit möglichen Duplikaten). Das Ergebnis wäre dann eine hohe Auswahl- und Registrierungsautonomie, aber eine niedrige Übersetzungsautonomie, weil zur Übersetzung eines Namens erst alle möglichen Knoten konsultiert werden müssen.

Garcia-Molina und Kogan führen den Begriff der *Foreign Request Autonomy* ein, um den Freiheitsgrad eines Knotens zu beschreiben, externe Anfragen zu beantworten und deren Priorität festzulegen. Entfernte Anfragen ermöglichen das Sharing von Ressourcen, wie Daten, CPU-Zeit, Speicher usw., um die Kooperation in einem verteilten System zu verwirklichen. So gibt es Lastbalancierungsalgorithmen, die ein niedriges Maß an Autonomie voraussetzen mit dem Ziel, das Gesamtsystem zu optimieren. Ein Knoten hat ausreichend lokale Kontrolle, um autonom zu entscheiden, welche lokalen Daten mit dem Gesamtsystem geteilt werden sollen. Der Eigentümer der Daten kann auch frei entscheiden, ob er eine entfernte Anfrage eines einzelnen Benutzers ablehnen will. Zum Beispiel möchte ein Knoten seine Daten nicht mit anderen Knoten teilen, wenn dies dazu führt, daß der Client über jedes Update der Daten informiert werden muß, was eine beträchtliche Einschränkung der Autonomie der Instanz darstellt, die den Dienst erbringt.

Garcia-Molina und Kogan betrachten auch Knoten mit Transaktionsschnittstelle. So diskutieren sie vier Aspekte der Transaktionskontrollautonomie: Die *Transaction Type Autonomy* charakterisiert die Fähigkeit einzelner Knoten, Transaktionen eines beliebigen Typs zu starten. Die *Execution Autonomy* berechtigt einen Knoten, Transaktionen zu jeder Zeit auszuführen und zu beenden ohne Synchronisation mit Transaktionen, die gerade sonst noch im System laufen. Abbruchautonomie (*Abort Autonomy*) ist die Fähigkeit eines Knotens, einseitig eine verteilte (d.h. globale) Transaktion abzubrechen, vielleicht sogar noch nach dem Votum für ein Commit. Ein Beispiel dafür findet sich in [Dav84], wo die Abbruchautonomie für autonome Knoten benötigt wird, die an einem optimistischen Protokoll für partitionierte Netzwerke teilnehmen. Abbruchautonomie erfordert den Start kompensierender Aktionen, weil die Ergebnisse der abgebrochenen Transaktion mittlerweile sichtbar geworden sein könnten. Der vierte Typ der Transaktionsautonomie nach [GK88] heißt Sperrenautonomie (*Lock Autonomy*) mit der Bedeutung, daß einzelne Knoten das Recht haben, Sperren auf lokalen Daten freizugeben, die durch nicht-lokale Transaktionen gesetzt wurden. Dies kann nützlich sein im Fall von Kommunikationsfehlern, die dazu führen, daß Remote Locks für unbestimmte Zeit gehalten werden. Anders als die Abbruchautonomie bewirkt die Sperrenautonomie mehr Flexibilität, indem sie dem Knoten gestattet, selektiv Sperren freizugeben, ohne gleich die Transaktion vollständig abzubrechen, was aber zu Lasten der globalen Konsistenz gehen kann.

4.3.2 Autonomie in Multidatenbanksystemen

In [HM85] beschreiben Heimbigner und McLeod eine generische föderierte Datenbankarchitektur und gehen auch auf den Aspekt der Autonomie von Komponentensystemen ein. Sie diskutieren den Konflikt zwischen der notwendigen Autonomie der lokalen Knoten in einem föderierten Datenbanksystem und ihrer Fähigkeit, ein bestimmtes Maß an Informationen zu teilen. Sie unterscheiden vier Anforderungen, die Autonomie charakterisieren:

1. Eine Komponente darf nicht gezwungen werden, eine Aktivität für eine andere Komponente auszuführen. Anstelle einer zentralen Gewalt müssen entsprechende Protokolle kooperative Aktivitäten der Komponenten unterstützen.
2. Jede Komponente bestimmt die Daten, die mit den anderen Komponenten geteilt werden, d.h. jede Komponente muß in der Lage sein, die Information zu spezifizieren, die verfügbar gemacht wird und welche anderen Komponenten auf sie wie zugreifen können.
3. Jede Komponente bestimmt, wie sie Daten sehen und kombinieren kann. In einer lose gekoppelten Föderation von Datenbanksystemen muß jede Komponente ihr eigenes "globales" Schema bauen, das ihren Anforderungen am besten genügt.
4. Eine Komponente muß die "Freiheit der Assoziation" mit Bezug zur Föderation haben, d.h. sie muß dynamisch die Föderation verlassen oder ihr beitreten können, nachfolgend als *Assoziationsautonomie* bezeichnet.

Gligor und Popescu-Zeletin [GP86] listen die Anforderungen an autonome Systeme wie folgt auf:

1. Die lokalen Operationen der Komponenten-DBS werden nicht beeinflußt durch ihre Teilnahme an einem Multidatenbanksystem.
2. Die Art und Weise, in der einzelne DBMS Queries verarbeiten und optimieren, sollte nicht durch die Ausführung globaler Queries beeinflußt werden.
3. Die Systemkonsistenz sollte nicht beeinträchtigt werden, wenn einzelne DBMS der Föderation beitreten oder sie verlassen.

Veijalainen und Popescu-Zeletin diskutieren eine Einteilung von Autonomie in drei Kategorien: Entwurf, Kommunikation und Ausführung. Diese Kategorien sind mittlerweile in der Multidatenbank-Literatur recht etabliert, auch wenn sie nur eine grobe Charakterisierung geben.

Entwurfsautonomie (Design Autonomy) kennzeichnet die Freiheit, über den Entwurf eines Komponenten-DBMS zu entscheiden. Dazu zählen:

- Diskursbereich
- Repräsentation (Datenmodell, Anfragesprache) und Benennung der Datenelemente $\rightarrow$ führt zu syntaktischer Heterogenität
- Semantische Interpretation der Daten $\rightarrow$ trägt zur semantischen Heterogenität bei
- Konsistenzbedingungen: semantische Integritätsbedingungen, Serialisierbarkeitskriterien
- Funktionalität des Systems (d.h. welche Operationen sollen unterstützt werden)
- Systemimplementierung: Record- und Filestrukturen, Concurrency-Control-Algorithmen u.a.

Die Entwurfsautonomie ist also die primäre Ursache für Heterogenität in einem Multidatenbanksystem.

Kommunikationsautonomie bezieht sich auf die Entscheidungsfähigkeit eines Komponenten-DBMS, ob es mit anderen Systemen kommunizieren möchte. Dazu zählt die Entscheidung, wann und wie auf eine Anforderung eines anderen DBMS geantwortet werden soll.

Ausführungsautonomie (Execution Autonomy) kennzeichnet die Fähigkeit eines Komponenten-DBMS, lokale Operationen auszuführen ohne Interferenz globaler Operationen. Die Ausführungsautonomie ist dafür verantwortlich, daß keine bestimmte Ausführungsreihenfolge auf dem lokalen DBMS erzwungen werden kann. Dies impliziert zugleich, daß das lokale System

darüber entscheiden kann, ob es extern aufgerufene Operationen ausführen will oder irgendeine Operation abbricht. Weiterhin brauchen die Komponentensysteme nicht die Föderation über die Ausführungsreihenfolge von globalen als auch lokalen Operationen zu informieren.

Der von Litwin u.a. beschriebene Multidatenbank-Ansatz [LA86, LA87] geht aus von einem Zugriff auf mehrere autonome Datenbanken ohne ein gemeinsam genutztes globales Schema (lose Kopplung). Das Fehlen eines globalen Schemas verursacht Redundanzen und Unstimmigkeiten zwischen Daten verschiedener Datenbanken, bei Namen, Datenstrukturen und Wertetypen. Die Autoren diskutieren vier Teilaspekte der Entwurfsautonomie, die die Definition eines gemeinsamen föderierten Schemas erschweren, auch wenn die Autonomie von den lokalen Benutzern gewünscht ist. Dazu zählen die *Datendefinitionsautonomie*, die *Duplikationsautonomie*, die *Restrukturierungsautonomie* (auf logischer und physischer Ebene) sowie die *Wertetypautonomie*.

Eine wichtige Untersuchung über die Auswirkungen der Autonomie in heterogenen Datenbanksystemen findet man bei Du u.a. [DELO89]. In [DEK90] wird lokale Autonomie als die Fähigkeit jedes lokalen DBS (LDBS) bezeichnet, den Zugriff auf seine Daten durch andere LDBS zu kontrollieren sowie seine eigenen Daten zu lesen und zu verändern unabhängig von anderen LDBS. Es werden zwei Arten lokaler Autonomie unterschieden: Operationsautonomie und Serviceautonomie. Die *Operationsautonomie* definiert die Fähigkeit jedes LDBS, verschiedene Arten von Operationen auszuführen und verschiedene Arten von Kontrolle über die eigene Datenbank auszuüben. Das ist die Voraussetzung dafür, daß lokale Applikationen ungeachtet ihrer Integration weiter laufen können. Die Autoren unterscheiden zwei Kategorien von Operationen auf einem LDBS: Transaktionsmanagement und Datenbankadministration.

Die Operationsautonomie beim Transaktionsmanagement definiert das Recht des lokalen Systems, die Ausführung lokaler Transaktionen zu kontrollieren, um die korrekte Ausführung zu garantieren und lokale Datenbankkonsistenz zu wahren. Lokale Concurrency Controller (LCC) können frei jede Transaktion, die auf dem lokalen Knoten läuft, durch Commit beenden oder mit Restart wiederholen ohne Kontrolle durch den Globalen Concurrency Controller (GCC). Deshalb kann kein externes System eine bestimmte Ausführungsreihenfolge der Kommandos auf den LDBS erzwingen, die auch selbst über Aborts von global abgesandten Kommandos entscheiden können. Du u.a. betrachten auch die Fähigkeit lokaler Benutzer, auf die lokalen Daten zuzugreifen. Nach der Integration verliert das LDBS teilweise die Kontrolle über den Zugriff auf seine eigenen Daten, weil z.B. einige der Daten nach der Integration auf anderen Knoten repliziert sein können. Die Frage bleibt offen, welcher Verlust an Zugriffsmöglichkeiten tolerierbar ist.

Die *Serviceautonomie* nach [DEK90] definiert das Recht jedes LDBS, über die Dienste und den Typ von Informationen zu bestimmen, die dem globalen System bereitgestellt werden. Diese Informationen können in Benutzerdaten und Kontrollinformationen unterschieden werden, z.B. Informationen über lokale Ausführungen für den GCC. Ein wichtiges Beispiel für den Gebrauch lokaler Kontrollinformation ist verteilte Anfrageoptimierung. Die Optimierung beinhaltet nicht nur die referenzierten Daten, sondern auch Data Dictionaries and statistische Informationen (z.B. letzte Zugriffe, Selektivität) des lokalen Knotens.

Operations- und Serviceautonomie sind zusammen Aspekte von lokaler Autonomie. Die Art des Dienstes, den ein LDBS bereitstellt, kann davon abhängen, inwieweit Kontrolle lokal aufgegeben wird. Zum Beispiel vermindert die Replizierung lokaler Daten zugleich die Kontrolle über sie. Den Konflikt zwischen Replikationstransparenz und lokaler Autonomie untersuchen Abbott und McCarthy in [AM88] anhand einiger Parameter, wie z.B. Ressourcenkontrolle und Zugriffskontrolle.

Du u.a. schlagen vor, lokale Autonomie nach statischen und dynamischen Aspekten zu bewerten entsprechend den statischen und dynamischen Eigenschaften eines Multidatenbanksystems. Die statischen Aspekte von Autonomie sind bereits vor der Integration eines LDBS bekannt und von daher unabhängig von speziellen Applikationen. Sie entsprechen der auf Seite 59 skizzierten Entwurfsautonomie, die zur Heterogenität der Komponentensysteme in der Föderation führt. Die dynamischen Aspekte betreffen Entscheidungen zur Laufzeit von Multidatenbank-Applikationen und umfassen somit Ausführungs- und Kommunikationsautonomie (vgl. Seite 59). Im allgemeinen ist es leichter, die Effekte statischer Autonomie zu verstehen, die statisch analysiert werden kann. Es ist aber sehr schwer, diese einzuschränken, wenn z.B. das Concurrency-Protokoll oder das Datenmodell betroffen sind, die fest vom zugrundeliegenden DBMS abhängen. Demgegenüber sind die Auswirkungen dynamischer Autonomie schwerer zu verstehen, weil sie von den Ausführungen der Applikationen abhängen (beispielsweise die Erkennung eines indirekten Konfliktes zwischen zwei globalen Transaktionen).

Chrysanthis und Ramamritham untersuchen in [CR93] die Semantik der Ausführungsautonomie aus der Sicht von Transaktion und Transaktionsmanager und leiten daraus Folgerungen für den Entwurf von Multidatenbankprotokollen ab. Die Autoren verstehen unter Autonomie die Fähigkeit von Transaktionen und Datenbanksystemen, Events ohne Beschränkungen auszuführen. Grundsätzlich wird von zwei Eventtypen in einem Datenbanksystem, nämlich den signifikanten Events einer Transaktion (Begin, Commit, Abort) und Events, die Operationen auf Objekten entsprechen, ausgegangen. Dabei wird Autonomie in zwei Dimensionen analysiert: Datenzugriffsautonomie und Transaktionsmanagement-Autonomie. *Datenzugriffsautonomie* erfaßt alle Aspekte von Objekt-Events, die durch Transaktionen ausgelöst werden. *Transaktionsmanagement-Autonomie* erfaßt Aufrufe von signifikanten Events, die zur Transaktion gehören, die unter Kontrolle des DBMS ausgeführt wird. Nach [CR93] kann die Verletzung der Autonomie durch das Vorschreiben eines bestimmten Events (*Violation through Prescription*) oder durch das Verbieten bestimmter Events (*Violation through Proscription*) erfolgen. Basierend auf diesen Begriffen, geben die Autoren eine verfeinerte Definition von Ausführungsautonomie:

Ein Knoten *n* hat Transaktionsmanagement-Autonomie (TM-Autonomie) gegenüber Transaktion t_i, wenn er weder gezwungen noch gehindert wird, ein zu t_i gehörendes signifikantes Event auszuführen.Ein Knoten *n* hat Transaktionsmanagement-Autonomie, wenn er sich gegenüber allen Transaktionen TM-autonom verhält.
Eine Transaktion *t* hat Datenzugriffsautonomie gegenüber Knoten *n*, wenn sie weder gezwungen noch gehindert wird, ein Objekt-Event (d.h. einen Datenzugriff) auszuführen. Dementsprechend hat eine Transaktion *t* Datenzugriffsautonomie, wenn sie gegenüber allen Knoten datenzugriffsautonom ist.
Zusammenfassend wird die Ausführungsautonomie in einem Multidatenbanksystem dadurch definiert, daß alle Transaktionen Datenzugriffsautonomie und alle Knoten Transaktionsmanagement-Autonomie haben.

Kemper u.a. führen den Begriff *autonomes Objekt* in einem verteilten Objektsystem ein [KLM+91]. Ein autonomes Objekt hat Kontrolle über seine Kooperation mit der Außenwelt, seinen Speicherort auf einem Knoten und seinen Lebenszyklus, d.h. Zustand und Verhalten entsprechend dem Fortgang seiner Entwicklung.

4.3.3 Verallgemeinerung des Begriffs Autonomie in interoperablen Systemen

Bei der Definition der Autonomie stellt sich die Frage nach dem Wesen von Autonomie, das sehr ausführlich von Veijalainen untersucht wurde, der mit seiner Arbeit [Vei90] Grundlagen für ein besseres theoretisches Verständnis von Autonomie gelegt hat. Diese Erkenntnisse wurden gesammelt bei der Entwicklung des Semantischen Transaktionsmodells (*S-Transaction Model*) für das SWIFT-Projekt zur Integration der Informationssysteme einiger europäischer Banken.

Um das Phänomen der Autonomie zu verstehen. müssen wir so unterschiedliche Dinge wie Organisationen, Menschen, Software, Datenbanken usw. betrachten, nachfolgend Entitäten genannt. Eine Entität hat eine bestimmte Struktur, die zugleich ihr Verhalten bestimmt. Reale Entitäten sind in der Lage, Aktionen auszuführen, ihr mögliches Verhalten entspricht einer endlichen Anzahl von endlichen Aktionsfolgen. Einige der Aktionen sind sichtbar für die Außenwelt, andere sind intern. Die sichtbaren Aktionen bilden die Grundlage für die Interaktion mit der Umgebung. Einige sichtbare Aktionen sind beschränkt, d.h. sie hängen von sichtbaren Aktionen anderer Entitäten ab und modellieren die Abhängigkeit der Entität von ihrer Umgebung. Formal kann nach [Vei90] eine Entität durch einen markierten, gerichteten Baum unendlicher Höhe modelliert werden. Die Kanten sind durch ein Paar (x, P_i) beschriftet, wobei x eine Aktion ist und P_i ($0 \leq P_i \leq 1$) die Wahrscheinlichkeit, mit der diese Aktion in dieser Situation ausgeführt wird. Jeder Pfad im Baum, der aus einer unendlichen Menge von Aktionen besteht, repräsentiert eine mögliche Zukunft der Entität.

Autonomie und externe Kontrolle sind miteinander verbundene, aber gegensätzliche Konzepte. Die Autonomie einer Entität kann als reale Möglichkeit gesehen werden, in einer bestimmten Situation unter verschiedenen Zukünften ein Verhalten zu wählen. Ihr Gegenteil ist Kontrollierbarkeit, d.h. die Unmöglichkeit für eine Entität, frei ein Verhalten in einer bestimmten Situation anzunehmen. Von daher bedeutet Autonomie einer Entität in gewissem Sinne die Nicht-Kontrollierbarkeit durch die Außenwelt. Das Maß an Selbstbestimmung einer Entität kann ausgedrückt werden durch die Wahrscheinlichkeit für ein Verhalten, das unabhängig von Aktionen der Umgebung ist.

Effektive Autonomie ist ein Verhältnis zwischen zwei Entitäten, bei dem die kontrollierende Entität *X* von der kontrollierten Entität *Y* ein bestimmtes Verhalten *B* in einer gegebenen Situation erwartet, wenn die Bedingung *C* erfüllt ist. Wenn *Y* das erwartete Verhalten zeigt, dann ist es extern kontrollierbar, ansonsten ist es effektiv autonom in dieser Situation gegenüber der kontrollierenden Entität. Die effektive Autonomie ist meßbar, indem man die Anzahl der Situationen bestimmt, in denen eine Entität ein autonomes Verhalten zeigt. Durch Angabe der Häufigkeit des autonomen Verhaltens können Fuzzy-Maße für jedes Paar von Entitäten definiert werden, um auszudrücken, wie oft das tatsächliche Verhalten einer Entität von der extern erwarteten Reaktion abweicht.

Absolute Autarkie bedeutet Abwesenheit von Interaktion mit der Umgebung. Eine Gleichsetzung von Autarkie und Autonomie ist nicht sinnvoll, weil Entitäten voneinander abhängen und während ihrer Existenz interagieren müssen. Jede Entität muß zumindest beobachtbar sein, was zwar einen Verlust an Selbstbestimmung bedeutet, aber keinen signifikanten Einfluß auf die Eigenschaften der Entität hat.

Veijalainen benutzt die Klassifikation aus [GP86] und wendet sie auf interagierende aktive und passive Entitäten an. In [Vei90] werden verschiedene Arten von Interaktion zwischen Entitäten untersucht. Nachfolgend werden Organisationen und Computersysteme betrachtet, die passiv

sind in dem Sinn, daß sie ihre Struktur nicht ändern können. Organisationen als aktiver Bestandteil bestehen aus Menschen, die den *Entwurf* (d.h. die Struktur) und den *Gebrauch* der Systeme kontrollieren, die zum Gesamtsystem gehören. Kontrolle der Struktur bedeutet die Auswahl oder Entwicklung von Hard- und Software und Applikationen des Computersystems. Die Kontrolle des Gebrauchs umfaßt Entscheidungen über die Rechte der Anwender und die Benutzungsbedingungen des Systems, das den Benutzern Dienste bereitstellt, z.B. die Funktionalität einer Datenbank. Die Organisationen der Menschen, alle Systeme und Telekommunikationseinrichtungen zusammen bilden die Gesamtorganisation.

Als Grundlage aller Arten von Autonomie wurde in [VEH92] die *O-Autonomie* (organisationelle Autonomie) eingeführt, weil Organisationen die Grundlage der Existenz und der Benutzung von Systemen sind. O-Autonomie charakterisiert die Kontrollierbarkeit einer Organisation durch eine andere Organisation durch Interaktionen wie Vereinbarungen, Konkurrenz und Kooperation. Eine Organisation *X* verhält sich effektiv O-autonom, wenn sie sich nicht so verhält, wie es eine andere Organisation *Y* während einer Interaktion von ihr erwartet. Abbildung 4.3 skizziert eine interoperable Umgebung aus autonomen Komponenten.

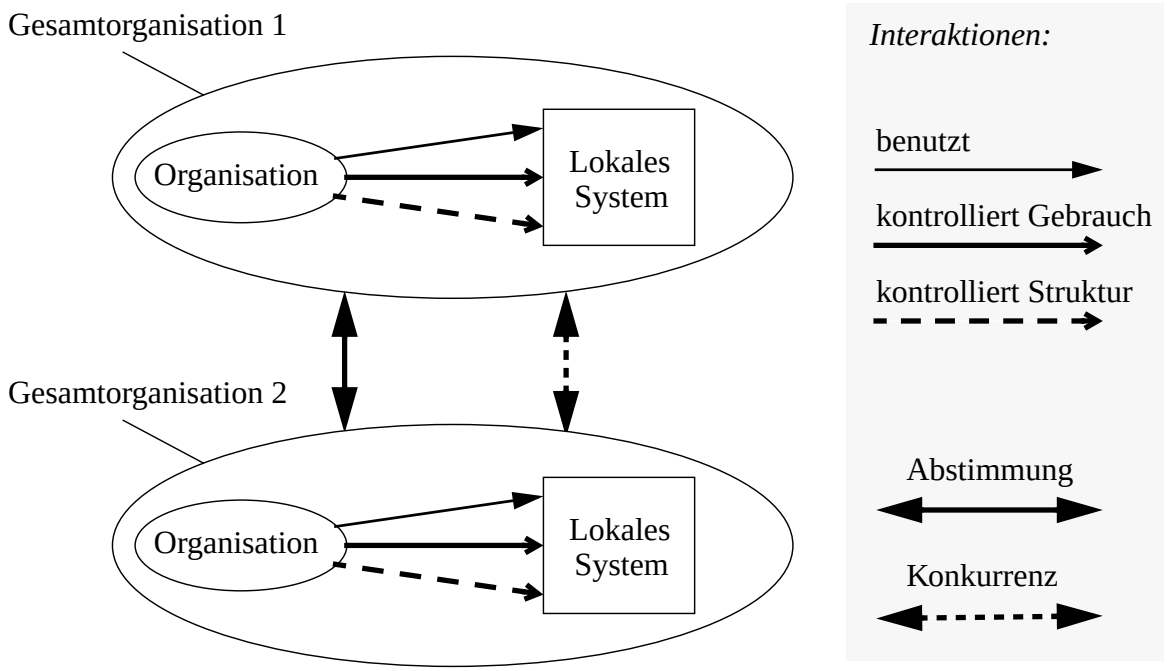


Abbildung 4.3: Interaktionen in einer autonomen Umgebung

In [Vei90, VEH92] wird die Entwurfsautonomie als *D-Autonomie* (Design-Autonomie) definiert. Eine Organisation ist demnach D-autonom gegenüber einem Computersystem, wenn sie die volle Kontrolle über die Struktur des Systems ausüben kann (z.B. Auswahl von Hard- und Software, Schemaentwurf). Eine Organisation kann zugleich D-autonom gegenüber einem Teil des Systems sein, nicht aber gegenüber einem anderen Teil. Eingeführt wird die *M-Autonomie* (Management-Autonomie), die eine Organisation besitzt, wenn sie den Gebrauch eines Systems vollständig kontrollieren kann. Dies setzt voraus, daß die Organisation Eigentümer oder Mieter des Systems ist. M-Autonomie kann auf mehreren Systemebenen betrachtet werden, z.B. am lokalen Datenbank-Interface: Welche Transaktionen darf ein bestimmter Benutzer ausführen? Wer darf welche Daten verändern?

Veijalainen nennt ein System *S* kommunikations-autonom (*C-Autonomie*) gegenüber einem anderen System *S'*, wenn dieses *S* nicht zwingen kann, die Kommunikation mit einem beliebigen System *S"* zu beginnen, zu beenden oder fortzusetzen.

Kooperative Interaktionen zwischen Systemen werden durch Protokolle unterstützt, definiert durch Typen von Protokollmessages und ihre Reihenfolge. Nach [Vei90] ist ein System ausführungs-autonom (*E-Autonomie*) für einen bestimmten Protokollmessage-Typ, wenn es entscheiden kann, ob es die ankommende Nachricht ausführen will oder nicht bzw. wie und wie schnell sie innerhalb des Systems verarbeitet werden soll.

Wenn eine Organisation für ihre Systementwicklung ihre eigenen Software Engineering- und Design-Tools frei auswählen kann, so wird dies in [VM92] als *T-Autonomie* (Tool-Autonomie) bezeichnet.

Eine interessante Fragestellung für das globale Transaktionsmanagement ist, ob lokale Applikationen bzw. deren Transaktionen durch lokale Benutzer ohne Einschränkung programmiert und durch das lokale Datenbank-Interface verarbeitet werden können. Eine Organisation ist *LT-autonom* (lokal transaktions-autonom), wenn sie lokale Applikationen modifizieren und neueentwickeln kann und diese als lokale Transaktionen starten kann. Die Fähigkeit, lokale Transaktionen an das DBMS ohne Kenntnis des Multidatenbanksystems abzusetzen, wird in [SKS91] auch als *Kontrollautonomie* bezeichnet, die aber auf unterschiedliche Weise wieder eingeschränkt werden kann. Aus der LT-Autonomie ergeben sich eine Reihe von Problemen für die globale Konsistenz, eine darauf basierende Klassifikation für Multidatenbanksysteme geben die Autoren in [VM92].

Zusammenfassung

Die Aufzählung unterschiedlicher Autonomiebegriffe führt uns zu der Frage, inwieweit all diese Definitionen zusammengefaßt werden können. Für eine Klassifikation der zitierten Definitionen stellen wir folgende Fragen:

- In welchem Kontext (auf welchem Abstraktionsniveau) ist die Autonomie definiert ?
- Handelt es sich um einen Entwurfs- oder laufzeitdynamischen Aspekt von Autonomie ?
- Für wen ist die Eigenschaft der Autonomie definiert ?
- Welche Freiheit wird durch die Autonomie charakterisiert ?

Tabelle 4.2 gibt einen Überblick über die zuvor behandelten Arten von Autonomie. Es wird in Analogie zu den vorhergehenden Abschnitten eine Unterscheidung zwischen homogenen verteilten Systemen und Multidatenbanksystemen beibehalten, eine verallgemeinerte Sicht davon stellen interoperable Systeme dar. Die Spalte Aspekt ordnet jeder Autonomie-Art einen Aspekt zu, um eine grobe Einordnung zu ermöglichen. Dabei handelt es sich entweder um **D** (Design-Aspekt), **X** (Ausführungs-Aspekt) oder **TX** (Transaktionsausführungs-Aspekt).

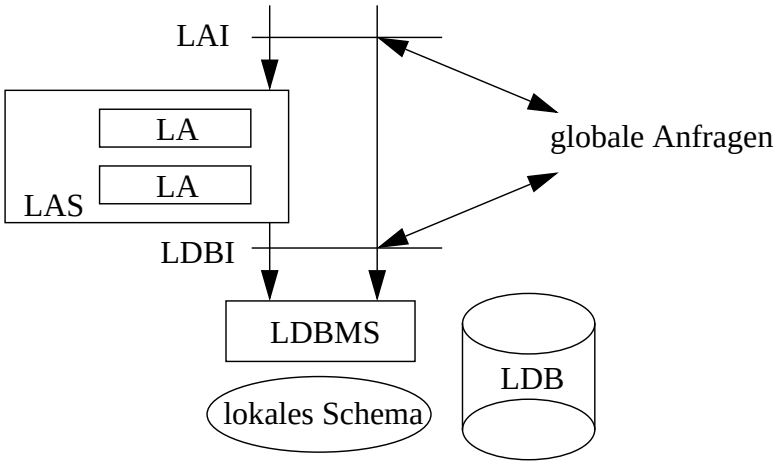
Bezeichung: ... Autonomy	Asp.	Eigenschaft von ...	beschreibt Freiheit bei ...	Autor
in homogenen verteilten Systemen				
Cooperative	X	Knoten	Kooperation (dezentralisierte Kontrolle)	[Ens78]
Site	X	Knoten	Kontrolle	[LS80]
Name Creation	X	Knoten	Benennung von Objekten (Namensauswahl und -registrierung)	[GK88]
Translation	X	Knoten	Umwandlung von Objektnamen in -adressen	[GK88]
Foreign Request	X	Knoten	Ausführung von Anfragen	[GK88]
Transaction Type	TX	Knoten	Start von Transaktionen	[GK88]
Execution	TX	Knoten	Ausführung von Transaktionen	[GK88]
Abort	TX	Knoten	Abbruch verteilter Transaktionen	[GK88]
Lock	TX	Knoten	Freigabe global gesetzter Sperren	[GK88]
in Multidatenbanksystemen				
Association	D	LDBS	Beitritt zur Föderation	[HM85]
Design	D	LDBS	Entwurf	[GP86]
Execution	X	LDBS	Ausführung lokaler Operationen	[GP86]
Communication	X	LDBS	Kommunikation	[GP86]
Data Definition	D	LDBS	Datendefinition	[LA86]
Duplication	D	LDBS	Replizierung von Daten	[LA86]
Restructuring	D	LDBS	Datenstruktur	[LA86]
Value Type	D	LDBS	Datentypen	[LA86]
Operation	X	LDBS	Ausführung von Operationen	[DEK90]
Service	D	LDBS	Bereitstellung von Diensten	[DEK90]
Control	TX	LDBS	Ausführung lokaler Transaktionen	[SKS91]
Data Access	X	Transaktion	Datenzugriff (Objekt-Event)	[CR93]
Transaction Management	TX	LDBS	Ausführung von Transaktions-Events	[CR93]
LT-	D, TX	Organisation	lokale Applikationen und lokale Transaktionen	[VM92]
in interoperablen Systemen				
effective	X	Entität	Verhalten	[Vei90]
O-	D	Organisation	Verhalten	[VEH92]
D-	D	Organisation, System	Kontrolle über die Struktur des Systems	[Vei90]
M-	D	Organisation	Kontrolle über den Gebrauch eines Systems	[VEH92]
C-	X	System	Kommunikation mit anderen Systemen	[Vei90]
E-	X	System	Ausführung externer Aufrufe	[Vei90]
T-	D	Organisation	Auswahl von Tools	[VM92]

Tabelle 4.2: Überblick über Arten lokaler Autonomie

4.4 Der Zusammenhang von Autonomie und globaler Konsistenz in Legacy-Systemen

Eine wesentliche Zielsetzung dieser Arbeit ist die Unterstützung globaler Konsistenzkontrolle in heterogenen Informationssystemen, deren Komponenten schon vor der Integration existiert haben. Diese wollen wir nachfolgend als *Legacy-Systeme* bezeichnen.

Definition 4.1 (Legacy-System)
Ein Legacy-System ist ein Informationssystem, das bereits vor der Integration in ein komplexeres (föderiertes) System bestanden hat und dessen Autonomie zum Zwecke der Integration nur in sehr geringem Maße eingeschränkt werden kann. Datenbankbasierte Legacy-Systeme bestehen aus einem lokalen Applikationssystem (LAS) und einem lokalen Datenbanksystem (LDBS), die Details sind in Abbildung 4.4 dargestellt.



LAS	Lokales Applikationssystem	LDBMS	Lokales Datenbankmanagementsystem
LA	Lokale Applikation	LDB	Lokale Datenbank
LAI	Lokales Applikationsinterface	LDBI	Lokales Datenbankinterface

Abbildung 4.4: Struktur eines lokalen Legacy-Systems

Diese Definition ist somit noch etwas weiter gefaßt als bei Brodie [Bro93], der weitere Kriterien für ein Legacy-System definiert (Alter, Größe, Plattform). Die Integration verschiedener Legacy-Systeme kann zu schwerwiegenden globalen Konsistenzproblemen führen, wie sie insbesondere in Abschnitt 2.4 diskutiert wurden. In diesem Abschnitt wollen wir nun die Anforderungen globaler Integritätssicherung den verschiedenen Graden an Autonomie der lokalen Legacy-Systeme gegenüberstellen.

Zur Wahrung globaler Konsistenz kann Autonomie auf unterschiedlichen Ebenen eines Legacy-Systems verletzt werden. Diese Ebenen sind:

- lokales DBMS
- lokale Datenbank (Schema, Daten)
- lokales Applikationssystem.

In verteilten heterogenen Umgebungen ist zusätzlich noch die Middleware zu betrachten, die über die Funktionalität des DBMS hinausreicht (z.B. Transaktions-Monitore, Name-Services in verteilten Systemen).

Autonomie wird dabei jeweils als Eigenschaft einer bestimmten Systemkomponente auf einer Ebene betrachtet. Der Begriff Komponente kann einen Applikationsmodul, ein Datenbankobjekt (z.B. Tabellen, Trigger, Benutzer, Zugriffsrechte, Daten) oder eine lokale DBMS-Komponente (z.B. Scheduler, Lock-Manager, Query-Prozessor, Transaktionsmanager) einschließen.

Autonomie umfaßt die Entscheidungsfreiheit der Komponente über:

- Struktur (Design-Aspekt)
- Verhalten (Ausführungs-Aspekt)
- Kommunikation (Design- und Ausführungs-Aspekt)

innerhalb des föderierten Systems.

4.4.1 Beschreibung von Autonomie-Verletzung

Strukturautonomie

Strukturautonomie umfaßt alle Aspekte, die vor der Integration eines Legacy-Systems in eine Föderation berücksichtigt werden. Alle Anpassungen von DBMS (Konfiguration), Applikationscode, Datenbankschema oder Daten berühren Strukturautonomie. Zum Zwecke der Integration müssen Komponenten modifiziert oder auch ergänzt werden. Eine Modifikation stellt die stärkste Beeinträchtigung der Autonomie dar, eine Ergänzung eine schwache.

1. Hinzufügen von Komponenten (*Add*)
Um heterogene Systeme zu integrieren, kann es erforderlich sein, das lokale DBS um zusätzliche Funktionen (auf Schemaebene) oder zusätzliche Daten anzureichern, um die Abbildung unterschiedlicher Wertebereiche zu ermöglichen.
Ebenfalls eine schwache Einschränkung ist die Ergänzung von Softwarekomponenten oberhalb des lokalen DBMS, das dabei unverändert bleibt.

2. Verändern von Komponenten (*Modify*)
Denkbar ist eine Veränderung von Daten als notwendige Anpassung an die Föderation oder die Veränderung des Schemas, was außerdem eine Veränderung von darauf operierenden Applikationen erfordert.

Verhaltensautonomie

Nach der Integration eines lokalen Systems hat die Verhaltensautonomie Einfluß auf die globale Konsistenz. Die Verhaltensautonomie entspricht der Fähigkeit des lokalen Systems, Events ohne Beschränkungen auszuführen. Von daher beschreibt die Verhaltensautonomie die Freiheit eines lokalen Systems, über seine Aktionen selbst zu entscheiden. Die Aktionen werden unterschieden nach Datenbankoperationen am lokalen Datenbankinterface (LDBI), lokalen Transaktionsoperationen (LT) oder sonstigen Operationen auf dem zugrundeliegenden lokalen System (LS). Die Verhaltensautonomie kann unterschiedlich eingeschränkt werden: durch Vorschreiben oder Verbieten bestimmter Aktionen oder durch Einführung von Restriktionen bei der Ausführung bestimmter Aktionen.

Ein hohes Maß an Verhaltensautonomie kann dazu führen, daß Konsistenzbedingungen über längere Zeiträume verletzt bleiben und Daten, die in globalen Abhängigkeiten stehen, sich unabhängig voneinander entwickeln. Bei einer asynchronen Verarbeitung lokaler Events müssen folglich zusätzliche Komponenten vorhanden sein, die das Wissen über die Kontrollabhängig-

keiten zwischen den lokalen Datenbanken zur Verfügung stellen. Dies ist zugleich die Grundlage für die Ausführung von Abgleichstrategien zwischen divergierenden Objekten.

1. Beschränkung von lokalen Aktionen (*Restrict*)
Wenn die Ausführung einer lokalen Aktion die Koordination mit anderen Teilnehmern erfordert, sprechen wir von einer Beschränkung. Ein Beispiel hierfür ist die Ausführung lokaler Transaktionen. Lokale Transaktionen müssen z.B. beschränkt werden, wenn sie das 2-Phase-Commit-Protokoll unterstützen sollen, um strenge Konsistenz zu realisieren. Dieser synchrone Ansatz führt zu einer hohen Datenqualität, aber einer verminderten Performance. Nur wenige Anwendungen, wie z.B. Bankanwendungen, benötigen ein solch hohes Maß an Konsistenz. Replizierte Datenbanksysteme, die Kohärenzbedingungen zwischen Replikaten kontrollieren (One Copy Serializability), können als ein Spezialfall angesehen werden, sie realisieren Strategien (z.B. Read Once Write All), um eine permanente Korrespondenz zwischen Kopien desselben Objektes zu gewährleisten.

Ein anderes Beispiel ist das Erzeugen eines neuen Datenbankobjektes in einer relationalen Datenbank, bei dem der Wert des Schlüssels nicht lokal festgelegt werden kann, sondern in der Entscheidung eines *Global Key Managers* liegt, der somit für eine globale Eindeutigkeit von Schlüsselwerten eines bestimmten Typs sorgt.

2. Verbieten lokaler Aktionen (*Proscribe*)
Lokale Updates können Probleme dadurch verursachen, daß sie globale Korrektheitskriterien (Serialisierbarkeit) verletzen. Somit können Transaktionen verboten werden, die nur über das lokale Datenbankinterface, ohne Kenntnis der Föderation, abgesetzt werden. Am stärksten wird die Autonomie eingeschränkt, wenn keine lokalen Transaktionen mehr ausgeführt werden, stattdessen dem Multidatenbanksystem die vollständige Kontrolle über die Ausführungsreihenfolge überlassen wird.

3. Vorschreiben von Aktionen (*Prescribe*)
Der globale Benutzer (d.h. der Benutzer der Föderation) benötigt das Recht, dem lokalen System Aktionen vorzuschreiben, wenn globale Integritätsbedingungen kontrolliert werden. So muß die lokale Ausführung von Aktionen zur Wiederherstellung erzwungen werden können.

Kommunikationsautonomie

Die Kommunikationsautonomie in unserem Modell beschreibt die Freiheit des lokalen Systems, über die Informationen zu entscheiden, die es an die Föderation weitergeben will. Dabei muß unterschieden werden, ob es sich um Informationen handelt, die einmalig zum Integrationszeitpunkt benötigt werden, oder solche, die regelmäßig ausgetauscht werden müssen. Insgesamt lassen sich drei Dimensionen unterscheiden: Ausführungsinformationen des lokalen DBMS (Statusinformationen, Commit-Ordnung der lokalen Transaktionen), Daten und Schemainformationen.

1. Bereitstellung von Informationen an das Multidatenbanksystem ohne lokale Modifikation (*Provide*)
In einem solchen Fall wird die Autonomie kaum verletzt, weil das globale System wie ein zusätzlicher Benutzer auf dem lokalen DBS agiert. Hierfür können z.B. externe Views auf den lokalen Datenbanken definiert sein. Ein anderes Beispiel ist die Bereitstellung von Statusinformationen an das MDBS, das dazu eine Anfrage an das lokale System richtet, ohne dessen Autonomie einzuschränken.

2. Bereitstellung zusätzlicher Informationen an das föderierte System (*Provide_add*)
Eine stärkere Einschränkung von Autonomie kann darin bestehen, Daten anzupassen. Das können z.B. Abbildungsfunktionen oder Daten sein, die im globalen System diese zusätzlichen Informationen zur Überbrückung der Heterogenität repräsentieren.

Die drei genannten Kategorien von Autonomie sind nicht orthogonal zueinander. So läßt sich feststellen, daß zumeist eine Restriktion, die an das Verhalten gestellt wird, Konsequenzen hat für die Strukturautonomie des LDBMS. Ebenso verhält es sich mit der Kommunikationsautonomie, die u.U. erfordert, daß strukturelle Veränderungen / Ergänzungen im LDBS zur Erleichterung der Kommunikation vorgenommen werden müssen.

Abbildung 4.5 gibt noch einmal zusammenfassend einen Überblick darüber, welche Komponenten wie in ihrer Autonomie eingeschränkt werden können.

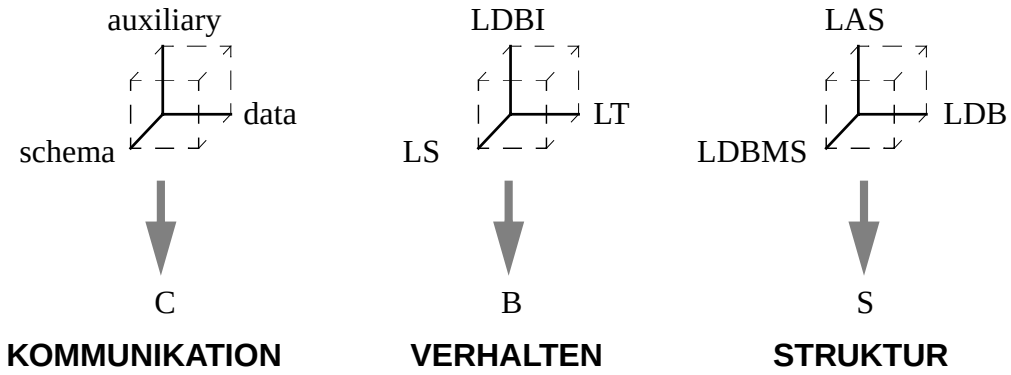


Abbildung 4.5: Kategorien von Autonomie

4.4.2 Der Trade-Off zwischen lokaler Autonomie und globaler Konsistenz

Nachfolgend wollen wir speziell den Konflikt, der bei globaler Konsistenzwahrung gegenüber der lokalen Autonomie von Legacy-Systemen auftritt, behandeln. Dazu definieren wir Konsistenztypen, die jeweils unterschiedliche Beschränkungen eines bestimmten Autonomiegrades erfordern. Diese lassen sich meist direkt aus der Spezifikation der Konsistenz ableiten.

4.4.2.1 Anforderungen aus Sicht der Konsistenzwahrung

In unserem Modell wird globale Konsistenz zwischen heterogenen Datenbanken in drei zueinander orthogonalen Dimensionen mit den dazugehörenden Wertebereichen beschrieben:

- Integritätsbedingungen zwischen Datenbanken (Abschnitt 2.4 / Abschnitt 2.7) **D**
  - Existenzabhängigkeiten (E)
  - Wertabhängigkeiten (V)
  - Strukturabhängigkeiten (S)
  - Verhaltensabhängigkeiten (B)
- Kontrollabhängigkeiten (Abschnitt 2.6) **C**
  - statisch (s)
  - dynamisch (d)
- Abschwächung von Konsistenz (Abschnitt 2.5) **R**
  - keine (0)

- zeitlich (t)
- extensional (x)

Globale Konsistenz kann somit durch einzelne Punkte oder Segmente innerhalb des durch die Dimensionen D, C und R aufgespannten Raumes charakterisiert werden. In Kurzschreibweise läßt sich eine bestimmte Konsistenzart durch ein Tripel (dep, ctrl, relax) $\in D \times C \times R$ beschreiben. Hierbei führen wir zusätzlich noch das Symbol '*' ein. Es steht jeweils stellvertretend für alle Punkte der Dimension, an dessen Position es notiert wird. Das können wir verwenden, um Konsistenzarten einzuführen, die vollständig für eine bestimmte Dimension gelten bzw. um mehrere Konsistenzarten zusammenzufassen.

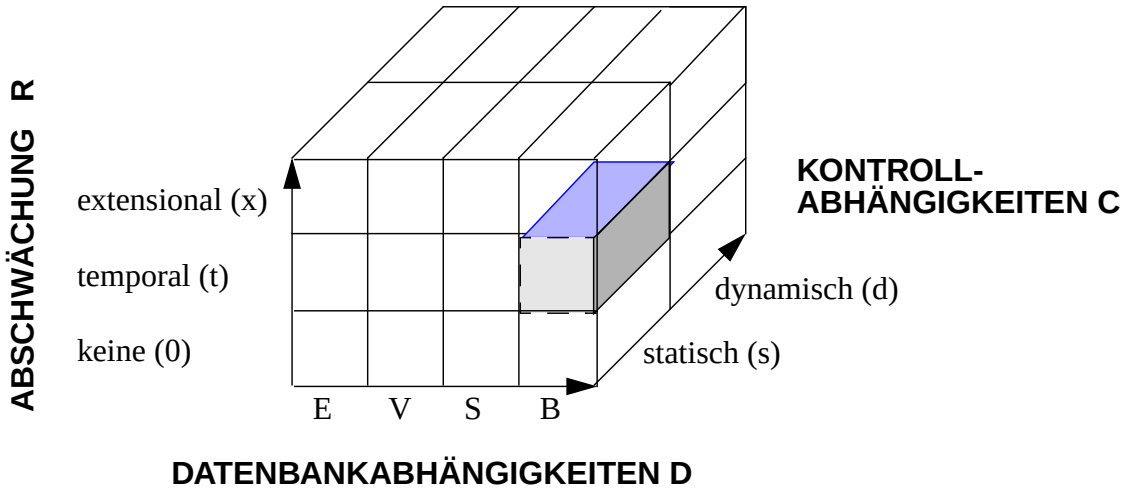


Abbildung 4.6: Dimensionen globaler Konsistenz

Abbildung 4.6 veranschaulicht noch einmal die Dimensionen, in denen globale Konsistenz spezifiziert werden kann. In der Darstellung ist beispielhaft ein Fall hervorgehoben, die Konsistenzart (B,s,t). Diese kann wie folgt interpretiert werden: Es handelt sich um eine Verhaltensabhängigkeit B, d.h. die Änderung des Verhaltens eines Objekts hat Einfluß auf das Verhalten eines anderen Objektes. Die Kontrollbeziehung zwischen den beteiligten Objekten ist statisch festgelegt. Die Konsistenzbedingung kann temporal abgeschwächt sein, d.h. es muß keine sofortige Propagierung der Verhaltensänderung erfolgen.

Aus einer gegebenen deklarativen Beschreibung muß eine prozedurale Spezifikation abgeleitet werden, um Konsistenz durch ECA-Regeln auszudrücken. Diese umfaßt im einzelnen:

- Definition der lokalen Events, die zu Konsistenzverletzung führen können und detektiert werden müssen:
  - Datenbank-Events auf Instanzenebene
  - Datenbank-Events auf Schemaebene (explizite Änderung der Semantik relevanter Daten, umfaßt Modifikationen von Struktur und Verhalten der lokalen Objekte)
  - implizite Änderung der Semantik von relevanten Daten (durch Modifikation von Applikationen)
- Definition der Bedingungen, die auf dem globalen DB-Zustand ausgewertet werden müssen
- Definition der Aktionen, die im Falle von Konsistenzverletzung ausgeführt werden müssen, prinzipiell sind zwei Möglichkeiten vorhanden:
  - Vorschreiben bestimmter lokaler Aktionen (Propagierung von Änderungen)

- Verbieten / Zurückweisen lokaler Aktionen, die zu Konsistenzverletzungen führen

- bei zeitlicher Abschwächung:
  - Definition der Events, die die Überprüfung der Konsistenzprädikate auslösen[7]
  - Beschreibung der Informationen über das aufgetretene Event für eine spätere Verarbeitung

- bei dynamischen Kontrollabhängigkeiten:
  - Definition der Entscheidungskriterien als Grundlage von Abgleichstrategien, z.B. Zeitpunkt einer Operation

4.4.2.2 Anforderungen gegenüber der Autonomie

Die aus der prozeduralen Spezifikation der Konsistenzbedingungen abzuleitenden Maßnahmen haben gleichzeitig auch einen direkten Effekt auf die lokale Autonomie. Dabei wird deutlich, daß einzelne Einschränkungen von lokaler Autonomie dazu dienen, die Gewährleistug bestimmter Aspekte von globaler Konsistenz zu unterstützen. Tabelle 4.3 zeigt die typischen Fälle in Legacy-Systemen, wo jeweils ein gewisser Grad an Autonomie aufgegeben werden muß, um den erforderlichen globalen Konsistenztyp einzuhalten.

Maßnahme	Beschränkung der Autonomie	Autonomiegrad	Konsistenztyp
Definition der Abhängigkeitsprädikate	Bereitstellung von lokalen Schemata (z.B. als externe Views)	*C-Provide*(schema)	(*,*,*)
Anpassung der Administration	Gewährung von Zugriffsrechten an externe Benutzer auf den relevanten Daten	*S-Modify*(LDB)	(E,*,*) (V,*,*)
Definition von Wert- und Objektkorrespondenzen	Bereitstellung von Informationen über lokale Daten, insbes. Wertebereiche	*C-Provide*(data)	(E,*,*),(V,*,*)
	Bereitstellung semantischer Informationen über lokale Daten	*C-Provide_add* (data)	(B,*,*)
Detektion von konsistenzverletzenden Events	Erzwingung der Signalisierung von Datenbank-Events bei lokalen Aktionen	*B-Restrict*(LDBI)	(E,*,*),(V,*,*)
	Detektion von Schemaveränderungen		(S,*,*),(B,*,*)
Detektion regelauslösender Events	Erzwingung der Signalisierung lokaler Datenbank-Events	*B-Prescribe*(LDBI)	(*,*,t)
Detektion lokaler Transaktions-Events	Teilnahme lokaler Transaktionen an globalen Protokollen (2PC, DTP)	*B-Restrict*(LT)	(*,*,0)
Erkennen semantischer Änderungen in lokalen Objekten	Übermitteln von Veränderungen in den Applikationen	*C-Provide_add* (data)	(B,*,*)

Tabelle 4.3: Autonomie vs. Konsistenz

7 Bei strenger Konsistenz wäre das auslösende Ereignis das Commit der lokalen Transaktion, d.h. es bedarf eines verteilten Transaktionsprotokolls, um 1 Copy Serializability zu garantieren.

Maßnahme	Beschränkung der Autonomie	Autonomiegrad	Konsistenztyp
Ausführung globaler Bedingungsprüfungen	Vorschreiben lokaler Aktionen durch ein externes System	*B-Prescribe*(LDBI)	(E,*,*),(V,*,*)
Propagierung von Änderungen			(E,*,*),(V,*,*)
Zurücksetzen konsistenzverletzender Aktionen	Ausführung verbieten; Zurückrollen der triggernden Transaktion; Ausführung kompensierender Aktionen	*B-Proscribe*(LDBI)	(E,*,*),(V,*,*)
Auswertung von Ausführungsinformationen	Signalisierung des Status von Transaktionen und DB-Verbindungen	*C-Provide*(aux)	(*,*,0),(*,*,e)
Abgleich divergierender Werte bei asynchronen Änderungen	Überschreiben von Daten auf untergeordneten Knoten / Datenbanken	*B-Prescribe*(LDBI)	(*,*,t)
Synchronisation lokaler Uhren	Eingriff in das Betriebssystem eines Knotens	*B-Restrict*(LS)	(*,d,t)
Ausführung von Regeln zu bestimmten Zeitpunkten	Vorschreiben lokaler Aktionen durch ein externes System	*B-Prescribe*(LDBI)	(*,*,t)
Protokollierung lokaler Events	Vorschreiben von Protokollierungsaktionen bei Detektion	*B-Prescribe*(LDBI)	(*,*,t)
	Bereitstellung der Event-Historie zur Regelausführung	*C-Provide_add (*data)	(*,*,t)

Tabelle 4.3: Autonomie vs. Konsistenz (Forts.)

4.4.3 Quantitative Betrachtung von Autonomie vs. Konsistenz

Eine Möglichkeit zur quantitativen Bestimmung von Autonomie besteht darin, die in einem Legacy-System verwalteten Daten dahingehend zu untersuchen, inwieweit sie Bestandteil globaler Integritätsbedingungen sind. Dabei lassen sich drei Kategorien identifizieren, in denen Autonomie quantitativ charakterisiert werden kann:

- Zugreifbarkeit der lokalen Daten
- Relevanz der lokalen Daten für die globale Konsistenz
- Häufigkeit externer Zugriffe auf die lokalen Daten

Die Zugreifbarkeit der Daten gegenüber externen Benutzern läßt sich in drei Stufen einteilen: schreibbar, nur lesbar und nicht zugreifbar.

Eine lokale Datenbank kann aus Daten bestehen, die nur lokal relevant sind, d.h. nicht Bestandteil globaler Konsistenzbedingungen sind. Als Bestandteil globaler Constraints können die Daten Primärdaten (Quelldaten) oder abhängige Daten (Zieldaten) sein. Handelt es um ungerichtete bzw. dynamische Kontrollabhängigkeiten, können Daten sowohl Quell- als auch Zieldaten in einem globalen Constraint sein.

Durch die Bestimmung der Häufigkeit, mit der auf lokale Daten bei der Konsistenzkontrolle zugegriffen wird, läßt sich auch ein laufzeitdynamischer Faktor in die quantitative Bestimmung der Autonomie aufnehmen. Dabei kann die Anzahl der Zugriffe auf die Daten gezählt werden, die bei der Sicherstellung globaler Konsistenz stattfinden.

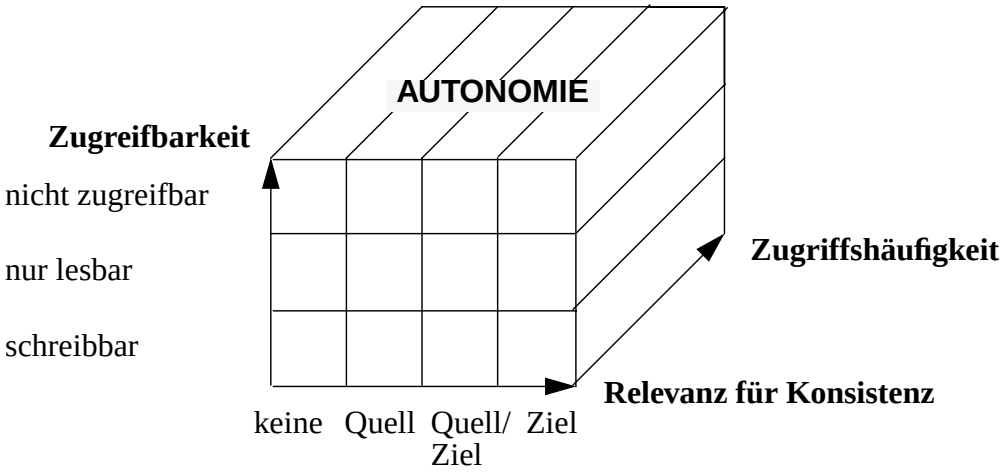


Abbildung 4.7: Quantitative Darstellung von Autonomie

Durch Kombination der Eigenschaften Zugreifbarkeit, Relevanz und Zugriffshäufigkeit kann man die Objekte einer lokalen Datenbank charakterisieren. Ein Objekt, das extern beschreibbar ist, das in globalen Integritätsbedingungen als Zieldatum von anderen abhängig ist und auf das sehr häufig zugegriffen wird, weist nur geringe Autonomie auf. Wenn ein Datum kein Bestandteil globaler Integritätsbedingungen ist und auch nicht extern verfügbar gemacht wurde, hat es maximale Autonomie. Eine Datenbank bzw. ein darauf operierendes lokales Applikationssystem kann in seiner Autonomie gegenüber einer externen Instanz durch den Anteil autonomer Objekte bewertet werden.

4.4.4 Zusammenfassung und Fazit

Die Diskussion über das Verhältnis von Autonomie und Konsistenz hat gezeigt, daß für die Realisierung eines Konsistenzkontrollmechanismus in einem heterogenen System zu klären ist, inwieweit an bestimmten Stellen Autonomie aufgegeben werden muß. Notwendige Einschränkungen der Verhaltens- und Kommunikationsautonomie haben zugleich Konsequenzen für die Struktur eines lokalen Systems. Da wir die Betrachtung auf Legacy-Applikationssysteme konzentrieren, sind aber die Möglichkeiten zu nachträglichen strukturellen Veränderungen dort sehr begrenzt. Ein DBMS bietet lediglich die Möglichkeit, die Konfiguration zu modifizieren, z.B. um die Performance zu verbessern. Veränderung bzw. Erweiterung von Applikationen, um globale Funktionalität zu unterstützen, bedeutet einen sehr hohen Aufwand und ist äußerst inflexibel bei notwendigen Erweiterungen der bestehenden Systeme.

Als eines der Haupthindernisse bei globaler Konsistenzkontrolle erweist sich die Detektion von Events, die einen Eingriff in die Autonomie von nichtkooperativen Systemen erfordert. Hierbei sind Lösungen anzustreben, die bestehende Systeme um zusätzliche Komponenten erweitern, die nur dann aktiviert werden, wenn eine Eventdetektion erforderlich ist. Demgegenüber erscheint die Ausführung globaler Konsistenzregeln einfacher, die in gewissem Sinne als eine zusätzliche lokale Anwendung im jeweiligen Applikationssystem realisiert werden können.

Es hat sich gezeigt, daß eine ständige Wahrung globaler Abhängigkeiten zwischen den Daten nicht primäres Ziel in einem Verbund autonomer Datenbanken sein kann. Abweichungen müssen toleriert werden und erfordern deshalb einen Abgleich zu bestimmten Zeitpunkten. Dem Problem der Kontrollabhängigkeiten zwischen den Systemen kommt deshalb beim Zusammenführen divergierender Daten eine besondere Bedeutung zu.

Ein existierendes lokales Schema kann zwar im Hinblick auf darauf operierende lokale Applikationssysteme nur schwerlich verändert werden. Es bietet aber oft die Möglichkeit zur Erweiterung bzw. kann die Grundlage für die Bereitstellung von Informationen an andere Teilnehmer eines Verbundes sein. Einschränkung der Kommunikationsautonomie heißt in diesem Zusammenhang, vom System bestimmte Informationen zu verlangen. Autonomie kann hier als ein Ausdruck der Selbstbestimmung der Organisationen gesehen werden, die ein lokales System betreiben. Der Informationsbereitstellung an externe Benutzer geht somit immer die Abstimmung zwischen den Organisationen voraus mit den daraus resultierenden Festlegungen für die jeweilige lokale Datenbankadministration.

Kapitel 5

Aktive Objekte zur Konsistenzkontrolle in heterogenen Systemen

Die Idee, die diesem Kapitel zugrunde liegt, besteht darin, lokale Datenbanksysteme (allgemein: lokale Datenspeicher) durch ein gemeinsames Objektmodell zu integrieren, wobei die Teilnehmer als aktive Objekte modelliert werden, die in der Lage sind, auf Ereignisse zu reagieren. Dazu ist die Definition eines Objektmodells als auch eines Regel- und Ausführungsmodells erforderlich.

In Abschnitt 5.2 wird das Regelmodell erläutert. Dazu diskutieren wir zunächst den Begriff des Events im Kontext von Multidatenbanken. In Analogie zu Transaktionen, die lokal oder global gestartet werden können, lassen sich lokale und globale Events in Abhängigkeit von der Ebene definieren, auf der sie detektiert werden. Damit wird die aus aktiven Datenbanken bekannte Event-Klassifikation erweitert. Die Unterscheidung einer lokalen und globalen Ebene erlaubt auch die Definition von ECA-Regeltypen, die durch unterschiedliche Reichweite gekennzeichnet sind. Dabei wird der Zusammenhang zwischen Regeltyp und den dabei notwendigen Beschränkungen lokaler Autonomie deutlich. Multidatenbanksysteme, darunter heterogene aktive Objektsysteme, lassen sich in ihren aktiven Eigenschaften anhand der von ihnen unterstützten Regeltypen charakterisieren.

Das Ausführungsmodell für Regeln von aktiven Objekten wird in Abschnitt 5.3 dargestellt. Prinzipiell wird zwischen einer direkten und einer indirekten Verarbeitung lokaler Events unterschieden, je nachdem, ob die Events direkt unter Kontrolle eines globalen Systems stehen oder ihre Auswertung vom Auftreten entkoppelt ist. Zur Beschreibung des Ausführungsmodells werden einige Grundbegriffe eingeführt, mit denen der “Weg” eines Events vom Auftreten über die Signalisierung (möglicherweise über mehrere Abstraktionsebenen hinweg) bis zur Interpretation beschrieben werden kann. Dabei lassen sich den einzelnen “Zwischenstationen” Zeitpunkte zuordnen, deren Ordnung in unterschiedlichen Situationen variieren kann. Der aus homogenen aktiven DBS bekannte Begriff des Kopplungsmodus wird hierbei in einer erweiterten Form verwendet.

Im Abschnitt 5.4 betrachten wir die semantischen Ausdrucksmöglichkeiten von Datenmodellen, die als kanonische Modelle in Multidatenbanksystemen vorgeschlagen wurden. Dabei sind Integritätsbedingungen von Interesse, die entweder modellinhärent sind oder explizit spezifiziert werden müssen. Da die Modelle datenbankübergreifende Sachverhalte beschreiben, handelt es sich um globale Constraints. Am Beispiel eines semantisch sehr reichhaltigen Modells

(BLOOM) wird gezeigt, daß sich prinzipiell alle Arten von Bedingungen auf ECA-Regeln abbilden lassen. Daraus lassen sich zwei Schlüsse ziehen: Einerseits sind ECA-Regeln als Basismechanismus zur Kontrolle von modellinhärenten oder auch expliziten Integritätsbedingungen einsetzbar. Andererseits könnte man bei der Entwicklung eines Multidatenbanksystems ein semantisch relativ armes Modell auswählen und für die Realisierung von Constraints auf ECA-Regeln zurückgreifen. Wir sind diesen Weg gegangen und haben uns für ODMG-93 als etabliertes Referenzmodell für OODBMS entschieden. Die wichtigsten Charakteristika dieses Modells werden in Abschnitt 5.5 erläutert.

5.1 Lösungsansatz: Aktive Objekte

Um in einer Föderation existierender Systeme Mechanismen zur Konsistenzkontrolle zu realisieren, müssen die lokalen Applikationen, Datenbanksysteme und Daten durch ein geeignetes Objektmodell modelliert werden, das zusätzlich mit Regelmechanismen auszustatten ist. Eines der Haupthindernisse, die lokale Autonomie, wirkt sich erschwerend für die Integration von Legacy-Systemen aus.

Objektorientierte Datenmodelle eignen sich als kanonische Datenmodelle am besten, weil sie Datenabstraktion und Einkapselung unterstützen und es somit ermöglichen, die Heterogenität durch die Bereitstellung von abstrakten Datentypen zu überwinden [CL88]. Die zu integrierenden Komponenten weisen als Objekte im globalen System unterschiedliche Granularität auf. In Analogie zur Darstellung in Abbildung 4.4 auf Seite 66 können die jeweils zugänglichen Schnittstellen (LAI, LDBI) bzw. die Operationen auf den Datenbankobjekten als Methoden globaler Proxy-Objekte modelliert werden (in [Buc90] als *Local Application Interface Object*, LAI-Objekt, bezeichnet). Die Idee der Einkapselung läßt sich somit geeignet auf Legacy-Systeme übertragen, deren Struktur nicht zugänglich ist.

In [Buc90] wird die Idee eines aktiven Objektraums für die Modellierung heterogener Systeme skizziert, wobei ein aktives Objekt dadurch definiert ist, daß es in der Lage ist, autonom und asynchron auf Ereignisse durch Ausführung von Regeln zu reagieren.

Dabei haben globale Proxy-Objekte die Aufgabe, eine objektorientierte Sicht auf lokale Daten zu repräsentieren [HZ90, BSKW91]. Somit existiert auf der globalen Ebene ein föderiertes Schema, das aus den lokalen Schemata gewonnen werden muß (eine Übersicht über Schemaintegration geben Batini u.a in [BLN86], siehe auch [DH84]). Dieser Ansatz ist insbesondere für die Behandlung von “primitiven” lokalen Datenbankereignissen geeignet. Die Proxy-Objekte können in einem Cache gehalten werden oder auch persistent abgelegt sein. Letzteres würde es ermöglichen, die Sicht in einer objektorientierten Datenbank zu verwalten. Rosenthal und Seligman diskutieren verschiedene Alternativen für die Architektur von heterogenen Systemen, die semantisch verwandte Informationen verwalten [RS94]. Dabei werden neben dem Sichtkonzept auch ein Datenfluß zwischen bestehenden interoperablen Systemen sowie eine Migration als mögliche Szenarien dargestellt.

Bei der Modellierung des Verhaltens einer lokalen Datenbank oder eines Applikationssystems durch globale Proxy-Objekte, deren Methodenschnittstelle jeweils auf den lokalen Interfaces beruht, sind auch benutzerdefinierte Ereignisse gut zu integrieren. Allerdings erfordert das einen höheren Aufwand für die Interpretation der Message-Parameter. LAI-Objekte mit aktiven Eigenschaften sind folgendermaßen charakterisiert [Buc90]:

1. Reaktion auf Ereignisse in der Umgebung als auch auf interne Zustandsveränderungen
2. Parallele Ausführung von mehreren Aktivitäten, unter ihnen Event-Detektion

3. Möglichkeit der Definition der zu detektierenden Events sowie der zugehörigen Reaktionen
4. Synchronisation innerhalb des Objekts sowie zwischen den Objekten
5. Persistenz (soweit erforderlich)

Diese Anforderungen sind zugleich Voraussetzung für die Kontrolle globaler Integritätsbedingungen in einem heterogenen System. Es bedarf einer Möglichkeit der Eventdetektion in der Umgebung, d.h. im lokalen System (1), die Definition der Events und der zugehörigen Reaktionen entspricht der Semantik von ECA-Regeln (3). Die Ausführung der Aktivitäten, die durch den Benutzer veranlaßt werden, sowie der Regeln muß parallel erfolgen und erfordert eine entsprechende Synchronisation (2,4). Die Eigenschaft der Persistenz ist für Objekte der globalen Metadatenbank zu fordern (5).

5.2 Ein ECA-Regelmodell für aktive Objekte in Multidatenbanken

5.2.1 Eventmodell

5.2.1.1 Eventbegriff

Die Dynamik der Realwelt muß sich in der Dynamik der Objekte in der Datenbank widerspiegeln, die durch Ereignisse (Events) bestimmt wird. Ein Event geschieht zu einem bestimmten Zeitpunkt und kann eine oder mehrere Zustandsänderungen von Datenbankobjekten zur Folge haben. Zur Klassifizierung von Events in aktiven Datenbankystemen wurde eine Reihe von Arbeiten veröffentlicht (siehe hierzu u.a. [BBKZ93]). Events können primitiv oder zusammengesetzt sein. Ein zusammengesetztes Event entsteht durch Komposition primitiver Events durch Anwendung einer abgeschlossenen Algebra, wie sie z.B. in Snoop vorgeschlagen wurde [CM91]. Weitere Beispiele finden sich in HiPAC, SAMOS, Ode und REACH [DBM88, GD94, GJS92, BZBW95].

Bei der Erkennung und Verarbeitung von Ereignissen lassen sich verschiedene Abstraktionsebenen definieren, was sich am Beispiel eines herkömmlichen Applikationssystems zeigen läßt. Ein Event wird ausgelöst durch eine Intervention des Benutzers an einem User Interface (z.B. Maus-Klick), dieses Event wird transformiert in einen Funktionsaufruf des zugrundeliegenden lokalen Applikationssystems. Ein solcher Funktionsaufruf wird übersetzt in eine Operation, die über die Datenbankschnittstelle (Database Interface) ausgelöst wird. Die Ausführung dieser Operation führt zu einem Datenbank-Ereignis, d.h. der Zustand der Datenbank verändert sich. Auch in umgekehrter Richtung kann eine Transformation eines datenbankinternen Ereignisses in Ereignisse höherer Abstraktionsebenen stattfinden (z.B. Ausgabe einer Nachricht über eingetretene Zustandsveränderungen). Abbildung 5.1 zeigt den prinzipiellen Ablauf bei Erzeugen eines Events x, der äquivalente Events x', x" ... auf darunterliegenden Ebenen auslöst.

In der Ebene 0 finden *datenbankinterne* Events statt (nachfolgend als 0-Events bezeichnet), die direkt zu einer Zustandsveränderung in der Datenbank führen können. Datenbankinterne Events umfassen Datenbank-Events und Transaktions-Events. Die Events auf den Ebenen 1, 2, ..., n werden als *datenbankexterne* Events bezeichnet (in Kurzform *j*-Events mit *j* = Nr. der Abstraktionsebene). Die Menge der Events auf allen Ebenen des Systems wird mit *E* bezeichnet.

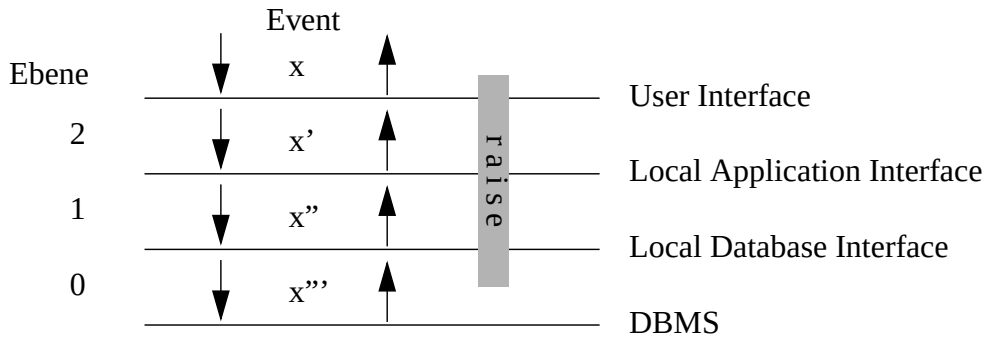


Abbildung 5.1: Eventdetektion in mehreren Ebenen

Definition 5.1 (Datenbank-Event)
Die Menge der Datenbank-Events E_{DB} ist definiert als $E_{DB} = \{e, c, p\}$ mit $e \in$ {insert, delete, update, read}, von dem eine Instanz der Klasse c (instanzenorientiert) oder eine Menge von Instanzen betroffen ist, die das Prädikat p erfüllen (mengenorientiert).

Definition 5.2 (Transaktions-Event)
Die Menge der Transaktions-Events E_T in einer Datenbank ist definiert durch E_T = {begin, commit, save, abort}. Diese können über eine lokale (E_{LT}) oder eine globale Transaktionsschnittstelle (E_{GT}) aufgerufen werden.

Folgende Funktionen werden für Events $E \in E$ definiert.

- occur(E)) Auftreten eines Events, bewirkt Zustandsveränderung (bei Transaktionen wird diese erst bei commit sichtbar), E = {0-Event}.
- raise(E_1,E_2) Signalisierung eines j-Events E_1 durch einen j+1-Event (oder j-1-Event) E_2 (E_1,$E_2 \in E$). Voraussetzung ist die Äquivalenz von E_1 und E_2.
- detect(E) Erkennung einer Events E, dies kann bei Signalisierung (raise) oder Auftreten (occur) erfolgen.
- interpret(E) Interpretation eines Events $E \in E$, d.h. Ausführung der zugehörigen Regel.
- visible (E_2,E_1) Sichtbarwerden des Events E_1 nach Eintreten des Events E_2 ($E_1 \in E_{DB}$, $E_2 \in E_T$). E_1 und E_2 können als eine Sequenz eines Datenbank-Events und eines Commit-Events aufgefaßt werden.

5.2.1.2 Events in Multidatenbanken

Multidatenbanksysteme wurden bisher typischerweise nach dem Integrationsgrad unterschieden (vgl. Abschnitt 4.1), wobei aktive Fähigkeiten einzelner Systeme nicht betrachtet wurden. Diese werden im folgenden für die Definition von Klassen aktiver Multidatenbanksysteme einbezogen. Ein lokales System (Applikationssystem, Datenbanksystem) in der Föderation wird als “aktiv” bezeichnet, wenn es in der Lage ist, auf bestimmte Datenbankereignisse mit Hilfe von Regeln zu reagieren.

Der eingeführte Eventbegriff läßt sich auch auf eine Datenbankföderation übertragen, in der eine Interaktion zwischen lokalen Komponenten und einer globalen Ebene stattfindet. Aus der Sicht eines lokalen Systems kann die globale Ebene als eine zusätzliche Ebene betrachtet werden, auf der Events ausgelöst werden können (z.B. durch globale Queries) bzw. an die Events signalisiert werden. Letzteres erlaubt die Definition lokaler Systeme als aktive Objekte, wenn die Fähigkeit zur selbständigen Detektion von Events gegeben ist.

Ein allgemeines Modell für aktive Multidatenbanken unter dem Namen RIMM (*Reactive Integration Multidatabase Model*) wurde von Pissinou u.a. [PRV95] vorgestellt, wobei das Verhalten von lokalen und globalen Objekten spezifiziert ist. Die Eventdetektion muß in zwei Ebenen, der lokalen Ebene und der globalen Ebene, wie sie in einem föderierten Datenbanksystem bestehen, betrachtet werden. Ein globales Event wird an der globalen Schnittstelle detektiert, ein lokales Event an einer lokalen Schnittstelle.

In Fortführung dieser Idee klassifizieren wir Events in einer Multidatenbankumgebung, was in nachfolgender Abbildung 5.2 dargestellt ist.

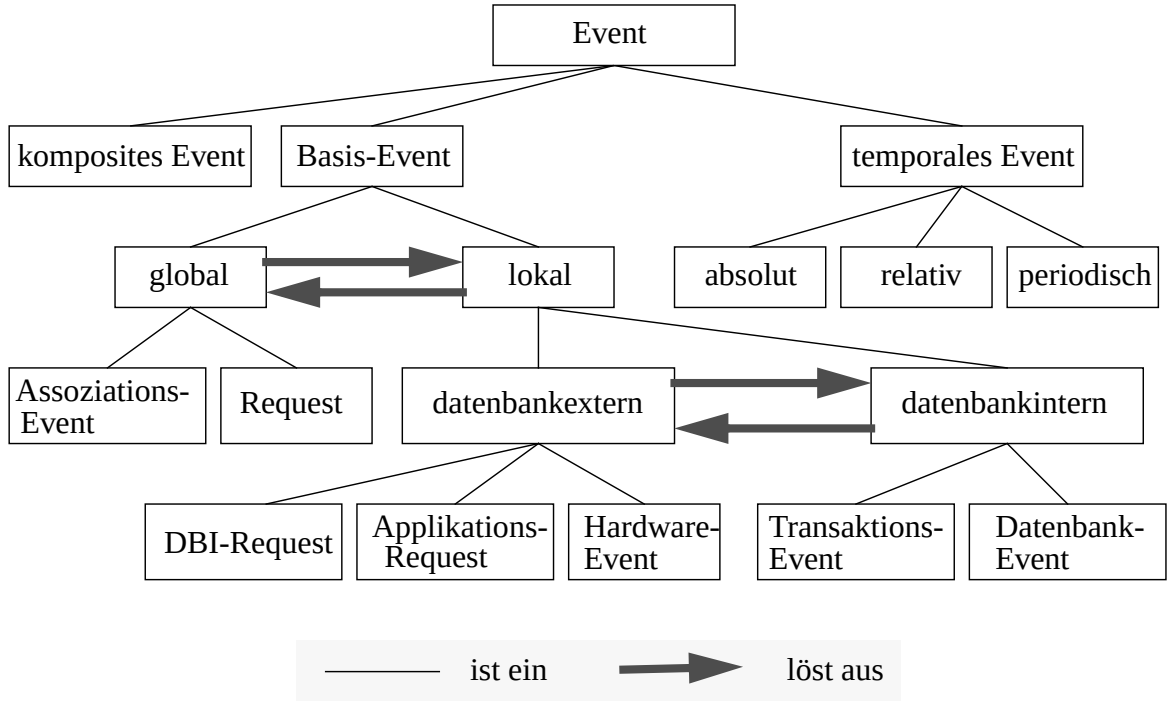


Abbildung 5.2: Event-Hierarchie in einer Multidatenbankumgebung

Events werden im allgemeinen durch einen Datenbankbenutzer ausgelöst, der eine Operation aufruft, was auch als *Request* bezeichnet wird. Ein Operationsaufruf kann ein oder mehrere Datenbank-Events auslösen. Diese Requests können am lokalen Datenbankinterface (*DBI Request*) oder an einem lokalen Applikationsinterface (*Application Request*) abgesetzt werden. Entsprechend gehören sie zu den datenbankexternen Events. Zu dieser Kategorie zählen auch hardwarebedingte Events (z.B. Signale), die aber nicht näher betrachtet werden.

Globale Events werden in *Assoziations-Events* und *Request*-Events eingeteilt. Assoziations-Events beziehen sich auf die Teilnahme einer Datenbank an der Föderation, wobei die Ereignisse Verbinden (*connect)* und Lösen der Verbindung (*disconnect)* unterschieden werden. Request-Events werden als globale Operationsaufrufe detektiert. Diese lösen lokale Datenbank-Events an der Schnittstelle aus, die für das Multidatenbanksystem zugänglich ist.

Die in Abbildung 5.2 dargestellte Beziehung “löst aus” zwischen einem globalen Request und einem lokalen datenbankexternen Event setzt das Vorhandensein einer entsprechenden Abbildungsfunktion voraus. Eine wesentliche Anforderung ist somit, daß zur Laufzeit globale Requests als lokale datenbankexterne Events detektiert werden können bzw. auch umgekehrt ein lokales Datenbank-Event einen mit ihm korrespondierenden globalen Request auslösen kann.

5.2.2 Regelmodell

Die in Abschnitt 5.2.1 gegebene Charakterisierung von Events behält ihre Gültigkeit auch für den Bedingungs- bzw. Aktionsteil von Multidatenbank-Regeln, in denen wiederum Ereignisse ausgelöst werden können.

In Analogie zu einem Ereignis wird eine Bedingung / Aktion als *lokal* bezeichnet, wenn deren Auswertung bzw. Ausführung innerhalb einer lokalen Datenbank unter Verwendung von Funktionen des lokalen Datenbank- oder Applikations-Interfaces erfolgt. Wir sprechen von einer *globalen Bedingung*, wenn diese nur unter Verwendung der globalen Schnittstelle getestet werden kann (z.B. bei Zugriff auf Objekte aus mehreren lokalen Datenbanken über globale Proxy-Objekte GP). Das kann auch den Zugriff auf semantische Objekte S aus der globalen Metadatenbank einschließen (vgl. hierzu auch Seite 35).

Cond (X) mit $X = \{x_1, x_2, \ldots, x_n\}$
Cond (X) = global $\Leftrightarrow (\exists\, x_i)\, ((\, x_i \in X \wedge x_i \in GM)$

Ebenso wird eine Aktion als *globale Aktion* definiert, wenn sie über eine globale Schnittstelle aktiviert wird. Bei Objekten aus dem globalen Data Dictionary liegt die Annahme zugrunde, daß auf sie immer über eine globale Schnittstelle zugegriffen wird, somit sind Aktionen auf ihnen immer als global anzusehen.

Die (sinnvollen) Kombinationen der Charakteristika von Ereignis, Bedingung und Aktion gestatten die Einführung von Typen von ECA-Regeln in Multidatenbanken (vgl. Tabelle 5.1). Dabei wird der Zusammenhang zur lokalen Autonomie deutlich. Lokale Regeln (Typ 1) lassen sich in aktiven DBMS durch Definition von Triggern u.ä. im Schema ausdrücken; das bedeutet eine Datenbankerweiterung, *S-add*(LDB).

Regeln von Typ 2 und 3 erfordern die Vergabe von Zugriffsrechten an einen globalen Benutzer, *S-modify*(LDB), sowie die Möglichkeit, einem lokalen System von außen bestimmte Aktionen vorzuschreiben, *B-prescribe*(LDBI). Optional kann Autonomie durch das Verbieten lokaler Aktionen verletzt werden, insbesondere solchen, die zu Konsistenzverletzung führen (*B-proscribe*(LDBI)). Um globale Aktionen definieren zu können, ist es hilfreich, Informationen über Struktur und Semantik der lokalen Daten zu erhalten.

Die Definition von Regeln, die durch globale Ereignisse ausgelöst werden (Typ 4), erfordert die Möglichkeit, lokale Ereignisse auch global detektieren zu können. Das erfordert jedoch die Einführung von Restriktionen bei der Ausführung lokaler Datenbankoperationen (*B-restrict*(LDBI)). Sollen lokale Transaktionsereignisse bei der Ausführung globaler Regeln berücksichtigt werden (z.B. als Kopplungsmodi), so sind auch Beschränkungen lokaler Transaktionen erforderlich (*B-restrict*(LT)).

Regeltyp	Ereignis	Bedingung	Aktion	Autonomiebeschränkung	
				notwendig	optional
1	lokal	lokal	lokal	*S-add*(LDB)	-
2	lokal	lokal	global	*S-modify*(LDB) *B-prescribe*(LDBI)	*C-provide*(data) *C-provide_add*(data) *B-proscribe*(LDBI)

Tabelle 5.1: Typen von ECA-Regeln in Multidatenbanken

Regeltyp	Ereignis	Bedingung	Aktion	Autonomiebeschränkung	
				notwendig	optional
3	lokal	global	global	*S-modify*(LDB) *B-prescribe*(LDBI)	*C-provide*(data) *C-provide_add*(data) *B-proscribe*(LDBI)
4	global	global	global	*S-modify*(LDB) *B-prescribe*(LDBI) *B-restrict*(LDBI)	*C-provide*(data) *C-provide_add*(data) *B-proscribe*(LDBI) *B-restrict*(LT)

Tabelle 5.1: Typen von ECA-Regeln in Multidatenbanken (Forts.)

Tabelle 5.2 ermöglicht eine Charakterisierung des aktiven Verhaltens einer Multidatenbankumgebung. Betrachtet werden müssen die aktiven Eigenschaften sowohl der lokalen Systeme als auch des föderierten Systems. Dabei sind von einem lokalen DBMS nicht unbedingt aktive Fähigkeiten zu erwarten und können auch nicht nachträglich in ein kommerziell verfügbares DBMS integriert werden, dessen Autonomie gewahrt werden soll. Dafür sind Lösungen zur Event-Detektion in höheren Schichten zu suchen (vgl. hierzu auch Abschnitt 6.3).

Fall	Multidatenbanksystem	Lokales System	Unterstützte Regeltypen
1	passiv	passiv	-
2	passiv	aktiv	1
3	aktiv	passiv	4
4	aktiv	aktiv	{1, 4},{1, 2, 3, 4}

Tabelle 5.2: Einteilung aktiver Multidatenbanken

Fall 1 kennzeichnet die typische Situation in heterogenen Umgebungen, in denen keine aktiven Eigenschaften vorhanden sind. Um globale Integritätsbedingungen einzuhalten, werden z.B. in periodischen Abständen Jobs gestartet, die Aufgaben der Konsistenzwiederherstellung wahrnehmen. Für die Triggerung dieser Funktionen ist entweder eine lokale Applikation verantwortlich, oder es werden Dienste des Betriebssystems genutzt.

Im Fall 2 können globale Constraints bei Auftreten von lokalen Ereignissen zwar nicht unmittelbar geprüft werden, aber die für die Integrität relevanten Ereignisse bzw. Zustandsveränderungen können aufgezeichnet und zu einem späteren Zeitpunkt ausgewertet werden. Lokale Trigger können lokale Integritätsbedingungen kontrollieren. In Abschnitt 9.2 werden Algorithmen skizziert, wie damit auch globale Constraints geprüft werden können.

Der Fall 3 beschreibt eine Situation, in der nur das föderierte System in der Lage ist, auf Ereignisse durch Ausführung von Regeln zu reagieren. Dies können allerdings nur Ereignisse sein, die an der globalen Schnittstelle auftreten. In Umgebungen mit autonomen Subsystemen ist dieser Fall aber nicht wünschenswert, da lokal ausgelöste Konsistenzverletzungen nicht behandelt werden können.

Nur im vierten Fall ist auch wirklich gewährleistet, daß alle Arten von Integritätsbedingungen durch das föderierte System überwacht werden, wenn jeder Typ von ECA-Regeln ausgeführt werden kann. Gegenüber den ersten drei Fällen ist es aber erforderlich, daß lokale Ereignisse auch Regeln feuern, die eine globale Reichweite haben, d.h. letztendlich, daß ein zusätzlicher Kommunikationsmechanismus zwischen lokalem und globalem System notwendig wird. Es ergeben sich beispielsweise verschiedene Ansätze zur Integration aktiver Mechanismen in die

Middleware, wie sich am Beispiel CORBA zeigen läßt [BKK96]. Sind hingegen nur Regeln vom Typ 1 und 4 verfügbar, so müssen Lösungen, wie im zweiten Fall skizziert, gefunden werden unter Inkaufnahme einer zeitweiligen Konsistenzverletzung.

5.3 Ausführungsmodell für aktive Objekte in Multidatenbanken

Für die Detektion und die Verarbeitung lokaler Events durch ein aktives Objektsystem in Multidatenbanken müssen prinzipiell zwei Fälle unterschieden werden:

- direkte Verarbeitung lokaler Events
- indirekte Verarbeitung lokaler Events

Bei eintreffenden lokalen datenbankexternen Events kann der zum Event gehörende Aufruf (z.B. ein Kommando des LDBI) detektiert und **direkt** unter Kontrolle eines aktiven globalen Systems verarbeitet werden. Das Prinzip der Verarbeitung besteht somit in einer Weiterleitung des lokalen datenbankexternen Events an das aktive System zum Zeitpunkt der Detektion.

Das Prinzip der **indirekten** Eventverarbeitung besteht darin, das lokale Event nach dessen Auftreten (occur(E_L)) zunächst lokal zu registrieren, um es zu einem späteren Zeitpunkt weiter zu verarbeiten. Aufgrund der möglichen zeitlichen Verzögerung zwischen Auftreten und Verarbeitung des Events müssen Informationen über die Event-Parameter bzw. den Zustand der Datenbank zum Auftrittszeitpunkt gespeichert werden.

5.3.1 Direkte Verarbeitung lokaler Events

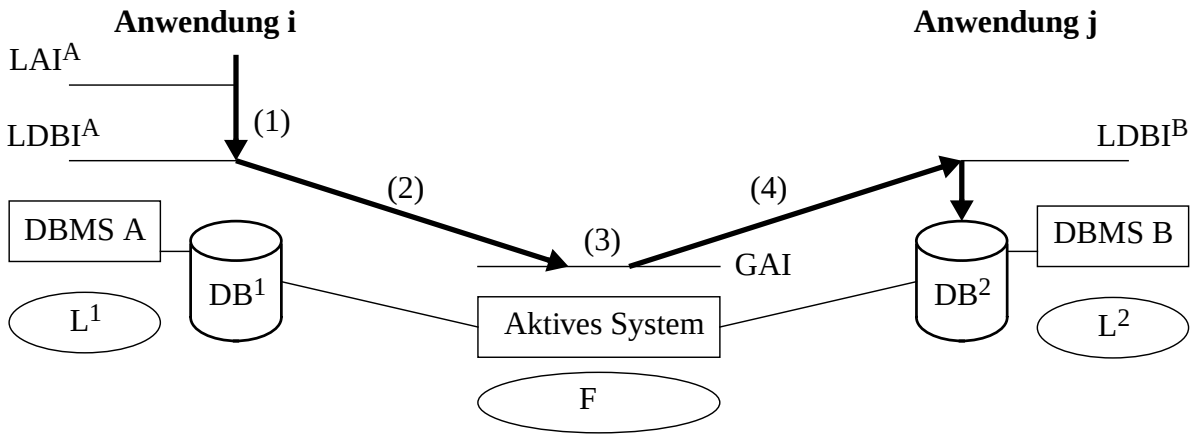
5.3.1.1 Ablaufmodell

Am Beispiel zweier Objekte o_1 und o_2 ($o_1 \in \text{inst}(c_{1i})$, $o_2 \in \text{inst}(c_{2j})$) aus zwei verschiedenen Datenbanken DB^1 und DB^2, die jeweils den gleichen Wert v aufweisen sollen, sei der Ablauf der Regelverarbeitung bei direkter Eventverarbeitung demonstriert. Abbildung 5.3 illustriert die Abfolge der Phasen von der Detektion bis hin zur Propagierung eines Events.

1. Detektion des lokalen Events: detect(E_L)
   Die Detektion des Events, z. B. update c_{1i}, kann am lokalen Applikationsinterface LAI^A erfolgen, was aber Kenntnisse über die Semantik der Funktionen voraussetzt, die an dieser Schnittstelle zur Verfügung stehen.
   Eine andere Möglichkeit besteht in der Detektion des Update-Events direkt am lokalen Datenbank-Interface $LDBI^A$, d.h. außerhalb der lokalen Applikation.

2. Signalisierung des lokalen Events als globales Event: raise(E_G,E_L)
   Die Signalisierung (über Prozeß- und Systemgrenzen) kann auch als Aktionsteil einer lokal getriggerten ECA-Regel angesehen werden
   ON E_L
   DO “signal E_L to global rule manager”[8]

8 Hierbei handelt es sich um eine entartete ECA-Regel ohne Bedingungsteil C.



Phasen der Eventverarbeitung:
(1) Detektion eines lokalen Events
(2) Signalisierung
(3) Detektion des globalen Events
(4) Regelausführung

Komponenten des Multidatenbanksystems:

L^i	Lokales Schema von DB^i
F	Föderiertes Schema
$LDBI^A$	Local Database Interface von DBMS A
LAI^A	Local Application Interface von DBMS A
GAI	Global Application Interface

Abbildung 5.3: Direkte Verarbeitung lokaler Events in einem Multidatenbanksystem

3. Detektion des globalen Events: detect(E_G)
   Vorausgesetzt wird die Existenz einer mit der lokalen Klasse c_{1i} korrespondierenden globalen Klasse (***c**lass **g**lobal*) cg_{1i}, die eine Sicht auf die lokale Klasse repräsentiert. Das lokale Update-Event muß auf ein Update-Event auf der globalen Klasse cg_{1i} abgebildet und anschließend als globales Event detektiert werden.

4. Interpretation des globalen Events E_G
   Die Detektion des globalen Events update cg_{1i} führt zur Ausführung einer globalen Regel, die im globalen aktiven System verwaltet wird. Im speziellen Fall erfolgt eine Propagierung der Update-Operation, indem eine Operation auf der mit der Zielklasse c_{2j} korrespondierenden globalen Klasse cg_{2j} aufgerufen wird. Die Umsetzung der Operation erfolgt dann über das lokale Datenbank-Interface $LDBI^B$ der Zieldatenbank DB^2.

Diesem Regelausführungsszenario liegt die Annahme zugrunde, daß die Instanzen jeweils nur lokal gespeichert sind, es aber einen Cache auf globaler Seite zur Zwischenspeicherung von Anfrageresultaten geben kann.

5.3.1.2 Auswirkungen auf lokale Autonomie

Die einzelnen Phasen der Eventverarbeitung haben einen Einfluß auf die lokale Autonomie der lokalen Systeme i bzw. j. Dabei erweist sich insbesondere die notwendige Eventdetektion als schwierig, weil alle lokalen DB-Operationen und Transaktionen bei ihrer Ausführung entsprechend angepaßt werden müssen. Tabelle 5.3 stellt noch einmal die einzelnen Verarbeitungsschritte den notwendigen Einschränkungen lokaler Autonomie gegenüber.

Phase		Autonomiebeschränkung	Anwendung
1	Detektion des lokalen Events	*B-Restrict*(LDBI) *B-Restrict*(LT)	i
2	Signalisierung des lokalen Events als globales Event	*B-Prescribe*(LS)	i
3	Detektion des globalen Events	-	-
4	Interpretation des globalen Events	*B-Prescribe*(LDBI)	j

Tabelle 5.3: Direkte Eventverarbeitung vs. Autonomie

5.3.1.3 Semantische Aspekte

Zeitliche Aspekte

Nur wenige Autoren [PDW+93] diskutieren die temporale Semantik der Parameter von detektierten Events in einem heterogenen aktiven System. Prinzipieller Unterschied zu einem homogenen aktiven System ist, daß eine Abbildung zwischen lokalem und globalem Event (d.h. einem Event auf der globalen Applikationsschnittstelle) definiert sein muß, so daß lokale Events globale Regeln feuern können, die an globale Ereignisse gebunden sind. Die Ausführung der Regel kann relativ zum Auftreten des Datenbank-Events zu unterschiedlichen Zeitpunkten stattfinden.

t(detect(E_L))	Zeitpunkt der lokalen Eventdetektion; Erkennung der Auslösung eines Datenbank-Events E_L durch ein korrespondierendes Event E'_L auf einer Ebene j ($j > 0$), d.h. Operation raise(E_L, E'_L)
t(raise(E_G,E_L))	Zeitpunkt der Eventsignalisierung an das globale System
t(detect(E_G))	Zeitpunkt der globalen Eventdetektion
t(occur(E_L))	Zeitpunkt des Auftretens eines Datenbank-Events
t(interpret(E_G))	Zeitpunkt der Ausführung der globalen Regel
t(visible(E_{LT},E_L))	Commit-Zeitpunkt der lokalen Transaktion, in der ein lokales Event auftrat
t(visible(E_{GT},E_G))	Commit-Zeitpunkt der globalen Transaktion, in der ein globales Event detektiert wurde

Dabei können folgende Fälle auftreten:

Fall 1:
Event-Signalisierung und Ausführen der Regel vor Auftreten des lokalen Datenbank-Events
• t(detect(E_L)) < t(raise(E_G,E_L)) < t(detect(E_G)) < t(interpret(E_G)) < t(occur(E_L))

Fall 2:
Event-Signalisierung vor Auftreten des lokalen Datenbank-Events, Ausführen der Regel nach Auftreten des lokalen Events
• t(detect(E_L)) < t(raise(E_G,E_L)) < t(detect(E_G)) < t(occur(E_L)) < t(interpret(E_G))

In Fall 1 und 2 wird ein lokales Event E'_L an einer lokalen Schnittstelle (LAI, LDBI) als datenbankexternes Event detektiert. Das kann z.B. der Aufruf eines SQL-Befehls sein, der an einen Server geschickt wird. Dabei wird vorausgesetzt, daß die Funktion raise(E_L,E'_L) bekannt ist, so daß daraus auf das ausgelöste lokale Datenbank-Event E_L geschlossen werden kann. Das lokale Event wird zum föderierten System gesendet und in ein globales Event übersetzt, d.h. es erfolgt der Aufruf einer Operation, die mit dem lokalen Ereignis korrespondiert. Frühestens zu diesem Zeitpunkt kann das lokale Event durch Auslösen des globalen Events wirksam werden

(occur). Das globale Event kann durch eine globale Regel interpretiert werden, entweder vor dem lokalen Event (Fall 1) oder danach (Fall 2).

Kopplungsmodi

In [DBM88, Buc94, BZBW95] werden verschiedene Arten von Kopplungsmodi homogener aktiver Datenbanksysteme diskutiert, die jeweils die Existenz eines einheitlichen Transaktionsmanagers voraussetzen. Das heißt, der Kopplungsmodus definiert den Interpretationszeitpunkt des Events relativ zur Transaktion, innerhalb der das Event auftritt.

In heterogenen Datenbanken muß davon ausgegangen werden, daß die lokalen Datenbanksysteme jeweils über unabhängige lokale Transaktionsmanager (LTM) verfügen. Für lokale Transaktionen, in denen Datenbank-Events auftreten, die vorher oder nachher detektiert werden, müssen die Kopplungsmodi auf andere Weise definiert sein. Zu beachten ist, daß die Ausführung der Regel in einer von der triggernden Transaktion unabhängigen Transaktion erfolgen kann.

Ein Kopplungsmodus in einem aktiven föderierten System definiert den Ausführungszeitpunkt einer globalen Regel $t(\text{interpret}(E_G))$ relativ zum Zeitpunkt der Detektion eines globalen Events. Im Gegensatz zu einem homogenen System ist die Anwendung der Kopplungsmodi *immediate* und *detached parallel* kritisch, wenn gilt:

$$t(\text{raise}(E_G,E_L)) < t(\text{interpret}(E_G)) < t(\text{visible}(E_{LT},E_L)),$$

Damit würde die Transaktionssemantik einer Update-Propagation über Datenbanksystemgrenzen hinweg bei auftretenden Fehlern verletzt werden. Für die Behandlung dieses Problems wird prinzipiell folgende Lösung vorgeschlagen:

In den Fällen 1 und 2 können alle Arten von Events an den lokalen Schnittstellen detektiert werden, was auch Transaktions-Events einschließt. Auch wenn ein lokales Event vor Auftreten schon an das globale System signalisiert wurde, kann die Ausführung der Regel solange verzögert werden, bis ein lokales Commit-Ereignis signalisiert wird. Dieses wird auf einen Commit-Aufruf am Global Application Interface abgebildet, wobei die globale Commit-Behandlung das Feuern von Regeln einschließt, die durch Ereignisse getriggert wurden, die innerhalb der Transaktion stattfanden. Das entspricht der Semantik von *deferred*. Bei einer Event-Signalisierung vor lokalem Auftreten ist dann die Reihenfolge der Zeitpunkte wie folgt:

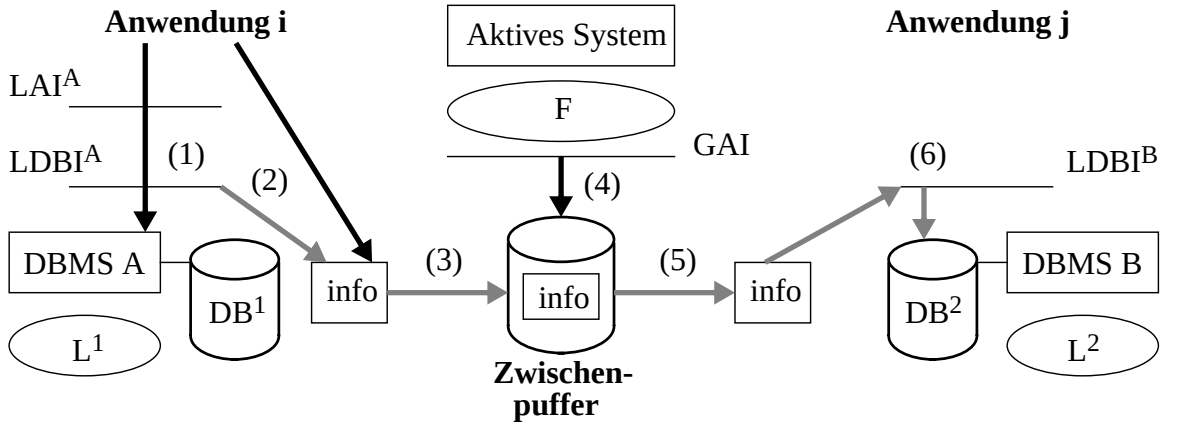
$$t(\text{detect}(E_L)) < t(\text{raise}(E_G,E_L)) < t(\text{detect}(E_G)) \ldots < t(\text{occur}(E_L)) \ldots$$
$$< t(\text{visible}(E_{LT},E_L)) < t(\text{raise}(E_{GT},E_{LT})) < t(\text{detect}(E_{GT})) < t(\text{visible}(E_{GT},E_G))$$
$$< t(\text{interpret}(E_G))$$

5.3.2 Indirekte Verarbeitung lokaler Events

5.3.2.1 Ablaufmodell

Abbildung 5.4 illustriert den Ablauf bei der indirekten Verarbeitung lokaler Events, die einen Puffer zur Zwischenspeicherung eventbezogener Informationen erfordert.

1. Detektion des lokalen Events: detect(occur(E_L))
   Eine bestimmte Aktion (nachfolgend triggernde Aktion genannt), die von einer Anwendung i initiiert wird, nimmt an einem lokalen Datenbestand Modifikationen vor. Damit wird ein lokales datenbankinternes Event ausgelöst, das bei seinem Auftreten detektiert wird.



Phasen der Eventverarbeitung:

(1) Detektion eines lokalen Events
(2) Generierung Änderungs-Info
(3) Anbieten der Änderungs-Info
(4) Start einer globalen Regel
(5) Annahme der Änderungs-Info
(6) Verarbeitung und Einbringung der Änderungs-Info

Event

Datenfluß

Abbildung 5.4: Indirekte Verarbeitung lokaler Events in einem Multidatenbanksystem

2. Generierung einer Änderungsmeldung
   Bei Ausführung der triggernden Aktion wird registriert, daß die geänderten Daten globale Relevanz für eine andere Anwendung j haben, der die aus diesen Modifikationen resultierenden Zustandsänderungen mitzuteilen sind. Deshalb wird in einer zusätzlichen lokalen Aktion eine entsprechende Änderungsmeldung generiert.

3. Anbieten der Änderungsinformationen
   Die Änderungsinformationen werden an einen imaginären Zwischenpuffer übergeben und damit global für Anwendung j sichtbar. Die aus der Durchführung der triggernden Aktion resultierende Modifikationsinformation kann einerseits sofort an den Zwischenpuffer übergeben werden (*direkt*) und steht somit unmittelbar externen Anwendungen zur Verfügung. Alternativ dazu kann die Modifikationsinformation erst bei einem bestimmten Ereignis auf lokaler Seite verfügbar gemacht werden (*indirekt*). Solche Ereignisse können lokale Transaktions-Events sein (z.B. das Commit der Transaktion, die die triggernde Aktion enthält).

4. Auslösen einer globalen Regel
   Globale ECA-Regeln können dafür verantwortlich sein, die Annahme bzw. Weiterverarbeitung der Änderungsinformationen zu steuern, wenn es sich um eine asynchrone Kommunikation zwischen triggernder und getriggerter Aktion handelt. Ausgelöst werden solche Regeln durch temporale Events. Informationen über die Kontrollabhängigkeiten zwischen den Anwendungen i und j bzw. Datenbanken DB^1 und DB^2 können im CA-Teil der Regel enthalten sein.

5. Annahme der Änderungsinformationen
   Die getriggerte Aktion entnimmt die Änderungsinformationen aus dem Zwischenpuffer. Entsprechend den Realisierungsformen für das Anbieten von Informationen gibt es auch zwei verschiedene Strategien bei der Annahme dieser Informationen: Bei der *synchronen* Annahme erfolgt das sofortige Auslesen der Information aus dem Zwischenpuffer und Einarbeitung in den lokalen Datenbestand durch die getriggerte Aktion. Bei einer *asynchronen* Verarbeitung erfolgt die Auslösung der getriggerten Aktion erst zu einem späteren Zeitpunkt, dabei können sich im Zwischenpuffer bereits Informationen angesammelt haben, die von mehreren triggernden Aktionen stammen.

6. Verarbeitung / Einbringen der Änderungsinformationen
   Im Falle einer asynchronen Verarbeitung werden die Änderungsinformationen gemäß einer Einbringstrategie in die Datenbank DB^2 eingearbeitet. Mögliche Einbringstrategien wurden bereits bei der Diskussion von Kontrollabhängigkeiten skizziert (vgl. Abschnitt 2.6 auf Seite 30).

Durch Kombination der Kommunikationsformen beim Propagieren bzw. Annehmen der Datenmodifikationen ergeben sich vier verschiedene Ablaufstrategien. Die loseste Form der Kopplung zwischen triggernder und getriggerter Aktion ist bei der indirekten asynchronen Verarbeitung anzutreffen und somit gut für autonome und heterogene Anwendungssysteme geeignet. Dabei kann die Propagierung als auch die Verarbeitung der Modifikationsinformationen durch ECA-Regeln anwendungsspezifisch gesteuert werden. Ein zusätzlicher Aufwand entsteht jedoch für die Verwaltung dieser Informationen auf dem Knoten, wo die Events anfallen.

5.3.2.2 Auswirkungen auf lokale Autonomie

Die einzelnen Phasen der indirekten Eventverarbeitung haben Einfluß auf die Autonomie der lokalen Systeme i bzw. j. Bezogen auf die Detektion lokaler Events wird die Verhaltensautonomie in geringerem Maße verletzt, da lokale Operationen unverändert ablaufen können. Wenn Informationen zu auftretenden lokalen Events (Art, Zeitpunkt, Parameter) benötigt werden, so müssen allerdings zusätzliche Anforderungen an das zu beobachtende System gestellt werden, die die Informationsbereitstellung betreffen. Tabelle 5.4 stellt noch einmal die einzelnen Verarbeitungsschritte den notwendigen Einschränkungen lokaler Verhaltensautonomie gegenüber.

Phase		Autonomiebeschränkung	Anwendung
1	Detektion des lokalen Events	-	i
2	Generierung der Änderungsmeldung	*B-Prescribe*(LDBI) oder *B-Prescribe*(LS)	i
3	Anbieten der Änderungsinformationen	*B-Prescribe*(LDBI) oder *B-Prescribe*(LS)	i
4	Auslösen einer globalen Regel	-	
5	Annahme der Änderungsinformationen	*B-Prescribe*(LDBI)	j
6	Verarbeitung / Einbringen der Änderungsinformation	*B-Prescribe*(LDBI)	j

Tabelle 5.4: Indirekte Eventverarbeitung vs. Autonomie

5.3.2.3 Semantische Aspekte

Fall 1.a:
Event-Signalisierung nach Auftreten des lokalen Datenbank-Events
• $t(\text{occur}(E_L)) < t(\text{detect}(E_L)) < t(\text{raise}(E'_L,E_L)) < t(\text{detect}(E'_L)) < t(\text{interpret}(E'_L))$

Fall 1.b:
Event-Signalisierung nach Auftreten des lokalen Datenbank-Events
• $t(\text{detect}(E_L)) < t(\text{occur}(E_L)) < t(\text{raise}(E'_L,E_L)) < t(\text{detect}(E'_L)) < t(\text{interpret}(E'_L))$

Fall 2:
Erkennen lokaler Events ohne Signalisierung
• $t(\text{occur}(E_L)) < t(\text{detect}(E_L)) < t(\text{interpret}(E_L))$

Im Fall 1 wird davon ausgegangen, daß das Datenbank-Event im LDBMS schon stattgefunden hat und dieses somit nach der lokalen Detektion nur noch an das globale System signalisiert zu werden braucht. Dabei wird ein korrespondierendes Event E'_L ausgelöst, das eine globale Regel feuert. Das Event E'_L zeigt dabei lediglich das (bereits stattgefundene) Auftreten des lokalen Events E_L an. Die Steuerung verbleibt hier beim jeweiligen lokalen System, das die vollständige Kontrolle über die triggernde Aktion behält.

Im Falle der Signalisierung eines Events erst nach dessen lokalem Auftreten muß ebenfalls die Bedingung $t(\text{visible}(E_{LT},E_L)) < t(\text{interpret}(E'_L))$ beachtet werden. Das setzt voraus, daß lokale Transaktionsereignisse detektiert und signalisiert werden können, was jedoch eine Modifikation des lokalen Transaktionsmanagers voraussetzt, wenn keine Ereignisdetektion an LAI oder LDBI stattfindet.

Im Fall 2 kann eine Detektion eines lokalen Events nur dadurch erfolgen, daß zu irgendeinem Zeitpunkt $t > t(\text{occur}(E_L))$ geprüft wird, ob das Ereignis stattgefunden hat, z.B. durch Testen eines Logs. Dieser Zeitpunkt t kann durch ein temporales Event vorgegeben sein.

5.4 Datenmodell für aktive Objekte in Multidatenbanken

5.4.1 Semantische Beziehungen in kanonischen Datenmodellen

In Datenmodellen, die als kanonische Modelle für föderierte Datenbanksysteme vorgeschlagen wurden, spielen semantische Abstraktionsbeziehungen zur Modellierung globaler Integritätsbedingungen eine große Rolle. Eine Vielzahl von Modellen unterstützt “nur” die semantischen Beziehungen, die z.B. in [PM88] diskutiert werden. Entsprechend der Ausdrucksmächtigkeit für Constraints lassen sich kanonische Datenmodelle einer der folgenden drei Gruppen zuordnen.

1. Grundkonzepte:
   Klassifikation, Vererbung (Spezialisierung / Generalisierung), Aggregation, Assoziation (Mengenbildung), Relationships
   OMG-Objektmodell [OMG91];
   ODMG-93: objektorientiert [Cat94];
   FUGUE: funktional [HZ88];

2. Erweiterte Ausdrucksmöglichkeiten für semantische Beziehungen (modellinhärent)
   Komplexitätsrestriktionen von Beziehungen, Insertion-/Deletion-Constraints, Mengensemantik von Spezialisierungen (partiell vs. total, disjunkt vs. überlappend)
   VODAK: *CategoryGeneralization*, *RoleGeneralization* [KDN90];
   BLOOM: Verfeinerung von Aggregation und Spezialisierung, siehe Abschnitt 5.4.2;
   STEP / EXPRESS: benutzerdefinierte Constraints [ISO92];
   SDM: Insertion-/Deletion-Constraints als Teil der Beziehungsdefinition [HL81];
   OSAM*: benutzerdefinierbare semantische Beziehungen [SKL89];

3. Definition von ECA-Regeln, Triggern u.ä. (explizit)
   OSAM*: Modellierung von Datenabhängigkeiten durch Regeln [SDS95];
   FROOM: funktionales OO Modell, erlaubt Definition von ECA-Regeln [MB90];

5.4.2 Modellierung modellinhärenter Bedingungen durch ECA-Regeln

Aufgrund seiner reichhaltigen Ausdrucksmöglichkeiten wurde das BLOOM-Modell einer näheren Betrachtung unterzogen: Das BLOOM-Modell (**B**arce**LO**na **O**bject **M**odel) ist ein objektorientiertes Modell, bestehend aus Objekten und Klassen [CSG92]. Folgende Abstraktionen sind vorgesehen:

- Klassifikation / Instanziierung: Objekte, Klassen, Metaklassen, Meta-Metaklassen
- Generalisierung / Spezialisierung:
  - Disjunkte Spezialisierung: Jedes Objekt der Superklasse gehört zu höchstens einer Subklasse.
  - Komplementäre Spezialisierung: Jedes Objekt der Superklasse gehört zu mindestens einer Subklasse.
  - Alternative Spezialisierung: Jedes Objekt gehört zu genau einer Subklasse.
  - Allgemeine Spezialisierung: Ohne Beschränkungen.
- Aggregation:
  - Simple Aggregation: Zusammenfassung von Attributen in einem Objekt.
  - Collection Aggregation: Zusammenfassung von Objekten derselben Klasse zu einem komplexen Objekt.
  - Composition Aggregation: Bildung eines Aggregates durch Komponentenobjekte verschiedener Klassen.

Diese modellinhärenten Constraints sollen nun in der Schreibweise von ECA-Regeln notiert werden. Dazu sei *T* der Objekttyp der Superklasse *c*, *T'* der Objekttyp einer Subklasse *c'* und c_i' eine beliebige Subklasse von *c*. Damit lassen sich ECA-Regeln zum Einfügen (*Insertion Rules*), IR, und zum Löschen (*Deletion Rules*), DR, formulieren. Eine Operation kann blokkiert (*reject* <operation>) oder propagiert (<operation>) werden, wenn die Bedingung, die die Art der Spezialisierungsbeziehung kennzeichnet, verletzt wird. Dies kann beim Einfügen oder Löschen geschehen. Eine Delete-Operation kann auch auf Objekte der Superklasse propagiert werden, solange die zugrundeliegende Bedingung nicht verletzt ist. Hierbei wird angenommen, daß die durch eine Operation manipulierten Objekte in Transitionsklassen gehalten werden: *inserted*, *deleted*, *old-updated* und *new-updated*.

- Disjunkte Spezialisierung:

IR:	ON insert *c*	IF inserted in any inst(c_i') ($c_i' \neq c'$)	DO reject insert
DR:	ON delete *c'*		DO delete *c*

- Komplementäre Spezialisierung:

IR:	ON insert *c*		DO insert c_i'
DR_1:	ON delete *c'*	IF deleted not exists in any inst($c_i' \neq c$)	DO delete *c*
DR_2:	ON delete *c'*	IF deleted not exists in any inst($c_i' \neq c$)	DO reject delete

- Alternative Spezialisierung:

IR_1:	ON insert *c*		DO insert c_i'
IR_2:	ON insert *c'*	IF inserted in any inst($c_i' \neq c$)	DO reject insert
DR_1:	ON delete *c'*		DO delete *c*
DR_2:	ON delete *c'*		DO reject delete

Die verschiedenartigen Spezialisierungsbeziehungen drücken modellinhärente Existenzabhängigkeiten zwischen Objekten aus Superklasse und Subklassen aus. Diese können durch ECA-Regeln beschrieben werden, die von einem föderierten System automatisch aus der Definition des Datenmodells abgeleitet werden müßten, d.h. sie sind für den Benutzer nicht zugänglich.

Modellinhärente Existenzabhängigkeiten zwischen Komponenten- und komplexen Objekten finden sich auch bei den Aggregationsbeziehungen Collection und Composition, die sich ebenso durch ECA-Regeln ausdrücken lassen (siehe hierzu [PDW+93]).

5.4.3 Modellierung expliziter Integritätsbedingungen durch ECA-Regeln

Trotz einer Vielzahl angebotener Abstraktionsbeziehungen im Datenmodell von BLOOM ist vorgesehen, explizite (d.h. benutzerdefinierte) Existenz- und Wertabhängigkeiten zu deklarieren.

In [CSG92] wird ein Ansatz beschrieben, der Existenzabhängigkeiten nach verschiedenen Kriterien klassifiziert und das jeweilige Verhalten in Metaklassen erfaßt. An einer Existenzabhängigkeit ist mindestens ein abhängiges Objekt beteiligt, der sogenannte *Dependent* (**d**). Dieser hängt von der Existenz eines anderen Objektes ab, das nachfolgend als *Dependor* (**D**) bezeichnet wird. Eine Existenzabhängigkeit heißt *strict*, wenn der Dependent niemals existieren kann, wenn der Dependor nicht existiert. Die abgeschwächte Form der Existenzabhängigkeit (*relaxed*) besagt, daß ein Dependent nicht erzeugt werden kann, solange der Dependor nicht existiert. Nachdem beide erzeugt wurden, ist die Existenz des Dependent aber unabhängig von der des Dependors und kann sogar nach Verschwinden des Dependors noch fortdauern (siehe Abbildung 5.5).

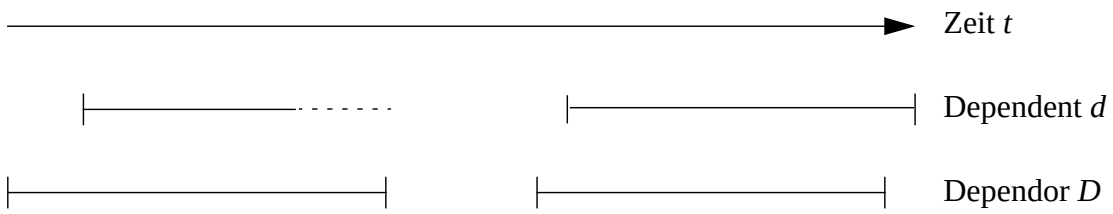


Abbildung 5.5: Existenzabhängigkeiten (strict vs. relaxed)

Eine Abhängigkeit ist *exclusive*, wenn der Dependent exklusiv von einer Instanz des Dependor-Objekttyps abhängt. Anderenfalls wird von einer *shared* Abhängigkeit gesprochen, bei der ein Dependent-Objekt von mehreren Dependors abhängen kann. Castellanos u.a. [CSG92] unterscheiden dabei weiterhin, ob bei einer Dependor-Menge mindestens eines oder alle Dependor-Objekte existieren müssen. Die beschriebene Semantik der Existenzabhängigkeiten läßt sich auch für komplexe Objekte (d.h. Aggregation) anwenden, z. B. auf das Konzept der *Composite Object References* in GOM [KM94].

Gerichtete Existenzabhängigkeiten

Am Beispiel der Deklaration exklusiver strikter Existenzabhängigkeiten (abgekürzt: ED) soll demonstriert werden, welche Möglichkeiten für die Spezifikation durch ECA-Regeln sich ergeben:

Eine gegebene ED soll nur für jene Mitglieder einer Dependent-Klasse gelten, die eine bestimmte Bedingung $\boldsymbol{C}$ erfüllen, somit enthält die Spezifikation optional eine Angabe von C. Um eine Korrespondenz zwischen Instanzen von d und D zu erfassen, muß eine Identifikationsfunktion $\boldsymbol{I}$ definiert sein, die von ihrer Semantik her der Wertkorrespondenzfunktion (vgl. Definition 2.14 auf Seite 24) entspricht. Bei globalen Integritätsbedingungen gehören die an einer ED beteiligten Klassen zu verschiedenen Datenbanken.

Die Operationen, die eine ED verletzen können, sind im einzelnen:

1. Einfügeoperation auf der Dependent-Klasse *d*: insert-*d*
2. Änderungen auf dem Dependent
   - auf Attributen, die in der Bedingung *C* enthalten sind: update-*C(d)*
   - auf Attributen, die in der Identifikationsfunktion *I* verwendet werden: update-*I(d)*
3. Löschoperation in der Dependor-Klasse: delete-*D*
4. Änderungen von Attributwerten in Dependor-Objekten, die in *I* genutzt werden: update-*I(D)*

Für jeden dieser Fälle sind mehrere Reaktionen möglich, sogenannte *Enforcement Policies*, die im Falle einer Konsistenzverletzung ausgelöst werden.

- insert-*d*-Effekt:

Wenn kein korrespondierender Dependor für das einzufügende Dependent-Objekt existiert, kann die Insert-Operation entweder propagiert oder verboten werden. D.h. ein korrespondierender Dependor wird eingefügt (*propagate*), oder die Operation wird blockiert (*block*).

- update-*C*(*d*)-Effekt:

Wenn der neue Wert des geänderten Attributs die Bedingung C erfüllt, wird das Objekt zu einem Dependent. Falls kein korrespondierender Dependor existiert, kann die auszulösende Reaktion entweder *insert* (Einfügen des korrespondierenden Dependors) oder *block* sein.

- update-*I*(*d*)-Effekt:

Wenn ein Attribut des Dependent, das in der Identifikationsfunktion *I* enthalten ist, geändert wird, so hat das Auswirkungen auf den korrespondierenden Dependor. Wenn dieser Dependor noch nicht existiert, so sind folgende Effekte möglich. Die Update-Operation wird auch auf dem alten korrespondierenden Dependor ausgeführt (*propagate*), ein neuer korrespondierender Dependor wird eingefügt (*insert*) oder die Operation verboten (*block*).

- delete-*D*-Effekt:

Wenn beim Löschen des Dependor noch Dependent-Objekte vorhanden sind, dann wird die Delete-Operation entweder dort ausgeführt (*propagate*) oder blockiert (*block*).

- update-*I*(*D*)-Effekt:

Eine Änderung in einem Attribut des Dependor, das Teil der Identifikationsfunktion *I* ist, führt zu einer Verletzung, wenn es kein Dependor-Objekt mehr gibt für die Dependents, die bisher mit dem alten Wert korrespondierten. Dabei sind folgende Reaktionen möglich: *propagate* (d.h. Ändern auch in den Dependents), *delete* (Löschen der Dependents) oder *block*.

Tabelle 5.5 enthält eine Zusammenfassung aller möglichen Reaktionen.

Aktion / Effekt		insert-*d*	update-*C*(*d*)	update-*I*(*d*)	delete-*D*	update-*I*(*D*)
propagate	**p**	X	-	X	X	X
insert	**i**	-	X	X	-	-
delete	**d**	-	-	-	-	X
block	**b**	X	X	X	X	X

('X' = anwendbar, '-' = nicht anwendbar)

Tabelle 5.5: Effekte bei konsistenzverletzenden Aktionen

Beispiel 5.1: (ED-Spezifikation)
Zugrunde liegt wiederum Beispiel 2.3 auf Seite 39 aus [CW93]:

Nodes [region = 'Milano' $\wedge$ function = 'plant']
strict depends_on Plants
with {p,i,p,p,b}

Damit ist folgendes Verhalten spezifiziert:

p Insert in Nodes wird auf Plants propagiert.
i Ein Update des Attributes region eines Kraftwerks auf den Wert 'Milano' in Nodes hat Auswirkungen auf die Bedingung, die die Dependent-Objekte beschreibt. In diesem Fall wird ein neues Kraftwerk in Plants eingefügt.
p Ein Update eines Attributes eines Kraftwerks aus der Region Milano in Nodes, das in der Identifikationsfunktion enthalten ist, wird propagiert.
p Eine Delete in Plants wird auf Nodes propagiert.
b Ein Update auf einem Kraftwerksattribut (in Plants), das in der Identifikationsfunktion enthalten ist, wird blockiert.

Das Verhalten läßt sich in gleicher Weise auf ECA-Regeln abbilden, wie in Abschnitt 5.4.2 gezeigt. Weitere Beispiele für die Umsetzung der Policies, auch am Beispiel von ungerichteten Existenzabhängigkeiten in einem Metaklassenansatz, beschreiben Castellanos u.a. in [CKSG94]. Wertabhängigkeiten lassen sich ebenso durch ECA-Regeln ausdrücken, wobei die typischen Verhaltensmuster bei Konsistenzverletzungen die Propagierung der Wertänderung (*propagate*) oder eine Abweisung der Änderung (*block*) sind.

5.5 ODMG-93 als globales Objektdatenmodell

Wahl des Objektdatenmodells

Die Entscheidung für ein Objektmodell basiert auf der Prämisse, kein neues Objektmodell zu entwerfen, sondern von einem bereits vorhandenen auszugehen, das "nur" die objektorientierten Grundkonzepte beinhaltet, und es durch ECA-Regeln zu erweitern. Die Wahl des globalen Datenmodells fiel hierbei auf ODMG-93 (kurz: ODMG) bzw. C^{++} aus folgenden Gründen:

- es ist Quasi-Standard für objektorientierte Datenbanken;
- es enthält alle notwendigen Eigenschaften objektorientierter Datenmodelle: Einkapselung, Vererbung, Beziehungen, Aggregation und Assoziation, Polymorphismus;
- die semantische Ausdrucksmächtigkeit ist durch Anreicherung mit ECA-Regeln erweiterbar;
- es ist erweiterbar im Hinblick auf unterschiedliche Granularität der zu modellierenden lokalen Komponenten;
- es bietet die Grundlage für die mögliche Einbeziehung von Middleware-Produkten (z.B. CORBA).

Die ODMG (*Object Database Management Group*) entstand 1991 als ein Zusammenschluß kommerzieller Anbieter objektorientierter Datenbanksysteme mit dem Ziel, einen Standard für objektorientierte Datenbanken zu definieren. Die ODMG ist ein Teil der OMG (*Object Management Group*), deren Ziel die Entwicklung eines CORBA-Standards ist (*Common Object Request Broker Architecture*). Als Ergebnisse liegen die Definition eines Objektmodells, die Datenbanksprachen ODL (*Object Definition Language*) und OQL (*Object Query Language*), Spracheinbettungen (*Bindings*) für C^{++} und Smalltalk sowie Bezüge zu OMG/CORBA vor

[Cat94]. Der nachfolgende Abschnitt enthält eine kurze Vorstellung des ODMG-Objektmodells.

5.5.1 Die Bestandteile des Objektmodells

Das ODMG-Objektmodell vereint in sich Eigenschaften, die als Anforderungen an objektorientierte Datenbanksysteme formuliert wurden. Dazu zählen:

- Basisprimitiv ist das *Objekt* (auch als Instanz bezeichnet). Jedes Objekt ist assoziiert mit einem eindeutigen systemdefinierten Identifikator (Surrogat) und optional durch benutzerdefinierte Schlüssel (*Key*) und Objektnamen zur Erleichterung des Zugriffs. Im Objektmodell gibt es zwei Arten von Objekten: *Immutable Objects* (Literale) werden durch ihren Wert identifiziert, der nicht geändert werden kann. Ein *mutable Object* besitzt ein Surrogat und einen Zustand. Objekte können atomar oder strukturiert sein.
- Objekte können in *Typen* kategorisiert werden. Ein Typ besitzt ein Interface, durch das alle Objekte dieses Typs nach außen hin charakterisiert werden. Dazu zählen die Attribute, auf deren Werte lesend oder schreibend zugegriffen werden kann, Beziehungen (*Relationships*) zu Objekten anderer Typen und Operationen, die auf den Instanzen aufgerufen werden können.
- Zu einem Typ kann es eine oder mehrere *Implementationen* geben, die durch eine *Klasse* (z.B. in C^{++}) beschrieben werden. Zur Implementation gehören Datenstrukturen für die physische Repräsentation der Objekte sowie Methoden, die darauf operieren und das im Interface spezifizierte Verhalten realisieren. Für einen Typ kann eine Extension definiert werden, in der alle Objekte dieses Typs verwaltet werden und somit für mengenorientierte Anfragen zur Verfügung stehen.
- Typen können in einer *Subtype* / *Supertype*-Hierarchie angeordnet werden. Alle Attribute, Beziehungen und Operationen können von einem Typ auf die Subtypen vererbt werden. Der Subtyp kann weitere Eigenschaften oder Operationen hinzufügen bzw. ererbte überschreiben.
- *Attribute* enthalten Werte eines Objekts. Sie können einen Basis-Datentyp haben oder einen komplexen Typ, der durch einen Typkonstruktor gebildet wird {*Set*, *Bag*, *List*, *Array*, *Structure*}.
- *Methoden* definieren das Verhalten der Objekte. Sie werden definiert durch eine Signatur, die aus Operationsname, Namen und Typen der Argumente und Rückgabewerte besteht. Es ist eine Reihe von Standard-Operationen zum Vergleichen, Zuweisen und Kopieren von Objekten und Werten vordefiniert.
- Beziehungen lassen sich als binäre 1:1, 1:m und m:n Relationships auf Typebene darstellen durch Anwendung von mengenwertigen Typkonstruktoren. Diese Beziehungen sind ungerichtet und und lassen sich über *inverse* Relationships in beiden Richtungen definieren, auch als relationale Integrität bezeichnet (vgl.. Abschnitt 2.3.2).

Abbildung 5.6 gibt eine vereinfachte Darstellung des Metamodells von ODMG-93 in Rumbaugh-Notation [LAC+93].

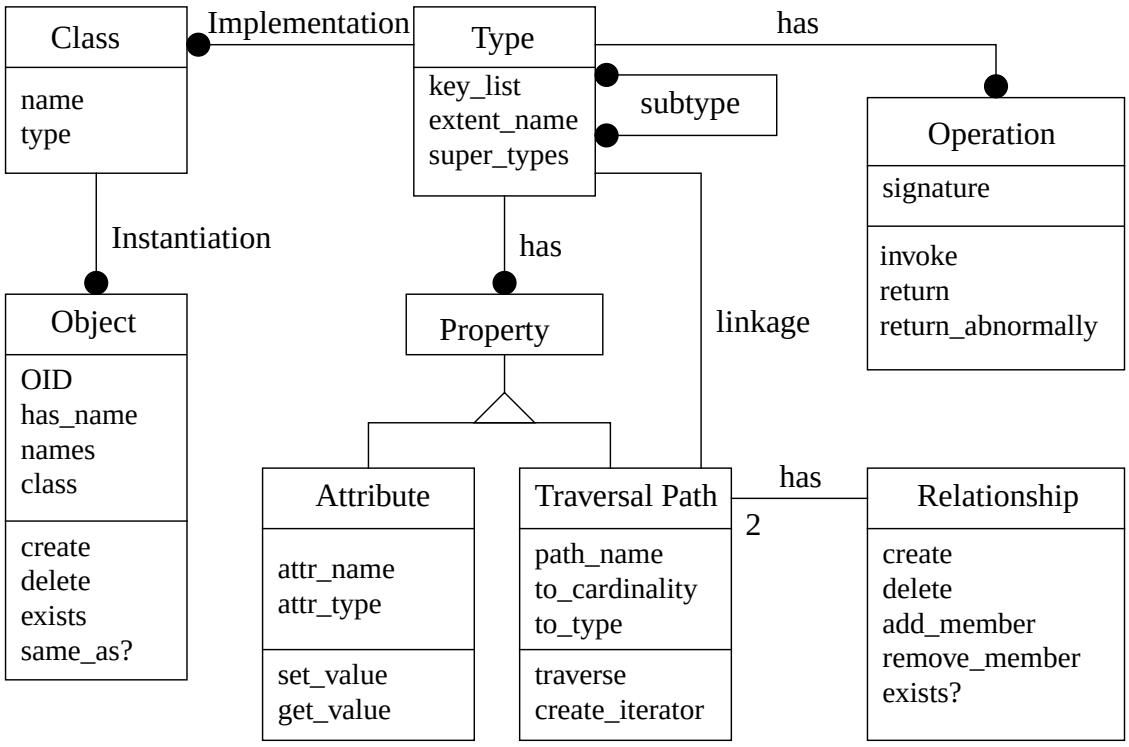


Abbildung 5.6: Das ODMG-93-Metamodell

5.5.2 Die Beziehung ODMG-Modell / C++

Wie bereits erwähnt, ist die Spracheinbettung in C++ Bestandteil der Arbeit am ODMG-Standard. Alle Bestandteile des Objektmodells lassen sich durch C++-Konstrukte ausdrücken.

Klassen und Objekte. Der `struct`-Typ wird in C++ zu einer Klassendefinition verallgemeinert, die dem Objekttyp in ODMG entspricht.

```
class Klassenname
  {
    private:  /* privateAttribute und Methoden */
    protected:/* geschützte Attribute und Methoden */
    public:   /* öffentliche Attribute und Methoden */
  };
```

Attribute und Methoden werden *Member* (Elemente einer Klasse) genannt. Methoden und Attribute können als `private` definiert werden, dann können sie nur von anderen Methoden derselben Klasse verwendet werden. Als `public` definierte Attribute und Methoden sind von außen frei zugänglich. Die Schnittstelle einer Klasse und ihre Implementierung werden separat in einer Header- und in einer Implementierungsdatei abgelegt. Das Erzeugen von Objekten erfolgt durch Aufruf des `new`-Operators, die Initialisierung erfolgt dabei durch einen Konstruktor. Dabei kann ein Objekt persistent erzeugt werden, indem der `new`-Operator überladen wird, das ist im Prinzip in einigen objektorientierten Datenbanksystemen so gelöst, z.B. ObjectStore.

C++ bietet die Möglichkeit zur Definition sogenannter `static` Variablen, womit alle Objekte einer Klasse die gleiche Information besitzen (entspricht den Typeigenschaften in ODMG-93).

Vererbung läßt sich in C++ durch eine Klassenhierarchie realisieren.

```
class Subklasse: [ private | public ] Superklasse
  {
  ...
  };
```

Die Unterklasse heißt abgeleitete Klasse, die Oberklasse Basisklasse. Die Eigenschaft `private` wird zusammen mit den Attributen vererbt, bei Definition von `public` bleiben die `public`-Elemente der Basisklasse auch in der abgeleiteten Klasse öffentlich. Durch dieses Konzept lassen sich auch Sichten realisieren, wie sie von Datenbanken her bekannt sind.

Virtuelle Klassen in C++ (auch als abstrakte Klassen bezeichnet) sind Klassen, die keine Instanzen besitzen; die Funktionen einer solchen Klasse sind im allgemeinen auch virtuell, d.h. es existiert noch keine Implementierung, sondern diese wird erst in den Subklassen eingeführt.

Zur Bildung komplexer Datentypen und damit komplexer Objekte sind generische Klassen (in C++ *Templates*) unerläßlich. Als Parameter können Standard-Datentypen und Klassennamen angegeben werden.

```
template <Typname>  class Klassenname
  {
  ...
  };
```

Wollen wir z.B. eine Menge von Werten eines beliebigen Typs in einer C++-Klasse definieren, so müssen wir eine Klasse derart definieren:

```
template <Typname> class Set { ... }
```

Daraus können spezielle Mengen definiert werden, z.B.

```
Set <Int> S1;
Set <Person> S2;
```

Die in ODMG-93 eingeführten Typkonstruktoren lassen sich also auf diese Weise zusammen mit den Methoden in C++ realisieren. Eine Reihe von C++-Klassenbibliotheken unterstützt dies bereits.

Objekte können einander über Referenzen oder sogenannte *Smart Pointer* referenzieren. Dafür kann man nach [Cat94] eine spezielle `Ref`-Klasse einführen, die die Integrität bei der Adressierung persistenter Objekte kontrolliert. Durch Überladen ist der Gebrauch der Referenzen analog dem von Pointern in C++.

Der *Traversal Path* zur Kontrolle von Integritätsbedingungen über inverse Beziehungen ist durch Definition von Funktionen in C++ möglich, wobei zwischen 1:1- und 1:n-Beziehungen unterschieden werden muß.

Tabelle 5.6 gibt noch einmal eine Zusammenfassung über die Beziehung der Konzepte von C++ und ODMG-93.

Semantisches Abstraktionskonzept	ODMG-93	C++
Klassenbegriff	`class`	`struct` / `class`
Einkapselung	`public`, `private`, `protected`	`public`, `private`, `protected`

Tabelle 5.6: Verhältnis von C++ und ODMG

Semantisches Abstraktionskonzept	ODMG-93	C++
Einkapselung	Interface und Implementation	Header- und Implementationsdatei
Generalisierung / Spezialisierung	Super -und Subtypen	Basisklassen und abgeleitete Klassen
Aggregation	`Structure`-Typkonstruktor	`struct`
Assoziation	`Collection`-Typkonstruktoren (`set`, `list`, `bag`, `array`)	Template-Klassen, z.B. `set<T>`
Relationships	Klasse `Ref`	Object Pointer

Tabelle 5.6: Verhältnis von C++ und ODMG (Forts.)

Kapitel 6

Umsetzung

Nach der Diskussion der Grundlagen von Konsistenz, aktiven und Multidatenbanken wollen wir in diesem Kapitel beschreiben, welche Konzepte für die Entwicklung eines aktiven Vermittlersystems zur globalen Konsistenzkontrolle notwendig sind.

Zunächst soll der Begriff des Vermittlersystems definiert und in der von uns verwendeten Funktion gegenüber anderen abgegrenzt werden. Die Vermittlerfunktion Konsistenzsicherung umfaßt die Kontrolle globaler Integritätsbedingungen als auch die Konsistenzwahrung zwischen Schemata (Struktur- und Verhaltensabhängigkeiten). Dabei sollen aktive Mechanismen zum Einsatz kommen, so wie sie in den vorangegangenen Kapiteln beschrieben wurden. Die gewählte Testumgebung umfaßt lokale relationale DBMS als etablierte Plattform zahlreicher Applikationsssyteme.

In Abschnitt 6.2 werden die Probleme benannt, die konzeptionell bzw. bei der Implementierung eines Prototypsystems zu lösen sind. Die nachfolgenden Abschnitte dieses Kapitels gehen dann im einzelnen auf diese Probleme ein.

Zunächst beschäftigen wir uns mit der Frage, wie autonome Systeme als Teilnehmer eines Verbundes, der durch einen Vermittler kontrolliert wird, aktiv gemacht werden können. Hierbei betrachten wir insbesondere die Fähigkeit zur Detektion und Weitergabe von lokal auftretenden Events. Wir untersuchen verschiedene in Frage kommende Detektionsmechanismen (Abschnitt 6.3). Neben der Detektion wird auch der Aspekt der Protokollierung diskutiert, wenn Events nicht sofort verarbeitet werden sollen. Dabei werden die wichtigsten zu protokollierenden Parameter skizziert (Abschnitt 6.4). Die erforderliche Signalisierung von Ereignissen an ein Vermittlersystem erfordert die Benutzung von Middleware-Produkten, eine Diskussion darüber erfolgt in Abschnitt 6.5.

Abschnitt 6.6 behandelt die Probleme bei der Integration relationaler Datenbanken mit einem objektorientierten System (in unserem Fall das zu entwickelnde Vermittlersystem). Die Formulierung globaler ECA-Regeln muß mit den Mitteln der globalen Zugriffsschnittstelle erfolgen, hierbei handelt es sich um Methoden auf Stellvertreter-Objekten im Objektraum des Vermittlers. Die Abbildungsproblematik zwischen objektorientiertem und relationalem Modell umfaßt Struktur und Verhalten der zugrundeliegenden Systeme, so z.B. die bidirektionale Umwandlung von C^{++}-Methodenaufrufen in SQL-Anweisungen. Es werden eine Reihe von Werkzeugen zum objektorientierten Zugriff auf relationale Datenbanken vorgestellt, die eine Abbildung zwischen beiden Datenmodellen unterstützen. Dazu zählt das System Persistence, das als Plattform des Vermittlersystems dient.

Die Anreicherung des Vermittlersystems um die Funktion der globalen Integritätssicherung erfordert, daß ECA-Regeln bzw. eine Regelausführungskomponente (*Rule Engine*) integriert werden. Dabei ist die Auswahl zu treffen zwischen bereits verfügbaren aktiven OODBMS, Produktionsregelsystemen oder einer eigenen Entwicklung (Abschnitt 6.7).

6.1 Ein aktives Vermittlersystem

6.1.1 Einführung in Vermittler

Viele Applikationen erfordern einen integrierten Zugriff auf heterogenene Informationen, die in Systemen mit heterogenen Datenmodellen und Zugriffsmechanismen abgelegt sein können. In [Wie92] werden eine Reihe von Aufgaben unter dem Begriff *Mediation* zusammengefaßt, die sich grob in zwei Kategorien einteilen lassen:

- Bereitstellung von Informationen für globale Applikationen durch Abstraktionsmechanismen wie Selektion, Aggregation, Transformation [Wie92].
- Verwaltung eines globalen Data Dictionary (vgl. Seite 35), um semantische Differenzen ausgleichen zu können und geeignete Übersetzungen und Abbildungen zwischen heterogenen Datenquellen bereitstellen zu können [Mad95].

Vermittler machen somit Endbenutzeranwendungen von darunterliegenden Datenressourcen unabhängig und helfen, die Lücke zwischen verfügbaren Daten und der daraus ableitbaren Nutzinformation zu verringern.

Definition 6.1 (Vermittler)
Ein Vermittler (*Mediator*) ist ein Software-Modul, der kodiertes Wissen über Mengen von Daten verwendet, um Informationen für eine höhere Ebene von Applikationen bereitzustellen.

In [Wie96] ist eine ganze Reihe von Vermittlungsdiensten (*Mediation Services*) aufgelistet, die sehr häufig auf Methoden der Künstlichen Intelligenz beruhen. Vermittler sind in ihrer Funktion und in ihrer Anwendungsdomäne spezialisiert. Für die nachfolgenden Ausführungen werden Vermittler in ihrer Funktion als Konsistenzkontroll-Werkzeug betrachtet. Die Konsistenz bezieht sich dabei zum einen auf globale Integritätsbedingungen als auch auf die Konsistenz der Wissensbasis des Vermittlers gegenüber den Basisdaten. Abbildung 6.1 zeigt eine allgemeine Schichtenarchitektur, bei der die Ebene der Vermittler als eine Softwareschicht zwischen zugrundeliegenden Basisdaten und den Applikationen angesiedelt ist.

Vermittlermodule sind dann am effektivsten einsetzbar, wenn sie eine Vielzahl von verschiedenen Anfragen (aus unterschiedlichen Applikationsprogrammen) bedienen können, die auch komponiert werden können. Entsteht jedoch ein neuer Informationsbedarf, der durch die Kombination der bisherigen Vermittler nicht abgedeckt ist, so sind neue Vermittlerdienste zu erstellen.

6.1.2 Ein Vermittlersystem zur Konsistenzkontrolle

Ein Vermittlersystem zur Konsistenzkontrolle besteht demnach aus Vermittlerprozessen, die entweder eine Menge globaler Integritätsbedingungen oder die Konsistenz der Schemata überwachen.

Integritätsbedingungen (Existenz- und Wertabhängigkeiten) werden durch ECA-Regeln auf einem globalen Schema ausgedrückt, das eine Sicht auf mehrere lokale Schemata darstellt. Zur

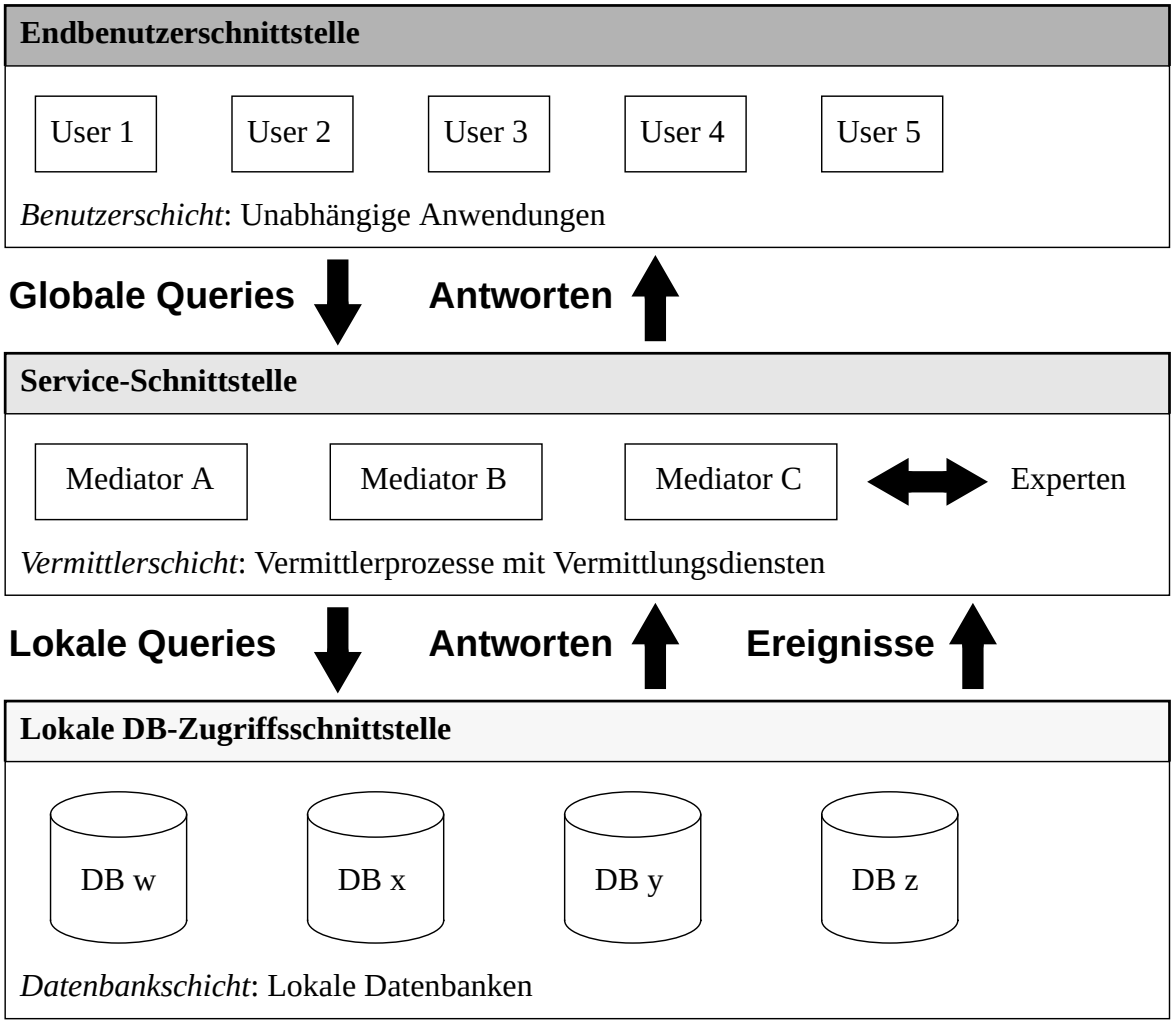


Abbildung 6.1: Schichtenarchitektur für Vermittlerprozesse

Realisierung abgeschwächter Konsistenzbedingungen lassen sich ebenfalls ECA-Regeln definieren. Die ECA-Regeln werden typischerweise durch Datenbank- oder temporale Events getriggert. Die dritte durch uns definierte Dimension von Konsistenz, die der Kontrollabhängigkeiten, wird als Bestandteil einer ECA-Regel oder durch Abgleichstrategien spezifiziert. Letzteres ist erforderlich, wenn es infolge asynchroner Verarbeitung zu divergierenden Werten kommt, die zu bestimmten Zeitpunkten wieder zusammengeführt werden müssen. Da die Integritätsbedingungen auf einem Datenbankzustand definiert sind, sind die auszuführenden Aktionen (bzw. auch die Ereignisse) typischerweise DML-Anweisungen; deswegen werden Vermittler dieser Kategorie als *DML-Vermittler* bezeichnet.

Die Konsistenzkontrolle zwischen lokalen Schemata und globalem Schema bei lokalen Schemaveränderungen muß ebenfalls durch einen aktiven, regelbasierten Mechanismus erfolgen, mit dem Unterschied, daß eine laufzeitdynamische Evolution des globalen Schemas nicht möglich ist. Regeln werden dabei auf der Ebene des Metaschemas notiert. Da die konsistenzverletzenden Ereignisse durch DDL-Anweisungen ausgelöst werden, werden diese Vermittler als *DDL-Vermittler* bezeichnet. Abbildung 6.2 veranschaulicht das dem Vermittlersystem zugrundeliegende Modell in OMT-Notation.

Für das zu entwickelnde Vermittlersystem lassen sich damit im wesentlichen drei Dienste (*Services*) formulieren:

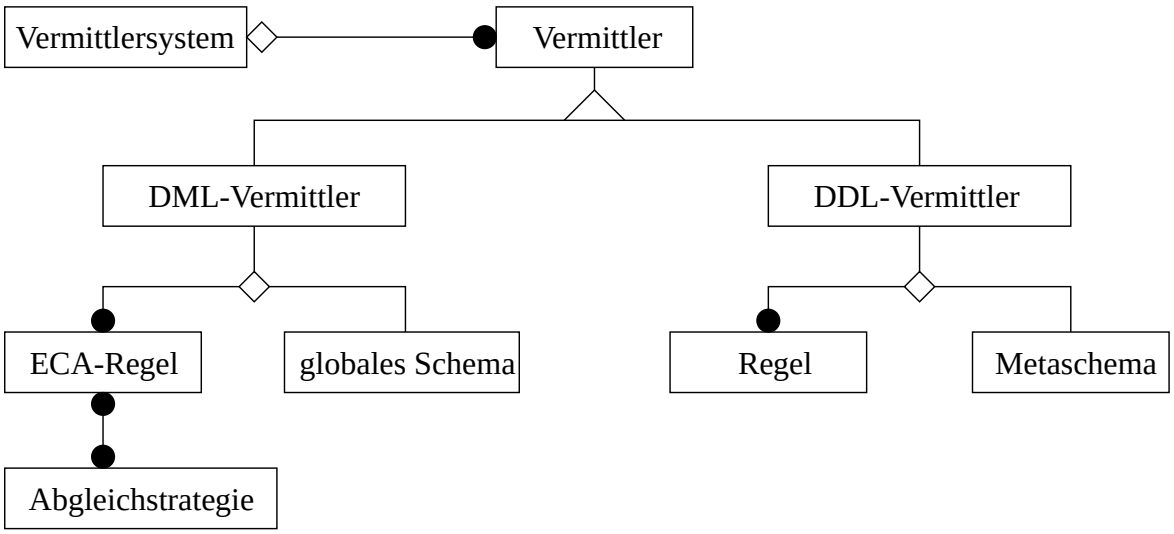


Abbildung 6.2: Modell des aktiven Vermittlersystems

- *Integrations-Service*: Ausführung von globalen Queries,
- *Integritäts-Service*: Kontrolle globaler Existenz- und Wertabhängigkeiten
  - bei Verletzung durch lokale Aktionen
  - bei Verletzung durch globale Aktionen
  - temporale Bedingungen (d.h. durch Zeitereignisse getriggerte Auswertung),
- *Schemakonsistenz-Service*: Kontrolle der Konsistenz zwischen globalem Schema und den zugrundeliegenden lokalen Schemata.

Der Integrationsservice, d.h. die Ausführung globaler Anfragen und Verarbeitung der Ergebnisse, entspricht der Funktionalität eines Multidatenbanksystems. Er bildet den Basismechanismus für den Integritätssicherungs-Service, der auch als eine Erweiterung des Integrationsservice aufgefaßt werden kann. Somit können Funktionen, die an der Service-Schnittstelle des Vermittlers (Global Application Interface) bereitstehen, auch zur Realisierung des Integritäts-Service verwendet werden.

Das zu realisierende Vermittlersystem soll die Konsistenzkontrolle zwischen mehreren relationalen Datenbanken übernehmen. In der Testumgebung (siehe Abbildung 6.3) werden als lokale DBMS Sybase System 11 und Informix Version 7 eingesetzt. Als globales Modell im Vermittler dient ein Objektmodell, um die nötige Offenheit bei Einbeziehung anderer Datenspeicher zu garantieren.

Die Verarbeitung lokaler Ereignisse muß danach unterschieden werden, ob sie Auswirkungen auf Instanzen haben oder sogar das Schema verändern. Dementsprechend werden DML- und DDL-Vermittler getrennt voneinander behandelt.

6.2 Zu lösende Probleme

Zugrunde liegt das in Kapitel 5 eingeführte Regel- und Ausführungsmodell für aktive Objekte in heterogenen Systemen. Hierbei wurde eine Reihe von Problemen identifiziert, wobei zu untersuchen ist, inwieweit auf bereits verfügbare Lösungen zurückgegriffen werden kann bzw. wo eigene Ansätze entwickelt werden müssen.

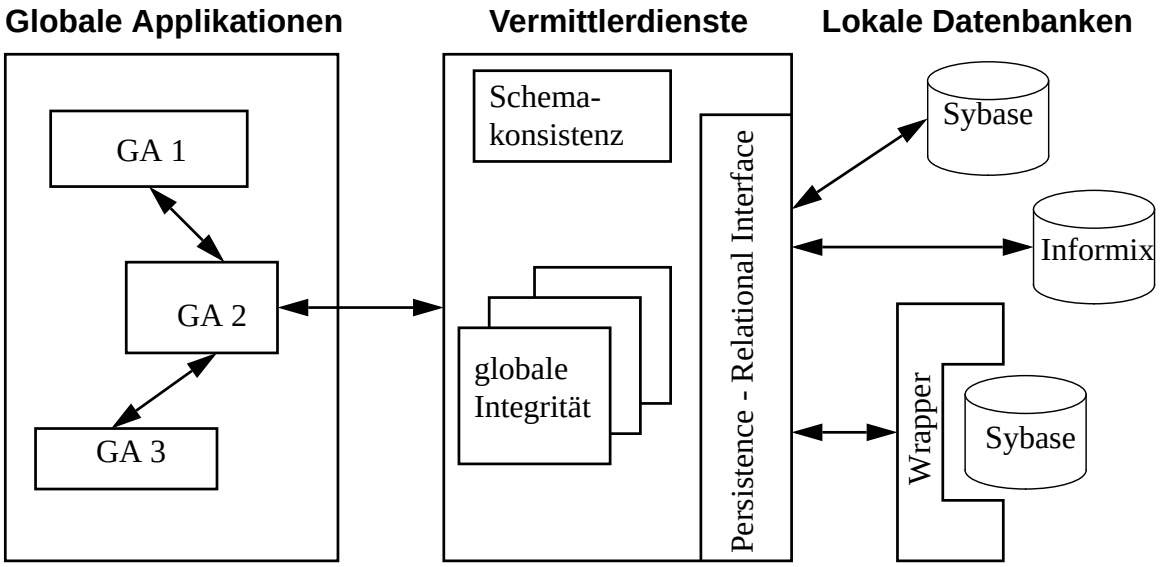


Abbildung 6.3: Aufbau der Testumgebung des Vermittlersystems

- **Detektion von lokalen Events in relationalen Systemen**

Zunächst einmal muß geprüft werden, welche Ereignisse in relationalen Datenbanksystemen Konsistenzverletzungen zur Folge haben können. Anschließend sind Lösungen zu finden, wie Events detektiert werden können unter größtmöglicher Wahrung der Autonomie auf verschiedenen Ebenen (Applikation, Datenbankschema).

- **Protokollierung von lokalen Events in relationalen Systemen**

Bei einer asynchronen Verarbeitung, wie sie in lose gekoppelten autonomen Systemen gegeben ist, muß das Problem gelöst werden, auf welche Weise Ereignishistorien geführt werden bzw. inwieweit auch Zustandshistorien relevant sind. Hierbei wird der Ansatz verfolgt, Detektion mit Protokollierung zu verbinden. In Abhängigkeit von der geforderten Konsistenz (z.B. Propagierung “alter” Änderungen) bzw. für die Anwendung von Abgleichalgorithmen bedeutet das einen zusätzlichen Overhead für das lokale System und damit in gewissem Sinne eine Einschränkung der Autonomie, da weitere Resourcen benötigt werden.

- **Signalisierung lokaler Events**

Die gewählte Kommunikationsform zwischen lokalem und globalem System beeinflußt die Wahl eines Signalisierungsprotokolls. Dieses hängt ab vom Grad der geforderten Konsistenz. Grundforderung dabei ist, daß netzwerkweit eine rechnerübergreifende Kommunikation zwischen allen Komponenten möglich ist und die lokalen Systeme entsprechende Betriebssystemprotokolle unterstützen.

- **Objektorientierte Integration relationaler Datenbanken**

Da die Beispielumgebung aus relationalen Datenbanksystemen besteht, das globale Datenmodell jedoch objektorientiert ist, entsteht das Abbildungsproblem relational / objektorientiert. Objektorientierte und relationale Zugriffsschnittstelle können als zwei unterschiedliche Eventdetektionsebenen betrachtet werden. Die Bestimmung korrespondierender Events (d.h. Methodenaufruf $\leftrightarrow$ SQL-Anweisung) setzt voraus, daß auch eine Abbildung vom Relationenschema auf das OO Schema existiert. Erforderlich ist somit eine Abbildung der Strukturen und Operationen in beiden Richtungen.

- **Integration der Regelverarbeitung / Detektion globaler Events**

Die aktive Komponente muß als Bestandteil des globalen Systems Regeln verwalten und bei Detektion globaler Events diese feuern. Dabei ist über die Auswahl einer *Rule Engine* zu entscheiden, die, eingebettet in das ODMG/C^{++}-Datenmodell, mit dem föderierten System interagiert. Überprüfung von Bedingungen und Ausführung von Aktionen über die objektorientierte Schnittstelle auf heterogenen relationalen Datenbanken setzt eine Kooperationsbereitschaft der lokalen Komponenten voraus. Zu beachten ist die größere Ausdrucksmächtigkeit von Integritätsregeln im Vergleich zur Wahrung der Schemakonsistenz.

6.3 Detektion lokaler Events in relationalen Systemen

6.3.1 Arten von Events

Wie bei der Beschreibung des Regel- und Ausführungsmodells aktiver Objekte bereits dargestellt, ist eine der wesentlichen Voraussetzungen für die Schaffung eines aktiven Multidatenbanksystems die Möglichkeit, lokale Ereignisse rechtzeitig zu detektieren und diese an das globale System zu signalisieren. Dabei soll die lokale Autonomie so wenig wie möglich beeinträchtigt werden.

Zur Betrachtung der für die globale Konsistenz kritischen Kommandos genügt im wesentlichen die SQL-Sprachschnittstelle. Darüber hinaus existieren Zugänge zum Datenbanksystem über Bibliotheken, deren Funktionen von den Clients aufgerufen werden können, so z.B. zum Einstellen von Optionen des Datenbankservers oder zur Verwaltung der Cursorbenutzung. Von besonderem Interesse ist die Aufnahme einer Verbindung vom Client zum Server sowie die Auflösung dieser Verbindung. Tabelle 6.1 enthält eine Übersicht über die zu detektierenden Datenbankereignisse an der SQL-Schnittstelle eines lokalen relationalen DBMS, die sich auf die globale Konsistenz auswirken können.

Datenbankereignis	Auswirkung auf die globale Konsistenz
DML-Befehle:	
INSERT, DELETE, UPDATE	Alle Arten von Existenz- und Wertabhängigkeiten
Transaktionskommandos: BEGIN, COMMIT, ROLLBACK	Kontrolle der Sichtbarkeit lokaler Änderungen für den globalen Benutzer
Queries: SELECT	Zugriff auf inkonsistente Daten möglich
Stored procedures	Auslösen anderer benutzerdefinierter Datenbankereignisse
DDL-Befehle:	
Schemaänderungen (Struktur)	Verletzung von Strukturabhängigkeiten von anderen Datenbanken; Inkonsistenz zum globalen Schema
Schemaänderungen (Constraints)	Interaktion mit globalen Constraints

Tabelle 6.1: Auswirkungen von SQL-Datenbankereignissen auf globale Konsistenz

Zu den eigentlichen datenmanipulierenden Aktionen gehören die SQL-Befehle DELETE, INSERT und UPDATE. Da auch die lokale Transaktionssemantik berücksichtigt werden muß, ist auch eine Detektion der Befehle BEGIN, COMMIT und ROLLBACK nötig.

Für die Kontrolle des Zugriffs auf die lokalen Daten ist ebenfalls SELECT von Interesse. Durch lokale Änderungen sind temporär globale Inkonsistenzen möglich, die aber erst dann

bedeutsam werden können, wenn Benutzer Zugriff darauf erhalten. Darauf basiert auch die Idee der *Lazy Evaluation* [SHP88], was aber die Kontrolle der SELECT-Kommandos voraussetzt.

Neben den "klassischen" DML-Befehlen können weitere Kommandos eine Veränderung der Datenbank oder auch nur einen Zugriff darauf auslösen, was gleichfalls einer Kontrolle unterliegen muß. Zu diesen gehören Aufrufe von vorübersetzten SQL-Prozeduren (*Stored Procedures*). Diese Prozeduren werden bei ihrer Definition in ausführbaren Code übersetzt und später mit ihrem Namen (eventuell mit Parametern) aufgerufen und vom Datenbankserver ausgeführt.

DDL-Kommandos, die das lokale Schema verändern, haben zur Folge, daß dieses mit dem globalen Schema (in dem die Integritätsbedingungen definiert sind) nicht mehr konsistent ist. D.h. die Struktur der Proxy-Objekte weicht von der der lokalen Basisobjekte ab (Verletzung einer Strukturabhängigkeit). Eine andere mögliche Schemamodifikation betrifft die Veränderung von lokal definierten Constraints (z.B. Fremdschlüsselbedingungen, CHECK-Klausel). Dabei können sich Interaktionen zu globalen Constraints ergeben, wenn die lokalen Bedingungen diese überlagern oder wenn lokale Aktionen getriggert werden, die selbst wiederum eine Auswirkung auf globale Constraints haben (zum Verhältnis lokaler und globaler Constraints siehe Abschnitt 2.6.2 auf Seite 33).

6.3.2 Lokale Detektionsmechanismen

In [KLB96a] werden verschiedene Mechanismen skizziert, die für die Detektion von Datenbank-Events genutzt werden können. Darunter sind Wrapping-Ansätze zu nennen sowie Mechanismen, die das relationale DBMS selbst bereitstellt.

6.3.2.1 Wrapping

Der Begriff *Wrapping* bedeutet in wörtlicher Übersetzung "verpacken" und erlangte weltweit Popularität 1995 im Zusammenhang mit der Verhüllung des Reichstages in Berlin ("Reichstag Wrapping" [Chr95]). Winsberg beschreibt Wrapping als Lösungsmöglichkeit für das Problem der Integration von Legacy-Systemen [Win95]. Unterschieden werden *Application Wrapping* und *Database Wrapping*. Im ersten Fall umgibt die Wrapping-Schicht sowohl Daten als auch Anwendungsprogramme des Legacy-Systems. Die Emulation der Funktionalität von verschiedenen Benutzeroberflächen bedingt hier spezialisierte und zeitaufwendige Lösungen. Auch aktive objektorientierte Datenbanksysteme, die keine Eventdetektion im Server unterstützen, verlangen vom Benutzer die Erweiterung seiner Applikationsprogramme um Methodenaufrufe zur Signalisierung eines Methodenaufrufes. Dieses muß manuell erfolgen [GD93] oder wird automatisiert durch Einsatz eines Präcompilers [BZBW95]. Dabei wird jedoch vorausgesetzt, daß bereits zur Übersetzungszeit eines Applikationsprogramms alle möglichen Funktionsaufrufe an einem LAI oder LDBI bekannt sind, was den Gebrauch dynamischer Anfragesprachen ausschließt. Die lokale Strukturautonomie eines lokalen Applikationssystems würde erheblich beeinträchtigt.

Der Ansatz des Database Wrapping basiert auf einer intelligenten Schicht zwischen dem DBMS und den lokalen Applikationsprogrammen. Die Hülle wird - applikationsunabhängig - um das DBMS gelegt. Für diesen Ansatz sind besonders Client-Server-Datenbanksysteme geeignet, die die Möglichkeit bieten, in die Kommunikation zwischen Client und Server einzugreifen. Die Detektion der relevanten Ereignisse erfolgt in einer Zwischenschicht durch eine syntaktische Analyse der über das lokale Datenbankinterface abgesetzten Anweisungen. Es geht somit keine Autonomie in bezug auf lokale Applikationen verloren. Ein weiterer Vorteil

gegenüber dem Application Wrapping besteht darin, daß die Kommandos erst zur Laufzeit analysiert werden müssen, was interpretativen Anfragesprachen wie SQL entgegenkommt und im Hinblick auf künftige dynamische Object Query Languages (OQL) von Nutzen ist.

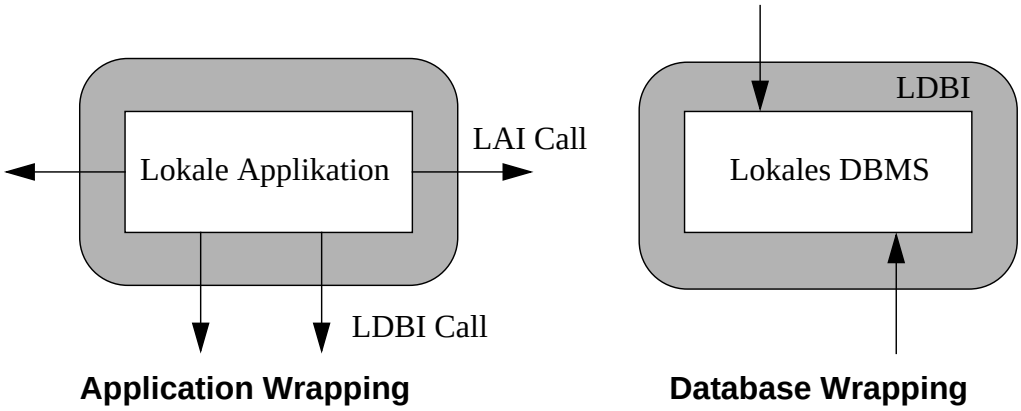


Abbildung 6.4: Wrapping-Mechanismen

Ein Nachteil des Database Wrapping ist jedoch darin zu sehen, daß Operationen, die vom lokalen Datenbankserver aktiviert werden, z.B. innerhalb von Triggern oder Stored Procedures nicht detektiert werden können. Diese Operationen werden nicht über das LDBI (Client-Server-Schnittstelle) aufgerufen, sondern verwenden eine DBMS-interne Schnittstelle, die nach außen nicht zugänglich ist. Die Detektion einer Triggerausführung wird dadurch erschwert, daß selbst der Aufruf des Triggers nicht mehr das LDBI passiert.

Ein verwandter Ansatz, wie durch Modifikation der DBMS-Software Ereignisse detektiert werden können, ist in [SKM92] beschrieben. Dieser Ansatz ist für Datenbanksysteme mit Client-Server-Architektur anwendbar. Hierbei wird das *SqlConnect* Kommando modifiziert, um die Datenbankoperationen, die an den Server gesandt werden, abzufangen. Dies gestattet die Analyse der Kommandos, um die Events zu identifizieren, die durch diese Operation ausgelöst wurden. Die erforderliche Modifikation des DBMS ist jedoch nur in wenigen Fällen möglich und bei Nicht-Client-Server-Datenbanken ein besonderes Problem.

6.3.2.2 Aktive Eigenschaften von relationalen Datenbanksystemen - Das Trigger-Konzept

Eine Möglichkeit zur Eventdetektion, die mittlerweile in allen kommerziell verfügbaren relationalen Datenbanksystemen zur Verfügung steht, ist das Triggerkonzept. Der Benutzer kann eine Folge von Anweisungen in der Art einer Stored Procedure angeben und die Ausführung dieser Befehlsfolge an DML-Operationen auf Objekten der Datenbank (d.h. im wesentlichen Tabellen) koppeln.

Ein Vorteil des Triggerkonzeptes ist die Integration seiner Mechanismen in den Datenbankserver. Dies sorgt für hohe Ausführungsgeschwindigkeit, da kein zusätzlicher Netzverkehr verursacht wird und die Triggerprozeduren vorübersetzt vorliegen. Zugriffe auf Tabellen werden in jedem Falle erkannt, also auch, wenn sie durch Trigger oder Stored Procedures verursacht wurden.

Gegen einen Einsatz von Triggern spricht deren Beschränkung auf die drei DML-Kommandos und die Kopplung an Tabellen. Da weitergehende Anforderungen an die lokale Eventdetektion gestellt sind (vgl. Tabelle 6.1), reichen die Fähigkeiten der Trigger hierfür nicht aus. So ist die Erkennung von lesenden Zugriffen (SELECT) nicht möglich. Ebenso kann die lokale Transak-

tionssemantik nicht unterstützt werden, da keine Detektion von Transaktionsbeginn oder -ende möglich ist. Somit wären Trigger nicht geeignet, um beispielsweise bei Änderung von Daten eine globale Update Propagation zu veranlassen, da sie implizit nur den Kopplungsmodus immediate verwenden. Außerdem sind Konflikte mit benutzerdefinierten Triggern zu erwarten, da in vielen Systemen für jedes DML-Event auf einer Tabelle nur je ein Trigger definiert werden kann.

Trigger bieten leider auch keine Möglichkeit, Veränderungen des lokalen Schemas zu beobachten, d.h. sie können nicht für Objekte des Data Dictionary definiert werden (dies gilt z.B. für Oracle, Informix, Sybase).

Trigger können, sofern sie die Möglichkeit dazu bieten, in ihrem Aktionsteil eine Benachrichtigung einer Detektionskomponente (Event-Monitor) beinhalten. Sollten die Trigger-Aktionen jedoch auf lokale Datenbankoperationen beschränkt bleiben, kann eine Detektion indirekt durch aktive Tabellen realisiert werden. Für jede (passive) Tabelle, die zu beobachten ist, wird eine aktive Tabelle definiert [SPAM91]. Bei jedem Zugriff auf die passive Tabelle fügt die Aktionskomponente des Triggers die Zugriffsinformationen (Datenbankoperationen und ihre Argumente) zur aktiven Tabelle hinzu. Diese aktiven Tabellen müssen regelmäßig durch einen Monitor untersucht werden (Polling), um lokale Events zu detektieren.

6.3.2.3 Auditing

Eine weitere Möglichkeit der Überwachung von Benutzeraktionen stellt das von einigen relationalen DBMS angebotene Auditing-System an. Bei Aktivierung dieses Systems werden in einer separaten Datenbank Daten über versuchte Zugriffe auf bestimmte Daten oder durch bestimmte Benutzer gespeichert. Je nach Konfiguration des Auditing-Systems ist neben einer globalen Überwachung die Auswahl einzelner Datenbanken, Tabellen, Sichten, Trigger oder Stored Procedures oder die Beschränkung auf Aktionen bestimmter Benutzer möglich. Das Auditing ist primär zur Aufrechterhaltung der Systemsicherheit gedacht und kann z.B. auch dazu dienen, eine Überlastung des DBMS oder einen Mißbrauch von Systemressourcen zu verfolgen. Die Aktivierung bzw. Deaktivierung dieser Mechanismen ist besonderen Datenbankbenutzern vorbehalten, z.B. in Sybase als *System Security Officer* (Sicherheitsbeauftragter) bezeichnet.

Das Auditing-System besitzt grundsätzlich die Fähigkeit, die erforderlichen Informationen über Datenmanipulationen und sogar über lesenden Zugriff auf kritische Daten aufzuzeichnen. Es hat jedoch den Nachteil, daß keine Zeitinformationen aufgezeichnet werden. Ebenso werden nicht die von einer DML-Operation betroffenen Tupel protokolliert, so daß keine Zustandshistorie der Datenbank entsteht. Unzureichend ist im allgemeinen die Protokollierung von schemamodifizierenden Anweisungen. Das Auditing-System kann aufgrund seiner Eigenständigkeit innerhalb des Datenbankservers nicht in einem föderierten System integriert werden, da es keine direkten Reaktionen auf Client-Anfragen erlaubt.

Fazit

Keiner der in diesem Abschnitt diskutierten Mechanismen allein ist perfekt, um die Anforderungen, die an die Eventdetektion gestellt sind, vollständig zu erfüllen. Die größte Flexibilität bietet der Database Wrapping Ansatz im Hinblick auf die Art der zu detektierenden Events und die dabei zu gewinnenden Informationen. Ein bedeutender Vorteil liegt in der größtmöglichen Wahrung der Autonomie eines lokalen Applikationssystems, das Hinzufügen einer Wrapping-Komponente bedeutet dabei nur eine Einschränkung vom Typ *S-add*(LDBMS). Die Entwick-

lung einer intelligenten Schicht zwischen Client und Server erfordert einen einmaligen Implementierungsaufwand für die syntaktische Analyse und Interpretation der Kommandos der Query Language. Eine solche Lösung muß für die DML-Events, die unterhalb dieser Schicht auf der Ebene des Servers auftreten, durch Trigger komplettiert werden.

6.4 Die Protokollierung von Datenänderungen

Die Protokollierung von kritischen Aktionen des Benutzers ist notwendig, um zu einem späteren Zeitpunkt alte Zustände des Datenbestandes rekonstruieren und auswerten zu können. Dies ist insbesondere für temporal abgeschwächte Integritätsbedingungen von Interesse. Dazu muß jede einzelne für die Konsistenz relevante Aktion protokolliert werden.

Der Ansatz, ein Transaktionsprotokoll (*Transaction Log*) für die nachträgliche Detektion und Interpretation von Events zu verwenden, erfordert Kenntnisse der internen Protokollstruktur. In einem solchen logischen Protokoll sind Informationen über Transaktionen, DML-Befehle und die durch sie verursachten Änderungen (alter Wert, neuer Wert) enthalten (ein Beispiel ist in [GMB+81] beschrieben). Transaktionsprotokolle werden vom Datenbankserver zum Zwecke des Recovery angelegt, können aber von den Clients nicht für Auswertungen u. ä. genutzt werden. Zwar gibt es DBMS, die einen lesenden Zugriff auf den Transaction Log erlauben, allerdings ist eine Interpretation der Einträge ohne Detailwissen über den Recovery Manager praktisch unmöglich. So enthält z.B. die Protokolltabelle von Sybase lediglich eine Transaktions-ID und eine sogenannte Update-Operationsnummer.

Aus den genannten Gründen wurde eine Lösung gewählt, die darin besteht, zusätzliche Protokollierungsbefehle transparent für den Benutzer zu erzeugen. Wir unterscheiden hierbei zwei Arten, das intensionale Protokoll (*Intensional Logging*) und das extensionale Protokoll (*Extensional Logging*).

Die **intensionale** Protokollierung erfaßt nur den Text des Kommandos. Die betroffenen Tupel werden in der WHERE-Klausel beschrieben, deren Interpretation zu einem späteren als dem ursprünglichen Ausführungszeitpunkt aber zu fehlerhaften Ergebnissen führt. Zum Beispiel kann die Aufzeichnung eines INSERT-Befehls, bei dem die Tupel aus einer anderen Tabelle kopiert werden (“`INSERT INTO T1 SELECT * FROM T2`”) nicht die tatsächlich eingefügten Tupel rekonstruieren, wenn nicht auch der Zustand von T2 zum Ausführungszeitpunkt bekannt war. Eine Parallele zeigt sich auch bei View Maintenance-Algorithmen von Data-Warehousing-Systemen [ZGHW95]: Anomalien treten auf, falls die zur Erneuerung einer View nötigen Anfragen später als die zugehörigen Datenmanipulationen erfolgen und keine zusätzlichen Maßnahmen getroffen werden. Dieses Fehlverhalten läßt sich durch Verwendung eines extensionalen Protokolls korrigieren.

Bei einer **extensionalen** Protokollierung werden alle von einer Manipulation betroffenen Daten so abgespeichert, daß die Rekonstruktion eines vergangenen Zustandes einer Tabelle ausschließlich unter Verwendung der Protokollinformationen dieser Tabelle möglich ist. Protokolliert werden müssen alle von einer Operation betroffenen Tupel mit der Angabe des Zeitpunktes, des Benutzernamens und der Operationsart. Da gleichzeitig mehrere Benutzer mit jeweils mehreren Anwendungen am Server arbeiten können, muß auch noch der Identifikator der Client-Server-Verbindung erfaßt werden. Um die Semantik lokaler Transaktionen zu berücksichtigen, sind weitere Schritte bei der Protokollierung notwendig. Prinzipiell muß verhindert werden, daß temporäre Zwischenzustände außerhalb eines lokalen Systems sichtbar werden, also müssen auch Transaktionsereignisse protokolliert werden.

Die Zusammenfassung der Log-Einträge in einem **verdichteten** Modifikationsprotokoll ermöglicht, daß bei einer Auswertung des Protokolls zu einem späteren Zeitpunkt keine inkonsistenten Zwischenzustände sichtbar werden, die nur vorübergehend innerhalb von Transaktionen gültig sind. Die Herstellung eines verdichteten (*condensed*) Protokolls ist vergleichbar mit Lösungsansätzen für die Wartung von *Database Snapshots* [KR87, LHM+86] oder materialisierten Sichten [BLT86]. Dabei werden die bis zu einem bestimmten Zeitpunkt protokollierten Änderungen an einem Objekt mit einer zu diesem Zeitpunkt eintretenden weiteren Änderung zu einem sogenannten Netto-Effekt (vgl. auch [WF90]) verschmolzen.

6.5 Eventsignalisierung durch Middleware

Wie bei der Diskussion des Eventmodells aktiver Objekte bereits diskutiert wurde, müssen auftretende lokale Events an das globale Vermittlersystem signalisiert werden, das die Konsistenzkontrolle vornimmt. Hierbei sollen die Möglichkeiten untersucht werden, die *Middleware* dafür bietet. Middleware wird die Software genannt, die beim Informationstransfer über ein Netzwerk hinweg unterstützend eingesetzt werden kann. Die Middleware-Schicht bewahrt den Entwickler davor, Unterschiede in den Kommunikationsprotokollen, den Betriebssystemen und den Hardware-Plattformen zu beachten und stellt somit eine hardware- und netzwerkunabhängige Infrastruktur bereit. Das Angebot an Middleware kann man in mehrere Bereiche einteilen:

- Database Middleware
- Remote Procedure Calls (RPC)
- Object Request Broker (ORB)
- Message Oriented Middleware (MOM)

6.5.1 Database Middleware

Dieser Ansatz setzt den Gebrauch eines (kommerziell verfügbaren) Datenbank-Gateways voraus, der vom Middleware-Hersteller geliefert wird. Ein Client-Prozeß sendet eine SQL-Anfrage an das Gateway, das diese dann an die Datenbank (d.h. den dortigen Server) weiterleitet. Abhängig von der Art der Anfrage wird dieses Gateway die Anfrage, in den Dialekt der Datenbank übersetzt, weitergeben. Datenbank-Middleware betreibt eine synchrone *point-to-point* Kommunikation. Es ist einer der am ausgereiftesten und verbreitetsten Middleware-Ansätze. Einen Überblick über kommerziell verfügbare Gateways gibt [Hüs95], Informationen über den Standard *Remote Database Access* (RDA) zur Verteilung von DB-Operationen in heterogenen Systemumgebungen findet man u.a. in [Pap91].

6.5.2 Remote Procedure Calls

Remote Procedure Calls (RPC) stellen einen seit längerem etablierten Ansatz dar, über ein Netzwerk hinweg zu kommunizieren. RPCs unterstützen eine prozedurale Sichtweise. Der ausgeführte Code eines Clients ruft seinerseits eine Prozedur auf einer anderen (remote) Plattform auf, deren Ergebnisse werden an die erstaufrufende Prozedur zurück übermittelt. Bei der Verwendung der RPCs ist wenig kommunikationsspezifischer Code notwendig. Ein Großteil des Codes wird seitens der Interface Definition Language (IDL) generiert (auch als XDR bezeichnet). Im allgemeinen werden die Einzelkomponenten einer solchen Applikation eine syn-

chrone Kommunikation in Form des *request-wait-reply* betreiben. Weitere Details sind in Abschnitt 7.2.1 beschrieben.

6.5.3 Object Request Broker

Object Request Broker (ORBs) können als eine Art objektorientierte RPCs aufgefaßt werden. Sie stellen einen wohldefinierten Standard in der Form von CORBA (Common Object Request Broker Architecture) dar, der von der Object Management Group (OMG) entwickelt wird. ORBs sind besonders für Systeme geeignet, bei denen Objektorientierung eine primäres Erfordernis ist. Wird ein ORB eingesetzt, legt eine Interface Definition Language (IDL) das Interface zwischen den Objekten fest (so wie im Falle der RPCs eine IDL das Interface zwischen den Prozeduren festlegt). ORBs arbeiten generell synchron in *point-to-point* Form.

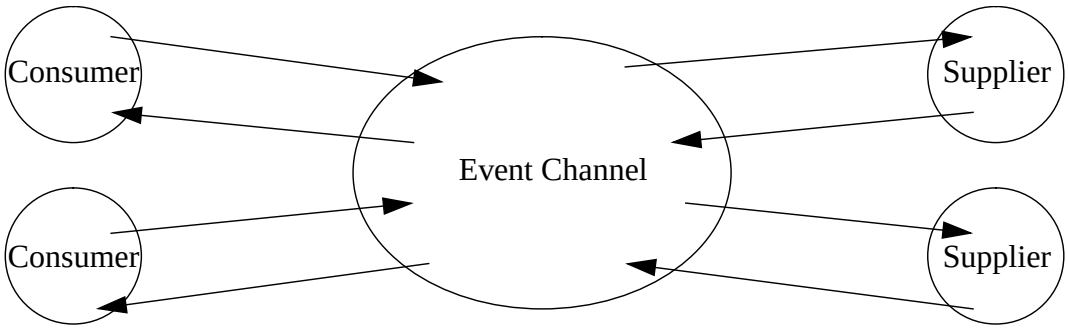


Abbildung 6.5: CORBA Event Services

Für die Realisierung einer Infrastruktur entsprechend CORBA wurden von der OMG Object Services definiert, von denen die Event Services als Grundlage einer ereignisbasierten Programmierung komplexer verteilter Applikationen näher betrachtet werden: Die CORBA Event Services unterstützen das *Push* Modell, bei dem das ereigniserzeugende Objekt den Versand des Events zum Verbraucherobjekt initiiert, als auch das *Pull* Modell, wo der Verbraucher vom Erzeuger die Ereignisdaten anfordert. Ein vermittelndes Objekt, der *Event Channel*, erlaubt die asynchrone Kommunikation zwischen mehreren Erzeugern und mehreren Verbrauchern, ohne daß diese die Identität der anderen kennen. Die Kommunikation kann generisch oder getypt sein, in letzterem Fall werden die Operationen, die am Interface in der IDL definiert wurden, benutzt. Es sind Standard-Interfaces vorgesehen, die es Erzeugern erlauben, Events einzutragen und in den Channel zu stellen, sowie Verbrauchern ermöglichen, Events zu abonnieren und aus dem Event Channel abzurufen.

6.5.4 Message Oriented Middleware

Message Oriented Middleware (MOM) kann als prozedurorientiert aufgefaßt werden, d.h es besteht ein Informationsfluß zwischen den Prozeduren. Hierbei werden Informationen in Form von Messages asynchron von einem Programm zu einem oder mehreren anderen Programmen versandt. Der Austausch von Nachrichten zwischen den Programmen basiert auf dem *Message Queuing*. Dabei legen Anwendungen Nachrichten für andere Anwendungen in Queues ab und lesen die Nachrichten von anderen Anwendungen aus Queues. Nachrichten an Partneranwendungen müssen nicht sofort, sondern können verzögert übertragen werden (*deferred delivery*). Die Nachrichten können in den Queues solange gespeichert werden, bis sichergestellt ist, daß der Kommunikationspartner erreicht werden kann und die Partneranwendung empfangsbereit ist. Die Kommunikation wird beim Message Queuing immer asynchron durchgeführt - es gibt dafür aber eine ganze Reihe von Übertragungsoptionen - nach Priorität, Zeit, Zahl der Messa-

ges in den Queues usw. - die auch eine nahezu synchrone Übertragung ermöglichen. Beim *transactional* Message Queuing erfolgt die Übertragung der Nachrichten transaktionsgesichert. Den Gebrauch eines Queuing Systems für die Ausführung von Transaktionen in kommerziellen verteilten TP-Systemen beschreiben Bernstein u.a. [BHM90].

MOM-Produkte beinhalten nicht nur Funktionalitäten für den Versand von Informationen (Messages), sondern darüber hinaus Dienstleistungen für das Übersetzen von Datentypen, Fehlerbehandlung und für die Lokalisierung von Ressourcen im Netzwerk. Sie werden von einer Vielzahl von Herstellern angeboten. Hierzu zählen MQSeries von IBM, DECmessageQ von DEC, Communications Integrator von Covia sowie SmartSockets von Talarian.

6.5.5 Auswahl der Kommunikationsplattform

Unter den genannten Ansätzen erwiesen sich Remote Procedure Calls für die Realisierung der Kommunikation zwischen dem Gateway und dem globalen Vermittler als sehr gut geeignet. Diese bieten den Vorteil der vollkommenen Ortstransparenz sowohl des SQL-Gateways (als RPC-Client) wie auch des Vermittlers (RPC-Server) in einer vernetzten UNIX-Umgebung. Die RPC-Schnittstellendefinition in Gestalt einer XDR-Protokollspezifikation ist übersichtlich, wartungsfreundlich und leichter erweiterbar als speziell definierte Protokolle und erlaubt die nötige Offenheit im Vergleich zu plattformabhängigen kommerziellen Lösungen (RDA, MOM). MOM-Ansätze erscheinen durch die mögliche asynchrone Kommunikation und transaktionsgesicherte Übertragung zwar sehr gut geeignet, eine Realisierung mit RPCs ist jedoch die billigere Lösung.

In der Perspektive ist eine Implementierung entsprechend dem CORBA-Standard denkbar als eine natürliche Weiterentwicklung zu einem “voll” objektorientierten Vermittlersystem. In einer solchen verteilten Umgebung können auch Datenbanksysteme neben anderen Ressourcen als Komponenten integriert werden. Aktive DBMS sind hierbei typischerweise Erzeuger von Datenbank-Events, sie können aber auch externe Events verbrauchen (d.h. solche, die in Applikationen generiert werden). Die entsprechenden IDL Interfaces werden somit auf einem aktiven DBMS definiert, so daß sie mit anderen Komponenten über die CORBA Event Services interagieren können.

6.6 Objektorientierte Integration von relationalen Datenbanken

6.6.1 Existierende objekt-relationale Ansätze

Um die Lücke zwischen relationalen und objektorientierten Datenbanksystemen zu schließen, wurde eine Reihe von Ansätzen bei DBMS-Anbietern, Anwendern und in der Forschung verfolgt, die folgendermaßen eingeteilt werden können:

- Erweiterung des relationalen Modells um objektorientierte Konzepte. Die umfangreichste Arbeit wird von der ANSI X3H2 Working Group geleistet bei der Entwicklung des SQL3-Standards [ISO95] (vgl. auch die Entwicklung von Illustra [Sto96]).
- Hinzufügen von Objektschichten auf der Basis von relationalen Datenbanken. Dies kann innerhalb der relationalen Datenbank oder in einem zusätzlichen Produkt realisiert werden (z.B. UniSQL/M, Subtleware, Persistence, vgl. hierzu Abschnitt 6.6.4).

- Hinzufügen von Objekt-Gateways zu relationalen Datenbanken, z.B. Gateways von einem OODBMS zu relationalen Datenbanken (z.B. ObjectStore [OK95]).
- Entwicklung von Objekt-Middleware zum Zugriff auf relationale Datenbanken. Diese können auf der OMG-Spezifikation für Object Services (z.B. Persistent Object Service [Ses96], Object Query Service, Object Transaction Service [OMG94]) basieren.

Die letzten drei Ansätze erfordern eine Abbildungsschicht zur Abbildung des Objektmodells auf das relationale Modell in beiden Richtungen. Die Unterstützung eines objektorientierten Zugriffs auf relationale Datenbanken weist zahlreiche Vorteile auf:

- Das relationale Datenbanksystem kann sowohl traditionelle Applikationen durch eine Standard-Schnittstelle (im allgemeinen SQL) unterstützen und zugleich durch ein objektorientiertes Interface fortgeschrittenen Anwendungen zur Verfügung stehen.
- Existierende Daten in relationalen Datenbanken können mehrfach verwendet werden sowohl in den traditionellen Applikationen als auch in objektorientierten Applikationen durch eine eindeutige Beziehung zwischen Objekt und Tupel.
- Vorteile der relationalen Datenbanktechnologie, wie z.B. Parallelverarbeitung, können in herkömmlichen und Non-Standard-Applikationen genutzt werden.

6.6.2 Leistungsumfang einer objektorientierten Zugriffsschicht auf relationalen Datenbanken

Unter den genannten Ansätzen besteht der von uns bevorzugte Weg darin, die Zugriffsfunktionen auf einer relationalen Datenbank auf objektorientierte Weise zu realisieren, d.h. das API einer relationalen Datenbank wird repräsentiert als eine Sammlung von Methoden auf den korrespondierenden Klassen des objektorientierten Datenmodells (Abbildung 6.6).

Somit können effiziente und zuverlässige relationale Datenbanksysteme (wie Sybase, Informix, Oracle, Ingres) als Speichermanager für persistente Daten verwendet werden. Die objektorientierte Applikation operiert mit Anwendungsobjekten, die Sichten auf den relationalen Daten entsprechen. Dabei ist sie allerdings von den technischen Details eines Zugriffs auf die relationale Datenbank, wie z.B. Cursorverwaltung und Return Codes, entlastet, und wird - durch Anpassung oder Austausch der Zugriffsschicht - unabhängig von konkreten Datenbanksystemen und somit portabel.

Eine individuelle Entwicklung (vgl. auch [HTW95]) ist jedoch sehr aufwendig und komplex. Es müssen dabei technische Lösungen für Objektidentität, Persistenz, komplexe Datentypen, die Abbildung von Klassen auf Tabellen, Polymorphismus und objektorientierte Datenbankanfragen realisiert werden. Die individuelle Realisierung anwendungsspezifischer Methoden zum Erstellen, Modifizieren, Löschen und Abfragen von Objektinstanzen (und der mit ihnen korrespondierenden Tupel) müßte für jede Klasse des Modells realisiert werden. Die damit verbundenen Nachteile sind u.a. Fehleranfälligkeit, Änderungsunfreundlichkeit, notwendige Erweiterungen um Konzepte wie Transaktionenmanagement, Sperrenverwaltung u.ä. Aus diesen Gründen ist ein kommerziell verfügbares Werkzeug vorzuziehen, das Unterstützung für diese Aufgaben bietet. Dazu zählt vor allem auch das Problem der Abbildung zwischen objektorientiertem und relationalen Modell. Dieses wollen wir deshalb nachfolgend in einem gesonderten Abschnitt skizzieren.

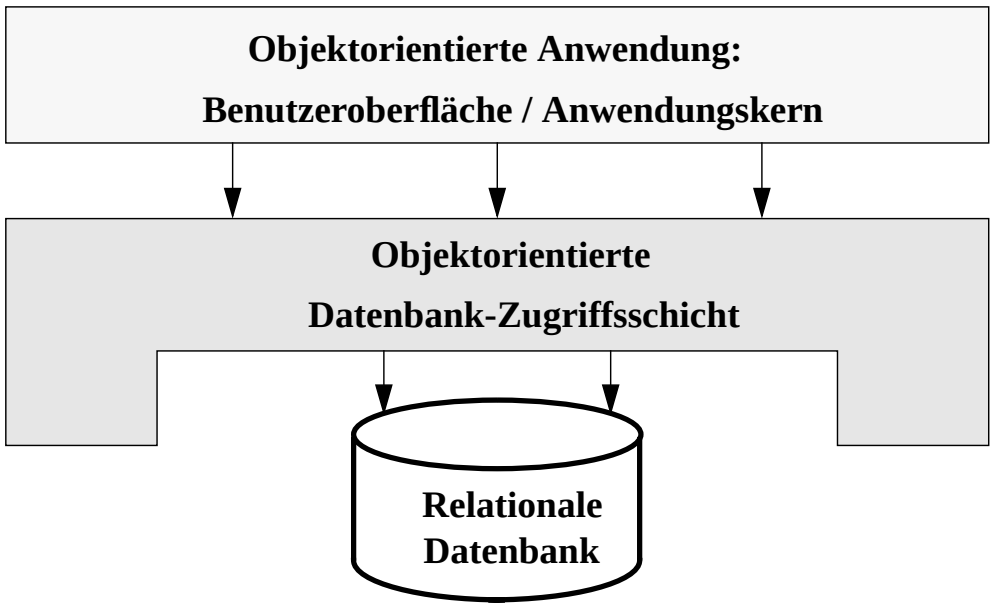


Abbildung 6.6: DB-Zugriffsschicht zwischen OO Anwendung und relationaler Datenbank

6.6.3 Abbildung zwischen objektorientiertem und relationalem Modell

6.6.3.1 Strukturelle Abbildung

a) Vorwärts-Abbildung (Forward Mapping)

Die Vorwärts-Abbildung beschreibt die Übersetzung eines objektorientierten Schemas in Relationen. Hierfür lassen sich eine Menge einfacher Grundregeln anwenden. Einen ausführlichen Algorithmus für die Abbildung eines erweiterten ER-Schemas (EER) in ein relationales Schema findet man in [EN94]. Diese Regeln lassen sich in analoger Weise auf ein ODMG/C^{++}-Datenmodell übertragen. Bei der Abbildung der Vererbungshierarchie bestehen mehrere Optionen für die Transformation der Subklassen in Relationen. Allgemein gilt, daß ein Teil der Semantik, die explizit in einem OO Schema repräsentiert ist, nur implizit im Relationenschema dargestellt werden kann.

b) Rückwärts-Abbildung (Reverse Mapping)

Eine Reihe von Arbeiten [BCN92, HHM+96, YL92, AAK+93] untersucht, wie semantische Modellkonstrukte in einem relationalen Datenbankschema identifiziert werden können. Die meisten dieser Methodologien verfolgen die Idee, die Relationen, basierend auf ihren Primärschlüsseln und Inklusionsabhängigkeiten, zu klassifizieren. Dieses ist zugleich die Grundlage für eine automatische Abbildung des Relationenschemas in ein EER oder OO Schema. Eine automatische Transformation erscheint jedoch nicht möglich. Auch wenn alle Primärschlüssel und Inklusionsabhängigkeiten für ein Relationenschema bekannt sind, ist es möglich, dieses auf unterschiedliche Weise in ein EER/OO Schema zu transformieren. Beispielsweise läßt sich eine Tabelle sowohl auf eine Klasse als auch auf eine Beziehung (Relationship) abbilden. Angebotene Methodologien bzw. kommerziell verfügbare Tools können nur als Hilfsmittel angesehen werden, deren Ergebnisse einer Anpassung durch den DB-Administrator bedürfen. Im Rahmen von IRO-DB, einem Projekt zur Entwicklung eines objekt-relationalen föderierten Datenbanksystems mit ODMG als kanonischem Datenmodell, wird die Abbildung eines lokalen Schemas in ein Exportschema in prozeduralen lokalen Adaptern realisiert [GSF+95]. Diese enthalten eine Menge von Funktionen für die Abbildung von Datentypen relationaler und ob-

jektorientierter Datentypen nach ODMG und erleichtern damit die Übersetzung von OQL Queries in lokale Anfragen.

6.6.3.2 Operationale Abbildung

a) Vorwärts-Abbildung (Forward Mapping)

In objektorientierten Systemen werden Operationen durch Aufruf von C++-Methoden ausgeführt. Anfragen können (im Stil von SQL) als assoziative Queries in einer Object Query Language (OQL) oder navigierend formuliert werden [Cat94]. In [HLW94] findet man einen Vergleich der Ausdrucksmöglichkeiten von Query Languages kommerzieller OODBMS.

Um Operationen aus einem objektorientierten System heraus in SQL-Befehle zu transformieren, gibt es prinzipiell zwei Möglichkeiten:
Eine Klasse (die mit einer Tabelle korrespondiert) enthält eine Menge von Methoden zum Zugriff bzw. Ändern der Instanzen dieser Klasse, die ihre Entsprechung in der DML von SQL haben. Dabei lassen sich Methoden unterscheiden, die auf einzelnen Instanzen oder auf mehreren bzw. allen Mitgliedern einer Klasse ausgeführt werden können. Methoden auf Instanzen erfordern den vorherigen Zugriff über einen Schlüssel, mengenorientierte Methoden erfordern die Auswahl der Objekte über ein Prädikat.
Alternativ läßt sich der Zugriff auf die relationale Datenbank über ein Call Level Interface (CLI) realisieren, wie es z.B. von der X/OPEN Group zusammen mit der SQL Access Group vorgeschlagen wurde. Die Idee des Interface besteht darin, eine Menge von Funktionen für den Aufbau einer Datenbankverbindung, das Absenden einer SQL Query, die Verarbeitung der Resultate sowie Transaktionskontrolle bereitzustellen. Diese lassen sich als Methoden auch in OO Sprachen wie C++ einbinden. Die Semantik eines solchen Methodenaufrufes ist im Unterschied zur ersten Variante vollständig im Befehlstext enthalten, der als Parameter übergeben wird. Das bedeutet allerdings, daß die Detektion relevanter Events, die durch eine solche Methode in einem objektorientierten System ausgelöst werden kann, eine Interpretation des Kommandotextes erfordert.

b) Rückwärts-Abbildung (Reverse Mapping)

Bei Vorhandensein einer deklarativen Object Query Language können relationale DML-Operationen 1:1 umgesetzt werden. Die Auswahl der Objekte setzt voraus, daß die Operationen der Relationenalgebra in eine äquivalente objektorientierte Algebra übersetzt werden können (siehe hierzu den Algorithmus in [PTR95]). Typischerweise erfolgt die Ausführung von Updates in der OO Zielsprache über Methodenaufrufe. Die Möglichkeit, C++ Methoden als Bestandteil einer laufzeitdynamischen Query aufzurufen, diskutieren Kiernan und Carey in ihrer Arbeit an OO-SQL/PESTO [KC95]. Wenn C++-Methoden innerhalb einer interpretativen Sprache aufgerufen werden sollen, muß die Methodenauswahl durch den Query-Prozessor simuliert werden (vgl. auch Abschnitt 8.2.1 auf Seite 147).

6.6.4 Kommerzielle objekt-relationale Produkte

Angesichts von Umfang und Komplexität der für eine individuelle Realisierung einer objektorientierten Zugriffsschicht zu lösenden Probleme haben eine Reihe von Anbietern kommerzielle Produkte vorgestellt, von denen die bekanntesten nachfolgend überblicksartig charakterisiert werden. Das Ziel besteht in der Auswahl eines solchen Produkts für einen Prototypen, mit dem herkömmliche relationale DBMS integriert werden können und das eine ODMG/C++-konforme objektorientierte Zugriffsschnittstelle beinhaltet.

6.6.4.1 UniSQL/M

UniSQL/M [Uni95] ist ein eng gekoppeltes föderiertes Datenbanksystem mit integriertem globalen Schema, an das sämtliche globale DB-Operationen zu richten sind. Die globalen Queries und Transaktionen werden in lokal ausführbare Teiloperationen übersetzt und dem lokalen DBMS zur Bearbeitung übergeben. Zur Integration lokaler relationaler Systeme (Ingres, Oracle, Sybase) werden jeweils Treiber benötigt, zur Kommunikation ist eine CSS-Komponente (*Communication Subsystem*) auf jedem Teilnehmerrechner notwendig. Zusätzlich kann das UniSQL-eigene objektorientierte Datenbanksystem UniSQL/X eingebunden werden.

Verteilte Änderungstransaktionen sind zwar möglich, setzen aber ein striktes Zwei-Phasen-Sperrprotokoll bei jedem LDBS voraus.

Mit dem UniSQL C++-Interface bietet UniSQL eine C++-Schnittstelle mit einer dem ODMG-Standard entsprechenden Architektur. Der Zugriff auf UniSQL wird dabei durch eine C++-Klassenschnittstelle und sogenannte Conversion Utilities ermöglicht. Die Schnittstelle unterstützt dabei zwei Arten der Klassendefinition: einen sprachenzentrierten und einen datenbankzentrierten Ansatz.

Der sprachenzentrierte Ansatz konvertiert die Klassendefinitionen aus existierenden C++-Headerdateien in die UniSQL-Datendefinitionssprache (DDL) und C++-Methoden, um den Zugriff auf persistente Daten dieser Klassen zu unterstützen. Der datenbankzentrierte Ansatz geht von existierenden UniSQL-Datenbank-Kataloginformationen (globales Data Dictionary) aus und erstellt daraus sowohl C++-Headerdateien als auch die benötigten Zugriffsmethoden für diese Klassen.

6.6.4.2 Subtleware

Subtleware unterstützt eine Vielzahl von Betriebssystem-Plattformen (DOS, Windows, mehrere UNIX-Dialekte), Datenbanksysteme (Oracle, Informix, Ingres, DB/2, dBase, Access, Foxpro, IMS, Btrieve, Rdb) und nahezu alle C++-Compiler [Sub95].

Subtleware besteht aus 4 Komponenten:

- dem Object Persistence API, einer am ODMG-Standard orientierten Schnittstelle für persistente Objekte zu darunterliegenden Datenbanken. Das API stellt erweiterbare Klassen und abstrakte Datentypen wie persistente Zeiger, Schlüssel, Beziehungen und Gruppierungen zur Verfügung, auf denen C++-Anwendungen erstellt werden können.
- SGEN, einem Forward-Engineering-Werkzeug, das die Abbildung von (durch ihre C++-Klassendefinitionen gegebenen) Objekten in die Datenbank unterstützt, indem es ein entsprechendes Datenbankschema anlegt.
- CGEN, einem Reverse-Engineering-Werkzeug, das die Erstellung von C++-Klassendefinitionen aus einem vorhandenen Datenbankschema automatisiert.
- SQLExec, ebenfalls ein C++-API, das auf SQL basiert und den Zugriff auf die Datenbanken erleichtert und standardisiert.

6.6.4.3 Persistence

Persistence von Persistence Software, Inc. [Per95b] realisiert die objektorientierte Zugriffsschicht in der Programmiersprache C++. Persistence generiert aus einem anwendungsspezifischen Objektschema C++-Klassenschnittstellen zwischen den C++-Objekten der Anwendung

und den Tabellen in der relationalen Datenbank. Die auf den Klassen generierten Methoden kommunizieren mit einem globalen Objektmanager, der Objekte und Methodenaufrufe in SQL-Kommandos und die Ergebnisse von SQL-Kommandos wieder zurück in Objektinstanzen abbildet.

Neben einem Objektschema-Editor mit graphischer Oberfläche besteht Persistence im wesentlichen aus zwei funktionalen Komponenten: dem *Relational Interface Generator* (RIG), der aus dem Objektschema datenbankunabhängige C++-Klassen für den Zugriff auf relationale Daten erzeugt, und dem *Relational Object Manager* (ROM), der den Datenbankzugriff über SQL, die Transaktionsverwaltung und die Sicherung der Objektintegrität implementiert.

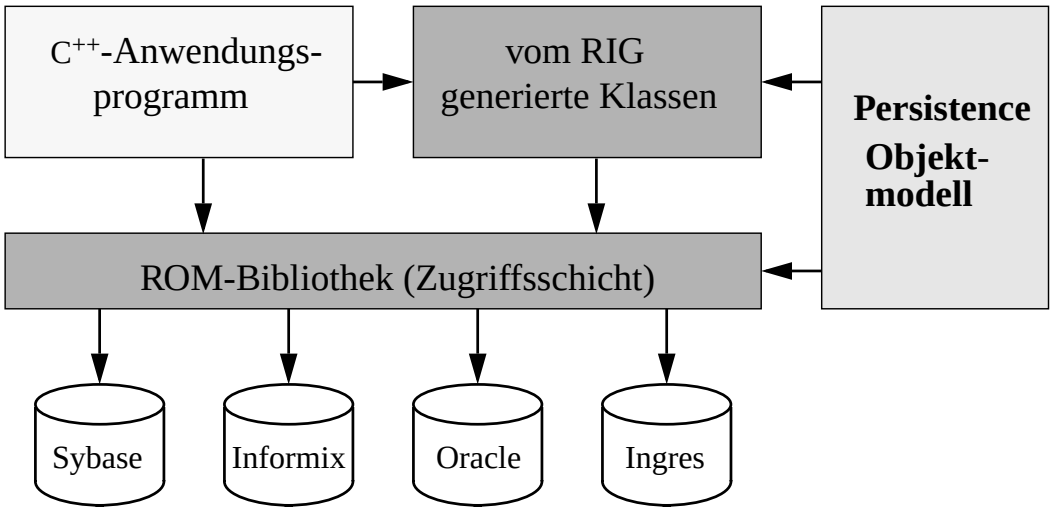


Abbildung 6.7: Systemarchitektur von Persistence im Überblick

Die Erstellung einer Persistence-basierten Anwendung besteht aus einem dreistufigen Prozeß (top-down):

- Definition eines anwendungsspezifischen Objektschemas
- Generierung der C++-Klassenschnittstelle aus dem Objektschema mit dem RIG
- Implementierung der Anwendungsfunktionalität auf der generierten Klassenschnittstelle

Eine bottom-up-Vorgehensweise, bei der durch Einsatz eines Data Dictionary Readers die Objektschema-Informationen gewonnen werden, wird später noch ausführlich auf Seite 140 beschrieben.

6.6.4.4 Weitere Systeme

Odapter von Hewlett-Packard Co. realisiert eine Softwareschicht zwischen den Object Views auf relationalen Daten und dem relationalen DBMS als Speichersystem. Auf Server-Seite wird ein Objekt-Manager zur Verfügung gestellt, auf Client-Seite ist ein Objekt-API verfügbar. Der Objekt-Manager bildet das Objektmodell auf relationale Tabellen ab. Das auf Client-Seite angebotene API besteht aus einer Klassenbibliothek, die allerdings nur sehr generische Objekte unterstützt, die vom Benutzer auf spezifische Applikationsobjekte angepaßt werden müssen. Vererbung, Assoziation und Aggregation müssen manuell kodiert werden. Nur eine beschränkte Anzahl von RDBMS (z.B. ORACLE) wird unterstützt.

Object Gateway von Sierra Atlantic, Inc. ist ein Tool, das Modellierung, Abbildung und Codegenerierung und somit den objektorientierten Zugriff auf relationale Datenbanken unter-

stützt. Die Komponente *Schema Genie* erlaubt den objektorientierten Entwurf im ODMG-Modell und generiert verschiedene Sprachschnittstellen (z.B. C++, C, OLE). *Object Engine* ist die Laufzeitkomponente, die das Objektmodell implementiert (Aggregation, Vererbung, Beziehungen, Cache-Management). Nur Windows-Clients werden unterstützt.

Die Bibliothek **Dbtools.h++** von Rogue Wave Software, Inc. enthält C++-Klassen, um relationale Objekte wie Tabellen, Tupel und Spalten zu modellieren. Vererbung, Assoziationen und Laufzeitverhalten müssen noch von Hand kodiert werden. Es werden die gängigen RDBMS unterstützt. Wenn eine Applikation primär ad-hoc-Queries ausführt, wird die notwendige Funktionalität erbracht. Bei navigierenden Anfragen bzw. Verknüpfung mehrerer Tabellen sind allerdings eher Tools geeignet, die Caching unterstützen.

6.6.5 Systemauswahl

Für die Auswahl eines objekt-relationalen Systems wurde eine Reihe von Kriterien berücksichtigt, wobei die am FG vorhandene Systemumgebung zu berücksichtigen war.

- Verfügbarkeit auf SUN-SPARC Workstations (Betriebssystem Solaris 2.x)
- Vorhandensein einer C++ Schnittstelle zum Zugriff auf lokale relationale Datenbanken
- Unterstützung einer großen Anzahl von relationalen Datenbanksystemen
- effizientes Forward Engineering (d.h. Unterstützung einer möglichst automatischen Abbildung des Objektmodells in C++-Klassen bzw. in durch sie repräsentierte Relationen)
- Möglichkeiten zum Reverse Engineering zur Integration vorhandener relationaler Datenbanken (d.h. Einbeziehung von Legacy-Systemen)
- Objekt-Cache zur temporären Aufbewahrung von Query-Resultaten
- systemeigene Datenhaltung zur Ablage von Metadaten oder Regeln

Bei der Bewertung der Systeme gemäß den oben formulierten Anforderungen ergab sich, daß das Produkt Persistence am besten allen Bedingungen gerecht wurde mit Ausnahme des letzten Punktes. Hierfür muß als alternative Lösung eine relationale Datenbank einbezogen werden, auf die in gleicher Weise wie auf die lokalen Datenbanken über Persistence zugegriffen werden kann.

6.7 Integration der Regelverarbeitung

Für die Auswahl eines Regelsystems, das mit einem objektorientierten C++-System interagiert, gibt es es prinzipiell drei Möglichkeiten:

1. Einsatz eines aktiven objektorientierten DBMS

Eine Anzahl von aktiven objektorientierten Prototyp-Systemen ist bereits verfügbar. Zur Auswahl stehen u.a. REACH [BZBW95], SAMOS [GD93] und Ode [GJ91]. Alle diese Systeme sind integriert in das C++-Datenmodell und basieren auf einem objektorientierten Datenbanksystem. Das bedeutet, daß lokale Daten, die sich wie aktive Objekte verhalten sollen, auch als Objekte in diesen aktiven OODBMS vorliegen müssen. Einer Lösung, die ein OODBMS zusammen mit Persistence als Schnittstelle zu relationalen Datenbanken eng integriert, sind derzeit Grenzen gesetzt. Die hierzu erforderliche Einbettung von Persistence in ein OODBMS als

Voraussetzung für eine gemeinsame Eventdetektion bzw. Regelausführung ist erschwert durch die Schnittstelle des in Persistence vorhandenen Relational Interface.

2. Einsatz von Produktionsregelsystemen / Expertensystem-Shells

Es gibt einige Regelsysteme auf dem Markt, die auch mit C/C^{++} kompatibel sind, d.h. einen Output in C oder C^{++} produzieren. Somit wäre eine Kombination möglich zwischen den Objekten eines Objekt-Cache und einer Rule Engine auf diesen Objekten, die als Proxy-Objekte der lokalen Datenbanken fungieren.

ILOG Rules stellt eine Regelsprache zur Verfügung sowie einen Compiler, der diese in C^{++}-Code transformiert [ILO96]. Eine Regelmenge beschreibt das Verhalten eines Agenten und wird in eine spezielle C++-Klasse *Context* übersetzt. Diese umfaßt eine Applikationsschnittstelle, die Regeln, die eine Bedingung erfüllen, sowie einen *Working Memory*, der die Objekte enthält, auf denen die Regeln ausgeführt werden. ILOG Regel-Agenten können wie C^{++}-Objekte behandelt werden. Jede übersetzte Regelmenge entspricht einer C^{++}-Klasse. Somit können mehrere Regelmengen genutzt werden und daraus Multi-Agenten-Systeme gebaut werden. Die Regelverarbeitung basiert hierbei auf dem RETE-Algorithmus. ILOG Rules läßt sich auf natürliche Weise in die Entwicklung von C^{++}-Systemen integrieren. Bereits bei der Definition von Regeln lassen sich C^{++}-Ausdrücke als Inline-Code im Bedingungs- und Aktionsteil verwenden.

CLIPS (**C L**anguage **I**ntegrated **P**roduction **S**ystem) ist ein Expertensystemwerkzeug, das eine vollständige Sprachumgebung für die Konstruktion von regelbasierten oder regel- und objektbasierten Expertensystemen bereitstellt [Ril95]. Es lassen sich Produktionsregeln, bestehend aus Bedingungs- und Aktionsteil, definieren. Im Bedingungsteil wird die Existenz von Fakten überprüft. Somit lassen sich eintreffende Events dadurch realisieren, daß die Faktenliste erweitert wird. Aktionen können auch benutzerdefinierte C-Funktionen sein. Somit bietet das System eine Voraussetzung für die Einbettung in eine C^{++}-Programmierumgebung, z.B. CORBA [BKK96] oder auch Persistence.

Intelligent Rules Element™ (IRE) ist Teil der Entwicklungsumgebung *Elements Environment* von Neuron Data [Neu96]. Es unterstützt die Entwicklung von Geschäftsregeln und besteht im Kern aus einem Regelprozessor, der sowohl vorwärts- als auch rückwärtsverkettet arbeitet. Die offene Architektur von IRE enthält API-Schnittstellen zu C/C^{++} und erlaubt die Einbettung in Applikationen, die Event Monitoring und entsprechende Reaktionen leisten sollen. Ein transparenter datenbankübergreifender Zugriff auf den gängigen RDBMS ist in IRE möglich.

3. Eigenentwicklung

Die Eigenentwicklung einer Regelverarbeitungskomponente gestattet eine flexible Realisierung der Konzepte aktiver Datenbanksysteme, die als Voraussetzung globaler Integritätskontrolle in Abschnitt 3.3 beschrieben wurden. Die reichhaltigen Ausdrucksmöglichkeiten aktiver DBS (z.B. Kopplungsmodi, Zeitereignisse, komplexe Ereignisse) lassen sich durch den Recognize-Act-Zyklus von Produktionsregelsystemen nicht adäquat realisieren. Ein Einsatz aktiver DBS wurde aufgrund der fehlenden Kopplung zum Persistence-System verworfen.

Aus den oben genannten Gründen heraus wurde der Eigenentwicklung einer aktiven Komponente, basierend auf der gewählten Systemplattform Persistence, der Vorzug gegeben. Diese Komponente soll in der Lage sein, ECA-Regeln zu verarbeiten, und funktional einen Bestandteil des zu entwickelnden Vermittlersystems bilden.

Kapitel 7

Detektion und Signalisierung lokaler Events

Dieses Kapitel beschreibt die Implementierung eines relationalen Datenbank-Gateways auf der Basis des SQL-Servers und des Open Servers von Sybase. Zugrunde liegt die Idee des Database Wrapping, die bereits im vorigen Kapitel beschrieben wurde. Dabei wird in die Kommunikation zwischen Client und Server eingegriffen, indem ein eintreffendes SQL-Statement durch einen entsprechenden Event Handler abgefangen und behandelt wird. Durch syntaktische Analyse des Befehls läßt sich die Relevanz der auszulösenden Datenbankoperation für die globale Konsistenz bestimmen. Wir zeigen die Architektur des Gateways, das nach der Analyse der Anweisungen auch verantwortlich ist für die Signalisierung der lokalen Ereignisse an den Vermittler bzw. deren lokale Protokollierung.

Eine besondere Behandlung erfahren DDL-Anweisungen, die an einen Vermittler signalisiert werden, der die Schemakonsistenz überwacht. In Sybase mögliche Schemaänderungen werden über eine RPC-Schnittstelle signalisiert.

Darüber hinaus wird auch auf das Problem verborgener Events eingegangen, die an der SQL-Sprachschnittstelle nicht erkannt werden können, weil diese erst durch den Datenbank-Server ausgelöst werden.

Weiterhin findet man in diesem Kapitel Details der Protokollierungsalgorithmen für DML-Befehle (Abschnitt 7.1.5) sowie einen Überblick über die derzeit vorhandenen Konfigurationsmöglichkeiten des Gateways (Abschnitt 7.1.7).

Interessant ist die Verallgemeinerbarkeit der hier vorgestellten Lösung auf andere RDBMS. Abschnitt 7.1.9 enthält eine kurze Diskussion hierzu.

Die Signalisierung der lokalen Ereignisse erfolgt über Remote Procedure Calls (RPC). Weitere Details über die Aufrufschnittstellen werden in Abschnitt 7.2 behandelt, der auch einen Überblick darüber enthält, wie ein entsprechender RPC-Server generiert wird.

7.1 Implementierung eines Datenbank-Gateways

7.1.1 Plattform: Sybase Open Server

Bei der Anforderungsanalyse, wie Änderungen in bestehenden DBS mit minimalem Verlust an Autonomie detektiert werden können, wurde der Ansatz des Database Wrapping favorisiert

und am Beispiel des relationalen DBMS Sybase umgesetzt. Dieser Ansatz ist prinzipiell auch in anderen RDBMS anwendbar (vgl. Abschnitt 7.1.9). Eine Sybase-Komponente, der sogenannte *Open Server*, enthält Werkzeuge und Schnittstellen, die die Entwicklung eigener Server ermöglichen. Zum Open Server gehören Bibliotheken mit Funktionen zur Entwicklung von Servern (*Server Library*) und von Client-Anwendungen (*Client Library*) sowie Routinen, die bei Entwicklung von Client- und Serveranwendungen nützlich sind (*CS Library*).

Anwendungen, die mit Funktionen der Open Server Library geschrieben werden, verhalten sich gegenüber den Client-Programmen wie der herkömmliche SQL-Server von Sybase, der natürlich keine Modifikation ermöglicht. Ein selbstgeschriebener Server kann jedoch auch noch für andere Aufgaben genutzt werden, so z.B. zur Steuerung von Hardware, zur Kommunikation mit beliebigen weiteren Anwendungen mit Hilfe verschiedener Protokolle oder einfach zum “Durchreichen” von Kommandos (nach entsprechender Bearbeitung) an den Sybase SQL-Server.

Eine Open Server-Anwendung kann auf drei verschiedene Arten arbeiten: als alleinstehender Server, als Hilfsserver oder als Gateway. Im Fall des alleinstehenden Servers nimmt der Client direkt Kontakt zum Server auf. Der Hilfsserver verarbeitet entfernte Prozeduraufrufe (RPCs) eines SQL-Servers. Als Gateway ist eine Open Server-Anwendung z.B. in der Lage, Client-Anwendungen und Server zusammenzuführen, die auf direktem Wege nicht miteinander kommunizieren können (vgl. Abbildung 7.1). Eine solche Open Server-Anwendung bildet die Grundlage für die Entwicklung eines Datenbank-Gateways, das in die Kommunikation zwischen Client und Server eingreift.

Abbildung 7.1: Open Server-Anwendung als Gateway

Zur Konstruktion einer Open Server-Anwendung (kurz: Server) gehören folgende Schritte:

- Definition der Betriebsumgebung des Servers durch Allokation verschiedener Datenstrukturen und das Setzen globaler Attribute (*Properties*).
- Definition der Fehlerbehandlung. Diese kann entweder durch Weitergabe der Fehlerinformation an die verursachende Client-Anwendung oder innerhalb einer eigenen Fehlerbehandlungsroutine erfolgen.
- Installation von Ereignisbehandlungsroutinen (*Event Handling Routines*), die vom Server aufgerufen werden, wenn Client-Kommandos sogenannte *Open Server Events* auslösen.
- Initialisierung des Servers und Aufruf des Servers, so daß er auf Anfragen von Clients wartet und diese abarbeitet.

Die Ereignisbehandlung macht den größten Teil der Verarbeitung im Server aus, da viele der Anfragen, die den Server erreichen, bei diesem Events auslösen. Der Event-Begriff wird allerdings hier anders gebraucht als bei ECA-Regeln. Die Open Server Events bewirken, daß derjenige Prozeß (*Thread*) im Server, der dem jeweiligen Client zugeordnet ist, eine entsprechende Behandlungsroutine ausführt; der Open Server ist *multithreaded* ausgelegt, so daß die Abarbeitung von Anfragen nach außen hin parallel erscheint.

7.1.2 Realisierung der Event-Detektion

Eine Grundaufgabe des Gateways ist, von den Clients kommende Befehle an den Server weiterzureichen. Dieses Prinzip läßt sich anhand der Ereignisbehandlungsroutine für SQL-Statements beschreiben, die als Kommandotext übermittelt werden. Den Anlauf bei Eintreffen eines SQL-Kommandos am Gateway veranschaulicht Abbildung 7.2.

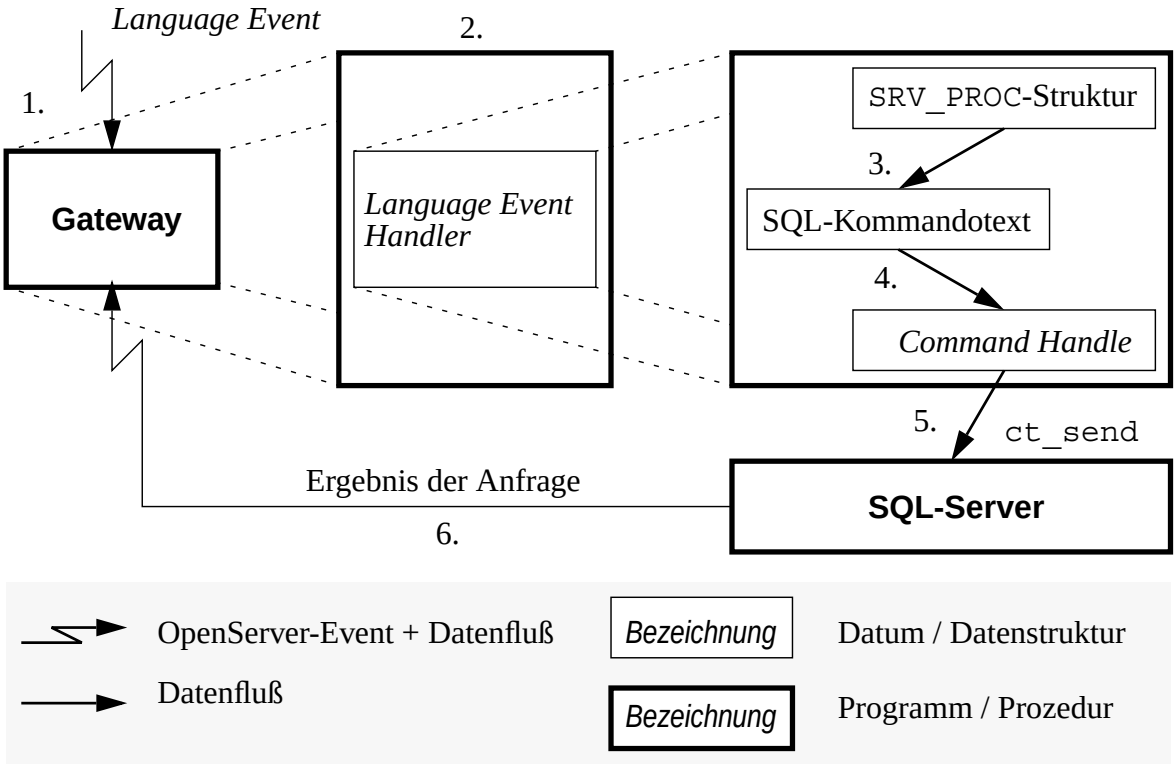


Abbildung 7.2: Ablauf der Behandlung eines Language Events

Nachdem die Behandlungsroutine für diese sogenannten *Language Commands* beim Starten des Open Server-Programms installiert wurde, wird sie beim Eintreten eines entsprechenden *Language Events* aufgerufen. Diese heißt `gw_langhandler` und erhält die Datenstruktur als Aufrufparameter, die die betroffene Client-Server-Verbindung identifiziert (`SRV_PROC`).

Unter Verwendung dieser Datenstruktur wird mit Hilfe von Routinen der Server-Bibliothek zunächst die Länge des SQL-Kommandotextes ermittelt (`srv_langlen`), ein entsprechend großer Speicherbereich allokiert (`srv_alloc`) und der Text des SQL-Kommandos dorthin kopiert (`srv_langcpy`). Anschließend wird durch den Aufruf einer Hilfsroutine ein Zeiger auf eine weitere Datenstruktur (*Connection Handle*) erzeugt, welche die eigentliche Implementierung einer Serververbindung darstellt (`gw_getchp`). Dieser Zeiger wird benötigt, um einen *Command Handle*, eine Art "Container" für ein SQL-Kommando, mittels Client-Bibliotheks-aufrufen zu allokieren und gegenüber dem Server bekanntzumachen (`ct_cmd_alloc`, `ct_command`). Er wird anschließend verwendet, um in einer weiteren Hilfsroutine (`ct_send`) den Text des SQL-Kommandos an den SQL-Server zu übertragen.

Im Abschluß an das Absenden werden mit der Hilfsroutine `gw_handleresults` die vom SQL-Server als erste Reaktion auf den Eingang des Kommandotextes übertragenen Daten ausgewertet und an die Clientanwendung gemeldet. Falls der SQL-Server dabei einen Fehler melden sollte, wird die Verarbeitung des aktuellen Kommandos beendet (`ct_cancel`). Anson-

sten wird die Ereignisbehandlungsroutine beendet, die die Kontrolle des Threads mit einem Erfolgscode an die Verwaltungskomponente der Open Server-Anwendung zurückgibt.

Auf ähnliche Weise werden auch andere Server-Aufrufe verarbeitet, bei denen nicht der Text des Kommandos übermittelt wird, sondern eine Bibliotheksroutine mit entsprechenden Parametern aufgerufen wird.

Eine Besonderheit bei der Weitergabe der Rückgabewerte (Anzahl der betroffenen Tupel) an die Clientanwendung besteht darin, daß diese innerhalb des Gateways “korrigiert” werden müssen, wenn zusätzliche Statements transparent für den Benutzer ausgeführt werden. Dies ist z.B. bei Protokollierungsoperationen der Fall, die quasi automatisch ohne Zutun des Benutzers in die Befehlsfolge eingefügt werden.

7.1.3 Die Architektur des Gateways

Das Gateway wurde als lokale Komponente des Vermittlersystems als virtueller SQL-Server mit den Funktionen der Open Server-Programmbibliothek implementiert. Erweitert wurde das Gateway um eine SQL-Analysefunktion (*Analyzer*) und je eine Komponente zur Protokollierung und Signalisierung von kritischen Befehlen, die vom Analyzer aus aufgerufen werden. Des weiteren sind Tools zur Datenbank-Initialisierung vorhanden, um die Protokollierung zu unterstützen. Diese werden logisch zum Gateway gezählt, obwohl es sich um unabhängige Programme auf Client-Seite handelt. Abbildung 7.3 zeigt, wie sich das Gateway zusammen mit dem eigentlichen SQL-Server den Client-Anwendungen gegenüber darstellt [Kra95].

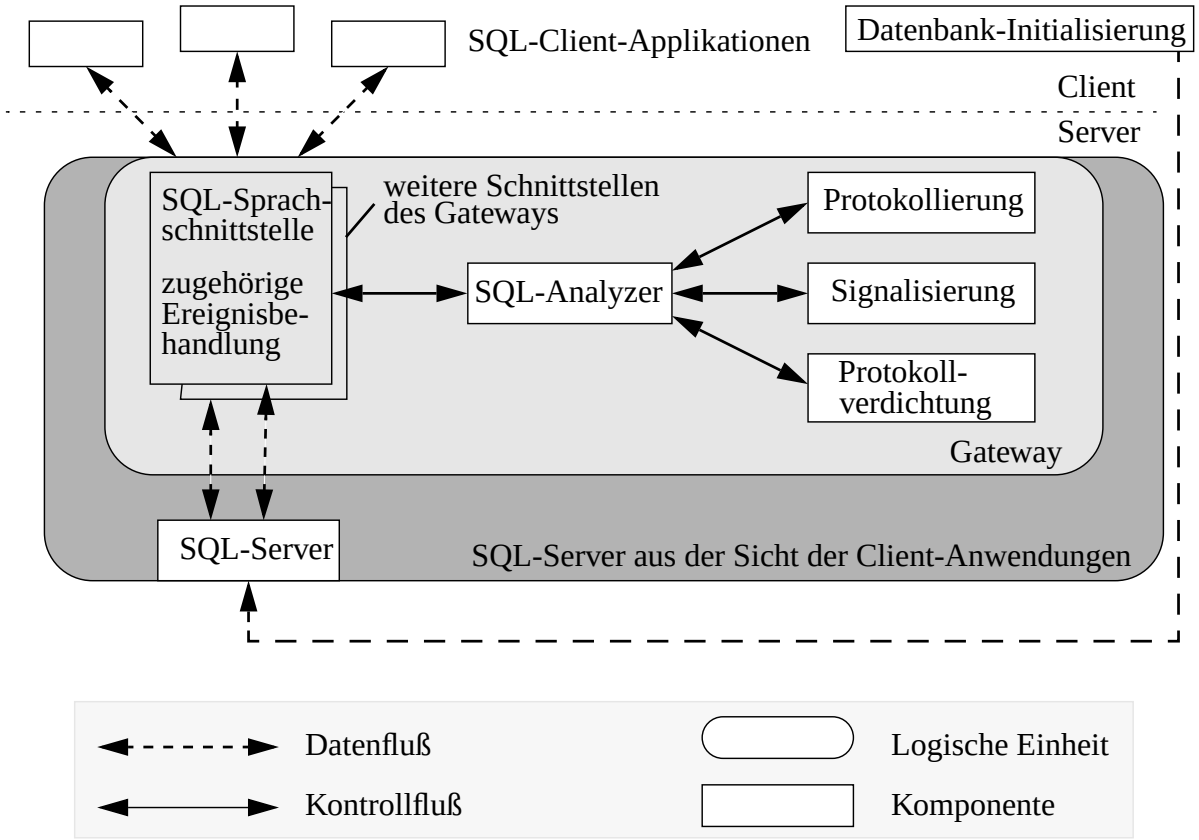


Abbildung 7.3: Architektur des Datenbank-Gateways

7.1.4 SQL-Syntaxanalyse

Zugrunde liegt der Sprachumfang von Transact-SQL, der SQL-Version, die vom Sybase-Datenbankserver verarbeitet wird. Zur Erstellung des SQL-Analyzers wurden die UNIX-Werkzeuge lex und yacc eingesetzt [LMB92].

Bei der Detektion bestimmter Benutzeraktionen ergaben sich einige Besonderheiten in der SQL-Syntaxanalyse: Da nicht alle Kommandos beobachtet werden sollten, wurde eine partielle Analyse angewandt, bei der bereits die lexikalische Analyse solche Befehle anhand der Schlüsselwörter erkennt und von einer weiteren syntaktischen Analyse ausschließt. Der SQL-Parser arbeitet in zwei Pässen: im ersten Paß werden die einzelnen Befehle eines Befehlsstapels (*Batch*) separiert, im zweiten Durchgang erfolgt dann die syntaktische Analyse der Einzelkommandos.

Im ersten Parserdurchgang wird anhand der Syntax ermittelt, ob es sich um ein DML- oder ein DDL-Kommando handelt und jedes Statement einer Batch separat als Element einer Warteschlange, `queue`, abgelegt. Da in SQL keine Terminatorsymbole für Kommandos existieren, ist die Zerlegung in Einzelbefehle notwendig. Die Verarbeitung der DDL-Befehle erfolgt bereits im ersten Durchgang (siehe Abschnitt 7.1.6).

Der analysierte Sprachumfang ist einfach durch eine Ergänzung der Grammatik-Regeln bis hin zu einer vollständigen Implementierung von Transact-SQL erweiterbar. Bei den derzeit nicht analysierten Kommandos handelt es sich im wesentlichen um administrative Kommandos zur Sicherung der Daten sowie zur Weitergabe von Rechten und Kommandos zum Einstellen von Optionen des DBMS. Da der Schwerpunkt dieser Arbeit auf globaler Konsistenzsicherung liegt, wurden vorrangig DML- und DDL-Kommandos behandelt. Das Gateway-Konzept bietet von seinem Prinzip her aber auch Möglichkeiten zur Behandlung aller Arten von lokalen SQL-Ereignissen. Die partielle Analyse hat zugleich den Vorteil, daß der SQL-Analyzer auch bei Versionswechseln des Datenbanksystems und damit verbundenen Erweiterungen von SQL wie gewohnt arbeitet.

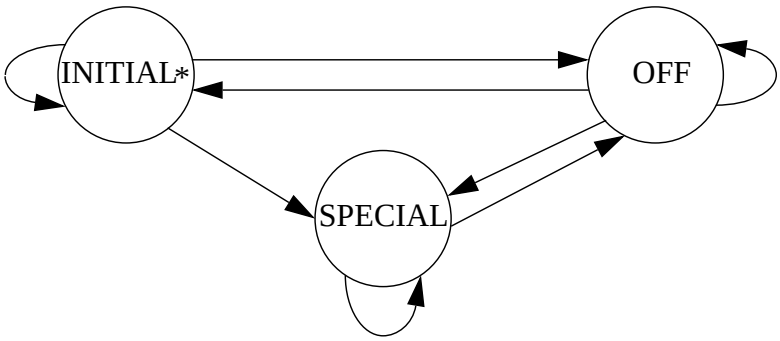
Die partielle lexikalische Analyse wird über mehrere Zustände gesteuert, so daß sich der Scanner durch einen Zustandsübergangsgraphen beschreiben läßt. In Abhängigkeit vom Zustand, in dem sich der Scanner befindet, werden beim Einlesen einer Zeichenfolge Token an den Syntax-Analyzer übergeben. Abbildung 7.4 zeigt die Zustände und deren Übergänge.

Die Befehlsbestandteile werden bei der syntaktischen Analyse im zweiten Parserdurchgang in Form eines Binärbaumes gespeichert, der bei einer *in-order*-Traversierung den Text des Kommandos wiedergibt.

7.1.5 Realisierung der Protokollierung von DML-Operationen

Zur intensionalen und extensionalen Protokollierung müssen aus den ursprünglich abgesetzten Kommandos zusätzliche Befehle generiert werden, bevor der modifizierte Kommandostapel an den SQL-Server übergeben wird.

Der Anstoß zu einer solchen Aktion kommt jeweils aus der syntaktischen Analyse der übergebenen Statements, von der aus eine Protokollierungsroutine, `Log` , aufgerufen wird, die als Argumente den Kommandotyp sowie einen Zeiger auf den zugehörigen Syntaxbaum erhält. Weitere zu protokollierende Informationen wie Login-Name und Client-Server-Identifikator lassen sich aus einer davor vorgesehenen Datenstruktur der Open Server-Bibliothek gewinnen (`SRV_PROC`), die für jeden Thread vom Server angelegt wird. Die Generierung des Operationszeitpunktes erfolgt durch Benutzung der Sybase-Systemfunktion `getdate()`, wobei der



INITIAL	Lesezustand, Umwandlung von Zeichenketten in Token und Weitergabe an die syntaktische Analyse (z.B. bei DML-Statements)
OFF	keine Umwandlung von Zeichenketten in Token (bei nicht zu detektierenden Statements)
SPECIAL	steht für mehrere Zustände, in denen keine Umwandlung von Zeichenketten in Token stattfindet, tritt typischerweise auf innerhalb von Kontrollflußanweisungen oder anderen Einschachtelungen / Klammerungen (z.B. zwischen BEGIN ... END, IF ... ELSE)

Abbildung 7.4: Zustandsübergangsdiagramm der lexikalischen Analyse

Zeitpunkt aufgezeichnet wird, der der tatsächlichen Ausführungszeit am nächsten kommt. Ein Identifikator wird vom System generiert, so daß eine Ordnung der Einträge hergestellt werden kann. Unter Verwendung all dieser Informationen wird eine Zeichenkette aufgebaut, die in den vorhandenen Kommandostapel eingefügt wird. Tabelle 7.1 zeigt den Aufbau der Tabelle zur Protokollierung der Kommandotexte.

Spaltenname	st_text	st_time	st_usr
Inhalt der Spalte	Text des Kommandos	Zeitpunkt der Operation	Login-Name des Benutzers

Tabelle 7.1: Intensionales Protokoll: `statement_log`

7.1.5.1 DML-Kommandos

Für jede zu beobachtende Tabelle wird eine Protokolltabelle angelegt, deren Struktur aus der Originaltabelle abgeleitet wird. Der Name des Protokolls wird aus dem der Tabelle gebildet, ergänzt um das Suffix `_log`. Tabelle 7.2 veranschaulicht die Struktur dieser Log-Tabelle.

Name	<*col 1*>	<*col 2*>	. . .	<*col n*>	mod_time	mod_way	mod_thr	mod_usr	mod_id
Inhalt	benutzerdefinierte Semantik (Spalten der Originaltabelle)				Zeitpunkt	Art der Änderung	C/S-Verbindung	Login-Name	Identifikator

Tabelle 7.2: Log-Tabelle zur Protokollierung von modifizierten Tupeln

Im einzelnen werden folgende Aktionen durchgeführt, bevor die tatsächlich vom Benutzer initiierte Operation zur Ausführung kommt: Im Falle eines DELETE wird jedes Tupel, das von der Löschung betroffen ist, mit der Operationsart ‘DEL’ protokolliert. Handelt es sich um ein INSERT, werden die neu einzufügenden Tupel als Duplikate ebenfalls in der Log-Tabelle eingetragen mit dem Kürzel ‘INS’. Falls ein UPDATE abgesetzt wurde, sind mehrere Maßnahmen erforderlich. Alle Tupel, die von der Änderung betroffen sind, werden mit der Kennung

‘UDE’ (***Update De****lete*) in die Protokolltabelle geschrieben. Die geänderten Tupel werden anschließend mit der Kodierung ‘UIN’ (***Update In****sert*) in die Protokolltabelle übertragen. Die Protokolldatei enthält somit für jede Änderungsoperation ein *Before Image* als auch ein *After Image*.

Bei der Markierung der Änderungsarten wird also unterschieden zwischen solchen Tupeln, die tatsächlich gelöscht oder neu eingefügt wurden, und solchen, die durch ein UPDATE “quasi entfernt” bzw. “neu eingefügt” wurden. Damit wird die Sichtweise der Tupel als Objekte unterstützt, die auch nach einer Änderung ihrer Attribute existieren, bei einem DELETE aber nicht nur ihren Inhalt, sondern auch ihre Identität verlieren. Diese Unterscheidung wird ebenso bei der Verdichtung des Protokolls bei der Interpretation der Änderungscodes beachtet. Alle Protokollierungsaktionen sind vor der eigentlichen durch den Benutzer beabsichtigten Aktion durchzuführen. Die Notwendigkeit ergibt sich daraus, daß die zu manipulierenden Daten nach Durchführung der DML-Operation nicht mehr oder nur noch verändert vorhanden sind. Tabelle 7.3 zeigt jeweils die aus den Originalkommandos generierten Befehlsfolgen.

Typ	Originalbefehl	Generierte Kommandofolge

SELECT	`select ...`
	`from ...`
	`where`
		`insert statement_log values`
		`('select ... from ... where ...',`
		`...)`
		`select ... from ... where ...`

INSERT	`insert <`*table*`>`
	`values (<`*values_list*`>)`
		`insert statement_log values`
		`('insert <`*table*`> values ...', ...)`
		`insert <`*table*`>_log`
		`values (<`*values_list*`>, 'INS', ...)`
		`insert <`*table*`>`
		`values (<`*values_list*`>)`

	`insert <`*table*`>`
	`<`*subquery*`>`
		`insert statement_log values`
		`('insert <`*table*`> <`*subquery*`>', ...)`
		`insert <`*table*`>_log`
		`<`*modified_subquery*`>`
		`insert` *table* `<`*subquery*`>`

UPDATE	`update <`*table*`>`
	`set <`*assignment_list*`>`
	`where <`*condition*`>`
		`insert statement_log values`
		`('update <`*table*`> set ... ', ...)`
		`insert <`*table*`>_tpl`
		`select * from <`*table*`> where <`*condition*`>`
		`insert <`*table*`>_log`
		`select *, 'UDE', ...`
		`from <`*table*`>_tpl`
		`update <`*table*`>_tpl set <`*assignment_list*`>`
		`insert <`*table*`>_log`
		`select *, 'UIN', ...`
		`from <`*table*`>_tpl`
		`delete <`*table*`>_tpl`
		`update <`*table*`> set <`*assignment_list*`>`
		`where <`*condition*`>`

Tabelle 7.3: Protokollierung von DML-Kommandos

Typ	Originalbefehl	Generierte Kommandofolge

DELETE	`delete <table> [where <condition>]`	`insert statement_log values ('delete <table> [where ... ]', ...)`
		`insert <table>_log select *, 'DEL', ... from <table> [where <condition>]`
		`delete <table> [where <condition>]`

Tabelle 7.3: Protokollierung von DML-Kommandos (Forts.)

Das SELECT-Kommando verursacht den geringsten Aufwand bei der Protokollierung, weil nur der Befehlstext in der Tabelle `statement_log` gespeichert wird und keine Datenmanipulation zu erfassen ist. Diese Befehlstext-Speicherung erfolgt auch für alle weiteren kritischen Kommandos. Im Falle der datenmanipulierenden Befehle DELETE, INSERT und UPDATE sind neben der Protokollierung des Kommandotextes bei der extensionalen Protokollierung weitere Aktionen zu veranlassen, die je nach Kommando die alten und neuen Zustände der betroffenen Tupel speichern.

Beim Löschen von Daten wird der Protokollierungsbefehl vor dem eigentlichen DELETE ausgeführt. Die Auswahl der als gelöscht zu protokollierenden Tupel erfolgt über die WHERE-Klausel des originalen DELETE-Befehls.

Sollen Tupel eingefügt werden, so werden sie auch in die Protokolltabelle geschrieben, jeweils ergänzt um die ‘INS’-Kennung für neue Tupel bzw. alle weiteren zu erfassenden Informationen (vgl. Tabelle 7.2).

Bei der Protokollierung von UPDATEs sind für die Speicherung von Before Images und After Images mehrere Schritte notwendig. In einer temporären Log-Tabelle (Suffix `_tpl`) werden die zu ändernden Tupel zwischengespeichert, anschließend protokolliert und auch dort geändert. Somit können die Tupel aus der temporären Tabelle anschließend als After Images (mit dem Kürzel ‘UIN’) in die Protokolltabelle geschrieben werden. Nach dem Löschen der temporären Tabelle erfolgt zum Schluß die Ausführung des originalen UPDATE-Befehls.

7.1.5.2 Transaktionskommandos

Da die Semantik lokaler Transaktionen berücksichtigt werden muß, sind weitere Schritte bei der Protokollierung notwendig. Prinzipiell muß verhindert werden, daß temporäre Zwischenzustände sichtbar werden. Zu diesem Zweck werden Transaktionskommandos ebenfalls in einer separaten Protokolltabelle (`transaction_log`) abgelegt. Diese besteht aus: Kürzel, Zeitpunkt, Benutzernamen, der Nummer der Client-Server-Verbindung sowie einem Identifikator. Im Kürzel werden folgende Operationen erfaßt: ‘BOT’ (***B**egin **o**f **T**ransaction*), ‘COT’ (***Co**mmit **T**ransaction*) und ‘ROT’ (***Ro**llback **T**ransaction*). Im Modus *chained* (nach ANSI-SQL), in dem eine Transaktion nicht explizit durch ein BEGIN-Statement eingeleitet wird, sondern implizit durch einen Datenbankzugriff, wird ebenfalls ein ‘BOT’-Eintrag geschrieben. Tabelle 7.4 gibt den Aufbau des Transaktionsprotokolls wieder.

Spaltenname	tr_stat	tr_time	tr_thr	tr_usr	tr_id
Inhalt der Spalte	Transaktions-kommando	Zeitpunkt	ID der C/S-Verbindung	Login-Name	Identifikator

Tabelle 7.4: Protokolltabelle für Transaktionskommandos `transaction_log`

Das Transaktions-Log dient zur Erstellung eines endgültigen, verdichteten Modifikationsprotokolls, welches Transaktionsgrenzen und damit die Sichtbarkeit von Datenänderungen berücksichtigt. Die Zusammenfassung der Log-Einträge ermöglicht, daß bei einer Auswertung des Protokolls zu einem späteren Zeitpunkt keine Zustände sichtbar werden, die nur vorübergehend innerhalb von Transaktionen gültig sind. Änderungen außerhalb von Transaktionen werden unverändert ins verdichtete Protokoll übernommen. Vom Benutzer zurückgesetzte Transaktionen werden nicht protokolliert, da die Protokollierung innerhalb der Originaltransaktion stattfindet. Bei Detektion von Transaktionsbeginn oder Transaktionsende werden Kommandos in die Batch eingefügt, die die Protokollverdichtung sowie die Aktualisierung der Tabelle `transaction_log` anstoßen. Tabelle 7.5 skizziert die Umsetzung von Transaktionskommandos in Protokollaktionen.

Kommando	Generierte Kommandofolge
`begin tran`	`begin tran` `insert into transaction_log values ('BOT', ....)`
`commit tran`	`insert into transaction_log values ('COT', ...)` `commit tran`
`rollback work`	`insert into transaction_log values ('ROT', ...)` `rollback work`

Tabelle 7.5: Protokollierung von Transaktionskommandos

7.1.5.3 Protokollverdichtung

Die anzuwendenden Verdichtungsregeln sind mit denen bei Refresh-Verfahren für *Differential Snapshots* zu vergleichen [KR87]. Im Gegensatz zu diesem Algorithmus bilden hier jedoch Transaktionen das Granulat der Verdichtung. Tabelle 7.6 veranschaulicht die Regeln, nach denen aus einem vorhandenen Änderungseintrag und einem folgenden Änderungseintrag mit höherem Zeitstempel ein neuer Eintrag im verdichteten Protokoll generiert wird.

In Abweichung von den Regeln in [KR87] wird ein DELETE-Eintrag mit einem nachfolgenden INSERT-Eintrag nicht zu einem UPDATE-Eintrag verschmolzen, da in der objektorientierten Sichtweise das Löschen eines Objekts und die anschließende Erzeugung (mit neuer Objektidentität) nicht als Modifikation des alten Objekts angesehen wird. Somit kann ein vorhandener DELETE-Eintrag nicht weiter verdichtet werden.

folgender Log-Eintrag	vorhandener Log-Eintrag			
	kein Eintrag	DELETE	INSERT	UPDATE
DELETE	DELETE	-	kein Eintrag	DELETE
INSERT	INSERT	-	-	-
UPDATE	UPDATE	-	INSERT	UPDATE

Tabelle 7.6: Regeln zur Verdichtung des Protokolls

Für die Übernahme der Zeitpunkte t in das verdichtete Protokoll wurden folgende Regeln festgelegt:

vorhander Eintrag	folgender Eintrag	Zeitpunkt im verdichteten Eintrag
UPDATE	DELETE	t(DELETE)
UPDATE	UPDATE	UDE: t(UPDATE 1) UIN: t(UPDATE 2)
INSERT	UPDATE	t(INSERT)

Tabelle 7.7: Zeitpunkte bei der Protokollverdichtung

Im ersten und dritten Fall wird jeweils der Zeitpunkt der resultierenden Operation übernommen. Für zwei aufeinanderfolgende UPDATEs gilt, daß für den Eintrag 'UDE' der Zeitpunkt des ersten, für den Eintrag 'UIN' der Zeitpunkt des letzten UPDATE im verdichteten Protokoll verwendet wird, da das Datentupel zum ersten Zeitpunkt seinen alten Wert verloren hat und erst ab dem letzten Zeitpunkt tatsächlich den neuen Wert hat.

Die Struktur des verdichteten Protokolls entspricht derjenigen der Log-Tabellen (vgl. Tabelle 7.2), mit Ausnahme der Client-Server-Verbindung, da diese Information nur zur Verdichtung benötigt wird.

Die Verdichtung des Protokolls erfolgt in vorübersetzten SQL Stored Procedures. Diese Lösung wurde aus Performance-Gründen gewählt, um die Interpretation der umfassenden Befehlsfolge des Verdichtungsalgorithmus zu vermeiden. Von den SQL-Prozeduren wird je eine für jede durch das Gateway überwachte Tabelle benötigt (`csp_<`*tabelle*`>`) außerdem wird noch eine weitere Prozedur aufgerufen, die nicht mehr benötigte Einträge aus der Tabelle `transaction_log` entfernt (`csp_clean`).

Das sequentielle Abarbeiten des Protokolls erfolgt anhand der Tupel-Identifikatoren. In einer temporären Tabelle (Suffix `_tmp`) werden 'INS'-Einträge sowie 'UDE'- und korrespondierende 'UIN'-Einträge zwischengespeichert. Eine Zwischenspeicherung in Variablen wäre nicht sinnvoll, da die jeweils unterschiedliche Tabellenstruktur in ihnen berücksichtigt werden müßte. Für jede zu protokollierende Tabelle wird eine verdichtete Protokolltabelle gepflegt (Suffix `_cdl`, für *condensed logging table*).

Die Zusammengehörigkeit zweier Protokolleinträge wird über die Primärschlüssel der Objekte ermittelt. Durch die Primäschlüsseleigenschaft sind einige Kombinationen ausgeschlossen, so z.B. zwei aufeinanderfolgende Inserts desselben Tupels. Falls keine Primärschlüssel existieren, so erfolgt der Vergleich sämtlicher Werte der beiden Tupel (zwei NULL-Werte werden hierbei als "gleich" interpretiert).

Der Aufruf der Protokollverdichtungsprozeduren hängt vom Verhalten des Benutzers ab. Ihr Aufruf wird - wie die Protokollierungsaktionen - in den vorhandenen Kommandostapel eingesetzt. Es wurde berücksichtigt, daß der Aufruf an Transaktionsgrenzen erfolgt (nicht am Ende einer Batch, deren Ende nicht mit dem der Transaktion übereinstimmen muß). In der gegenwärtig realisierten Form erfolgt sogar eine Verdichtung vor und nach jeder Transaktion, um den Aufwand bei jedem Verdichtungsvorgang zu verringern. Kommandos, die nicht durch Transaktionen geklammert sind, werden spätestens beim Beenden der Client-Server-Verbindung in das verdichtete Protokoll übernommen. Denkbar ist auch eine Parametrisierung der Ereignisse, die eine Verdichtung aktivieren. Weitere Details der Verdichtungsalgorithmen können [Kra95] entnommen werden.

7.1.5.4 Protokollierung im Überblick

Abbildung 7.5 zeigt die Protokollierung der unterschiedlichen Ereignisse und die betroffenen Protokolltabellen im Zusammenhang; dabei wird die Protokollierung für eine einzige Tabelle, `tabelle1`, betrachtet.

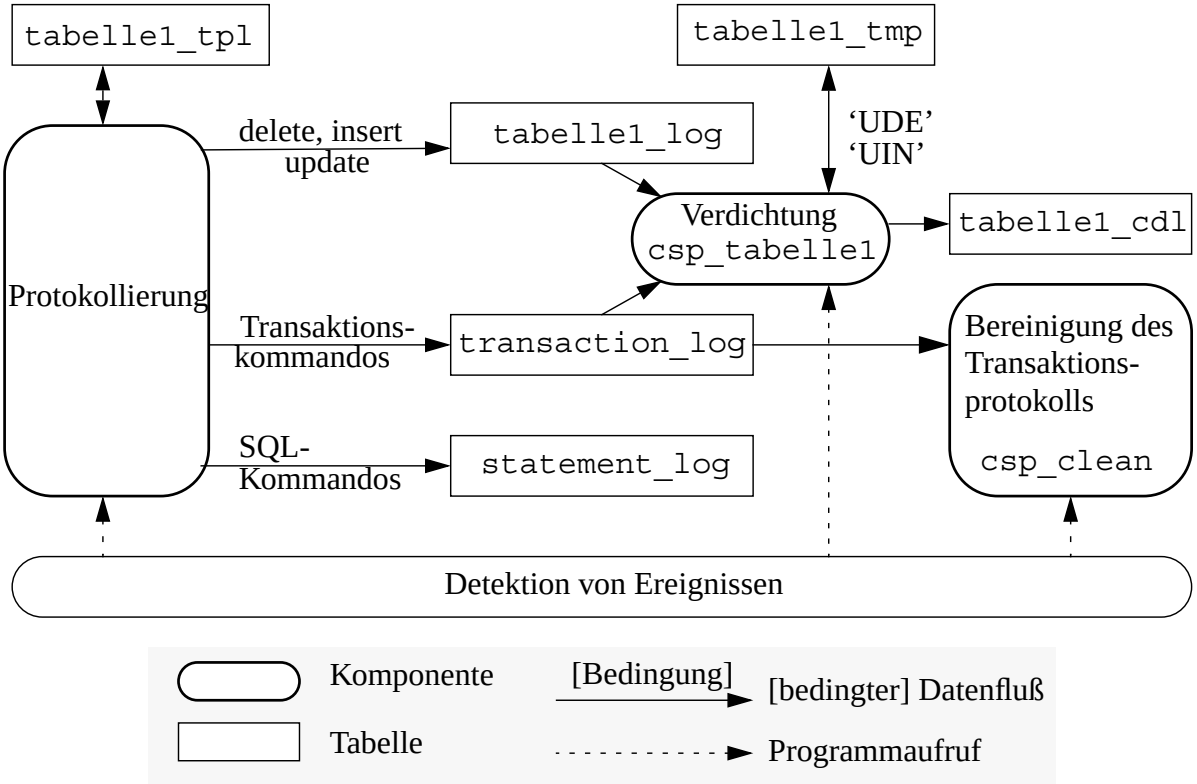


Abbildung 7.5: Zusammenhang von Protokollierungsaktionen und -tabellen

7.1.6 Behandlung von DDL-Befehlen

7.1.6.1 Extraktion relevanter Informationen

Im ersten Parserdurchgang werden die SQL-Befehle CREATE TABLE, DROP TABLE, ALTER TABLE, CREATE VIEW und DROP VIEW detektiert. Aus Kompatibilitätsgründen mit früheren DBMS-Versionen werden jedoch auch Sybase-Systemprozeduraufrufe zum Umbenennen von Datenbank-Objekten (`sp_rename`), zum Löschen von Schlüsseln (`sp_dropkey`) sowie zum Anlegen von Schlüsseln (`sp_commonkey`, `sp_foreignkey`, `sp_primarykey`) detektiert, deren Wirkung weitgehend den DDL-Befehlen in SQL entspricht [Rec96].

Als Ergebnis der SQL-Syntaxanalyse werden die Informationen in einer Operationsstruktur abgelegt, die eine Schemamodifikation geeignet beschreiben. Bei komplexen DDL-Statements (z.B. ALTER TABLE) wird eine Liste von Operationen erzeugt.

```
struct OpDDL {
    char *statement;
    char *database;
    int thread_id;
    int tran_level;
    char *table;
```

```
enum Operation OpCode;
parameter *par;
OpDDL *next }
```

Die Felder `thread_id` und `tran_level` dienen zur Identifizierung der eventauslösenden Transaktion, was für die Behandlung durch einen DDL-Vermittler wichtig ist. Aufbau und Arbeitsweise des DDL-Vermittlers sind in Abschnitt 8.7 beschrieben. Die betroffene Datenbanktabelle wird durch die Parameter `database` und `table` beschrieben.[9] Die Identifizierung der Operationen erfolgt durch den Operationscode `OpCode`. Verarbeitet werden die Statements, die bereits existierende Datenbank-Objekte modifizieren. Tabelle 7.8 gibt einen Überblick über die DDL-bezogenen Operationen:

OpCode	Beschreibung
TableRename	Umbenennen einer Tabelle
TableDrop	Löschen einer Tabelle
ColumnRename	Umbenennen einer Spalte
ColumnReplaceDefault	Ändern Defaultwert
PrimaryKeyCreate	Definition eines Primärschlüssels
StatementPrimaryKeyCreate	Definition eines Primärschlüssels und Erzeugen zusätzlicher Statements
ForeignKeyCreate	Definition eines Fremdschlüssels
StatementForeignKeyCreate	Definition eines Fremdschlüssels
ForeignKeyDrop	Löschen eines Fremdschlüssels
ViewDrop	Löschen einer Sicht
ConstraintDrop	Löschen eines Constraints (allgemein)

Tabelle 7.8: DDL-Operationscodes

Des weiteren werden im Operationscode auch Transaktionskommandos erfaßt, die vom DDL-Vermittler verarbeitet werden, um die lokale Transaktionssemantik zu wahren. Zur Behandlung von Prozeduraufrufen, deren Ausführung relevante Ereignisse beinhalten kann, wurden zusätzliche Operationscodes eingeführt.

Die Operationsbeschreibung kann mit einer Parameterliste verbunden sein in Abhängigkeit vom Operationscode. Durch die Parameter werden die vom DDL-Befehl betroffenen Datenbankobjekte näher charakterisiert (Name bzw. Typ).

7.1.6.2 Detektion “verborgener” Events

Ein Problem bei der Ereignisdetektion am Language Interface des Servers besteht darin, daß Events, die innerhalb des Datenbankservers ausgelöst werden, nicht erfaßt werden können. Für die Erkennung von Schemamodifikationen ist der Gebrauch von Triggern auf dem Data Dictionary in kommerziellen relationalen DBMS ausgeschlossen. Die im Gateway gewählte Lösung beruht auf einem Vergleich der Zustände des Data Dictionary bei Kommandos, in denen möglicherweise DDL-Operationen stattfinden.

Zur Detektion von Schemaveränderungen durch Prozeduren genügt ein Vergleich bestimmter Systemtabellen. Dies sind bei Sybase: `sysobjects`, `syscolumns` und `syskeys`.[10] Wenn

9 Zur Identifizierung des Datenbankobjekts muß in der Struktur noch der Eigentümer ergänzt werden.

bei der Analyse des Statements ein Prozeduraufruf erkannt wird, werden die Systemtabellen in einer Variablen vom Typ `s_tables` zwischengespeichert, der folgende Struktur aufweist:

```
struct s_tables {
    int count;
    char *thread;
    sysobjects *objects;
    syskeys *keys;
    syscolumns *columns }
```

Das Attribut `thread` dient dabei der Synchronisation des Zugriffs mehrerer Threads auf die Systemtabellen [Rec96]. Nachdem die Prozedur im Server ausgeführt wurde (spätestens am Transaktionsende), muß der aktuelle Stand der Systemtabellen mit dem gesicherten verglichen werden. Zu diesem Zweck wird der Inhalt der Systemtabellen in einer zweiten Variablen vom Typ `s_tables` gespeichert. Die Strukturen von `objects`, `keys` und `columns` entsprechen denen der korrespondierenden Sybase-Systemtabellen, wobei die Tupel in Form einer Liste, geordnet nach Schlüssel, gespeichert sind. Beim Vergleich der Listen läßt sich feststellen, ob Einträge verschwunden oder hinzugekommen sind. Als Ergebnis des Vergleichs von altem und neuem Zustand wird zu jedem Tupel eine Änderungsinformation gespeichert.

Folgende Veränderungen am Datenbankschema können detektiert werden: Datenbankobjekt-Modifikationen umfassen Umbenennen, Löschen und Hinzufügen von Tabellen und Views, die durch den Vergleich der Werte aus `s_tables.objects` bestimmt werden. Änderungsoperationen bei Schlüsseln sind das Entfernen und Hinzufügen, die durch den Vergleich von `s_tables.keys` erkannt werden. Eine Spalte kann in der verfügbaren Sybase-Version 11 nicht ergänzt oder entfernt, sondern nur umbenannt werden, was ggf. über die Veränderungen in `s_tables.columns` festgestellt werden kann.

7.1.6.3 Ausführung detektierter Operationen

Nach der Analyse der Batch wird die erzeugte Statement Queue abgearbeitet, bis sie leer ist. Handelt es sich um den Operationstyp DML, wird dieser an einen zweiten Parserdurchgang gereicht, da eine andere Weiterverarbeitung (Protokollierung, Signalisierung) erfolgt. Ansonsten werden folgende Schritte durchlaufen:

1. Zugriff auf Data Dictionary
In Abhängigkeit vom Operationscode werden fehlende Informationen aus dem Data Dictionary ergänzt. Das betrifft z.B. das Löschen von Constraints (`ConstraintDrop`), wofür die Art des Constraints aus der zugehörigen Systemtabelle bestimmt werden muß. Bei Prozeduraufrufen wird der alte Zustand der Systemtabellen gesichert (vgl. voriger Abschnitt).

2. Ausführung des Befehls am SQL-Server
Die Befehle werden einzeln ausgeführt, so daß eine Rückmeldung über deren erfolgreiche Ausführung eintrifft als Voraussetzung für die nachfolgenden Schritte.

3. Auswertung der Rückgabewerte
Nach erfolgreicher Ausführung des Statements werden zusätzliche Informationen aus dem Data Dictionary gewonnen, die für den DDL-Vermittler benötigt werden.

---

10 Aus Kompatibilitätsgrunden zum Persistence Dictionary Reader wird die Tabelle `syskeys` ausgewertet, bei einer Weiterentwicklung sollte jedoch mit der Systemtabelle `sysindexes` gearbeitet werden.

4. Anreicherung um zusätzliche Befehle
Es gibt Fälle, in denen die Batch um zusätzliche Statements angereichert wird, die in die Queue gestellt werden. Beim Löschen von Tabellen werden auch (falls vorhanden) die zugehörigen Log-Tabellen gelöscht, die vom Gateway gepflegt werden (siehe Abschnitt 7.1.5). Außerdem werden z. Zt. Statements ergänzt, die die Konsistenz zwischen den Sybase-Systemtabellen `sykeys` und `sysindexes` wahren [Rec96].

5. Ermittlung "verborgener" DDL-Events
Nach Prozeduraufrufen, die potentiell DDL-Events auslösen, werden die Zustände der Systemtabellen verglichen und ggf. eine Beschreibung der stattgefundenen Modifikationen erzeugt.

6. Senden der Informationen an den DDL-Vermittler
Die aufgetretenen DDL-Operationen werden dem DDL-Vermittler signalisiert, der die Konsistenz zwischen lokalen Schemata und dem globalen Schema überwacht (Details in Abschnitt 7.2.3).

7.1.7 Weitere Komponenten des Gateways

Initialisierung

Für die Inbetriebnahme des Gateways müssen alle benötigten Protokolltabellen für die zu überwachenden Tabellen in den betroffenen Datenbanken vorhanden sein. Außerdem müssen die SQL-Prozeduren zur Protokollverdichtung und zur Bereinigung des Transaktionsprotokolls erzeugt werden. Die Generierung all dieser Objekte erfolgt im Programm `loginit`. Beim Aufruf werden nur dann Objekte neu angelegt, wenn diese noch nicht vorhanden sind.

Konfiguration

Um eine höhere Flexibilität bei der Benutzung des Gateways zu erzielen, wurde eine Konfigurationsdatei `gateway.conf` vorgesehen, in der Festlegungen über die Objekte der Datenbank getroffen werden, auf denen Ereignisse zu protokollieren bzw. zu signalisieren sind. Das vorher erwähnte Programm `loginit` greift zur Erzeugung von Tabellen und Prozeduren ebenfalls auf diese Konfigurationsdatei zu. Die Konfigurationsdatei enthält eine Liste von Benutzern mit jeweils einer Liste von Datenbanken. Zu jeder Datenbank kann wiederum eine Liste von Tabellen angegeben, die überwacht werden sollen. Die Syntax der Konfigurationsdatei erlaubt in der Tabellenliste die Angabe von '`*`', um alle Benutzertabellen der Datenbank anzusprechen (die vollständige Syntax kann [Kra95] entnommen werden). Bei einer leeren Konfigurationsdatei werden keine lokalen Ereignisse protokolliert bzw. signalisiert.

Das Gateway als ein Open Server-Prozeß kann mit unterschiedlichen Optionen aufgerufen werden:
`ctosdemo [ -log | -stalog | -tablog ] [ -rpc <`*rechnername*`> ]`

`-log`	vollständiges Logging
`-stalog`	nur intensionale Protokollierung (`statement_log`)
`-tablog`	extensionale Protokollierung
`-rpc` <*rechnername*>	Signalisierung der Ereignisse als RPC zum Vermittler auf Rechner <*rechnername*> (siehe auch Abschnitt 7.2.4)

Die Auswahl des Server-Prozesses, an den die Anfragen zu richten sind, erfolgt durch Setzen einer Umgebungsvariable vor dem Starten der Clients.

7.1.8 Sonstige Aspekte

Die Ausführung der lokalen Befehle am Gateway erfolgt auf zweierlei Weise: durch eine direkte Weiterleitung an den lokalen Datenbankserver (bei Protokollierung oder Ausführung von DDL-Befehlen) oder durch eine Ausführung über einen DML-Vermittler (vgl. Abschnitt 8.2.2 auf Seite 149), der aber auch unabhängig davon Anfragen absetzen kann. Hierfür wurde eine Erweiterung der Protokolltabellen vorgesehen um eine Spalte für den Namen der Client-Applikation, die es erlaubt, den Ursprung einer Anfrage zu rekonstruieren.

Bei der Protokollierung werden die dafür notwendigen Kommandos innerhalb des Kommandostapels bzw. der Transaktion des Benutzers eingefügt, so daß keine Synchronisationsprobleme entstehen. Die Protokollierung einer vollständigen Zustandshistorie lokaler Datenbankobjekte bedeutet einen Verlust an Performance und sollte deshalb nur in ausgewählten Fällen zum Einsatz kommen (z.B. bei asynchroner Replikation). Somit bedeutet Einschränkung der lokalen Autonomie hier zweierlei: eine mögliche Verlängerung der Antwortzeiten sowie zusätzlicher Ressourcenverbrauch in der lokalen Datenbank für die Speicherung der Protokolle.

Die vorgestellte Gateway-Lösung hat einen Nachteil in bezug auf die Detektion von Datenbankereignissen, die durch Trigger oder innerhalb von Stored Procedures ausgelöst werden. Eine mögliche Lösung hierbei kann die Analyse des Quelltextes von Triggern und Stored Procedures bei deren Definition sein, um die erforderlichen Maßnahmen beim Aufruf bzw. im Fall von Triggern die auslösende DML-Operation festzustellen und diese Informationen zusätzlich zu verwalten und zur Laufzeit zu gebrauchen. Denkbar ist auch eine manuelle Pflege der Informationen über das (potentielle) Verhalten von Stored Procedures in einer Metadatenbasis. Außerdem verbleibt die Möglichkeit, Trigger für Detektion und Signalisierung von DML-Events ergänzend zu nutzen.

Die Konfigurationsmöglichkeiten des Gateways sind ausbaufähig, so z.B. die Angabe größerer Zeitgranulate für die Protokollverdichtung entsprechend den Benutzeranforderungen oder eine verfeinerte Spezifikation der gewünschten Protokollierung (pro Benutzertabelle).

7.1.9 Vergleich mit anderen Plattformen

Es ist zu untersuchen, ob die von uns prototypisch realisierte Lösung eines Eingriffs in die Client-Server-Kommunikation sich auch auf andere Datenbanksysteme uebertragen läßt:

In Ingres werden SQL-Anweisungen immer in der gleichen Weise an einen Server gesandt. Im lokalen Fall ist das der Datenbankserver, bei einer Client/Server-Trennung ist es der Kommunikations-Server (GCC). Die Umwandlung der *embedded* SQL-Anweisungen in entsprechende Bibliotheksaufrufe ist nicht sichtbar, d.h. es gibt kein unterstütztes API. Allerdings ist die Kommunikation zwischen den Ingres-Prozessen in der *Global Communication Architecture* (GCA) offengelegt. Somit ist es möglich, eine eigene Server-Klasse für einen “Zwischen-Server” zu definieren (so wie es Server-Klassen INGRES, STAR, COMSVR, etc. gibt) und die Datenbank-Verbindung über diese Server-Klasse zu realisieren. Dieser Zwischenserver ist ein echter Server, der in der Lage ist, mehrere Clients zu bedienen. Dann bleibt noch die Aufgabe, die eintreffenden Kommunikations-Pakete zu interpretieren, bei bestimmten Situationen die eigene Logik anzustoßen und in allen anderen Fällen das Paket an den Datenbank-Server bzw. in der Gegenrichtung an die Anwendung weiterzuleiten.

Informix bietet eine Möglichkeit des Eingriffs, die allerdings nicht offiziell dokumentiert ist. Bei der Konfiguration besteht die Möglichkeit, einen eigenen Zwischen-Server als Datenbank-Server anzugeben, der nach einer Behandlung der Daten diese an den Datenbank-Server weiterleitet. Das Protokoll zwischen Client und Server ist 2-Byte-wortorientiert und synchron mit fester oder variabler Nachrichtenlänge (maximal 64 K).

In Oracle gibt es das SQL*Net Utility, um Verbindungen vom Client zum Server herzustellen. SQL*Net ist eine Familie von Netzwerk-Protokollen, die transparent verschiedene Clients, Servers und Gateways integriert, um eine einheitliche Infrastruktur zu konstruieren. Zwar ist es nicht möglich, die Datenpakete innerhalb der SQL*Net Architektur abzufangen, aber man kann eine Middleware-Schicht schreiben zwischen Client-Programm und dem Client API, welches die Schnittstelle zu SQL*Net bildet. Als API stehen entweder Präcompiler zur Auswahl (z.B. Pro*C) oder das Oracle Call Interface (OCI).

7.2 Integration der Eventsignalisierung in die Systemumgebung

7.2.1 Remote Procedure Call im Überblick

Abbildung 7.6 veranschaulicht die Schritte bei einem Remote Procedure Call vom Aufruf des Client-Stubs, der über das Netzwerk mit einem Server-Prozeß kommuniziert, bis hin zur Rückgabe der Ergebnisse an den lokalen Client.

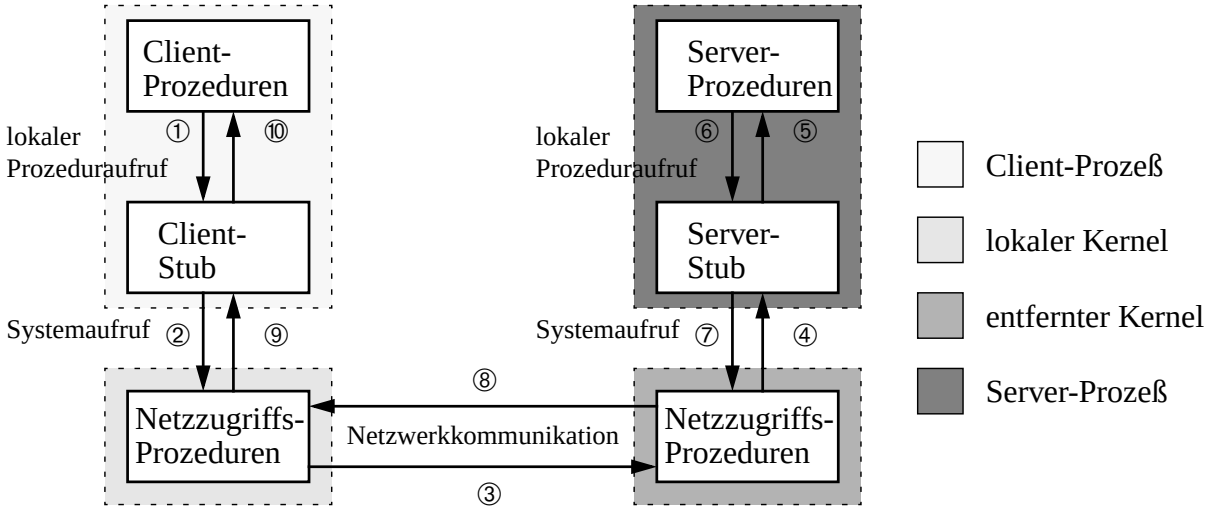


Abbildung 7.6: Modell eines Remote Procedure Call [Ste90]

Der Quasi-Industriestandard Sun RPC ist eine auf dem Industriestandard Open Network Computing - eXternal Data Representation (ONC XDR [Sun87]) aufbauende RPC-Implementation für eine verteilte UNIX-Umgebung. XDR beinhaltet Kodierungsregeln für die maschinenunabhängige Repräsentation von Daten und eine Beschreibungssprache, in der die benötigten Datentypen in C-ähnlicher Notation definiert werden können.

7.2.2 ExecDML: Die RPC-Schnittstelle zum DML-Vermittlerprozeß

Der Ablauf der Signalisierung von Benutzeraktionen an die zuständige Komponente des Vermittlersystems ähnelt zunächst dem der Generierung von Protokollierungsaktionen. Auch hier kommt der Anstoß zu einer Aktion aus der syntaktischen Analyse des übergebenen Statements. Von dort aus wird die Prozedur `Signal` aufgerufen, die als Parameter ebenfalls den Typ des Kommandos sowie einen Zeiger auf den zugehörigen Syntaxbaum erhält. Die Signalisierung erfolgt sofort durch den Aufruf einer entfernten Prozedur, für DML-Anweisungen die Prozedur `exec_dml`.

Um die Portabilität mit alten Versionen zu erhalten, nimmt die generierte RPC-Funktion nur einen Parameter als Argument, der die verschiedenen Informationen über das auszuführende DML-Kommando in den Komponenten einer Struktur vom Typ `ExecDMLParm` verpackt.

Die Aufrufparameterstruktur `ExecDMLParm` hat in jedem Fall zwei Komponenten: die Sybase Sitzungs-ID `session_id`, die das übergebene Kommando an die Datenbank abgeschickt hat, und die variante Struktur `op` vom Typ `OpUnion`. Diese gliedert sich in die feste Komponente `type`, die den Typ der auszuführenden Operation angibt und je nach dem Wert von `op.type` (d.h. für `OP_DELETE`, `OP_INSERT` und `OP_UPDATE`) noch eine Struktur vom Typ `StatementSpec`, in der die Parameter für das betreffende Kommando enthalten sind. Auszuführende Operationen können sein: `OP_BEGIN`, `OP_COMMIT`, `OP_CONNECT`, `OP_DELETE`, `OP_DISCONNECT`, `OP_INSERT`, `OP_ROLLBACK`, `OP_SELECT`, `OP_UPDATE`. Die Struktur `StatementSpec` besteht aus der Spezifikation `tuples` der zu selektierenden Tupel vom Typ `TuplesSpec` mit den Komponenten `tablename` (Tabellenname), `fromclause` (FROM-Klausel) und `whereclause` (Selektionsprädikat), jeweils als Strings. Außerdem (außer bei `OP_DELETE`) kann noch eine Liste von Attributnamen und auswertbaren SQL-Ausdrücken folgen, deren Werte den entsprechenden Spalten einer Datenbanktabelle zugewiesen werden, als variables Array `values[]` vom Typ `ValueSpec`. Diese wiederum gibt in ihrer integer-Komponente `values_len` die Anzahl der Wertzuweisungen an Tabellenspalten an und enthält die eigentlichen Werte in dem Array `values_val[]`, das die zwei Zeichenketten `colname` und `expression` mit den Spaltennamen und dem eigentlichen SQL-Wertausdruck enthält. Abbildung 7.7 zeigt die Datenstruktur der RPC-Parameter noch einmal im Überblick.

Zu beachten ist, daß neben den Benutzerkommandos, die auch protokolliert werden, bei der Signalisierung noch die Ereignisse “Aufnahme Client-Server-Verbindung” (`OP_CONNECT`) und “Beendigung Client-Server-Verbindung” (`OP_DISCONNECT`) signalisiert werden. Diese werden auf einer anderen Ebene des Gateways unterhalb der SQL-Sprachschnittstelle in eigenen Ereignisbehandlungsroutinen (*Connect Handler* bzw. *Disconnect Handler*) separat behandelt (siehe auch [Kra95]).

Die Resultatwertstruktur `ExecDMLResVal` hat drei Komponenten: den Fehlercode (“eigentlicher” Rückgabewert) in `resval`, ggf. die zum Fehlercode gehörende Fehlermeldung im Klartext in der Komponente `description` sowie die Anzahl der von der aktuellen Anweisung betroffenen Tupel in der Komponente `numrows`. Im Fall eines SELECT-Befehls enthält `description` die Ergebnismenge (mit Spalten- und Tupelseparatoren) im Zeichenkettenformat, die dann vom Gateway vor der Rückgabe an den aufrufenden Client aufbereitet werden muß (vgl. auch Abschnitt 8.2.2 auf Seite 149).

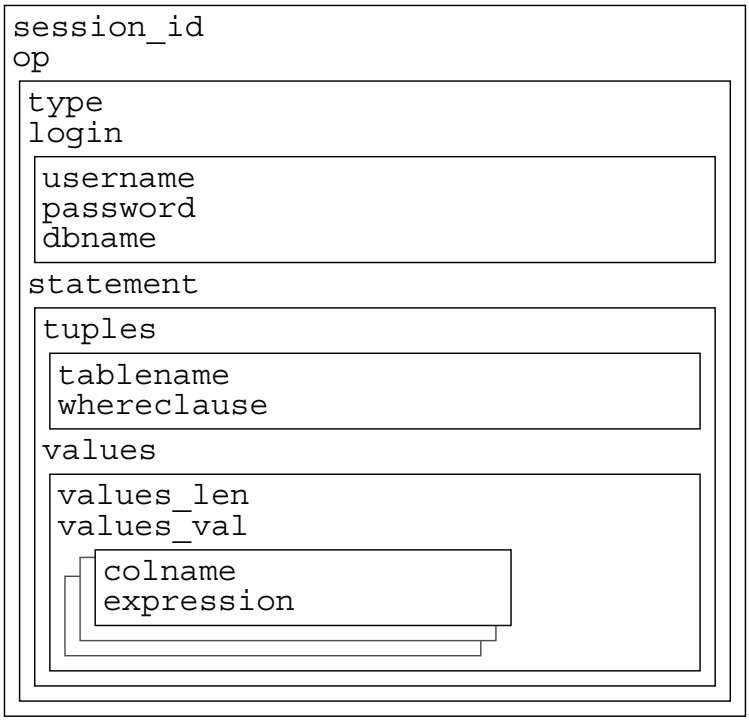


Abbildung 7.7: Struktur der RPC-Parameter von `ExecDMLParm`

7.2.3 ExecDDL: Signalisierung lokaler Schemaveränderungen

Die Signalisierung von DDL-Anweisungen an den dafür zuständigen DDL-Vermittlerprozeß erfolgt, in Analogie zu DML-Befehlen, durch Aufruf der entfernten Prozedur `exec_ddl`. Die Parameter für diese Prozedur sind eine transformierte Darstellung der `OpDDL`-Struktur (vgl. Seite 127). Die von der Operation betroffene Tabelle ist in `table` abgelegt, betroffene Spalten als Strings (ggf. weitere Tabellen) in `par`. Die RPC-Parameterstruktur beim Aufruf von `exec_ddl` sieht somit folgendermaßen aus:

```
struct OpRPC {
    int OpCode;
    int tran_level;
    int thread_id;
    char *database;
    char *table;
    char *par; }
```

Beispiel 7.1: (Transformation von DDL-Statements in RPC-Parameter)
An einem einfachen Beispiel soll noch einmal die Verarbeitung eines DDL-Statements im Gateway bis zur Aufbereitung für die Signalisierung demonstriert werden: Zunächst wird in der Tabelle `ort` die Spalte `plz` in `zip` umbenannt, was in Sybase durch Verwendung der Systemprozedur `sp_rename` erfolgen kann. Anschließend wird die Tabelle `student` mittels DROP TABLE gelöscht.

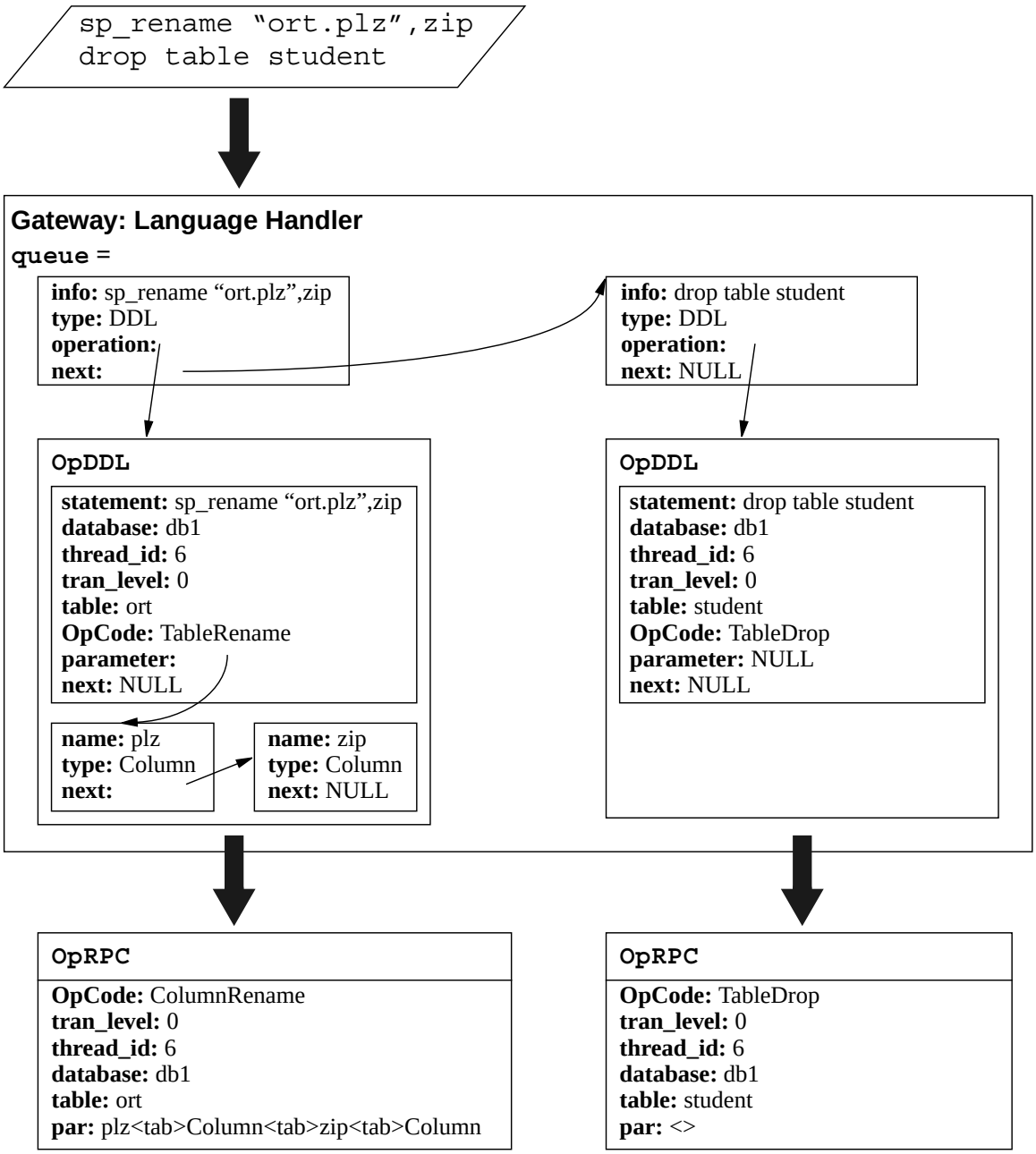


Abbildung 7.8: Transformation von DDL-Statements in RPC-Datenstrukturen (Beispiel)

7.2.4 Generierung eines RPC-Servers

Sun RPC besteht aus zwei Komponenten [Ste90]:

- `rpcgen`, dem RPC-Protokoll-Compiler, der aus der Definition der RPC-Aufrufschnittstelle (Remote Procedure Interface) in XDR-Notation C-Quelltexte für die Client- und Server-Stubs generiert sowie
- `portmap`, dem Port-Mapper, einem Hintergrundprozeß, der die Zuordnung der auf einem Rechner verfügbaren RPC-Prozeduren zu den Netzwerk-Ports des TCP/IP-Protokolls steuert.

Abbildung 7.9 veranschaulicht am Beispiel von `exec_dml`, welche Dateien aus der XDR-RPC-Protokollspezifikation (vgl. Abschnitt 7.2.2) generiert werden.

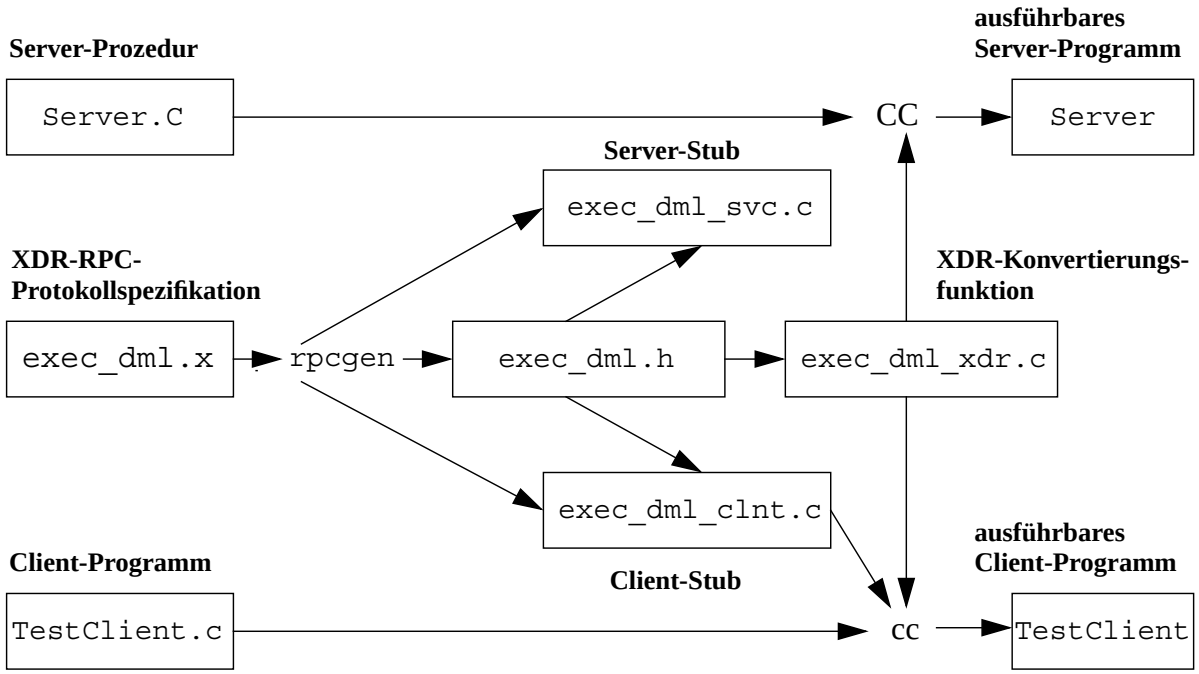


Abbildung 7.9: Dateien bei der Generierung eines Sun-RPC-Programms (Beispiel)

Der `rpcgen`-Compiler erzeugt aus der Datei `exec_dml.x` drei weitere C-Quelltextdateien: den Client-Stub `exec_dml_clnt.c`, den Server-Stub `exec_dml_svc.c` sowie Hilfsfunktionen für die Datenkonversion und den -transport gemäß dem XDR-Standard in der Datei `exec_dml_xdr.c`. Der Client-Stub wird zusammen mit dem Hauptprogramm für den Client und der XDR-Unterstützung zum ausführbaren Programm `TestClient` gebunden; der Server-Stub bildet zusammen mit der Server-Prozedur und den XDR-Routinen das Server-Programm `Server`.

Um das RPC-Programm zu starten, muß auf dem entfernten System der Port-Mapper als Hintergrundprozeß laufen. Der Serverprozeß kontaktiert nach seinem Start den Port-Mapper, um sich dort mit Programmnummer und -version zu registrieren und einen Netzwerkport für die Ein-/Ausgabe der Datenpakete zugewiesen zu bekommen.

Kapitel 8

Implementierung eines aktiven heterogenen Systems

In diesem Kapitel wollen wir die Realisierung eines aktiven Vermittlersystems beschreiben sowie die Probleme, die in deren Verlauf zu lösen waren. Begrifflich unterschieden werden DML-Vermittler (Behandlung globaler Integritätsbedingungen) und DDL-Vermittler (Behandlung von globaler Schemakonsistenz), jeweils benannt nach dem Typ von Ereignissen, die die Konsistenzverletzung auslösen.

Grundlage der Implementierung bildet das objekt-relationale Integrationswerkzeug Persistence, dessen Hauptmerkmale in Abschnitt 8.1 beschrieben werden. Zum Verständnis der entwickelten Komponenten des Vermittlersystem wird die zugrundeliegende Zugriffsschnittstelle skizziert, die von Persistence bereitgestellt wird.

Das Prinzip des DML-Vermittlers besteht darin, lokal auftretende Events als Methodenaufrufe auf der globalen Ebene zu interpretieren und gegebenenfalls an die Regelverarbeitung weiterzuleiten. Eine Lösung für die Abbildung von dynamischen SQL-Anweisungen in statische Methodenaufrufe (mit bekannten Namen und Parametern) in Persistence wird in Abschnitt 8.2 vorgestellt. Diese beruht auf der Analyse der Informationen des globalen Schemas, wobei diese Lösung für ein beliebiges Schema verallgemeinert werden kann, so daß daraus ein schemaspezifischer Vermittlerprozeß generiert werden kann. Die Implementierung eines solchen Generators im Detail ist in Abschnitt 8.3 beschrieben.

Die Darstellung der aktiven Komponente des Vermittlers erfolgt in Abschnitt 8.4. Zunächst werden die Arten von globalen Events genannt, die eine Regel auslösen können. Dazu zählen im wesentlichen Datenbank-Events, Transaktions- und Connect-Events sowie Time-Events. Anschließend wird die Repräsentation der ECA-Regeln in Persistence erörtert. Entsprechend dem in Kapitel 5 eingeführten Verarbeitungsmodell werden dabei auch Kopplungsmodi berücksichtigt, die die Bestandteile einer ECA-Regel voneinander entkoppeln (deferred, detached). Die Verwaltung von Ereignissen und Regeln erfolgt durch die Komponente MERU (**ME**diator's **RU**le System), deren Architektur nachfolgend dargestellt ist. Um auch zeitbehaftete Integritätsbedingungen auszudrücken, wurde das aktive System um eine temporale Komponente erweitert, die es gestattet, zeitabhängig Konsistenzprüfungen anzustoßen. Eine Implementierung beschreibt Abschnitt 8.4.5. Den Abschluß von Abschnitt 8.4 bildet eine Bewertung und Einordnung des implementierten Regelsystems. Dabei wird der im 3. Kapitel verwendete Kriterienkatalog zugrunde gelegt.

Die in Abschnitt 8.5 beschriebene Komponente, die eine asynchrone Replikationskontrolle durch den Vermittler ermöglicht, kann in zweierlei Hinsicht auch als eine Anwendung anderer Komponenten des Vermittlersystems angesehen werden. Zum einen werden die Informationen zum Abgleich der Daten aus Protokollen gewonnen, die durch das Datenbank-Gateway produziert werden (vgl. Abschnitt 7.1.5). Zum anderen läßt sich die Ausführung der Replikation durch ECA-Regeln spezifizieren.

Die Spezifikation von globalen Constraints wird durch die Sprache GISpeL (**G**lobal **I**ntegrity **Spe**cification **L**anguage) erleichtert, deren Aufbau in Abschnitt 8.6 beschrieben ist. Sie bildet zugleich die Grundlage für eine Benutzerschnittstelle, die es ermöglicht, aus einer Beschreibung heterogener Datenbanken und der Constraints zwischen ihnen automatisch den benötigten DML-Vermittler und die zugehörigen ECA-Regeln (C^{++}-Code und Datenbankobjekte) aus Templates zu generieren sowie die Gateway-Konfiguration anzupassen.

Die Wirkungsweise des DDL-Vermittlers wird am Ende dieses Kapitels in Abschnitt 8.7 beschrieben. Das Prinzip besteht darin, Modifikationsoperationen am lokalen Schema, die vom Datenbank-Gateway signalisiert werden, in Veränderungen des globalen Schemas umzuwandeln. Dieses Schema bildet die Grundlage für die eigentliche Anpassung von betroffenen Schemata und Applikationen, was im allgemeinen auch eine Neuübersetzung erforderlich macht.

8.1 Das Produkt Persistence als Plattform

8.1.1 Das Objektmodell von Persistence

Das Objektmodell von Persistence kann im wesentlichen als ein um das Konzept der Vererbung erweitertes “objektorientiertes” Entity-Relationship-Modell angesehen werden. Ein Schema wird in Gestalt einer formatierten Textdatei gespeichert, nach dem Dateinamens-Suffix auch als `.persist`-Datei bezeichnet.

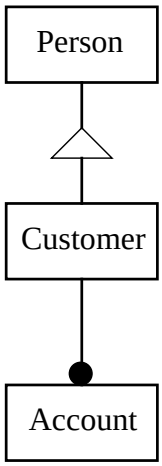
Die Basiskonstrukte des Persistence-Objektmodells sind:

- Klassen und Objekte
  Klassen sind der grundlegende Baustein sowohl im Objektmodell wie auch in der daraus generierten C^{++}-Schnittstelle. Sie repräsentieren die strukturellen Eigenschaften einer Menge von gleichartigen Objekten, die als Attribute und Methoden der Klasse zugeordnet werden. Objekte sind Instanzen einer Klasse. Als solche entsprechen sie Tupeln der Tabelle, auf die die Klasse abgebildet wird.

- Vererbung
  Durch Vererbung können Klassenhierarchien zur Darstellung von Generalisierungs- bzw. Spezialisierungsbeziehungen im Objektmodell gebildet werden. Subklassen “erben” als Spezialisierungen sämtliche auf ihren Superklassen definierten Attribute und Methoden. Persistence bildet nur die Blattklasse des Vererbungsbaums auf Datenbanktabellen ab; daher können auch nur neue Instanzen von diesen angelegt werden. Die einer Blattklasse zugeordnete Tabelle enthält zusätzlich zu ihren eigenen Attributen alle Attribute ihrer Oberklassen als Spalten, auch Beziehungen werden weitervererbt. Durch diese horizontale Partitionierung wird die Zahl der Tabellen minimiert und der Zugriff auf Instanzen der Blattklassen vereinfacht. Nachteilig ist, daß bei (gewöhnlich selteneren) Queries über Superklassen die Abfrage für jede Subklasse repliziert werden muß.

- Beziehungen
  Durch eine Beziehung (*Relationship*) werden Objekte einer anderen Klasse referenziert. Relationships werden wie (potentiell mengenwertige) Attribute vom Typ der Zielklasse behandelt und ermöglichen es somit, Objekte beliebiger Komplexität zu erzeugen (Aggregation). Persistence bietet spezielle Methoden zur Verwaltung dieser Beziehungen (Erstellen, Löschen bzw. Hinzufügen zu einer Beziehung, Objektzugriff über eine Beziehung). Diese Methoden sichern gleichzeitig die referentielle Integrität (keine *Dangling References*) und korrekte Kardinalität der Beziehungen. Darüber hinaus lassen sich sogenannte *Delete Actions* festlegen, wenn Objekte gelöscht werden sollen, die an einer Beziehung beteiligt sind. Diese entsprechen von ihrer Semantik her denen in SQL-92: *Block* (Blockieren), *Propagate* (Löschen) und *Remove* (NULL-Setzen der Beziehung). Beziehungen werden in relationalen Datenbanken durch Fremdschlüsselattribute dargestellt.

Beispiel 8.1: (Schemadefinition in Persistence)
Abbildung 8.1 veranschaulicht am Beispiel einer 1:n-Beziehung zwischen Kunden (`Customer`) als Subklasse von Personen (`Person`) und Konto (`Account`) die objektorientierte Modellierung und Repräsentation im Schema-File.




```
PersistentClass Person
{
  Key
  {
    int pers_no;
  }
  char name[40],
    Queries value;
  (...)
}

PersistentClass Customer:Person
{
  int cust_id,
    Queries value;
  int account_no,
    Queries value;

  ForeignKeyTo Account Through
  rel_account
  {
    Account_no MapsTo number
  }
}
```

```
PersistentClass Account
{
  Key
  {   int number;
  }
  PS_Money balance;
  (...)
}


Relationship
{
  Customer accounts
  min 0 max m Account;
  Account owner
  min 1 max 1 Customer;
}
```

Abbildung 8.1: Schema und Ausschnitt aus zugehöriger `.persist`-Datei

8.1.2 Dictionary Reader

Persistence wurde ursprünglich entworfen für eine top-down Vorgehensweise von der Definition eines OO Schemas bis zur Erzeugung relationaler Datenbanken. Für die beabsichtigte Integration von Legacy-Systemen erwiesen sich jedoch die Dictionary Reader zur Integration bestehender relationaler Datenbanken als wichtige Werkzeuge zum strukturellen Reverse Engineering.

Das Prinzip soll am Beispiel des Sybase Readers skizziert werden: Der Dictionary Reader filtert Informationen aus den Systemtabellen heraus und erstellt daraus ein OO Schema in einer Schemadatei, aus dem die entsprechenden Klassen durch den Relational Interface Generator (RIG) erzeugt werden. Für alle Benutzertabellen der relationalen Datenbank wird eine korrespondierende Klasse gleichen Namens angelegt; die Spalten der Tabelle werden in Attribute der entsprechenden Persistence-Basisdatentypen umgesetzt. Die Primärschlüssel der Tabelle bilden dabei die Attribute des Key Objects der jeweiligen Klasse (vgl. Seite 143). Fremdschlüsselbeziehungen werden - soweit das RDBMS deren Definition unterstützt - in Beziehungen zwischen den beteiligten Klassen umgesetzt (bei Sybase entnommen aus der Tabelle `syskeys`). Beim Einsatz neuerer Sybase-Versionen (ab System 10) entstehen dabei jedoch Probleme, da DDL-Statements Informationen über Schlüssel in der Systemtabelle `sysindexes` ablegen, die vom Reader aber nicht verarbeitet wird.

Der verfügbare Sybase Reader erwies sich bei der Bestimmung der korrekten Kardinalität der resultierenden Beziehungen als verbesserungswürdig. Obwohl Primär- und Fremdschlüsselspalten als *unique* oder *not null* definiert sein können, werden diese Informationen nicht berücksichtigt, sondern jede Beziehung mit der Komplexität (0,1),(0,n) generiert.

Falls die Fremdschlüsselspalte in der Tabelle NULL sein darf, setzt der Dictionary Reader die Delete Action für die Zielklasse (die Klasse, in der der Fremdschlüssel Primärschlüssel ist), auf *Remove*, d.h. das Löschen eines Objektes löscht zusätzlich auch die Beziehung. Ist die Fremdschlüsselspalte als NOT NULL definiert, wird die Löschaktion propagiert (*Propagate*) und auch das ehemals in Beziehung stehende Objekt mit dem entsprechenden Primärschlüssel gelöscht.

Sollten zwischen zwei Klassen mehrere Beziehungen existieren (Rollen), werden durch den Reader gleichnamige Relationships generiert, was durch den RIG fehlerhaft verarbeitet würde. Um ein OO Schema für existierende Datenbanken den realen Gegebenheiten anzupassen, müssen die erzeugten Schemadateien manuell nachbearbeitet werden. Dabei sollten vor allem folgende Punkte beachtet werden:

- Virtuelle Oberklassen
  Gemeinsame Attribute mehrerer Klassen können zu virtuellen Oberklassen zusammengefaßt werden.
- Beziehungen identifizieren
  Die korrekten Komplexitätsgrade der Beziehungen und die semantisch angemessenen Delete Actions sind zu modifizieren oder zu ergänzen (siehe oben).
- Umbenennung von Klassen und Attributen
  Wenn die Tabellen- und Spaltennamen so beibehalten werden, wie sie vom Reader aus der Datenbank gelesen worden sind, können die (Alias-)Namen für Klassen und Attribute gegenüber diesen Namen geändert werden. Somit können auch bei Namenskonflikten oder schlechter Namenswahl in der relationalen Datenbank ohne Umbenennung der Tabellen oder Spalten im RDBMS die Persistence-Anwendungen mit neuen adäquaten Klassen- und Attributnamen realisiert werden.

Als Konsequenz aus den Unzulänglichkeiten des verfügbaren Sybase Dictionary Reader wurde selbst eine überarbeitete Version entwickelt, die in mehreren Punkten Verbesserungen gegenüber dem herkömmlichen Tool aufweist [Jas97]:

- Verarbeitung der Systemtabellen, wie sie ab Sybase 10 Gültigkeit haben,
- korrekte Bestimmung der Komplexität von Beziehungen,

- Behandlung von mehreren Beziehungen zwischen gleichen Tabellen,
- gleichzeitige Verarbeitung mehrerer Datenbanken (erlaubt datenbankübergreifende Beziehungen).

8.1.3 Relational Interface Generator

Aus einem vorgegebenen Schema generiert der Relational Interface Generator (RIG) C++-Klassen für alle Entitäten im Schema. Jede Klasse implementiert ihre eigenen Methoden für den Zugriff auf die Datenbank, z.B. zum Erstellen von persistenten oder transienten Objekten, Ändern von Objektattributen oder -beziehungen, zur Abfrage über Attributwerte oder SQL oder zum Löschen von Objekten aus dem Speicher bzw. der Datenbank. Das Konzept der Vererbung von Attributen, Methoden und Beziehungen, insbesondere auch die Möglichkeit der Propagierung von Queries auf einer Klasse über alle ihre Subklassen wird ebenso unterstützt wie die Verwendung virtueller Methoden zur Realisierung von Polymorphismus oder die Integration benutzerdefinierter Klassen in die Vererbungshierarchie.

Abbildung 8.2 gibt einen Überblick über die aus einer Basisklasse (am Beispiel `Employee`) abgeleiteten Klassen.

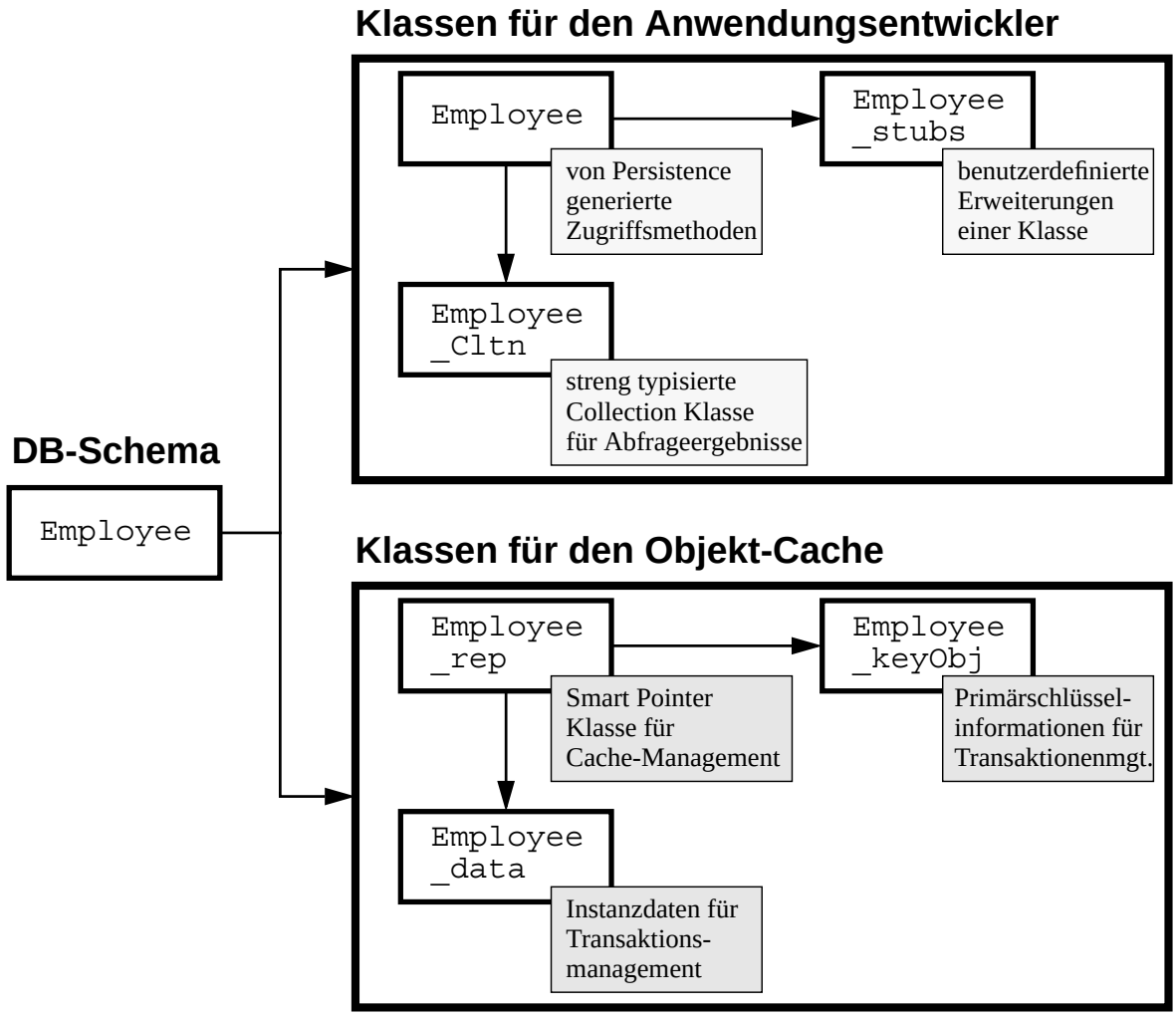


Abbildung 8.2: Klassen in Persistence

Aus einem Schema werden die folgenden Klassen generiert:

- `PersistenceObject`
  Persistente Basisklasse des Vererbungsbaums mit den grundlegenden Objektmethoden zum Anlegen, Löschen, Ändern und Lesen von Instanzen,
- <*Project*>
  Für jedes Schema generierte Klasse mit Methoden, die Operationen auf allen Klassen eines Schemas ausführen,
- Klassen für den Anwendungsentwickler (Subklassen von `PersistenceObject`): <*Class*>, <*Class*>`_Cltn`, <*Class*>`_stubs`,
- Klassen für das Objekt-Cache-Management: <*Class*>`_rep`, <*Class*>`_keyobj`, <*Class*>`_data`.

In Tabelle 8.1 ist eine Auswahl der Methoden enthalten, die vom RIG für jede Klasse generiert werden und die für nachfolgende Betrachtungen relevant sind.

Kategorie	Name	Wirkung
Konstruktor	<*Class*>`()`	4 Varianten: - Erzeugen neuer Instanzen - Kopieren einer Instanz - Lesen einer Instanz aus der RDB über den Primärschlüssel - Erzeugen eines neuen leeren nicht-persistenten Objekts (NPO)
Destruktor	`~`<*Class*>`()`	Löschen der Instanz aus dem Speicher ohne Effekt auf die RDB
Modifikatoren	`remove()`	Löschen einer Instanz aus der RDB, aber nicht aus dem Speicher
	`set`<*Attribute*>`()`	Setzen des Attributwertes
	`addTo`<*Relationship*>`()`	Hinzufügen zur Menge der Instanzen in der Beziehung
	`rmvFrom`<*Relationship*>`()`	Entfernen aus der Menge der Instanzen in der Beziehung
	`set`<*Relationship*>`()`	Zuweisen einer Menge von Instanzen (oder einer einzelnen) an die Beziehung
Zugriffsmethoden auf Klassen	`queryKey()`	Liefern einer Instanz, die im Primärschlüsselwert übereinstimmt
	`querySQLWhere()`	Liefern einer Menge von Instanzen (*Collection*), die die WHERE-Bedingung erfüllen
Zugriffsmethoden auf Instanzen	`get`<*Attribute*>`()`	Lesen des Attributwertes
	`get`<*Relationship*>`()`	Liefern der Objekte, die an der Beziehung teilnehmen

Tabelle 8.1: Methoden der Klasse <*Class*> (Auswahl)

Bestimmte Besonderheiten sind dabei zu beachten, z.B. können Werte von Primärschlüsselattributen eines Objektes nach dessen persistenter Speicherung nicht über den Aufruf von

`set<`*Attribute*`>()`-Methoden verändert werden, d.h. die entsprechenden Methoden werden vom RIG gar nicht erst generiert. Eine vollständige Beschreibung ist [Per95a] zu entnehmen.

8.1.4 Relational Object Manager

Der Relational Object Manager (ROM) ist verantwortlich für die Kontrolle nebenläufiger Verarbeitung und die Sicherstellung eines effizienten Datenzugriffs durch die Verwaltung eines Objekt-Cache im Hauptspeicher. Dabei ist es zwingend notwendig, Konsistenz auf der Objektebene sicherzustellen: Ein Objekt kann z.B. mehrere Datenbanktabellen referenzieren. Außerdem können mehrere Kopien desselben Objekts zu einem Zeitpunkt auf einmal im Hauptspeicher gehalten werden.

Der ROM stellt in diesem Fall die Objektidentität sicher: Wann immer der Objekt-Manager ein Datentupel für ein Objekt aus der relationalen Datenbank gelesen hat, registriert er die Schlüsselinformationen (Schlüsselobjekte der Klasse `<`*Class*`>_keyObj`) und speichert die Daten im Objekt-Cache. Dabei werden Fremdschlüsselattribute, die auf bereits im Cache befindliche Objekte aus anderen Klassen verweisen, automatisch in hauptspeicherinterne Zeiger umgesetzt (in [Per95b] als *Semantic Key Swizzling* bezeichnet). Dadurch wird die Systemleistung insbesondere bei navigierenden Abfragen, die Objekte aus verschiedenen Klassen über Beziehungen selektieren, wesentlich erhöht, sobald die Daten einmal aus der Datenbank gelesen wurden.

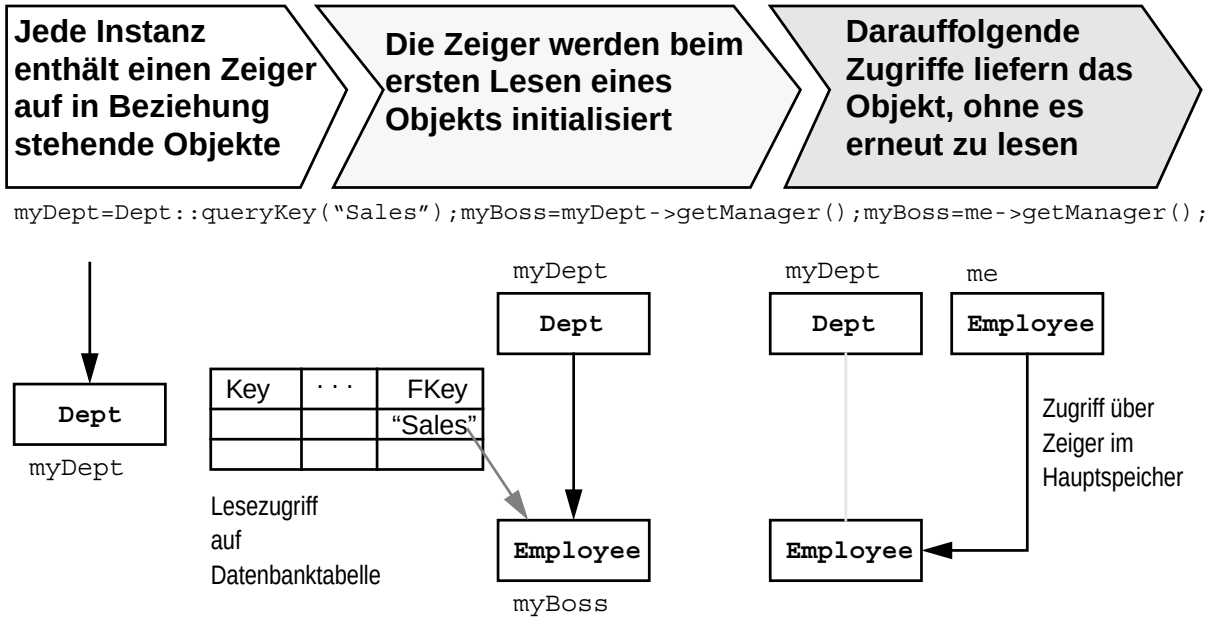


Abbildung 8.3: Konzept des Semantic Key Swizzling

Über die Registrierung der Schlüsselobjekte kann zudem das Problem der Objektidentität gelöst werden, wenn sich mehrere Kopien desselben Objekts zur gleichen Zeit im Hauptspeicher befinden: Die vom RIG generierten Methoden geben niemals Zeiger auf die eigentlichen Objektdaten zurück, sondern immer nur sogenannte *Smart Pointer* auf die Daten. Ein Smart Pointer beinhaltet einen Referenzzähler für die Anzahl der aktiven Kopien des Objekts und einen Zeiger auf die physikalischen Daten.

Die Datenintegrität zwischen Objekt-Cache und Datenbank (insbesondere bei Nebenläufigkeit im Mehrbenutzerbetrieb) wird folgendermaßen gesichert: Wird eine Transaktion ordnungsgemäß durch Commit beendet, werden die Daten für alle geänderten Objekte aus dem Objekt-

Cache in die Datenbank zurückgeschrieben. Der Cache wird geleert und alle Sperren freigegeben, die auf Objekten gehalten wurden, auf die in der Transaktion lesend und/oder schreibend zugegriffen wurde. Um allerdings zu verhindern, daß die noch im Speicher befindlichen Objekte und die darunterliegenden Daten in der Datenbank voneinander abweichen, werden die Smart Pointer für alle zurückgeschriebenen Objekte im Speicher zurückgehalten, so daß beim nächsten Zugriff auf eines dieser Objekte die entsprechenden Sperren wieder erworben und die Objektdaten wieder aus der Datenbank eingelesen (und gecacht) werden können.

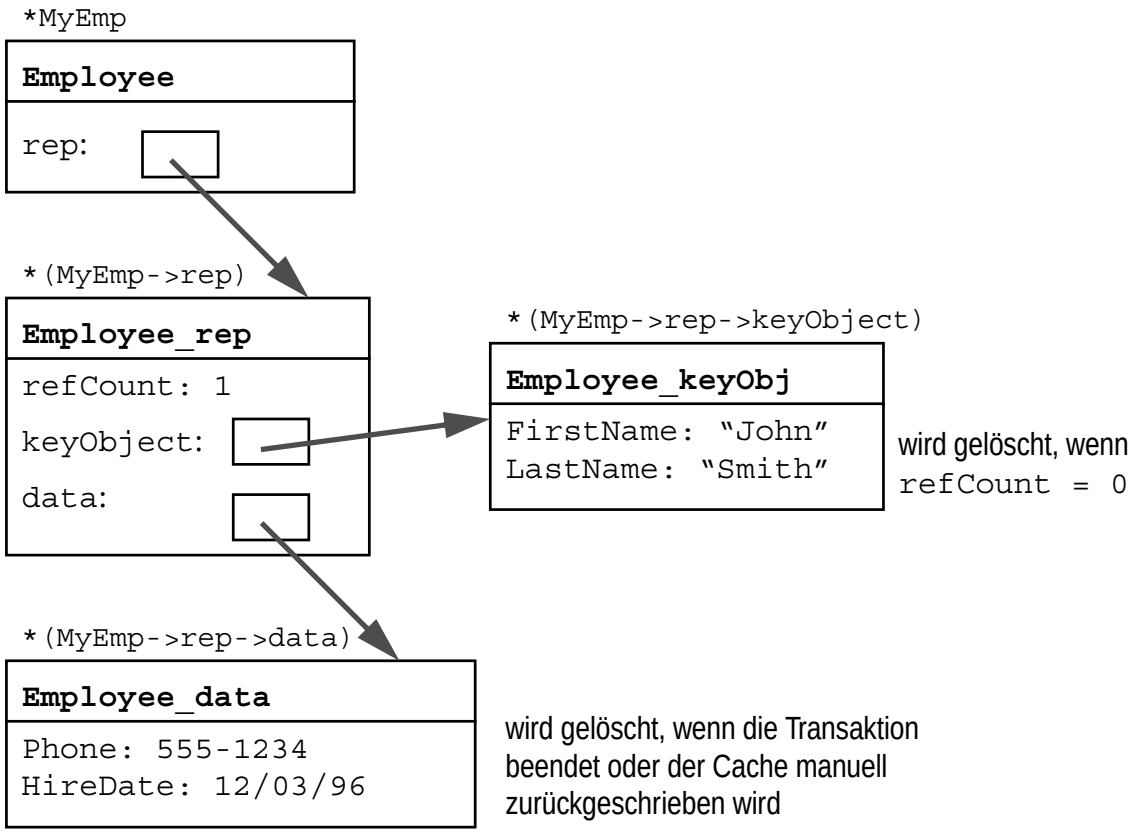


Abbildung 8.4: Konzept der Smart Pointer

Die durch den ROM implementierten Methoden umfassen u.a. die Verwaltung von Datenbankverbindungen (Klasse `PS_Connection`) und die Zuordnung der Klassen zu Datenbankverbindungen sowie Transaktionsverarbeitung (Begin, Commit und Rollback von Transaktionen, Setzen von Sicherungspunkten) und explizite Manipulation des Objekt-Cache (siehe Tabelle 8.2 auf Seite 145).

In Persistence werden drei Schreibmodi und zwei Sperrmodi unterstützt. Durch Schreibmodi wird kontrolliert, wann Änderungen auf Objekten im Cache auf die Datenbank zurückgeschrieben werden. Möglich sind NEVER (verbietet Updates), ONCOMMIT (Zurückschreiben bei Commit) und IMMEDIATE (sofortiges Schreiben in die Datenbank). Durch den Sperrmodus (pro Connection oder Klasse) wird festgelegt, ob für das Lesen eines Objekts aus der Datenbank eine Lesesperre angefordert werden muß (LOCK) oder nicht (NOLOCK). Transaktionen können nur auf Connections ausgeführt werden (vgl. Seite 145).

Voraussetzung ist, daß die Kombinationen der Schreib- und Sperrmodi in Persistence durch die Einstellung der Konsistenzgrade (*Isolation Level*) in den lokalen Datenbanksystemen ermöglicht werden.

Kategorie	Name	Wirkung
Konstruktor	`PS_Connection()`	Herstellen einer DB-Verbindung durch Einloggen
Destruktor	`~PS_Connection()`	Löschen der Verbindung zur Datenbank
Transaktionen	`beginTransaction()`	Beginn einer Transaktion
	`rollbackTransaction()`	Zurücksetzen zum Beginn oder Savepoint
	`saveTransaction()`	Setzen eines Savepoints
	`commitTransaction()`	Beenden der Transaktion
Cache	`clearCache()`	Löschen aller Objekte aus der Datenbank
	`flushCache()`	Leeren des Cache und und Zurückschreiben der geänderten Objekte
	`writeCache()`	Zurückschreiben aller Objekte in die DB
Zugriff über SQL	`selectMany()`	Ausführen einer beliebigen SELECT SQL-Anweisung auf der aktuellen Connection
	`sendSQL()`	Ausführen einer beliebigen SQL-Anweisung auf der aktuellen Verbindung

Tabelle 8.2: Methoden der Klasse `PS_Connection`

Um beliebige SQL-Kommandos an eine Datenbank zu senden und Anfrageergebnisse zu empfangen, müssen die Methoden `sendSQL()` bzw. `selectMany()` verwendet werden. Das Ergebnis einer SELECT-Anfrage wird als Instanz der Klasse `PS_ResultObject_Cltn` zurückgeliefert. In ihr sind alle Ergebnistupel der SELECT-Anfrage repräsentiert. Zur Beschreibung der Tupeleigenschaften (Datentypen, Attributnamen) des Resultats ist eine Klasse `PS_PropertyList` vorgesehen. Die einzelnen Tupel sind als Instanz der Klasse `PS_ResultObject` repräsentiert, deren Werte als Instanz von `PS_ResultValue` vorliegen und mit datentypspezifischen Operatoren extrahiert werden können.

8.1.5 Gleichzeitiger Zugriff auf mehrere Datenbanken

Eine besondere Eigenschaft von Persistence gegenüber anderen objekt-relationalen Werkzeugen besteht darin, mehrere Verbindungen zu heterogenen Datenbanksystemen gleichzeitig zu unterhalten. Jeder Klasse kann dynamisch zur Programmlaufzeit eine geöffnete Datenbankverbindung zugeordnet werden (*Connection Mapping*), die später auch noch geändert werden kann.

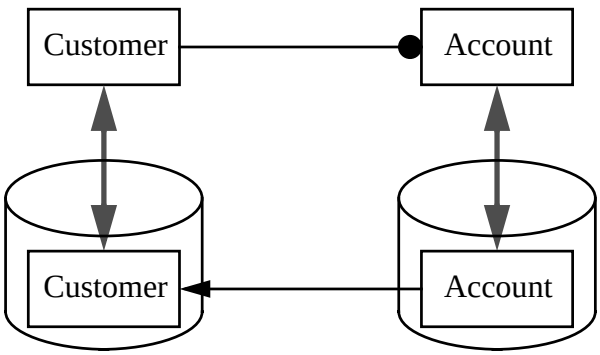


Abbildung 8.5: Gleichzeitiger Zugriff auf mehrere Datenbanken in Persistence

Wenn das Persistence-Schema Beziehungen (Relationships, Aggregationen) zwischen zwei oder mehr Klassen beinhaltet, die zur Laufzeit auf unterschiedliche Datenbanksysteme abgebildet werden, entsteht durch die implizite Sicherung der referentiellen Integrität bereits eine einfache Art globaler Integritätssicherung (siehe Abbildung 8.5).

Um lokale Datenbankzugriffe auszuführen, muß zuerst ein Objekt der Klasse `PS_Connection` erzeugt werden (gleichbedeutend mit einem Login auf der lokalen Datenbank). Anschließend wird eine Transaktion auf dieser Connection begonnen, bevor Methoden auf den vom RIG generierten Klassen aufgerufen werden können (vgl. Beispiel 8.2).

Beispiel 8.2: (Transaktionen in Persistence)

```
PS_SybConnection *conn;
conn = new PS_SybConnection(database, user, password);
conn->beginTransaction();
aEmployee->queryKey(100);
aEmployee->setName("Mike Stone");
conn->commitTransaction();
```

Um eine globale (*multithreaded*) Applikation in Solaris 2.x zu entwickeln, muß eine Verbindungsgruppe (Klasse `PS_ConnectionGroup`) definiert werden. Eine Instanz der Klasse `PS_ConnectionGroup` ist verantwortlich für einen Objekt-Cache, eine Menge von Connections und Mappings zwischen C^{++}-Klassen und DB-Tabellen. Zusätzlich muß jede Connection einer Connection Group zugeordnet werden. Die Instanz von `PS_ConnectionGroup` muß aktiviert werden, bevor es zu einem DB-Zugriff über eine ihrer Connections kommen kann. Eine weitere Klasse, `PS_GroupMap`, sichert den exklusiven Zugriff eines Threads über eine `PS_ConnectionGroup`-Instanz, so daß sich mehrere Threads eine Connection Group teilen können. Bei Erzeugen eines Threads muß diesem eine Connection Group über die Klasse `PS_GroupMap` zugewiesen werden. Ansonsten erfolgt die Synchronisation des Zugriffs auf den Objekt-Cache über die Methoden `use()` und `unuse()`. Abbildung 8.6 zeigt noch einmal den Zusammenhang zwischen allen Persistence-Klassen zum Management von Connections und Threads in OMT-Notation.

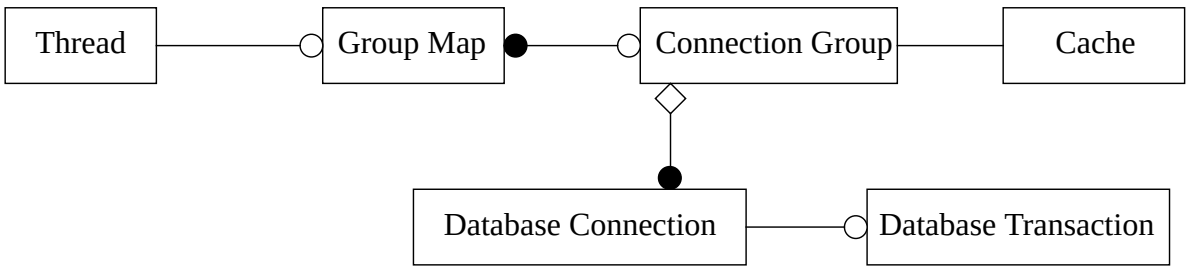


Abbildung 8.6: Datenbankverbindungen in Persistence

Persistence bietet jedoch keine Unterstützung von Transaktionen über mehrere Datenbankverbindungen: der Benutzer bleibt auf die verbindungsspezifischen Methoden der Klasse `PS_Connection` (siehe Tabelle 8.2) beschränkt, die die Transaktionssteuerung jeweils für die einzelnen beteiligten DBMS übernehmen. Wird z.B. eine Transaktion, die in zwei verschiedenen Datenbanken Änderungen vorgenommen hat, nur in einem der beteiligten Systeme zurückgesetzt, im anderen jedoch ordnungsgemäß beendet, können global inkonsistente Zustände entstehen.

Einen Ansatzpunkt, wie ECA-Regeln eingesetzt werden können, um strenge globale Konsistenz (1-Kopien-Serialisierbarkeit) zu sichern, bietet die Arbeit von Chrysanthis und Ramamritham [CR93]. Das Datenbank-Gateway ermöglicht es, Transaktions-Events an der SQL-Schnittstelle zu detektieren. Unter dieser Voraussetzung lassen sich die in [CR93] entwickelten Axiome zur Beschreibung des 2PC-Protokolls, die auf Events basieren, in ECA-Regeln ausdrücken.

Persistence unterstützt darüber hinaus die Integration von Transaktions-Monitoren (*TP-Monitore*) wie Tuxedo von Novell oder Encina von Transarc [GR93]. Somit wird es auch möglich, in Persistence globale Anwendungen zu schreiben, die auf verteilten Transaktionen basieren. Um die Atomarität einer verteilten Transaktion zu garantieren, müssen die TP-Monitore ein gemeinsames verteiltes Commit-Protokoll unterstützen. Die LDBMS sind an dem Protokoll über die TP-Monitore indirekt beteiligt, so daß ein hierarchisches Commit-Protokoll (2PC) erforderlich wird. Die dafür erforderliche Funktionalität ist im X/OPEN-Standard zur verteilten Transaktionsverarbeitung dokumentiert [XO91]. Die darin beschriebene XA-Schnittstelle wird von den genannten TP-Monitoren genutzt und muß auch von den lokalen DBMS unterstützt werden, damit sie in das Commit-Protokoll einbezogen werden können. Die führenden Anbieter relationaler DBMS bieten XA-Schnittstellen als Zusatzprodukte zu ihren Systemen an.

8.2 DML-Vermittler

8.2.1 Abbildung von SQL-Anweisungen auf die Persistence-Schnittstelle

Die von Persistence bereitgestellte Funktionalität liegt in Gestalt von statischen C++ Bibliotheken (bzw. Klassen) vor und kann somit nur in zu compilierenden Anwendungen eingesetzt werden. Deshalb können dynamische SQL-Zugriffe auf beliebige Datenbanken nicht ohne weiteres in eine der von Persistence angebotenen Methoden übersetzt werden. Eine Übersetzung von SQL-Anweisungen in äquivalente Methodenaufrufe setzt die vollständige Kenntnis der Klassen- und Attributnamen voraus. Um dieses Problem zu lösen, wird die Annahme getroffen, daß ein DML-Vermittler nur eine bestimmte Menge von Altanwendungen behandelt mit einem weitgehend unveränderlichen Objektschema (Persistence-Schemadatei), so daß a-priori die Menge der Klassen und der auf ihnen ausführbaren Methoden bekannt ist.

Die von einer Altanwendung in SQL abgesetzten Kommandos werden dem Persistence-basierten DML-Vermittler vom Gateway über den Aufruf der entfernten Prozedur `exec_dml` in einer simplifizierten SQL-artigen “Sprache” übergeben (vgl. Abschnitt 7.2.2 auf Seite 133). Der Vermittlerprozeß übersetzt dann die Parameter, die von der Prozedur `exec_dml` übergeben wurden, in äquivalente Methodenaufrufe und führt diese aus.

Die für das Verständnis des nachfolgend gegebenen Übersetzungsalgorithmus benötigten C++-Methoden am Relational Interface von Persistence wurden bei der Vorstellung von Persistence eingeführt.

Unterstützt werden zur Zeit die Operationstypen, so wie sie in `exec_dml` definiert wurden: Auf- und Abbau einer Client-Server-Verbindung, die DML-Anweisungen sowie die Transaktionskommandos BEGIN, COMMIT und ROLLBACK. Tabelle 8.3 zeigt die prinzipielle Abbildung der Operationstypen in äquivalente Methodenaufrufe in Persistence.

Kategorie	Operation	Methodenaufruf
Verbindungen	OP_CONNECT	`PS_Connection::PS_Connection()`
	OP_DISCONNECT	`PS_Connection::~PS_Connection()`
Transaktions-kommando	OP_BEGIN	`PS_Connection::beginTransaction()`
	OP_COMMIT	`PS_Connection::commitTransaction()`
	OP_ROLLBACK	`PS_Connection::rollbackTransaction()`
DML-Befehl	OP_INSERT	<*Class*>`::`<*Class*>`()`
	OP_UPDATE	<*Class*>`::set`<*Attribute*>`()`
	OP_DELETE	<*Class*>`::remove()`
SELECT	OP_SELECT	`PS_Connection::selectMany()`

Tabelle 8.3: Abbildung der Operationstypen auf Persistence-Methodenaufrufe

Während die Umsetzung der Transaktions- und Verbindungskommandos keine Probleme bereitet, müssen für die anderen Operationen zunächst die Klassen- und Attributinformationen ausgewertet werden. Die dafür erforderlichen jeweils identischen Codeabschnitte werden bei der Generierung des DML-Vermittlers erzeugt (siehe Abschnitt 8.3.2). Wenn die Klasse, auf der die betreffende Operation durchzuführen ist, bereits bestimmt wurde, ist folgendes Vorgehen notwendig:

- **`OP_SELECT`**

Queries werden in der aktuellen Vermittler-Version umgesetzt in einen Aufruf der Methode `PS_Connection::selectMany()`, mit der SELECT-Anfragen auf einer geöffneten Connection ausgeführt werden können. Das Resultat ist dabei immer eine Menge von Objekten vom Typ `PS_ResultObject`, die mit anderen Methoden weiterverarbeitet werden kann. (vgl. Seite 145). Der Vorteil besteht in der hohen Flexibilität, da jede beliebige Anfrage als Zeichenkettenparameter übergeben werden kann.

Eine alternative Abbildung würde darin bestehen, daß bei Anfragen auf einer einzigen Tabelle auf die Methode <*Class*>`::querySQLWhere()` bzw. bei Joins auf die Aufruffolge <*Class*>`::get`<*Relationship*>`()` und <*Class*>`::get`<*Attribute*>`()` abgebildet wird, was jedoch den Übersetzungsaufwand bei komplexeren SQL-Anfragen über mehrere Tabellen erhöhen würde. Ein wichtiger Vorteil der Übersetzung in `get()`-Methoden würde darin bestehen, diese Ereignisse als globale Datenbank-Events detektieren zu können (vgl. Abschnitt 8.4).

- **`OP_INSERT`**

1. Erstellen eines neuen nicht-persistenten Objektes der jeweiligen Klasse (Konstruktoraufruf);
2. Abfragen des zur Klasse gehörigen Schlüsselobjektes (*Key Object*);
3. In einer Schleife für alle Attribute, die im INSERT adressiert werden:
   falls Primärschlüsselattribut:
   Aufruf der Methode <*Class*>`_keyObj::set`<*Attribute*>`()`;
   falls Fremdschlüsselattribut:
   aus der in Beziehung stehenden Klasse das korrespondierende Objekt für den gewünschten Fremdschlüsselwert abfragen;
   Aufruf der `set`<*Relationship*>`()`-Methode auf dem neuen Objekt mit dem abgefragten Objekt als Parameter;
   falls kein Schlüsselattribut:
   Aufruf der `set`<*Attribute*>`()`-Methode auf dem neuen Objekt;
4. Zurückschreiben des geänderten Key Object in das neue Objekt;
5. Persistent-Machen des neuen Objektes durch Aufruf von <Class>`::insert()`;

- **OP_UPDATE**

1. Bereitstellen aller Objekte der jeweiligen Klasse, auf die das WHERE-Prädikat zutrifft, in einer Collection durch Aufruf der Methode <*Class*>::querySQLWhere();
2. In einer Schleife für alle Objekte aus der Collection:
3. In einer Schleife für alle Attribute:
   falls Primärschlüsselattribut:
   nicht-persistentes Objekt als Kopie des aktuellen persistenten Objekts erstellen;
   zugehöriges Key Object abfragen;
   Aufruf der Methode <Class>_keyObj::set<Attribute>();
   Löschen des alten persistenten Objektes;
   Persistent-Machen des geänderten nicht-persistenten Objektes durch Aufruf von <*Class*>::insert();
   falls Fremdschlüsselattribut:
   aus der in Beziehung stehenden Klasse das korrespondierende Objekt für den gewünschten Fremdschlüsselwert abfragen;
   Aufruf der set<*Relationship*>()-Methode auf dem aktuellen Objekt mit dem abgefragten Objekt als Parameter;
   falls kein Schlüsselattribut:
   Aufruf der set<*Attribute*>()-Methode auf dem aktuellen Objekt;

- **OP_DELETE**

1. Bereitstellen aller Objekte der jeweiligen Klasse, auf die das WHERE-Prädikat zutrifft, in einer Collection durch Aufruf der Methode <*Class*>::querySQLWhere();
2. In einer Schleife für alle Objekte aus der Collection:
   Löschen durch Aufruf der Methode <*Class*>::remove();

8.2.2 Einbettung in die Systemumgebung

SQL-Anwendungen, die für das DBMS Sybase geschrieben wurden, können mit dem DML-Vermittler direkt kommunizieren, indem sie ihre DML-Kommandos nicht als Clients zum lokalen SQL-Server schicken, sondern vom Datenbank-Gateway an “ihren” DML-Vermittlerprozeß. Die Schnittstelle wurde in Abschnitt 7.2.2 auf Seite 133 beschrieben. Der DML-Vermittlerprozeß läuft als RPC-Server, wandelt die Parameter in Methodenaufrufe um, die seinerseits als SQL-Kommandos am lokalen SQL-Server ausgeführt werden. Für die Wahrung der lokalen Transaktionssemantik müssen alle Anweisungen, die in einer Transaktion enthalten sind, über den Vermittler ausgeführt werden, insbesondere SELECT-Befehle. Anderenfalls würden Befehle derselben Transaktion über unterschiedliche Datenbankverbindungen zum SQL-Server geschickt werden mit dem Ergebnis, daß Zwischenergebnisse nicht sichtbar werden.

Dabei entsteht allerdings das Problem der Kommunikation der Abfrageergebnisse zwischen Vermittler und Gateway einerseits bzw. Gateway und SQL-Anwendung andererseits (siehe Abbildung 8.7). In der aktuellen Realisierung wird ein SELECT-Befehl des Clients sowohl an den Vermittler als auch direkt an den lokalen SQL-Server geschickt. Damit kann das Format der Resultate direkt aus der Datenstruktur der Server-Kommunikation entnommen werden und braucht nicht zusätzlich als Parameter der ExecDML-Prozedur mitgegeben zu werden.

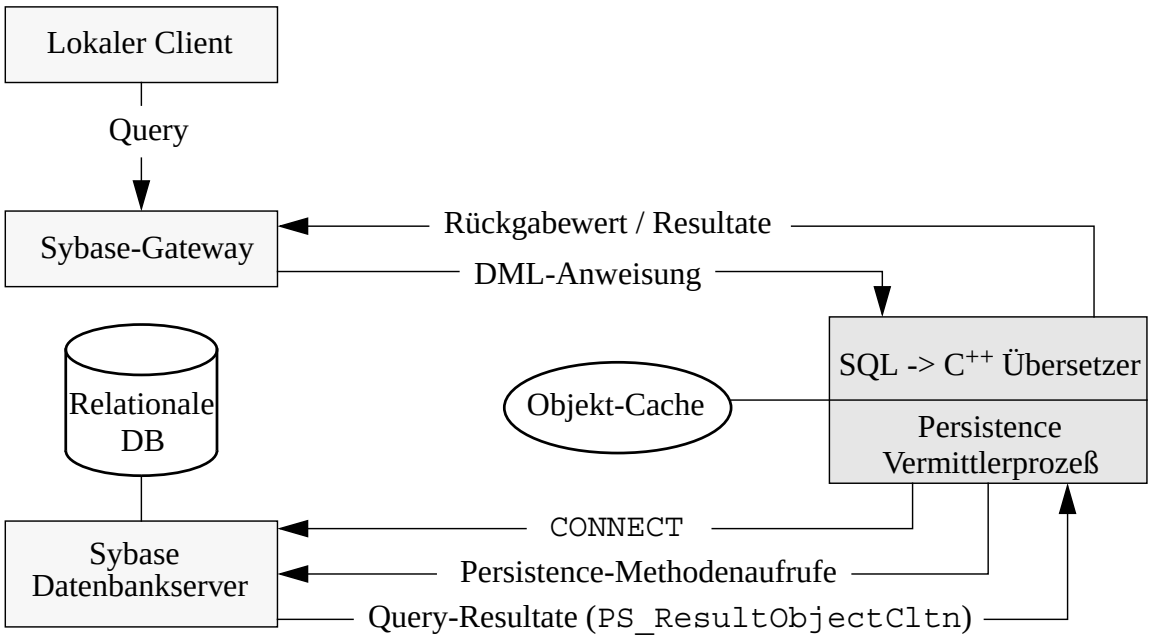


Abbildung 8.7: Kommunikation zwischen Gateway, Vermittlerprozeß und lokalem DB-Server

8.3 Vermittler-Generatoren

8.3.1 Das Konzept des "Vermittler-Generators"

Bei der Modellierung von DML-Vermittlern ergab sich, daß diese in einer Beziehung zum globalen Schema stehen, durch das zugleich alle möglichen Methodenaufrufe an der globalen Schnittstelle bestimmt sind (vgl. Abbildung 6.2 auf Seite 100). Die Menge der Klassen und ihrer Methoden (und Regeln) ist bei C++ statisch zum Übersetzungszeitpunkt festgelegt und erlaubt damit keine Modifikation zur Laufzeit. Zugrunde liegt dabei das Datenbankschema (Schema-File), das aus den lokalen Schemata durch den Gebrauch eines Data Dictionary Readers gewonnen werden werden kann. Wenn Regeln ausgeführt werden sollen, so müssen diese in sogenannten *Hook*-Methoden (siehe Abschnitt 8.4) spezifiziert werden. Hinzu kommen die Zuordnungsinformationen der Klassen zu den lokalen Datenbanksystemen. Im Ergebnis wird der Quelltext eines Vermittlers produziert, aus dem sich ein DML-Vermittlerprozeß generieren läßt, der für alle Anwendungen, die auf einem gemeinsamen globalen Schema operieren, spezifisch ist. Abbildung 8.8 zeigt die Schritte bei der Generierung eines DML-Vermittlers.

8.3.2 Die Implementierung des Vermittler-Generators

8.3.2.1 Architektur und Module des Vermittler-Generators

Der Vermittler-Generator `MediatorGenerator` wurde in C geschrieben und besteht aus drei Komponenten: dem Hauptprogramm, dem Objektschema-Parser und dem Präprozessor.

Das Hauptprogramm (`MediatorGenerator.c`) benötigt folgende Eingabeparameter:

- Name eines Persistence-Objektschemas (d.h. Name der `*.persist`-Datei),
- Name der Datei, die das *Connection Mapping* beschreibt
  (vgl. Abschnitt 8.1.5 auf Seite 145),

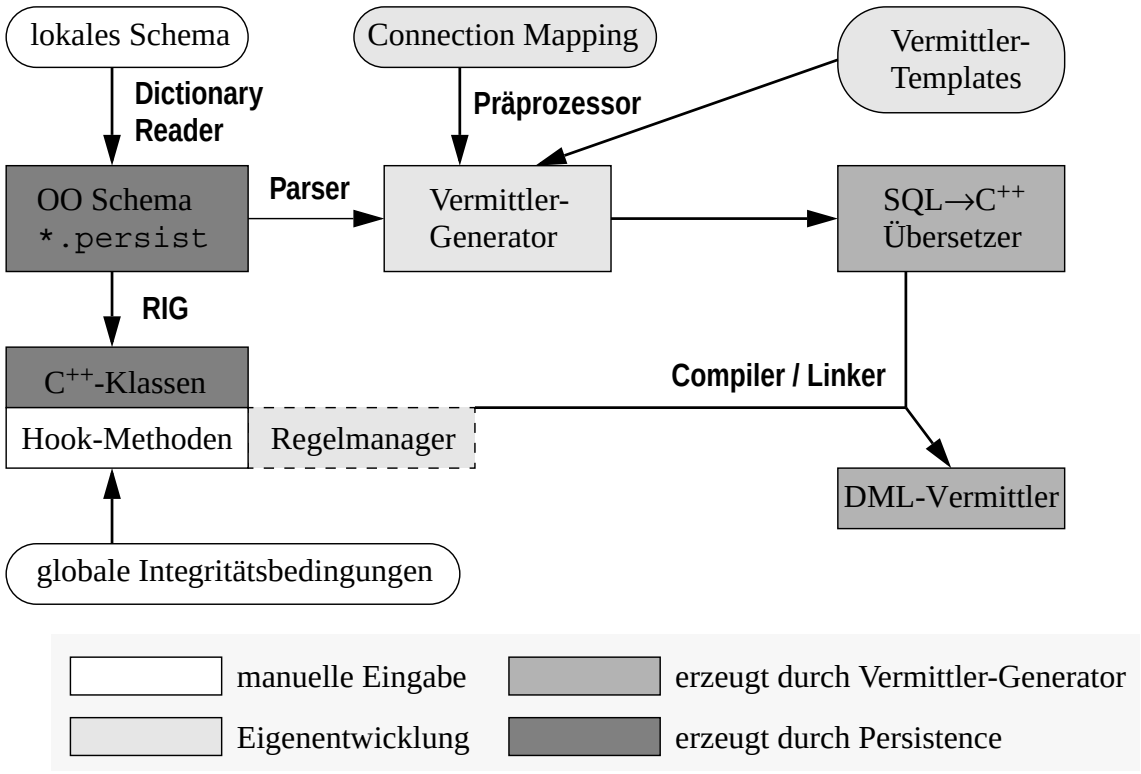


Abbildung 8.8: Generierung eines DML-Vermittlers (Prinzipdarstellung)

- RPC-Programmnummer, unter der der erzeugte Vermittler-Server vom Gateway angesprochen werden kann (vgl. Abschnitt 7.2.2 auf Seite 133),
- Name des Verzeichnisses für den erzeugten Vermittler-Quelltext.

Die Connection Mapping-Datei `*.connmap` enthält für jede Klasse eines Schemas einen Eintrag, der ihr das DBMS zuordnet, in dem sich die korrespondierende Tabelle befindet.

Im nächsten Schritt wird der Analysator des Objektschemas aufgerufen, um die relevanten Informationen aus dem Schema zu extrahieren und in Variablen zwischenzuspeichern. Das Hauptprogramm ermittelt dann aus der `*.connmap`-Datei die Zuordnung der Klassen auf die beteiligten Datenbanksysteme, bevor mit Hilfe des Vermittler-Präprozessors `TemplateTools.c` aus den Quelltextschablonen der Code des DML-Vermittlers erzeugt wird. Die Funktionsweise des Präprozessors wird ausführlich auf Seite 154 beschrieben. Abbildung 8.9 zeigt den Generierungsprozeß für einen DML-Vermittler mit den daran beteiligten Programmen, Dateien und Bibliotheken im Überblick.

Der erzeugte Vermittler-Quelltext wird im beim Aufruf des Generators angegebenen Zielverzeichnis angelegt. Im einzelnen werden für jeden Vermittler 7 Dateien angelegt:

- `Makefile_<`*Schemaname*`>`
  Makefile zum automatischen Übersetzen des generierten Vermittler-RPC-Servers,
- `exec_dml_<`*Schemaname*`>.x`
  RPC-Parameterdefinitionsdatei für den RPC-Server mit individueller RPC-Programmnummer (Parameter beim Aufruf des Vermittler-Generators),
- `exec_dml_tools_<`*Schemaname*`>.h` und `exec_dml_tools_<`*Schemaname*`>.c`
  Hilfsfunktionen für die Kontrolle der RPC-Kommunikation (Ausgabe der RPC-Daten über einen Dateideskriptor oder den *syslog*-Dienst),

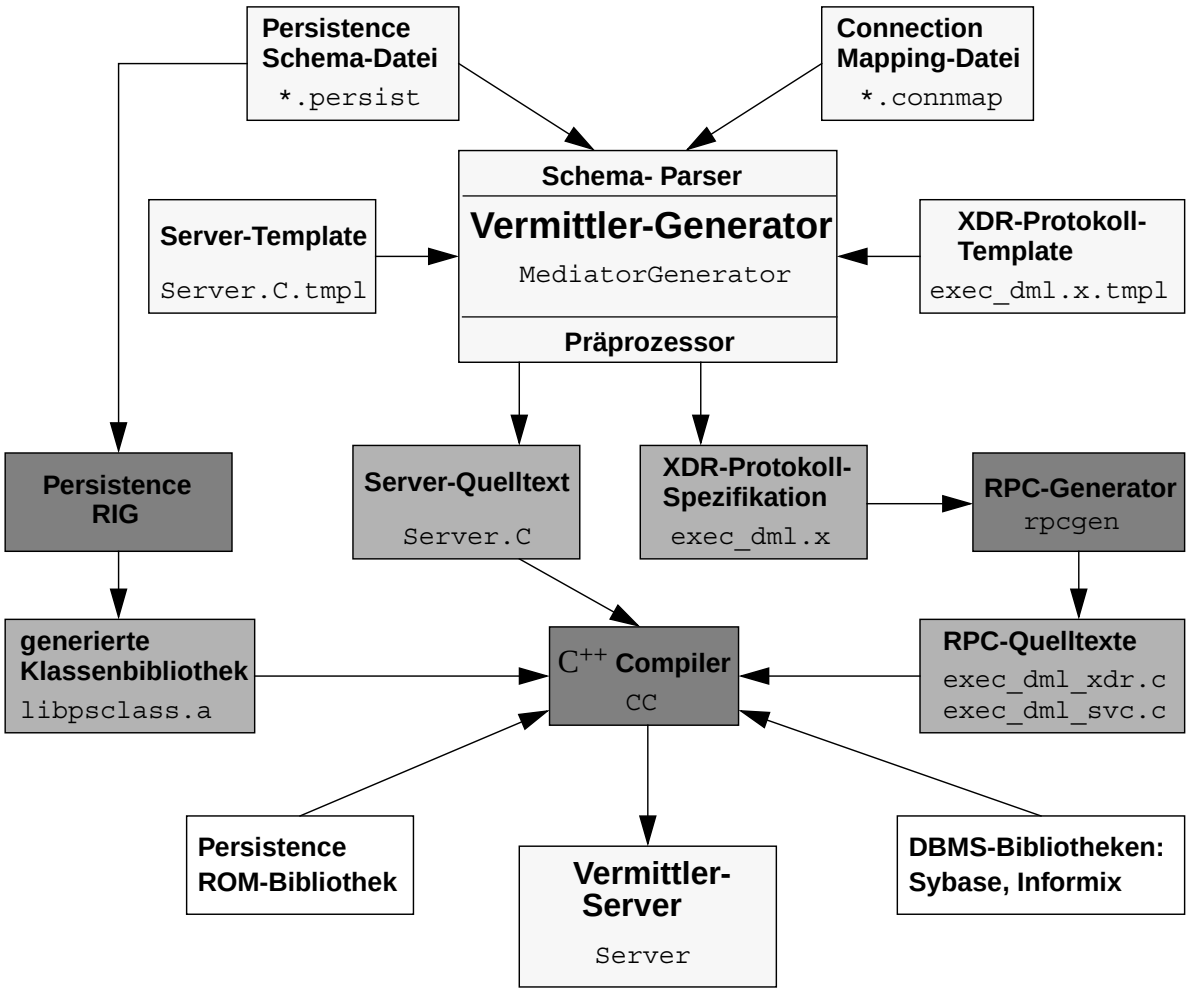


Abbildung 8.9: Vermittler-Generierungsprozeß im Überblick

- `TestClient_<`*Schemaname*`>.c`
  Test-Client (in C) für den erzeugten Vermittler-Server: interpretiert eine in `ExecDML`-ähnlicher Notation geschriebene Textdatei und erzeugt daraus die äquivalenten RPC-Aufrufe (eingesetzt zum Testen des Vermittlers),
- `Server_<`*Schemaname*`>.H` und `Server_<`*Schemaname*`>.C`
  Programmcode des eigentlichen Vermittler-Hauptprogramms; verarbeitet die als RPC-Parameter übergebenen `ExecDML`-Befehle und bildet diese auf die äquivalenten C++-Methodenaufrufe ab.

Aus der Parameterdefinitionsdatei werden über das Makefile mittels `rpcgen` die Dateien

- `exec_dml_<`*Schemaname*`>_xdr.c` (XDR-Repräsentation der `ExecDML`-Datentypen)
- `exec_dml_<`*Schemaname*`>_clnt.c`. (Client-Stub; Verwendung für den Test-Client)
- `exec_dml_<`*Schemaname*`>_svc.c` (Server-Stub)

generiert, die zusammen mit den übrigen Quelltexten und der Persistence-generierten Klassenbibliothek in die ausführbare Vermittler-Anwendung gebunden werden.

8.3.2.2 Der Objektschema-Analysator

Der Analysator für `*.persist`-Dateien wurde mit den UNIX-Werkzeugen lex und yacc geschrieben auf der Grundlage der Grammatik der Schemadefinition [Per95b]. Der Parser analysiert zunächst den einleitenden Schema-Block (u.a. mit globalen *Preferences* als Steuerungsinformation für die Code-Generierung durch den RIG). Anschließend wird eine Liste von Klassenbeschreibungen mit jeweils einer Liste von Attributen und Fremdschlüsseldefinitionen eingelesen, anschließend eine Liste von Beziehungen (Relationships). Ein kleines Beispielschema war in Abbildung 8.1 auf Seite 139 dargestellt.

Der Aktionscode des Parsers ist so gestaltet, daß beim Reduzieren der Regeln aus der Grammatik die für die spätere Generierung des Vermittlers relevanten Informationen zunächst in globalen Feldvariablen gespeichert werden. Die Variablen `classind` und `attrind` geben während des gesamten Parsing-Vorgangs jeweils den Index des nächsten freien Eintrags für eine Klasse bzw. ein Attribut an. Im einzelnen werden folgende Informationen in globalen Variablen gespeichert:

Globaler Variablenname	Bedeutung
`projname`	Schemaname in Persistence
`projdir`	Name des Verzeichnisses mit den von Persistence generierten Klassen
`classind`	Index der aktuellen Klasse
`attrind`	Index des aktuellen Attributs in der aktuellen Klasse
`classnum`	Anzahl der Klassen im Schema
`attrnum[i]`	Anzahl der Attribute in der aktuellen Klasse
`classname[i]`	Name der Klasse `i`
`tablename[i]`	Name der DB-Tabelle für Klasse `i` (falls verschieden)
`parentclass[i]`	Name der Vaterklasse von Klasse `i`
`classmapdb[i]`	Name des DBMS, wo `tabname[i]` gespeichert ist (wird nachträglich bei der Interpretation der ConnectionMap-Datei gesetzt)
`classhooks[i][h]`	ist auf Klasse `i` der Hook mit dem Hooktyp `h` definiert ? (bool)
`attrname[i][j]`	Name des Attributs mit dem Index `j` in Klasse `i`
`colname[i][j]`	Name der DB-Spalte für aktuelles Attribut in aktueller Klasse
`attrtype[i][j]`	Typ des Attributs `j` in Klasse `i`
`attrdim[i][j]`	(max.) Dimension des Attributs `j` in Klasse `i`
`attriskey[i][j]`	ist das Attribut `j` in Klasse i ein Primärschlüsselattribut ? (bool)
`attrhooks[i][j][h]`	ist auf Attribut `j` in Klasse `i` der Hook mit dem Hooktyp `h` definiert ? (bool)
	falls das Attribut `j` in Klasse `i` ein Fremdschlüssel ist:
`attrfkeyclass[i][j]`	Name der Klasse, zu der die Fremdschlüsselbeziehung besteht
`attrfkeyattr[i][j]`	Name des Attributs in Klasse `attrfkeyclass[i][j]`, zu dem die Fremdschlüsselbeziehung besteht
`attrfkeyrel[i][j]`	Name der Fremdschlüsselbeziehung zu dem Attribut `attrfkeyattr[i][j]` in Klasse `attrfkeyclass[i][j]`

Tabelle 8.4: Globale Variablen im Vermittler-Generator

8.3.2.3 Der Präprozessor des Vermittler-Generators

Der Präprozessor wird aus dem Vermittler-Generator heraus aufgerufen, der als Parameter u.a. der Name einer Ein- und einer Ausgabedatei übergeben wird. Er besteht im wesentlichen aus einer `while`-Schleife, die den Inhalt der Eingabedatei zeichenweise in die Ausgabedatei kopiert und dabei mehr als 30 verschiedene (jeweils durch eine Tilde '~' eingeleitete) Präprozessor-Direktiven abarbeitet. In [Loe95, S. 69 ff.] findet man eine ausführliche Auflistung aller Präprozessor-Direktiven.

Man kann drei Typen von Präprozessor-Direktiven unterscheiden:

- reine Ersetzungsdirektiven
- Schleifendirektiven, die den Kontrollfluß durch die Eingabedatei steuern
- Bedingungsdirektiven, die je nach Zutreffen einer Bedingung aus der Ein- in die Ausgabedatei kopieren

Bei Ersetzungsdirektiven werden die Direktiven rein textuell durch aus dem Objektschema gewonnene Informationen ersetzt, z.B. `~j` (Name des Persistence-Schemas) oder `~x` (gewünschte RPC-Programmnummer für den Vermittler).

Mit Hilfe der Schleifendirektiven wird der jeweils durch `~clb`/`~cle` (***c**lass **l**oop **b**egin* / ***c**lass **l**oop **e**nd*; Klassenschleife) bzw. `~alb`/`~ale` (***a**ttribute loop begin* / ***a**ttribute loop end*; Attributschleife) eingeschlossene Textblock aus der Eingabedatei in einer Schleife für jede Klasse (bzw. jedes Attribut einer Klasse) einmal in die Ausgabedatei kopiert. Dabei werden die Klassen- bzw. Attributindizes erhöht, so daß die innerhalb einer Schleife enthaltenen Ersetzungsdirektiven bei jedem Durchlauf jeweils durch den Namen der nächsten Klasse (`~cc`) bzw. des nächsten Attributs (`~ac`) ersetzt werden.

Die Bedingungsdirektiven `~akp`/`~akf`/`~akn`/`~ake` erlauben es, den durch zwei Direktiven eingeschlossenen Textblock jeweils nur beim Zutreffen einer Bedingung in die Ausgabedatei zu kopieren und sonst zu überspringen. Der Block zwischen `~akp` (***a**ttribute **k**ey **p**rimary*) und `~akf` (***a**ttribute **k**ey **f**oreign*) wird nur für Primärschlüsselattribute in die Ausgabedatei kopiert, der Block zwischen `~akf` und `~akn` (***a**ttribute **k**ey **n**o*) nur für Fremdschlüsselattribute und der Block zwischen `~akn` und `~ake` (***a**ttribute **k**ey **e**nd*) nur für Nichtschlüsselattribute. Somit können für das Ändern eines Attributwertes je nach dessen Schlüsseleigenschaften jeweils unterschiedliche Code-Blöcke in den Server-Quelltext kopiert werden.

8.4 Die aktive Komponente des Vermittlers

8.4.1 Aktive Verarbeitung mit Hooks in Persistence

Persistence selbst bietet die Möglichkeit, sowohl auf Klassenebene als auch für einzelne Objektattribute und Beziehungen sogenannte *Notification Hooks* als aktive Elemente zu definieren. Analog zum Triggerkonzept in relationalen Datenbanken werden dabei auf den betroffenen Objekten spezielle Methoden (mit vordefinierten Namen) automatisch aufgerufen, wenn bestimmte Ereignisse entweder direkt bevorstehen (*Pre*-Hooks) oder gerade eingetreten sind (*Post*-Hooks). Als Ereignisse zählen dabei das Hinzufügen und Löschen, die Abfrage oder Änderung von Objekten in einer Klasse, die Änderung von Werten für einzelne Attribute sowie das Lesen und Ändern von Beziehungen zwischen Objekten. In Persistence werden diese als CRUD-Methoden (***C**reate **R**ead **U**pdate **D**elete*) bezeichnet.

Wenn der RIG den C++-Code für die Klassen eines Objektschemas erzeugt, werden in der Datei <*Class*>_stubs.C standardmäßig "leere" Methodenstümpfe (*Stubs*) für die definierten Hooks generiert, die modifiziert werden können, um beliebigen Anwendungscode auszuführen.

Die Namen der automatisch aufgerufenen Methoden werden dabei aus dem Präfix `will` für Pre- bzw. `did` für Post-Hooks, dem Namen des Datenbankereignisses und ggf. dem Attribut- oder Beziehungsnamen gebildet. So wird z.B. die Methode `Employee::willStore()` direkt vor dem Speichern eines neuen `Employee`-Objekts in der Datenbank aufgerufen, `Customer::didRemove()` unmittelbar nach dem Löschen einer Instanz von `Customer`. Bei Wertänderungen von Attributen oder Beziehungen werden die alten (bei Post-Hooks) bzw. neuen Werte (bei Pre-Hooks) als Parameter an die Hook-Methoden übergeben.

Das Konzept der Hook-Methoden ist durch seine Berücksichtigung von Datenbankereignissen besonders zur Integritätswahrung geeignet. Methoden, die zusätzlich vom Benutzer geschrieben werden, rufen die elementaren, vom RIG automatisch generierten Zugriffsmethoden auf, die mit Hook-Methoden gekoppelt sein können. Die Kopplung zwischen dem auslösenden Event (Aufruf einer Zugriffsmethode) und der Ausführung der Hook-Methode ist implizit *immediate* (zur Realisierung komplexerer Kopplungsmodi siehe Seite 161).

8.4.2 Behandlung von Events

8.4.2.1 Integration von Datenbank-Events

Die Einteilung der Datenbank-Events, die durch den Regelmanager im Vermittler zu verarbeiten sind, basiert auf den im vorigen Abschnitt skizzierten Hook-Methoden. Tabelle 8.5 gibt einen Überblick über die Datenbank-Events.

Event-Typ	Beschreibung
Create Instance	Erzeugen einer neuen Instanz
Delete Instance	Löschen einer Instanz aus der DB
Fetch Instance	Lesen einer Instanz aus der DB
Store Instance	Schreiben einer Instanz in die DB
ChangeAttribute	Ändern eines Attributwertes auf einer Instanz
ChangeRelationship	Ändern der Beziehung zwischen Objekten
FetchRelationship	Lesen des durch die Beziehung referenzierten Objektes

Tabelle 8.5: Datenbank-Events in Persistence

Datenbank-Events werden durch folgende Attribute beschrieben (Klasse `dbEvent`):

- `EventID` Event-Identifikator
- `ClassName` Name der Klasse, die vom Event betroffen ist
- `AttributeName` Name des Attributes (irrelevant bei instanzbezogenen Events)
- `OP` Event-Charakteristik (3 Bytes), zusammengesetzt aus:
  Zeitpunkt ::= **`b`** (*before*) | **`a`** (*after*)
  Objektart ::= **`r`** (*relationship*) | **`i`** (*instance*) | **`a`** (*attribute*)
  Operationstyp ::= **`c`** (*create*) | **`d`** (*delete*) | **`f`** (*fetch*) | **`s`** (*store*)

Globale Event-Detektion

Wie bei der Diskussion des ECA-Regelmodells für aktive Objekte in Multidatenbanken bereits skizziert wurde, bedarf es auf der globalen Ebene (hier: auf der Ebene des Vermittlers) einer Möglichkeit zur Eventdetektion, wobei die Events durch lokale oder globale Benutzer ausgelöst worden sein können. Zur Signalisierung an einen Regelmanager werden die Hook-Methoden benutzt. Dazu muß nach Aufruf des RIG, der den Code der Hook-Methoden im File <*Class*>`_stubs.C` erzeugt, eine Post-Processing-Routine gestartet werden, die den erzeugten Code nachträglich um Aufrufe einer Funktion `raise` zur Eventsignalisierung erweitert. Hierzu wurde ein Programm geschrieben (`InsertRaises`), das in 3 Schritten arbeitet:

1. Analyse des Schema-Files `*.persist` und Ermittlung der Hooks, die definiert wurden
2. Analyse alles <*Class*>`_stubs.C`-Files zur Bestimmung bereits generierter Hook-Methoden
3. Einfügen des `raise`-Funktionsaufrufs in den Code der Hook-Methode (falls noch nicht vorhanden)

Die Verarbeitung der Schema-Files erfolgt mit lex/yacc, der C^{++}-Code wird mit awk bearbeitet. Der Funktionsaufruf `raise` hat folgende Struktur:

```
raise (ClassName,AttributeName,EventType,CallingObject,
       AttributeValue,ConnectionAssignedToClass)
```

Parametername	Semantik	Herkunft
`ClassName`	Name der betroffenen Klasse	<*Class*>`::className()`
`AttributeName`	`a`: Name des Attributes `r`: Name der Relationship `i`: NULL (keine Attribute)	aus Hook-Methodennamen im Schema-File
`EventType`	Event-Charakteristik	Schema-File
`CallingObject`	Pointer zum aufrufenden Objekt	`this`-Pointer

`AttributeValue`	`a`: Wert (alt / neu)
	`r`: referenziertes Objekt (alt / neu) oder Objektmenge (alt / neu)
	`i`: NULL (keine Attribute)
		Hook-Aufrufparameter:
		`a`: `newValue` / `oldValue`
		`r`: `relItem` oder `relCltn`

`ConnectionAssignedToClass`	zugewiesene DB-Verbindung	<*Class*>`::getConnection()`

Tabelle 8.6: Parameter bei der Signalisierung von Datenbank-Events

Das Programm zur Einfügung der `raise`-Aufrufe ist auch verantwortlich für die Wertebelegung der Event-Parameter. Dazu werden entweder Informationen aus dem Schema-File extrahiert oder zur Laufzeit aus den Parametern der Hook-Methoden gewonnen. Bei Before-Events (Pre-Hooks) reflektieren die übergebenen Werte den Zustand vor Ausführung der CRUD-Methode, bei After-Events (Post-Hooks) den Zustand danach. Tabelle 8.6 veranschaulicht die Semantik der Event-Parameter, wobei die Objektart `a`, `r` oder `i` die Interpretation der Parameter beeinflußt (vgl. Definition der Klasse `dbEvent` auf Seite 155).

Die generierten `raise`-Aufrufe müssen alle Klassen und Datentypen unterstützen, die von Persistence genutzt werden können. Das Interface der `raise`-Funktion sollte deshalb allgemeingültig sein und die Erweiterbarkeit um neue Klassen erlauben. Die Klassenhierarchie von Persistence unterstützt diesen Entwurf, da jede Klasse als Subklasse der allgemeinen Basis-

klasse PersistenceObject vom RIG generiert wird (siehe Seite 142). Somit ist der Parameter CallingObject immer vom Typ PersistenceObject. Unter den Attributtypen werden die in C++ bekannten Basisdatentypen unterstützt sowie die Persistence-spezifischen Typen PS_Money und RWTime.

Das automatische Einfügen der raise-Funktionsaufrufe wird durch einen erweiterten Aufruf des RIG aktiviert (fett dargestellt).

persistence -gen **-raise <*Filename*>.persist**

Globale Event-Protokollierung

Datenbank-Events können zusammen mit Transaktions-Events (siehe nächster Abschnitt) protokolliert werden. Das Eventprotokoll hat folgende Struktur (Klasse LogDbEvent):

- theDbEventID — Event-ID
- theTimeOfEvent — Auftrittszeitpunkt des Events (Genauigkeit ms)
- theOnDatabase — DBMS-Typ (Sybase, Informix, ...)
- theSessionID — ID der DB-Verbindung
- theTransDepth — Transaktionstiefe
- CommitOrRollb — {BG, CM, RB} (Begin, Commit, Rollback)

8.4.2.2 Detektion von Transaktions- und Connect-Events

Transaktionsereignisse werden durch Methodenaufrufe auf der Klasse PS_Connection ausgelöst. Folgende Methoden sind verfügbar: beginTransaction(), commitTransaction(), rollbackTrannsaction().

Um Transaktions-Events automatisch zu registrieren, müssen die Methoden der Connection-Klassen redefiniert werden. Die Methoden commitTransaction() und rollbackTransaction() müssen zusätzlich zum vorhandenen Code das eintretende Event signalisieren. Die Transaktionsmethoden sind allerdings in Persistence in der abstrakten Basisklasse PS_Connection definiert, deren Code an der Programmierschnittstelle nicht zugänglich ist.

Ansätze, die bestehende Klassenhierarchie zu erweitern, führten nicht zum Erfolg, weil entweder die Original-Methoden aufgerufen wurden oder Laufzeitfehler beim Verlassen einer lokalen DB auftraten [Bit96]. Eine praktikable Lösung bestand darin, die Connection-Klassenhierarchie vollständig auf eine neue Hierarchie abzubilden mit Modifikationen in den Transaktionsmethoden. Betroffen davon sind die Klassen PS_ConnectionGroup, PS_Connection sowie alle DBMS-spezifischen Subklassen (z.B. PS_SybConnection), die als OPS_ConnectionGroup, OPS_Connection usw. neu definiert wurden. Als Konsequenz daraus müssen Persistence-Anwendungen, die Transaktions-Events verarbeiten sollen, diese neuen Klassen verwenden. Zusätzliche Aufrufe sind dabei nicht notwendig. Der Nachteil dieser parallelen Klassenhierachie wird dadurch gemindert, daß die Übersetzung des Codes mit den erweiterten Klassen unabhängig von lokalen Applikationen geschieht.

In Analogie zu den Transaktions-Events lassen sich auch Disconnect-Events behandeln. Dies macht eine Redefinition des Destruktors der Klasse PS_Connection erforderlich, der beim Verlassen einer lokalen Datenbank aufgerufen wird. Darin muß kontrolliert werden, ob die Arbeit des Vermittlerprozesses sinnvoll fortgesetzt werden kann, da die Ausführung globaler Regeln geöffnete DB-Verbindungen voraussetzt. Unerwartete Disconnects sind allerdings nur

beim Feuern der Regeln zu ermitteln, indem vor ihrer Ausführung auf Vorhandensein der nötigen DB-Verbindungen getestet wird.

8.4.2.3 Time-Events

Das Auftreten von Time-Events ist durch genau einen Zeitpunkt auf der Zeitachse spezifiziert. Ein Zeitpunkt wird folgendermaßen dargestellt:

Tag : Monat : Jahr : Stunde : Minute : Sekunde

Time-Events lassen sich in absolute, periodische und relative Time-Events klassifizieren; entsprechend werden Subklassen einer allgemeinen Klasse `TemporalEvent` gebildet. Ein absolutes Time-Event besteht aus genau einem Zeitpunkt (Attribut: `AlarmTime`). Hingegen ist ein relatives Time-Event durch ein Datenbank-Event (`dbEventID`) und eine Zeitspanne (*Offset*) gekennzeichnet (`AdditionTime`). Somit ergibt sich der Auftrittszeitpunkt des relativen Events aus der Summe der Zeitpunkte von Datenbank-Event sowie Offset. Ein periodisches Event wird durch zwei Events (`BeginOfPeriod`, `EndOfPeriod`) und einen Offset (`Period`) charakterisiert. Die Events markieren Beginn bzw. Ende des Zeitraums der Periode, der Offset enthält die Länge einer Periode. Der Offset wird immer zum letzten Auftrittszeitpunkt des periodischen Events hinzuaddiert (beim ersten Mal zum Beginnzeitpunkt). Periodenbeginn und -ende werden im allgemeinen entweder durch ein absolutes Time-Event oder durch ein Datenbank-Event definiert, das Ende kann auch undefiniert sein. Abbildung 8.10 zeigt den Ausschnitt der Event-Klassenhierarchie mit Bezug auf Time-Events.

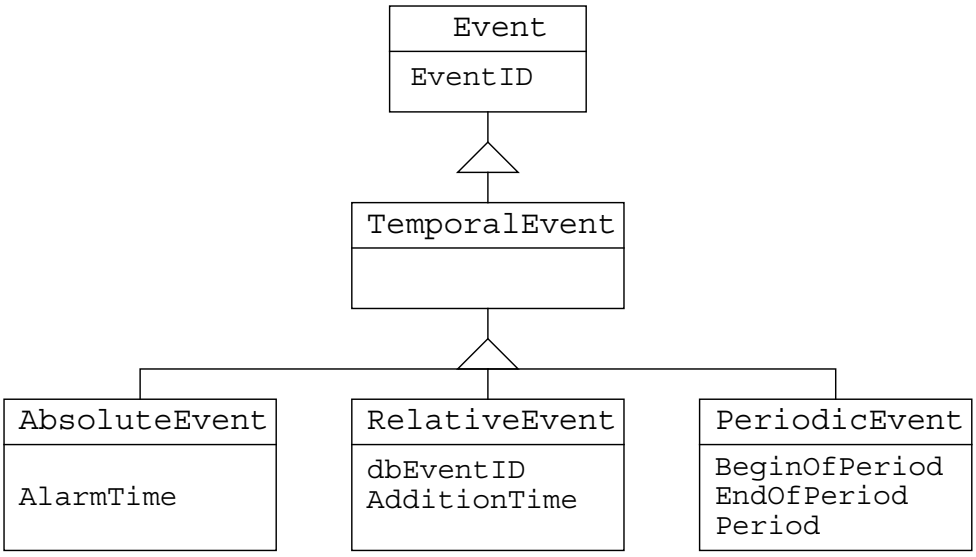


Abbildung 8.10: Klassenhierarchie für Zeitereignisse

8.4.3 Repräsentation der Regeln

In [DBM88] wird die Idee entwickelt, Regeln als erstklassige Objekte zu repräsentieren. Dafür werden zwei Argumente genannt: 1. Regeln können Attribute haben und mit anderen Objekten in Beziehung stehen, Regelgruppierungen sind dadurch möglich. 2. Regeln können als Objekte in gleicher Weise wie alle anderen Objekte der Datenbank behandelt werden (gleiche Methoden, gleiche Transaktionssemantik beim Zugriff).

Die Umsetzung dieser Idee erfolgt, indem eine abstrakte Basisklasse `Rule` zugrunde gelegt wird, die die Gemeinsamkeiten aller Regeln beinhaltet. Eine neue Regel wird eingeführt durch

Definition einer abgeleiteten Klasse <*Rulename*>`::Rule` und Erzeugung einer Instanz davon (*Single Instance Class*). Die virtuellen Methoden, die das Verhalten der Regel im Bedingungs- und Aktionsteil beschreiben, müssen in jeder Regelklasse redefiniert werden, da jede ECA-Regel i.allg. eine spezifische Condition bzw. Action aufzuweisen hat. Das beschriebene Vorgehen findet seine Analogie bei der Definition von Triggern in RDBMS, wobei jeder Trigger als eigenständiges Datenbankobjekt mit seinem spezifischen Verhalten (Stored Procedure) in der Datenbank abgelegt ist.

Die Basisklasse `Rule` hat folgende Attribute:

- `RuleID` Regel-Identifikator
- `PreConditionCoupl` Kopplungsmodus Event-Condition
- `PreActionCoupl` Kopplungsmodus Condition-Action
- `Active` Status der Regel (ENABLED / DISABLED)[11]
- `Priority` Priorität, bei Auswahl aus einer Regelmenge (höchste Priorität 0)

Folgende Methoden sind in der Klasse `Rule` definiert:
Die Methode `fire` startet die Ausführung der Regel in Abhängigkeit von den definierten Kopplungsmodi und dem Zeitpunkt relativ zur Transaktion, der über ein Flag mitgeteilt wird. Die `condition`-Methode prüft ein Prädikat auf dem aktuellen Datenbankzustand und liefert einen Rückgabewert (TRUE oder FALSE), um über die Ausführung des Aktionsteils zu entscheiden. Die `action`-Methode beinhaltet den Code des Ausführungsteils und wird in Abhängigkeit vom Condition-Action-Kopplungsmodus in der `fire`-Methode aufgerufen. Die EC- bzw. CA-Kopplungsmodi haben Einfluß darauf, ob eine Bedingung oder Aktion verzögert ausgeführt werden soll (DEFERRED oder DETACHED). In einem solchen Fall wird die Ausführung der Regel suspendiert und stattdessen der *Non-Immediate-RuleManager* (siehe Seite 161) durch Aufruf der `notifyNIR`-Methode verständigt. Die Methode `makeRuleNPO` liefert eine nichtpersistente Kopie eines Regelobjektes, die als solches in verschiedenen Listen des Regelmanagers temporär gehalten wird.

Die Methodenschnittstellen sind wie folgt definiert:

```
void fire(PersistenceObject *aPersistenceObject,
  void *aVptr, int *aAttType, int flags)
int condition (PersistenceObject *aPersistenceObject,
  void *aVptr)
void action (PersistenceObject *aPersistenceObject,
  void *aVptr)
void notifyNIR(PersistenceObject *aPersistenceObject,
  void *aVptr, int *aAttType, char *themode)
Rule* makeRuleNPO()
```

Die Parameter haben folgende Bedeutung:

- `aPersistenceObject` Instanz, die ein DB-Event verursacht hat
- `aVptr` Void-Pointer zum Attributwert, der vom Event betroffen ist
- `aAttType` Attributtyp
- `flags` Zeitpunkt relativ zur Transaktion:

11 Die semantischen Probleme beim Ein- und Ausschalten von Konsistenzregeln wurden bereits auf Seite 50 diskutiert.

	IMM	immediate
	EOT	nach Transaktion, vor Commit
	COMMIT	nach Commit der Transaktion
- `themode`	EC- oder CA-Kopplungsmodus (in Abhängigkeit vom Aufrufzeitpunkt innerhalb der Regel): {IMMEDIATE, DEFERRED, DETACHED}	

Die Semantik des Kopplungsmodus DETACHED entspricht der von *detached sequential causally dependent*, wie er in REACH [BBKZ93] eingeführt wurde. Das bedeutet, erst nachdem die triggernde Transaktion erfolgreich beendet wurde, erfolgt die Ausführung der Regel. Die erlaubten Kombinationen der EC- und CA-Kopplungsmodi basieren auf denen von HiPAC [DHW95].

8.4.4 Architektur und Funktionalität von MERU

Die Klassen zur Darstellung der Regeln und der Events werden in einem Persistence-Schema definiert, woraus die korrespondierenden C++-Klassen generiert werden. Die Methoden müssen dannn jeweils im File <*Class*>`_stubs.C` ergänzt werden. Da Persistence keine eigene persistente Datenhaltung unterstützt, werden alle Regeldaten als Teil des globalen Dictionary im RDBMS Sybase gespeichert. Bei der Definition des Schema-Files wird festgelegt, welche Events welche Regeln triggern. Dieses sind m:n-Beziehungen, die allerdings nicht direkt im Persistence-Objektmodell definiert werden können. Eine Hilfsklasse dient dazu, die m:n-Beziehung in zwei 1:n-Beziehungen aufzuteilen (siehe Abbildung 8.11).

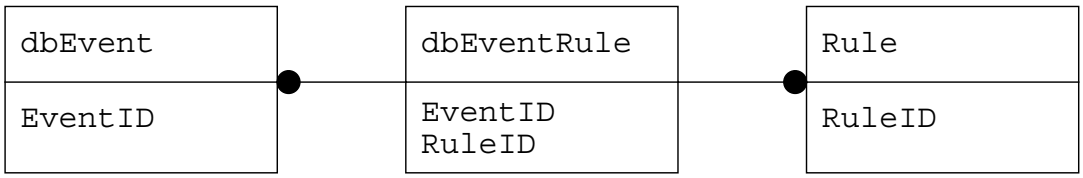


Abbildung 8.11: Beziehungen zwischen Regeln und DB-Events

Um die Ausführung der zu den Events gehörenden Regeln zu erleichtern, wurden einige Manager-Klassen implementiert. Ihr Aufruf erfolgt jeweils in unterschiedlichen Threads. Das ermöglicht eine erhöhte Parallelität zwischen eventauslösender Applikation und Regelausführung.

Zu den Managern gehören:

- *EventManager*
- *DB-EventManager*
- *Non-Immediate-RuleManager* (*NI-RuleManager*)

Der **EventManager** (Klasse `EventM`) detektiert die Events, die durch die `raise`-Funktion aus den Hook-Methoden heraus gemeldet werden. In der Methode `EventM::notify` wird zum detektierten Event aus der Klasse `dbEvent` mit Hilfe der Parameter `ClassName`, `AttributeName`, `OP` die `EventID` bestimmt und geprüft, ob es sich um ein gültiges Event handelt. Das Event wird (falls gewünscht) im globalen Log protokolliert. Anschließend erfolgt der Aufruf des DB-EventManagers, der in einem anderen Thread läuft, sowie der Aufruf des TemporalEventManagers (Abschnitt 8.4.5).

Der **DB-EventManager** (Klasse `dbEbentM`) wird aktiviert durch den Aufruf der Methode `dbEvent::notify`. Als Parameter bekommt er die Event-ID, das eventverursachende Objekt und ggf. Wert und Typ des geänderten Attributes. Diese Parameter können dann auch an die `fire`-Methode der zugehörigen Regel weitergereicht werden.

```
void dbEventM::notify
  (PS_OID *adbEventID, PersistenceObject *aPersistenceObject,
  void *aVptr, int *AttType)
```

Der DB-EventManager ermittelt aus der Klasse `dbEventRule` die ID's der zu den Events gehörenden Regeln und speichert diese temporär in der Liste `aListRuleObject` zusammen mit den Event-Parametern, die durch `notify` übermittelt wurden. Dies erfolgt unter der Voraussetzung, daß überhaupt Regeln gefunden werden bzw. diese auch eingeschaltet sind (`Active=TRUE`). Danach erfolgt für jede Regel in `aListRuleObject` der Aufruf der Methode `Rule::fire` mit den übernommenen Parametern. Anschließend erfolgt das Entfernen der Regel aus der Liste. Nach Ausführung der Aktion der Regel geht die Steuerung wieder an die eventauslösende Methode zurück. Die Auswahl der Regeln mit derselben Event-ID ist durch ihre Prioritäten bestimmt. Die Ausführungsstrategie für kaskadierende Regeln ist mit *depth-first* festgelegt.

Der **NI-RuleManager** ist verantwortlich für die Regeln mit dem Kopplungsmodus *deferred* oder *detached*. Weil diese Regeln erst am Ende der triggernden Transaktion gefeuert werden, müssen sie durch einen eigenständigen Manager verwaltet werden. Jede Regel ist der Transaktion zugeordnet, innerhalb der das zugehörige Event auftrat. Der NI-RuleManager wartet auf die Signalisierung des Transaktionsendes. Dies geschieht entweder bei Aufruf der Methode `commitTransaction()` oder `rollbackTransaction()` auf der jeweiligen Connection. Der NI-RuleManager verwaltet die Regeln mit den dazugehörigen Parametern in Listen, wobei ein Listenelement folgende Struktur aufweist:

```
struct ListRuleNIRElement
{
  Rule                          *aRule;
  PersistenceObject             *aPersistenceObject;
  void                          *aVPtr;
  int                           *aAttType;
  PS_Connection::DatabaseType *aDatabaseType;
  unsigned short                *aDBSessionID;
  int                           *aTransDepth;
}
```

Zum Regelobjekt und seinen Parametern kommen also noch die Informationen über den DBMS-Typ (z.B. Sybase oder Informix), die ID der Datenbankverbindung und die Transaktionstiefe. Diese drei Parameter identifizieren die eventauslösende Transaktion, mit denen bestimmt werden kann, für welches Transaktions-Event eine Regel gefeuert oder aus der Liste entfernt werden muß. Sie werden bei der Aktivierung des NI-RuleManagers am Ende der Transaktion übergeben. Der NI-RuleManager, der in einem separaten Thread läuft, wählt nach Erhalt der Parameter die zur Transaktion gehörigen Regeln aus und feuert diese entsprechend ihrer Priorität bzw. dem Auftrittszeitpunkt des zugehörigen Events. Abbildung 8.12 zeigt noch einmal (vereinfacht) das Zusammenwirken aller Manager bei der Regelausführung

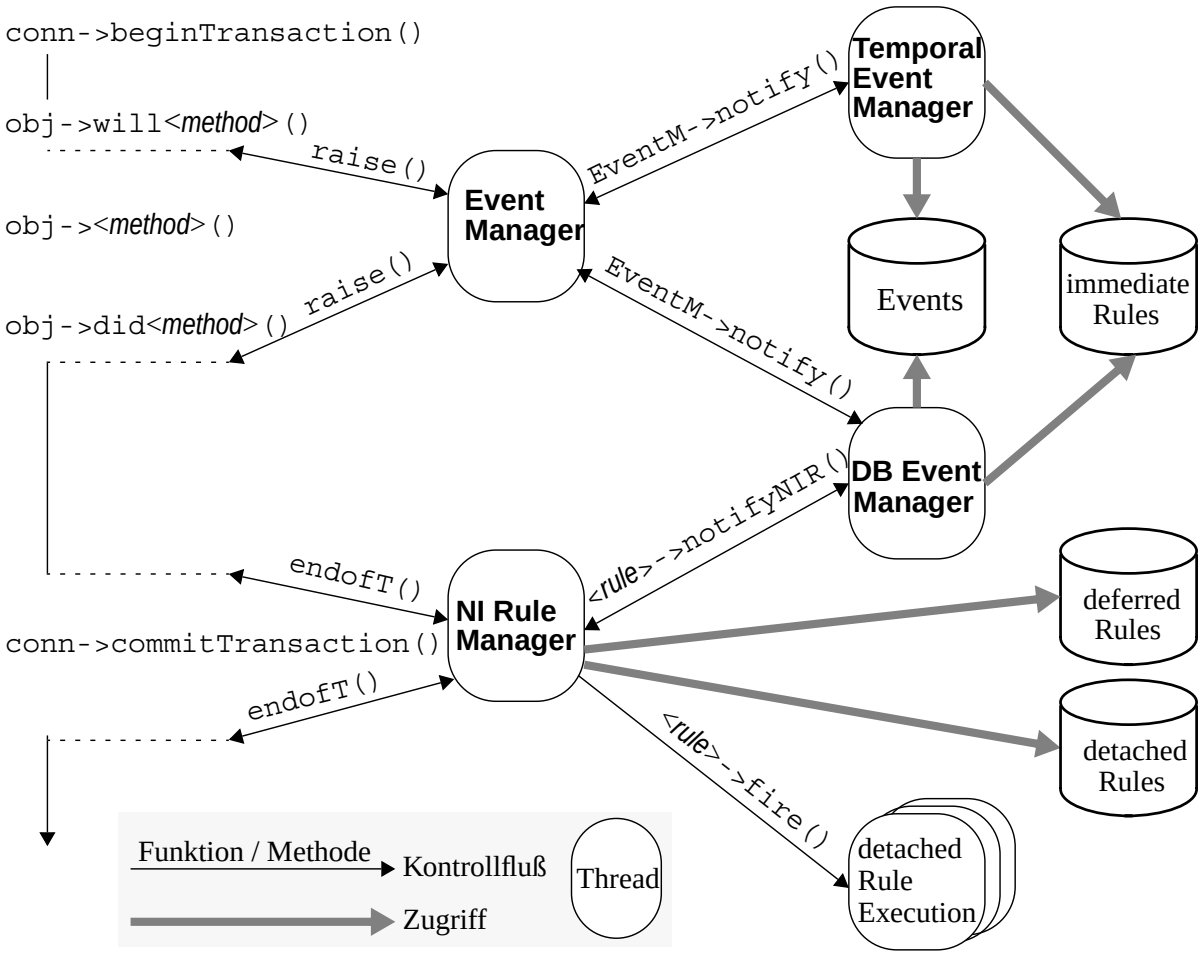


Abbildung 8.12: Architektur von MERU

8.4.5 Funktionsweise und Architektur des TemporalEventManagers

Der **TemporalEventManager** ist verantwortlich für das Management von Zeitereignissen. Dazu zählen die Eingabe und Speicherung von Time-Events, die Berechung und Detektion von relativen und periodischen Time-Events, die Detektion der Zeitpunkte sowie die Ausführung der an die Time-Events gebundenen Regeln. Zur Erhöhung der Parallelität wurde die Funktionalität des TemporalEventManagers in mehrere Threads aufgeteilt. Damit eine Applikation die temporale Komponente benutzen kann, muß sie diese aktivieren durch Aufruf der Prozedur `initdetector`. Folgende vier Threads werden dabei erzeugt:

- *StoreTempEvent* — Parsen und Speichern der eingegebenen Time-Events
- *RelativeTimeEvent* — Berechnung relativer Time-Events
- *DetectAlarm* — Detektion von Zeitpunkten der Time-Events im globalen System
- *FireRules* — Ausführen der Regeln

Abbildung 8.13 veranschaulicht die Thread-Architektur des TemporalEventManagers, die nachfolgend erläutert wird. Weitere Implementierungsdetails des TemporalEventManagers können [Hab96a] entnommen werden. Die hier skizzierte Realisierung beruht in Teilen auf der Implementierung des Zeitereignis-Managements im REACH-Prototypen [Zim97].

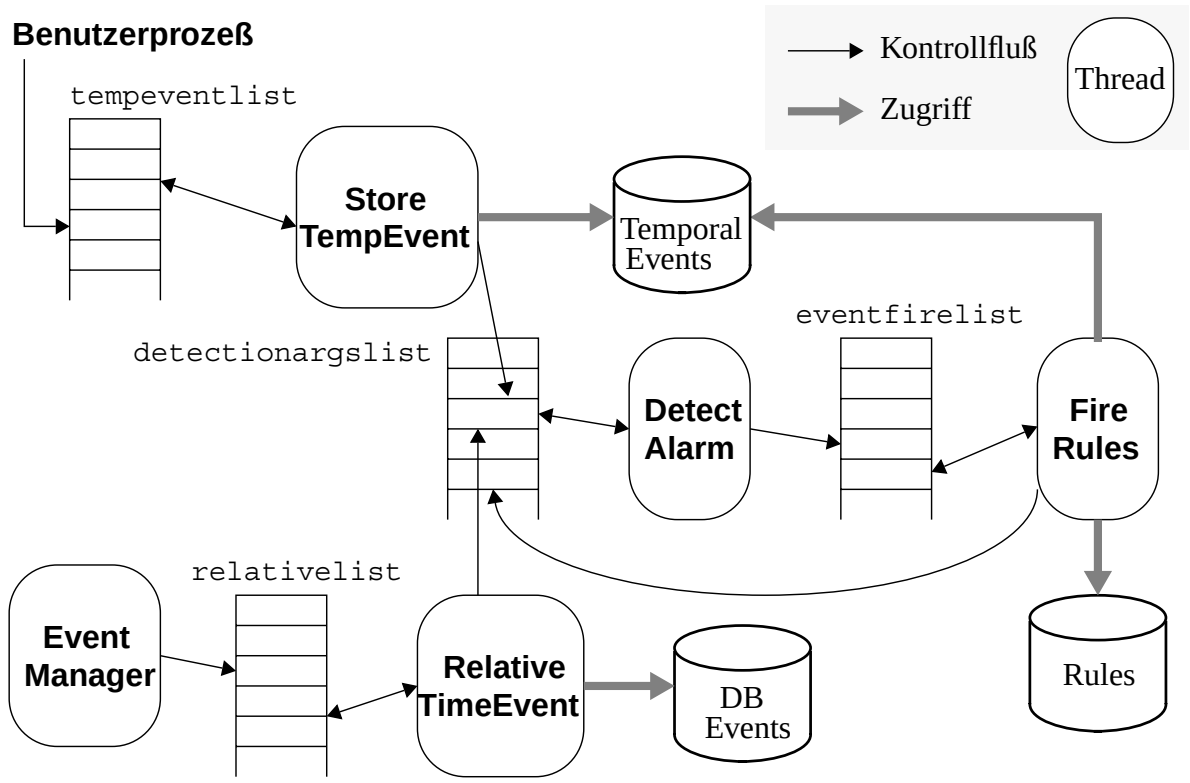


Abbildung 8.13: Thread-Architektur des TemporalEventManagers

Eingabe und Speicherung der Time-Events

Die Time-Events sind vom Benutzer in einem Objekt der Klasse `TimeEventArguments` zu spezifizieren, d.h. die entsprechenden Attribute müssen belegt sein. Die vollständige Attributbeschreibung kann [Hab96a] entnommen werden. Die Übergabe dieser Struktur an den TemporalEventManager erfolgt durch Aufruf der Prozedur `InsertTempEvent(TimeEvent-Argument *arguments)`. In dieser Prozedur wird das Event in einer Liste, `tempevent-list`, abgelegt und dem `StoreTempEvent`-Thread signalisiert, daß ein Time-Event detektiert werden soll. Nach Erhalt des Signals holt der `StoreTempEvent`-Thread den String aus der Liste und parst ihn, um die Art des Time-Events zu bestimmen. Im Anschluß daran wird das Time-Event in der Datenbank des Vermittlers mit den dazugehörigen Regeln gespeichert. Dies geschieht in Analogie zu den DB-Events..

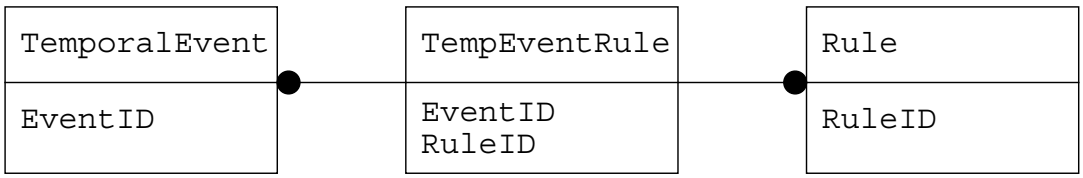


Abbildung 8.14: Beziehungen zwischen Regeln und Time-Events

Falls es sich um ein absolutes oder periodisches Time-Event (mit einem absoluten Beginnzeitpunkt) handelt, wird der Detektionszeitpunkt und die Event-ID des Time-Events in einer Liste, `detectionargslist`, eingefügt, wo alle eingehenden Alarmaufträge verwaltet werden.

Ein Listenelement hat folgende Struktur:

- `EventID` Identifiziert das Time-Event
- `Alarmtime` Zeitpunkt, der entdeckt werden soll (in s)
- `aPersistenceObject` Instanz, die das DB-Event verursacht hat (NULL, falls absolutes Time-Event)
- `aVptr` Void-Pointer zum Attributwert, der vom Event betroffen ist (NULL, kein Attribut zugewiesen oder absolutes Time-Event)
- `aAttType` Attributtyp

In der verketteten Liste sind die eingehenden Alarmaufträge aufsteigend sortiert, so daß der nächste zu detektierende Zeitpunkt an erster Stelle steht. Für gleiche Alarmzeitpunkte werden jeweils Unterlisten gebildet. Im Anschluß daran wird dem `Alarm`-Thread signalisiert, daß ein Time-Event detektiert werden soll.

Detektion von relativen Time-Events

Der `RelativeTimeEvent`-Thread berechnet relative Zeitereignisse aus Datenbank-Event und Offset bzw. periodische Time-Events, die durch ein Datenbank-Event ausgelöst wurden. Dabei wird der Zeitpunkt der Detektion des globalen Events im Vermittlersystem verwendet. Ebenso kann das Ende einer Periode festgestellt werden, wenn es durch ein DB-Event charakterisiert ist. Der EventManager (siehe Seite 160) trägt ein aufgetretenes DB-Event in die Liste `relativelist` ein und signalisiert es dem `RelativeTimeEvent`-Thread. Dies geschieht durch Aufruf der Prozedur `InsertRelativeEvent`, die folgende Schnittstelle aufweist:

```
void InsertRelativeEvent (PS_OID *dbEvent, PersistenceObject
  *aPersistenceObject, void *aVPtr,int *aAttType, long *TimeOf-
  dbEvent)
```

Die Parameter haben folgende Bedeutung:

- `dbEventID` Event-ID des Datenbank-Events
- `aPersistenceObject` Instanz, die ein DB-Event verursacht hat
- `aVptr` Void-Pointer zum Attributwert, der vom Event betroffen ist
- `aAttType` Attributtyp (NULL, falls kein Attribut zugewiesen)
- `TimeOfdbEvent` Auftrittszeitpunkt des DB-Events (in s)

Fall sich der `RelativeTimeEvent`-Thread im Wartezustand befindet und er das Signal bekommt, daß ein DB-Event vorliegt, holt er sich dieses aus der Liste `relativelist`. Damit wird in der Klasse `RelativeEvent` geprüft, ob Einträge für diese Event-ID existieren. Ist das der Fall, wird für jedes relative Time-Event der dazugehörige Offset aus der Klasse bestimmt. Damit wird dann der zu detektierende Zeitpunkt berechnet. Dieser Detektionszeitpunkt wird zusammen mit der ID des relativen Time-Events sowie den Parametern des DB-Events (Instanz, Attributwert und -typ) in die Liste `detectionargslist` eingefügt. Anschließend wird dem `DetectAlarm`-Thread signalisiert, daß ein relatives Time-Event detektiert werden soll.

Im Anschluß daran wird mit Hilfe des DB-Events ebenso in der Klasse `PeriodicEvent` geprüft, ob ein periodisches Time-Event ausgelöst oder beendet werden soll. Falls ein DB-Event zum Start eines periodischen Time-Events aufgetreten ist, wird sofort die Ausführung der zugehörigen Regel veranlaßt und der Startzeitpunkt gespeichert. Mit Hilfe des Startzeitpunktes wird eine wiederholte Detektion desselben periodischen Events vermieden.

Detektion von Zeitpunkten der Time-Events

Der Thread `DetectAlarm` ist für die Detektion der Zeitpunkte aller Time-Events verantwortlich. In der Liste `detectionargslist` sind sämtliche Zeitpunkte enthalten. Die Zeitpunkte sind chronologisch in dieser Liste geordnet; der nächste Zeitpunkt in der Liste ist somit auch zeitlich der nächste. Wenn ein Zeitpunkt dort eingefügt wird, wird dem `DetectAlarm`-Thread signalisiert, daß ein Time-Event detektiert werden soll. Falls sich der Thread im suspendierten Zustand befindet, weil kein Time-Event detektiert werden muß, holt er nach Empfang des Signals den Zeitpunkt des Time-Events aus der Liste, anderenfalls hat das Signal keine Wirkung. Mit dem Zeitpunkt des Time-Events programmiert der `DetectAlarm`-Thread durch den Systemaufruf `alarm()` die Stoppuhr des Prozesses und suspendiert seine Ausführung, indem er durch ein MUTEX in den Wartezustand versetzt wird. In `alarm()` wird als Parameter die Anzahl der Sekunden bis zum Senden des UNIX-Signals `SIGALRM` angegeben. Dies erfolgt durch Berechnung der Differenz von Alarmzeit und aktueller Zeit.

Empfängt der Prozeß das Signal `SIGALRM`, führt er eine Signal-Handler-Routine aus, in der der MUTEX wieder freigegeben wird, so daß der Thread fortgesetzt werden kann. Der `Alarm`-Thread holt den ersten Eintrag aus `detectionargslist` (d.h. das soeben detektierte Time-Event) und fügt es in die Liste `eventfirelist` ein. Anschließend wird dem Thread `FireRules` signalisiert, daß ein Time-Event aufgetreten ist und die entsprechenden Regeln ausgeführt werden müssen.

Im Anschluß daran holt er den nächsten Zeitpunkt eines Time-Events aus der Liste `detectionargslist` und programmiert erneut die Stoppuhr des Prozesses. Falls kein weiterer Alarmzeitpunkt angegeben ist, begibt er sich in den Wartezustand. Das Time-Event wird also nochmals in einer Liste abgelegt, um die Regeln ausführen zu lassen. Die Regelausführung in einem separaten Thread geschieht zur Erhöhung der Parallelität bzw. um Verzögerungen bei der Detektion nachfolgender Alarmzeitpunkte zu vermeiden. Diese können dann auftreten, wenn das Intervall zwischen zwei Alarmzeitpunkten im Verhältnis zur Ausführungszeit einer Regel sehr kurz ist. Eine solche Realisierung verringert in einem solchen Fall zumindest die Wahrscheinlichkeit einer Verzögerung, wenn es auch nicht immer eine Lösung ist.

Beim Eintragen des Time-Events in die Liste `detectionargslist` wird auch überprüft, ob der Alarmzeitpunkt dieses Events vor demjenigen des ersten Events in der Liste liegt. In diesem Fall wird die Stoppuhr des Prozesses durch den Aufruf von `alarm()` einfach neu programmiert, der ursprüngliche Wert geht dabei verloren. Nach Ausführung der zugehörigen Regel wird der Timer auf den nachfolgenden Event in der Liste programmiert.

Regelausführung

Der Thread `FireRules` ist für die Ausführung der Konsistenzregeln zuständig. Bei Eintreffen eines Signals vom `DetectAlarm`-Thread entnimmt er den entsprechenden Eintrag aus der Liste `eventfirelist`. Im Anschluß daran holt er sich aus der Datenbank die zu dem Time-Event gehörigen Regeln und ruft auf jeder Regel (sortiert nach deren Priorität) die Methode `fire` auf.

Periodische Events, bei denen das Ende-Event eingetreten ist, sowie absolute Time-Events werden aus der Datenbank gelöscht. Für ein periodisches Time-Event, das sein Ende noch nicht erreicht hat, wird der nächste Alarmzeitpunkt berechnet, in die Liste `detectionargslist` eingetragen und dem Thread `DetectAlarm` signalisiert, daß ein Time-Event detektiert werden soll. Anschließend begibt sich der `FireRules`-Thread wieder in den Wartezustand.

8.4.6 Besondere Implementierungsaspekte

Recovery

Nach einem Systemabsturz werden bei einem Neustart die absoluten Time-Events, die noch nicht detektiert wurden, neu initialisiert. Im Fall, daß deren Zeitpunkte bereits überschritten sind, werden die zugehörigen Regeln nachträglich ausgeführt.

Für periodische Time-Events sind nach einem Systemausfall mehrere Strategien für die Ausführung der zugeordneten Regeln denkbar: verspätetes Feuern aller Regeln, keiner oder einer Regel. Im Vermittlersystem wurde die Lösung gewählt, den letzten Auftrittszeitpunkt des periodischen Events vor dem Restart zu berechnen und für diesen sofort die Regel auszuführen. Eine Erweiterungsmöglichkeit besteht darin, unterschiedliche Recovery-Varianten als Konfigurationsinformation zu definieren.

Synchronisation der Uhren

In einem verteilten System mit Client/Server-Architektur besitzt jeder Rechner einen eigenen Zeitgeber. Das kann bei zeitkritischen Anwendungen Probleme verursachen, wenn nicht garantiert ist, daß alle Zeitgeber synchron laufen, wie es insbesondere für autonome Teilnehmer zutrifft. Ein Zeitgeber kann eine logische oder physikalische Uhr sein. Wenn es ausreicht, daß alle Rechner mit einer Zeit übereinstimmen, spricht man von *logischen Uhren*. Wenn die Systemzeit von der realen Zeit nur unwesentlich abweichen darf, so handelt es sich um *physikalische Uhren*.

Bei der Integration heterogener Systeme mit unabhängigen Zeitgebern gibt es prinzipiell zwei Ansätze, eine globale Übereinstimmung der Zeit gemäß den Korrektheitsanforderungen zu garantieren:
Eine Möglichkeit ist, einen bestimmten Rechner als Time-Server auszuwählen, der als Zeitgeber für alle anderen Teilnehmer fungiert. Mögliche Probleme dabei sind eine Überlastung des Servers bzw. die notwendige Gewährleistung der Ausfallsicherheit.
Im zweiten (dezentralen) Ansatz benutzt jeder Rechner weiterhin seinen lokal verfügbaren Zeitgeber. In bestimmten zeitlichen Intervallen findet eine Synchronisation der Uhrzeit zwischen allen Rechnern statt. Dabei existiert weiterhin ein spezifizierter Time-Server als globaler Zeitgeber.

Die Synchronisation logischer Uhren in einem verteilten System wurde durch den Algorithmus von Lamport gezeigt [Lam78], ein Algorithmus zur Synchronisation in einem verteilten System mit einem Time-Server kann z.B. [Cri89] entnommen werden.

Die Synchronisation der Zeitgeber in verteilten Systemen ist eigentlich Aufgabe des Netzwerkbetriebssystems und nicht die eines DBMS. Zur Lösung des Synchronisationsproblems wurde im Vermittlersystem der erste Ansatz gewählt. Ein Rechner stellt als Time-Server die global gültige Zeit bereit, die in einem Vermittlerprozeß genutzt wird. Die Zeitanforderung erfolgt nur beim Setzen der Alarmzeitpunkte.

Synchronisation der Threads

Die Kommunikation der Threads untereinander bzw. mit dem Benutzerprozeß erfolgt durch *Condition Variablen*. Die Zugriffe auf die unterschiedlichen Listen werden durch *Mutual Exclusion Locks* (MUTEX) synchronisiert.

Der allgemeine Ablauf von Synchronisation und Kommunikation im TemporalEventManager verläuft nach dem Erzeuger-Verbraucher-Modell:
In der Erzeuger-Prozedur wird mit einem MUTEX der Zugriff auf die Liste für andere gesperrt. Danach kann der Auftrag in die Liste eingetragen und einem evtl. wartenden Thread signalisiert werden, daß ein Job vorliegt. Im Anschluß daran wird die Liste mittels UNLOCK auf dem MUTEX wieder freigegeben.
Im Verbraucher befindet sich der Thread in einer Endlosschleife. Er sperrt den MUTEX und überprüft, ob ein Auftrag in der Liste ist. In diesem Fall wird der nächste Job aus der Liste geholt und ausgeführt. Ist die Liste leer, gibt er die Liste wieder frei und begibt sich in den Wartezustand, um auf einen Auftrag zu warten. Bei Ankunft eines entsprechenden Signals holt er sich dann den Job aus der Liste, gibt diese wieder frei und führt den Auftrag aus.

Außer den Listenzugriffen muß auch der Zugriff auf den Objekt-Cache von Persistence synchronisiert werden. Hierzu können jedoch die von Persistence angebotenen Klassen genutzt werden.

8.4.7 Bewertung

In diesem Abschnitt soll eine Einordnung und Bewertung des entwickelten Regelmanagers MERU im Vergleich zu aktiven Datenbanksystemen vorgenommen werden. Die zugrundeliegenden Kriterien sind [PDW+93] und [CB95] entnommen.

MERU wurde “on top of” Persistence implementiert und kann als Klassenbibliothek optional mit Persistence-Applikationen integriert werden. Diese Applikation kann der DML-Vermittler sein, der lokal ausgelöste Events verarbeitet, aber auch ein “rein globales” Anwendungsprogramm, das in Persistence geschrieben ist. Somit haben die Möglichkeiten und Restriktionen von Persistence Einfluß auf die Eigenschaften des darauf basierenden aktiven Systems.

Zur Charakterisierung des Systems werden die Eigenschaften, die im Feature Benchmark für aktive DBS (vgl. Abschnitt 3.2) diskutiert wurden, zugrunde gelegt. Dementsprechend erfolgt die Bewertung auf fünf Ebenen: Event, Condition, Action, Exekutionsmodell und Management-Fähigkeiten (bezogen auf das Management der Regeln).

Event

In MERU müssen Events definiert sein und als Objekte der Klasse `Event` existieren (*mandatory*). MERU unterstützt die Detektion von Datenbank-Operationen, wobei diese entweder von einem lokalen Benutzer aufgerufen oder direkt über die Schnittstelle von Persistence aktiviert werden durch den Aufruf einer CRUD-Methode. Transaktions-Events werden zwar detektiert und auch protokolliert, können aber explizit nicht in Regeln verwendet werden. Sie dienen ausschließlich dazu, die Steuerung der Regelausführung entsprechend der Kopplungsmodi zu unterstützen. Zeitereignisse können in MERU behandelt werden. MERU unterstützt nur primitive Events, ist aber im Hinblick auf komposite Events erweiterbar. Hierfür notwendig wäre die Realisierung eines *CompositeEventManagers* in Analogie zu den existierenden, der über jedes aufgetretene Event benachrichtigt werden müßte. In MERU beziehen sich Datenbank-Events immer auf Klassen, die zugleich Teil der Eventdefinition sind. Durch Ausnutzung der Hook-

Mechanismen von Persistence können Datenbank-Events vor oder nach ihrem Auftreten detektiert werden. Bei Commit-Events wird die before- und after-Semantik dadurch unterstützt, daß die Zeitpunkte vor oder nach Aufruf der Methode `commitTransaction()` erkannt werden. Dies wird ausgenutzt für die Realisierung der Kopplungsmodi *deferred* (nach letzter Aktion, vor Commit) und *detached dependent* (nach erfolgreichem Commit).

Condition

In Regeln kann ein Bedingungsteil auch weggelassen werden kann (*optional*), die Methode `condition()` wird zwar in jedem Falle aufgerufen, braucht aber keinen Code zu enthalten. Für MERU gilt, daß der *Scope* alle Datenbanken (aus möglicherweise heterogenen DBMS) umfassen muß. Dies ist nur unter der Voraussetzung möglich, daß die Verbindungen zu allen Datenbanken geöffnet sind. In MERU kann außer dem aktuellen Datenbankzustand auch der des betroffenen Objekts zum Zeitpunkt des Events ausgewertet werden. Ältere Datenbankzustande stehen zur Verfügung, wenn die Applikation über das lokale Datenbank-Gateway mit eingeschalteter Protokollierung ausgeführt wird (vgl. Abschnitt 7.1.5 auf Seite 121). Auf die Protokolle kann in gleicher Weise wie auf die Objekte selbst zugegriffen werden (eine Anwendung wird nachfolgend für asynchron replizierte Objekte beschrieben).

Action

Aktionen in MERU können Datenbanken und somit eingeschränkt auch die Regelbasis verändern (*update-db*, *update-rules*). Um den Benutzer im Fehlerfalle zu verständigen, besteht auch die Möglichkeit, den *Exception Handling*-Mechanismus von Persistence zu nutzen (*inform*). Ein Blockieren bzw. Zurückweisen von Operationen ist in MERU nur durch das Zurücksetzen der eventauslösenden Transaktion möglich (*abort*), in der `action`-Methode läßt sich dafür auch eine Alternative angeben (*do-instead*). Ansonsten gelten dieselben Aussagen wie bei der Condition (C-2 bis C-4).

Ausführungsmodell

In MERU wird nur die Granularität auf Instanzen-Ebene unterstützt, was sich aus der Verwendung der Hook-Methoden erklärt, die Instanzen-Methoden sind.
Für die Bindung der Event-Parameter gilt in MERU: Das betroffene Objekt (*instance*) kann an die Regel als Parameter geschickt werden (Instanz der Klasse `PS_Object`).[12] Eine Ausnahme bildet (seit Persistence Version 3.0) die Änderungsmethode auf Relationships. Diese erlaubt nun auch Objektmengen als Parameter, die eingefügt, gelöscht oder zugewiesen werden können. Die Erweiterung der Parameterschnittstelle der Methode `Rule::fire()` ist einfach, da die Klassen <*Class*>`_Cltn` Subklassen der abstrakten Basisklasse `PS_Collection` sind, die als Parametertyp verwendet werden kann.[13] Der Zugriff auf Attributwerte unmittelbar vor deren Änderung ist ebenfalls möglich (*prior*).
MERU unterstützt zeitliche Beschränkungen bzw. Alternativen explizit nicht. Allerdings ist eine Erweiterung des Systems denkbar, die bereits vorhandenen Zeitereignisse in der Weise auszunutzen, um zu prüfen, ob eine Regel rechtzeitig ausgeführt wurde, vergleichbar mit der Idee der *Milestones* [BBKZ93]. In MERU werden prinzipiell alle Regeln ausgeführt. MERU erlaubt die Definition absoluter Prioritäten. Die Auswahl einer Regel bei gleichen Prioritäten

12 Seit Version 3.0 wird in Persistence der Pointer auf das gelöschte Objekt im `didRemove()`-Hook nicht mehr belegt, so daß kein Zugriff auf den alten Zustand mehr möglich ist.

13 Diese Erweiterung ist z. Zt. in MERU nicht integriert.

ist in MERU nicht spezifiziert. Die Aufruftiefe kaskadierender Regeln ist nicht festgelegt. Da die Regeln nicht auf Triggern der lokalen DBMS basieren, werden deren Beschränkungen nicht übernommen (wie z.B. die Aufruftiefe 16 von Sybase-Triggern). Geschachtelte Transaktionen werden in MERU unterstützt, weil das Transaktionsmodell des Relational Object Manager von Persistence zugrundeliegt. Die Transaktionen sind allerdings jeweils nur auf einem DBMS ausführbar.

Management der Regeln

Das Datenmodell, auf dem ein Regelsystem basiert, hat Einfluß auf den Charakter des Gesamtsystems (z.B. Art der Events, Art der Objekte). Das Datenmodell des Vermittlersystems ist ein auf C^{++} basierendes objektorientiertes Modell, allerdings besteht die Besonderheit gegenüber aktiven objektorientierten Systemen [GD93, Zim97] darin, daß nur eine eingeschränkte Menge von Methoden-Events behandelt wird. Diese entspricht eher dem Triggerkonzept relationaler Systeme.
Für die Repräsentation der Regeln werden alle Möglichkeiten (*programming language*, *query language*, *objects*) in Kombination genutzt. Regeln werden als Klassen mit einer Instanz definiert und in ihrem strukturellen Teil in einer Datenbank gespeichert. Der Methodenteil (Bedingung und Aktion) wird in der Programmiersprache C^{++} beschrieben. Durch ein Attribut ist die Deaktivierung von Regeln möglich. Eine Besonderheit ist die Möglichkeit der Einbettung von SQL-Anweisungen in den Code.
Die Attribute der Regeln (Kopplungsmodi, Priorität, Aktivierungsflag) können dynamisch wie alle anderen DB-Objekte geändert werden. Auf das Problem einer dynamischen (De-)Aktivierung von Regeln wurde bereits in Abschnitt 3.3 hingewiesen. Eine Änderung des Verhaltens einer Regel macht eine Modifikation der entsprechenden Methode und Neuübersetzung der zugehörigen Regelklasse erforderlich. Es ist allerdings möglich, daß die Event- und Regelbibliotheken dynamisch gebunden werden, so daß bei einer Änderung nicht die Applikationen recompiliert werden müssen. Eine echte Dynamik des Regelcodes läßt sich dadurch erreichen, daß diese nur mit SQL-Zugriffsmethoden wie `selectMany()` oder `sendSQL()` programmiert werden, so daß eine rein interpretative Ausführung erfolgt.
In MERU wird die Zugriffskontrolle von Sybase ausgenutzt. Die Regeln liegen in einer separaten Datenbank, `DBMEDI`, auf die nur der Benutzer `medi` schreibend zugreifen kann. Regeln können nur mit den Mitteln des C^{++}-Datenmodells zusammengefaßt werden (*data model*), z.B. durch Bildung spezieller Regelklassen.

Tabelle 8.7 zeigt zusammenfassend die Bewertung von MERU anhand der zuvor aufgestellten Kriterien.

Event	
E-1	Role $\in$ {mandatory}
E-2	Source $\subset$ {db-operation, transaction, clock}
E-3	Type $\subset$ {primitive}
E-4	Scope $\subset$ {collection}
E-5	WhenRaised $\subset$ {before, after}
Condition	
C-1	Role $\in$ {optional}
C-2	Mode $\subset$ {immediate, deferred, detached dependent}
C-3	Scope $\subset$ {instance, target, database, all databases}
C-4	Available State $\subset$ {event, condition}

Tabelle 8.7: Charakterisierung des Regelmanagement-Systems MERU

Action	
A-1	Options $\subset$ {update-db, update-rules, inform, abort, do-instead}
A-2	Mode $\subset$ {immediate, deferred, detached}
A-3	Scope $\subset$ {instance, database, all databases}
A-4	Available State $\subset$ {event, action}
Execution Model	
X-1	Transition Granularity $\in$ {instance}
X-2	Binding Model $\subset$ {instance, prior}
X-3	Constraints $\subset$ {}
X-4	Scheduling $\in$ {all fired}
X-5	Priorities $\in$ {numerical}
X-6	Conflict Resolution {unspecified}
X-7	Run-Time Depth Limit $\in$ {unspecified}
X-8	Nested Transaction $\in$ {yes}
Management	
M-1	Data Model $\in$ {object-relational}
M-2	Description $\subset$ {programming language, extended query language, objects}
M-3	Operations $\subset$ {activate, deactivate}
M-4	Adaptability $\in$ {compile time, run time}
M-5	Authorization $\in$ {yes}
M-6	Modularization $\in$ {data model}

Tabelle 8.7: Charakterisierung des Regelmanagement-Systems MERU

8.5 Implementierung der asynchronen Replikationskomponente

8.5.1 Ablaufmodell

Die nachfolgend skizzierte asynchrone Replikationskomponente ist ein Beispiel dafür, wie das Ausführungsmodell für indirekte Eventverarbeitung aktiver Objekte, das im 5. Kapitel vorgestellt wurde (Seite 85), bei autonomen Komponenten Anwendung finden kann. Abbildung 8.15 veranschaulicht dabei die konkrete Umsetzung der im allgemeinen Modell beschriebenen Verarbeitungsschritte.

Voraussetzung für die Implementierung der Replikationskontrolle ist die Protokollierung der DML-Kommandos bzw. der Zustandshistorie. In einer Anwendung i wird ein Kommando aufgerufen, das zur Veränderung einer lokalen Tabelle führt (1). Wenn diese Tabelle in einer anderen Datenbank als Replikat existiert und somit Teil eines globalen Copy Constraints ist, wird vom auslösenden Kommando das Schreiben in eine Protokolltabelle initiiert (2). Das Protokoll wird global sichtbar nach Aufruf einer Verdichtungsprozedur (`csp_<`*Table*`>`), die aus einer DML-Protokolltabelle eine verdichtete Log-Tabelle anlegt (Suffix `_cdl`), in [Jab90] als indirektes Anbieten bezeichnet (3). Dies geschieht zu benutzerdefinierten Zeitpunkten, z.B. beim Commit, spätestens jedoch beim Lösen der Verbindung zum DB-Server. Die Annahme der Modifikationsinformation erfolgt ereignisgesteuert, in unserem System sind dafür Zeitereignisse vorgesehen, wie sie durch den TemporalEventManager als Teil des Vermittlersystems verwaltet werden. Die Verarbeitung der Modifikationsinformation, d.h. das Lesen der Log-Tabelle (5) bzw. die Propagierung zur korrespondierenden Tabelle in der Ziel-Datenbank (6) wur-

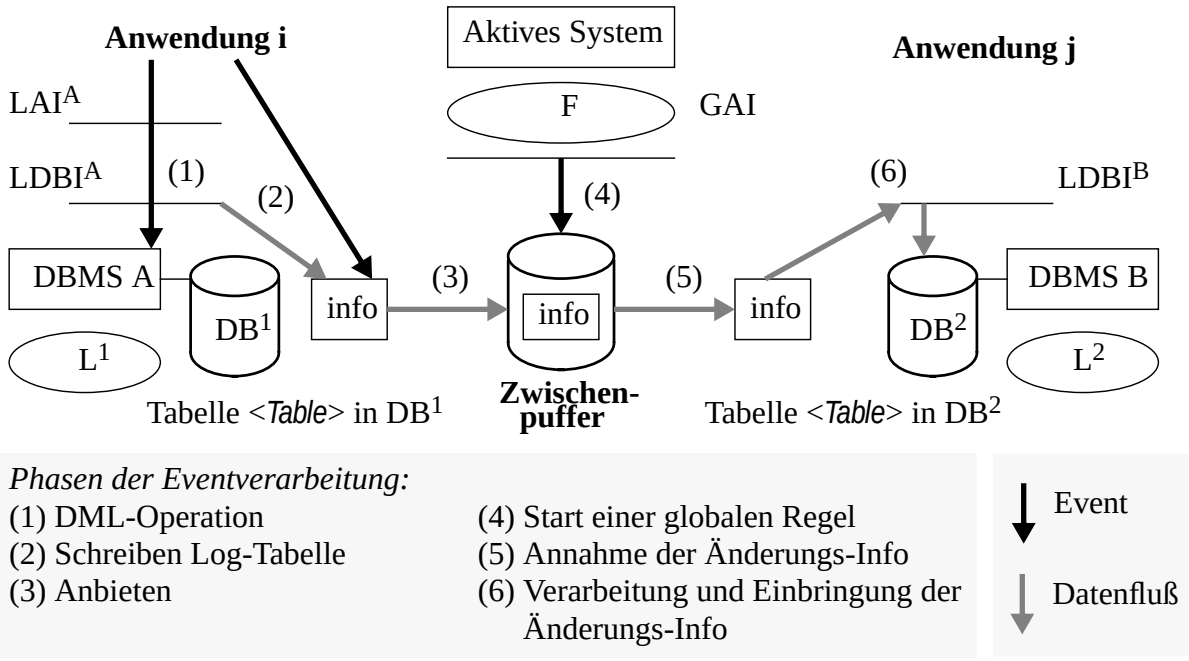


Abbildung 8.15: Verarbeitungsmodell für Replikationskontrolle im Vermittlersystem

de in Persistence realisiert. Dafür eignet sich dessen dynamische SQL-Schnittstelle, so daß erst beim Aufruf der Abgleichmethoden die Replikat-Tabellen als Parameter übergeben werden.

8.5.2 Spezifikation der Replikate und Algorithmen

Replikationsklassen

Für jede Abgleichstrategie wurde eine Klasse definiert als Subklasse der abstrakten Klasse `Replication`, die gemeinsam verwendete Methoden zum Zugriff auf die Log-Tabellen bzw. zum Propagieren der geänderten Tupel enthält (siehe Abbildung 8.16.). Um eine Replikationsbeziehung zwischen zwei Tabellen aus unterschiedlichen Datenbanken zu definieren, muß ein Objekt einer Subklasse von `Replication` erzeugt werden, wobei die Beziehung als String-Parameter an den Konstruktor übergeben wird. In Analogie zu ECA-Regeln enthält jede Replikationsklasse eine Methode `Condition` und `Action`. Somit lassen sich diese Methoden direkt im Bedingungs- bzw. Aktionsteil einer ECA-Regel aufrufen. In der Methode `Condition` wird geprüft, ob überhaupt ein Abgleich der Replikate infolge von lokalen Datenmodifikationen erforderlich ist. Ist dies der Fall, wird die `Action`-Methode aufgerufen, in der der eigentliche Abgleichalgorithmus abläuft.

Definition der Replikate

Die Definition der Replikationsbeziehung erfolgt über einen Parameter, `replicas`, bei Aufruf des Konstruktor einer Replikationsklasse [Hab96b]. Hierbei können mehrere Tabellen als Replikate angegeben werden (Bedingung: gleiche Anzahl von Attributen und Schlüsselattributen). Eine Tabelle wird in folgendem Format qualifiziert:

<DB_Server>.<DB_Name>.<Login>.<Tabelle>

Die Identifikation erfolgt also durch den Namen, den Login des Eigentümers sowie den Datenbanknamen. Hinzu kommt noch der DBMS-Name (‘SYB’ = Sybase, ‘INF’ = Informix). Die an erster Stelle stehende Tabelle wird als Bezugsrelation (BR) bezeichnet, der bei asymmetri-

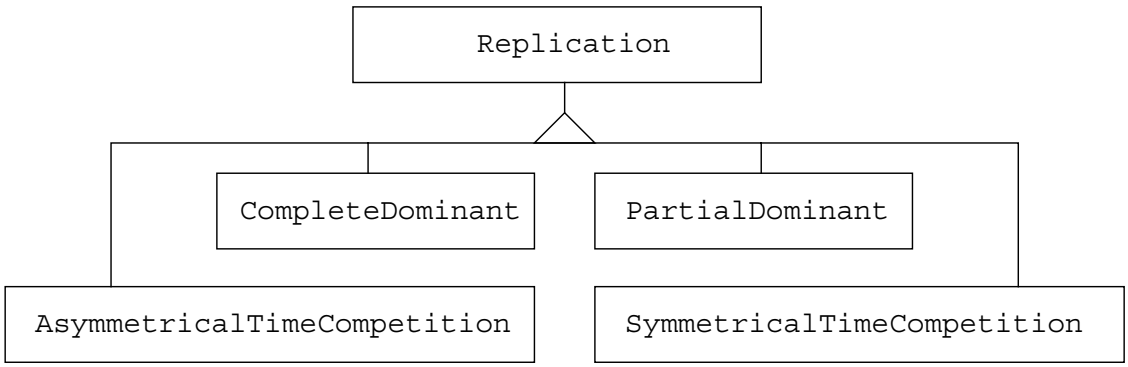


Abbildung 8.16: Hierarchie der Replikationsklassen

schen Abgleichstrategien besondere Bedeutung zukommt. Bei unterschiedlichen Spaltennamen müssen explizite Zuordnungen getroffen werden, ebenso bei Homonymen. Dies erfolgt in einem zusätzlichen String-Parameter des Konstruktors. Das Format der Zuordnung sieht folgendermaßen aus:

<Attribut_BR> = <DB_Server>.<DB_Name>.<Login>.<Tabelle>.<Attribut>

Voraussetzung ist natürlich eine Kompatibilität derAttributtypen. Hierbei gibt es eine Reihe von Besonderheiten zu beachten, die sich bei heterogenen DBMS ergeben und in [Hab96b] beschrieben sind.

Beispiel 8.3: (Definition einer Replikationsbeziehung)
Die Relationen `emp1`, `Ang` und `emp2` sollen als Replikate mit der Replikationsstrategie partielle Dominanz verwaltet werden, wobei `emp1` die Bezugsrelation ist. Die Attribute der Relationen sowie deren Korrespondenzen sind in Tabelle 8.8 dargestellt (äquivalente Attribute sind in gleicher Weise unterlegt).

Relation	Spalten			
emp1	ID	name	SSN	address
Ang	ID	IDA	name	adresse
emp2	name	SSN	ID	address

Tabelle 8.8: Replikationsbeziehung zwischen 3 Tabellen (Beispiel)

Die Parameter für den Aufruf des Konstruktors sehen hierbei aus wie folgt:

```
PartialDominant aStr (1, 3,
  "SYB staff1.user1.emp1 INF.Personal.Ben1.Ang
  SYB.db2.user2.emp2",
  "address=INF.Personal.Ben1.Ang.adresse",
  "ID=INF.Personal.Ben1.Ang.IDA",
  "SSN=INF.Personal.Ben1.Ang.ID");
```

Der erste Parameter ist die ID dieser Replikationsbeziehung, im zweiten Parameter ist die Anzahl der zusätzlichen Parameter gegeben, in diesem Fall drei Attributzuordnungen. Im dritten Parameter werden die an der Replikationsbeziehung teilnehmenden Tabellen aufgezählt. In den weiteren Parametern erfolgt die logische Zuordnung von Spaltennamen, soweit sie sich in den einzelnen Datenbanken unterscheiden. Für gleichnamige Attribute braucht keine explizite Zuordnung getroffen zu werden, mit Ausnahme von Homonymen (im Beispiel die Spalte `ID`). Die Attributzuordnung ist intern durch zwei Arrays (für Attribute und Schlüssel) organisiert,

die für jedes Replikat angelegt werden. Die Identifizierung der Spalten erfolgt hierbei über ihre Position in der Tabelle [Hab96b].

Hilfsdaten für die Replikationskomponente

In der Datenbank des Vermittlersystems (`dbmedi`) sind zusätzliche Informationen abgelegt, die von der Replikationskomponente benutzt werden:

In der Tabelle `LastChange` wird für jedes Replikat der Zeitstempel des (verdichteten) Protokolleintrags gespeichert, der beim letzten Abgleich noch berücksichtigt wurde. Die Identifizierung der Replikate erfolgt über die ID der Replikationsbeziehung sowie ihre Position im Parameter `replicas`.

Die Tabelle `Mod_ts_Log` hat die Funktion, nicht mehr benötigte Protokolleinträge zu identifizieren. Jeder Anwendung ist für jede Protokolltabelle, die für sie geführt wird, ein Tupel in `Mod_ts_Log` zugeordnet, die Protokolltabellen werden über die ID's von Datenbank, Besitzer und Tabelle eindeutig gekennzeichnet. Das Attribut `ID` identifiziert global eine Anwendung, hierbei handelt es sich um die ID der Replikationsbeziehung. Das Attribut `mod_ts` gibt an, bis zu welchem Zeitstempel die Einträge in den Protokolltabellen gelöscht werden können (zu Beginn mit 0 initialisiert).

Tabelle	Spalte				
`LastChange`	`ID`	`nr`	`mod_ts`		
`Mod_ts_Log`	`db_ID`	`user_ID`	`object_ID`	ID	`mod_ts`

Tabelle 8.9: Zusätzliche Tabellen für die Replikation

Condition-Methode

In der Condition-Methode der Replikationsklassen wird überprüft, ob ein Abgleich der Replikate notwendig ist. Hierfür werden die Einträge unterschiedlicher Log-Tabellen ausgewertet: alle bei vollständiger Dominanz und symmetrischer zeitlicher Konkurrenz, nur die der Bezugsrelation bei partieller Dominanz und asymmetrischer zeitlicher Konkurrenz.

Bei der Ausführung der Condition-Methode wird für jede zu berücksichtigende Protokolltabelle der Zeitstempel `mod_ts` aus der Tabelle `LastChange` geholt und nach Log-Einträgen gesucht, die einen größeren Zeitstempel als `mod_ts` besitzen. In diesem Fall sind neue Protokolleinträge hinzugekommen, was die Ausführung der Action erforderlich macht.

Action-Methode

Der allgemeine Ablauf in der Action-Methode sieht aus wie folgt:

1. Test bei jeder Tabelle, ob noch Log-Einträge im primären Protokoll existieren, die zunächst in das verdichtete Protokoll überführt werden müssen (z.B. für DML-Kommandos außerhalb von Transaktionen).
2. Je nach zugrundeliegender Replikationsstrategie Durchführung des Abgleichalgorithmus.
3. Bei fehlerfreier Durchführung Aktualisierung der Einträge in den Tabellen `LastChange` und `Mod_ts_log`.

Die Algorithmen werden am Beispiel einer Bezugsrelation (BR) und einer abhängigen Relation (AR) beschrieben und gelten in gleicher Weise für alle weiteren abhängigen Relationen (Ausnahme: symmetrische zeitliche Konkurrenz, ohne BR).

a) Vollständige Dominanz
Hierbei läßt sich durch einen Parameter der Action-Methode (`copy_all`) anwendungsspezifisch entscheiden, ob eine vollständige Übernahme der dominanten Relation geschehen soll, oder ob nur die Änderungen seit dem letzten Abgleichszeitpunkt propagiert werden sollen.
Im ersten Fall werden alle Tupel aus BR komplett nach AR übertragen, wo vorher alle Tupel gelöscht worden sind. Dieses Verfahren sollte dann gewählt werden, wenn die Gesamtanzahl der Tupel in den Replikaten relativ klein gegenüber der Anzahl der Modifikationen ist.
In der zweiten Variante werden nur beim ersten Aufruf des Algorithmus alle Tupel aus BR nach AR kopiert. Andernfalls wird für jedes Tupel in BR, das seit dem letzten Abgleich modifiziert wurde, der jüngste Protokolleintrag gesucht. Diese Modifikationen werden dann nach AR propagiert. Alle Modifikationen in AR werden dann zurückgesetzt, soweit sie nicht durch eine Propagierung einer Modifikation von BR entstanden sind. Voraussetzung ist, daß für jedes Replikat eine Protokolltabelle existiert.

b) Partielle Dominanz
Für jedes Tupel in BR, das seit dem letzten Abgleich modifiziert wurde, wird der jüngste Protokolleintrag gesucht. Diese Modifikationen werden nach AR propagiert. Es genügt, wenn nur die Modifikationen von BR protokolliert sind.

c) Asymmetrische zeitliche Konkurrenz
Für jedes Tupel in BR, das seit dem letzten Abgleichszeitpunkt modifiziert wurde, wird der jüngste Log-Eintrag gesucht. Diese Modifikationen werden dann nach AR propagiert. Hat in AR während des gleichen Zeitraums an einem korrespondieren Tupel auch eine Modifikation stattgefunden, dann wird die Modifikation aus BR nur dann nach AR propagiert, wenn ihr Zeitstempel größer ist. Die Korrespondenz zweier Tupel wird über gleiche Schlüsselwerte bestimmt. Voraussetzung für die Anwendung dieses Algorithmus ist, daß für jedes Replikat Log-Tabellen geführt werden.

d) Symmetrische zeitliche Konkurrenz
Für jedes Tupel, das seit dem letzten Abgleich in einem oder mehreren Replikaten modifiziert wurde, wird der jüngste Protokolleintrag in allen Log-Tabellen gesucht. Diese Modifikation wird dann auf alle anderen Replikate propagiert. Voraussetzung hierfür ist wiederum, daß für alle Replikate Log-Tabellen existieren.

Besondere Aspekte

Ein besonderes Problem bei der Implementierung stellen Fremdschlüsselabhängigkeiten dar. Eine solche Abhängigkeit kann auftreten, wenn in einem Replikat ein Tupel über einen Fremdschlüssel eine andere Relation referenziert oder der Primärschlüssel des Replikats einen Fremdschlüssel für eine andere Relation darstellt. Im ersten Fall kann ein Tupel nicht mit INSERT eingefügt, im zweiten Fall nicht mit DELETE gelöscht werden, wenn die lokale referentielle Integrität vom DBMS überwacht wird. Wenn beim Abgleich der Replikate ein solcher Fall auftritt, wird der Log-Eintrag einer Tabelle, der auf das Replikat propagiert werden soll, erneut in das verdichtete Protokoll eingefügt. Die Werte des Protokolleintrags bleiben hierbei identisch, bis auf den Zeitstempel `mod_ts`, der so erhöht wird, daß dieser Log-Eintrag beim nächsten Abgleich der Replikate wieder zu berücksichtigen ist. Durch dieses Verfahren wird das Einfügen oder Löschen eines Tupels bis zu einem Zeitpunkt verzögert, an dem die referentielle Integrität nicht verletzt wird.

Die Diskussion über die Seiteneffekte der Ausführung der Abgleichalgorithmen zeigt gleichzeitig die besondere Problematik bei asynchroner Verarbeitung. Wenn überschriebene Daten schon in lokalen Anwendungen verarbeitet wurden, so müssen ggf. die daraus produzierten Daten invalidiert werden. Das Überschreiben bzw. Löschen lokaler Daten kann lokale Constraints berühren. Von daher muß die Kontrollbeziehung zwischen globalen Copy Constraints und lokalen Abhängigkeiten (z.B. referentielle Integritätsbedingungen) definiert werden.

Der bei einigen Abgleichverfahren stattfindende Vergleich zeitbezogener Log-Einträge, die von autonomen Systemen geschrieben wurden, setzt eine angemessene Synchronisierung der lokalen Zeitgeber voraus. Dabei gehen wir von einer groben Zeitgranularität aus, so daß mögliche Abweichungen zwischen den lokalen Uhren in einem Toleranzbereich liegen (vgl. Seite 29).

8.5.3 Bewertung der Replikationskomponente

Die Implementierung der Replikationskomponente im Vermittlersystem ist unter einer Reihe von Aspekten erweiterbar:

Die Klassenhierarchie für Replikationsbeziehungen könnte um neue anwendungsspezifische Abgleichstrategien ergänzt werden, um außer dem Kriterium Zeit auch andere Heuristiken zur Auflösung von Konflikten (speziell bei symmetrischen Beziehungen) zu nutzen.

Bei der Definition der Replikationsbeziehungen ist eine größere Flexibilität sinnvoll. In der vorliegenden Implementierung liegt als Granulat immer eine Tabelle zugrunde. Denkbar sind dafür aber auch Datenbanken oder Sichten. Eine Replikationsbeziehung beruht im Vermittlersystem immer auf einer Wertgleichheit korrespondierender Objekte. Diese könnten jedoch ganz allgemein über eine beliebige arithmetische Funktion verknüpft werden, ggf. lassen sich auch SQL-Aggregationsfunktionen einbeziehen. Anstelle von Primärschlüsseln sind manchmal alternative Korrespondenzfunktionen über andere Spalten wünschenswert.

Ein wichtiges Problem stellt die Wahrung der globalen Transaktionssemantik bei der Durchführung der Abgleichstrategien dar, hierfür sollten die Möglichkeiten zur Realisierung verteilter Transaktionen genutzt werden (siehe Seite 146).

Durch Nutzung aktiver Mechanismen ist eine flexible ereignisgesteuerte Propagierung lokaler Modifikationen auf semantische Replikate möglich, bei loser Kopplung der Komponenten eignen sich primär asynchrone Verfahren, wie sie im Prototyp realisiert wurden. Die Wartung der lokalen Protokolltabellen bedeutet allerdings einen Performance-Verlust gegenüber einem Zugriff über den Transaktions-Log, bei dem Aufzeichnung und Transaktion synchron erfolgen. Eine solche Lösung, wie sie in zahlreichen kommerziellen Replikationsservern zum Einsatz kommt [Soe96], setzt allerdings Kenntnisse der internen Struktur der Logs voraus bzw. erfordert Eingriffe in den lokalen DBMS-Server, siehe z.B. [Inf94].

Einen Überblick über Replikationsverfahren gibt Adly in [Adl95], Ceri u.a. entwickeln eine Klassifikation von Replikationsmethoden für die Propagierung von Änderungen [CHKS91]. Dabei bezeichnen sie den asynchronen Fall als *Independent*, d.h. Updates können unabhängig und autonom auf einzelnen Knoten erfolgen, so daß Inkonsistenzen entstehen können, die von Zeit zu Zeit korrigiert werden müssen. Einen entsprechenden Abgleichalgorithmus, basierend auf *History Logs*, stellen die Autoren in [CHKS95] vor.

8.6 Ein Werkzeugkasten zur Eingabe globaler Integritätsbedingungen

8.6.1 Spezifikation globaler Integritätsbedingungen (GISpeL)

8.6.1.1 Sprachdefinition

Die Idee des Werkzeugkastens zur Definition von globalen Integritätsbedingungen besteht vor allem darin, die notwendigen Schritte zur Generierung eines lauffähigen Vermittlersystems zu vereinfachen. Zu diesem Zweck wurde die Sprache GISpeL zur Spezifikation von globalen Integritätsbedingungen entworfen (**G**lobal **I**ntegrity **Spe**cification **L**anguage). Die komplette Grammatik der Sprache kann Anhang A entnommen werden.

Eine GISpeL-Datei ist in zwei (optionale) Einheiten unterteilt, eine Tabellensektion und eine Regelsektion. In GISpeL werden vier verschiedene Kategorien von Objekten unterschieden. Untergeordnete Objekte werden durch übergeordnete Objekte qualifiziert. Da Klassen jeweils eine Tabelle repräsentieren, werden beide Begriffe nachfolgend synonym gebraucht.

- Datenbanksysteme
- Datenbanken
- Klassen
- Attribute

Tabellensektion

In der Tabellensektion können für Tabellen bestimmte Services angefordert und Abkürzungen für Objektnamen (Aliase) definiert werden. Die klassenbezogenen Dienste zum Protokollieren und Signalisieren, die im Datenbank-Gateway bereitstehen (siehe Beschreibung der Konfiguration auf Seite 130) werden syntaktisch ausgedrückt durch das Schlüsselwort `log`.

Beispiel 8.4: (Definition eines Log-Dienstes)
Um für eine Sybase-Tabelle `Table` der Datenbank `dbbase`, die durch die Klasse `Class1` repräsentiert ist, einen Log-Dienst anzufordern, genügt die Anweisung

```
Sybase.dbbase.Class1 log;
```

Dadurch wird bewirkt, daß in der Konfigurationsdatei des Gateways (`gateway.conf`) folgendes eingetragen wird:

```
user
    dbbase
        Table
```

Zusätzlich zu den Dienstanforderungen können Abkürzungen bzw. Aliase für Namen von Bezeichnern in der Tabellensektion definiert werden durch Zuweisungen, auf dessen linker Seite der Aliasname steht. Möglichkeiten dafür sind in nachfolgendem Beispiel illustriert.

Beispiel 8.5: (Tabellensektion)

```
#Tabellensektion einer Datei
    MN_Wire = Sybase.dbmn.MN_Wire log;
    v = Sybase.dbmn.mn_Wire.voltage;
    IN_Wire = Informix.dbin.IN_Wire;
```

Regelsektion

Die Regelsektion dient der Definition von globalen Integritätsbedingungen gemäß der Syntax von GISpeL. Dabei werden gerichtete und ungerichtete Existenz- und Wertabhängigkeiten unterschieden mit der Semantik, wie sie auch bei [CW93, CKSG94] zugrunde liegt.

Existenzabhängigkeiten werden allgemein so definiert:
gerichtet: *lhs* :- *rhs*
ungerichtet: *lhs* :- *rhs*
Wertabhängigkeiten verwenden folgende Operatoren:
gerichtet: *lhs* := *rhs*
ungerichtet: *lhs* :=: *rhs*

Die linke Seite (*lhs*) bzw. rechte Seite (*rhs*) einer Abhängigkeit kann wie folgt definiert werden. Dabei werden folgende Bezeichner verwendet: *dbs* (Name eines Datenbanksystems), *db* (Name einer Datenbank), *class* (Name einer Klasse), *a* (Name eines Attributs).

ein Attribut: *dbs.db.class.a*
mehrere Attribute einer Klasse: *dbs.db.class.*<a_1, a_2, . . ., a_n>
mehrere Klassen: *dbs.db.*<$class_1.a_1$, . . ., $class_n.a_n$>

Voraussetzung ist, daß die Typen korrespondierender Attribute kompatibel sind. Zusätzlich können Prädikate beliebiger Komplexität in GISpeL definiert werden (außer bei ungerichteten Existenzabhängigkeiten), eingeleitet durch das Schlüsselwort `WHERE`, siehe auch Anhang A.

Beispiel 8.6: (Definition einer Abhängigkeit in GISpeL)
Als Beispiel diene eine ungerichtete Wertabhängigkeit, durch die definiert wird, daß alle Kabeltypen, die sowohl im Netzwerk Italiens als auch im Netzwerk Mailands vorkommen, in den Attributen `wire_size` bzw. `cross_section` die gleichen Werte haben:

```
Sybase.dbmn.MN_WireType.cross_section
        :=: Informix.dbin.IN_WireType.wire_size
            WHERE MN_WireType.type = IN_WireType.type;
```

Aus einer GISpeL-Anweisung werden ECA-Regeln generiert, die von MERU verarbeitet werden können. Die Datenbankereignisse, die zur Konsistenzverletzung führen können, können blockiert (BLOCK) oder propagiert (PROPAGATE) werden. Diese beiden Strategien können in GISpeL für Existenz- und Wertabhängigkeiten gewählt werden. Insgesamt gibt es sechs verschiedene Ereignistypen, für die eine Strategie vorgegeben werden kann, wobei der Standardwert jeweils PROPAGATE ist. Diese Datenbankereignisse (und mögliche Abkürzungen) können Tabelle 8.10 entnommen werden. Die Strategien können linkerhand und rechterhand des Operators auch unterschiedlich festgelegt werden, wie nachfolgendes Beispiel 8.7 zeigt:

Bezeichner	Bedeutung
I(ns(ert))L(eft)	Einfügen eines Objekts in Klasse linkerseits des Operators
I(ns(ert))R(ight)	Einfügen eines Objekts in eine der Klassen rechterseits des Operators
U(pd(ate))L(eft)	Ändern eines der Attribute linkerseits des Operators
U(pd(ate))R(ight)	Ändern eines der Attribute rechterseits des Operators
D(el(ete))L(eft)	Löschen eines Objekts aus Klasse linkerseits des Operators
D(el(ete))R(ight)	Löschen eines Objekts aus einer der Klassen rechterseits des Operators

Tabelle 8.10: Datenbankereignisse in GISpeL

Beispiel 8.7: (Definition von Konsistenzwahrungsstrategien in GISpeL)
In folgender GISpeL-Anweisung wird eine gerichtete Existenzabhängigkeit so definiert, daß das Einfügen von neuen Kabeln in die Mailand-Datenbank erlaubt ist und diese in die Datenbank des Italien-Netzwerks propagiert werden, während das Löschen dieser Kabel aus der Italien-Datenbank verboten ist:

```
Informix.dbin.IN_Wire.wire_id :- Sybase.dbmn.MN_Wire.wire_id
        PROPAGATE InsertRight BLOCK DeleteLeft;
```

Die Sprache GISpeL wurde auch um die Definition von Zeitereignissen erweitert, um zeitlich abgeschwächte Konsistenzbedingungen ausdrücken zu können. Damit wird festgelegt, wann die Konsistenz überprüft und mit deren Wiederherstellung begonnen werden soll. Absolute Zeitpunkte werden mit dem Schlüsselwort `AT` eingeleitet und durch Datum und Uhrzeit definiert. Relative Zeitpunkte sind mit `AFTER` gekennzeichnet, dem sich die Definition eines Zeitintervalls anschließt. Das Zeitintervall besteht (optional) aus einer Anzahl von Zeiteinheiten sowie der Angabe der Zeiteinheit selbst (`MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`). Alternativ läßt sich das Intervall auch im Format einer Uhrzeit definieren. Die Definition periodischer Zeitpunkte umfaßt optional einen Startzeitpunkt (Schlüsselwort `AT`), die Angabe eines Zeitintervalls (Schlüsselwort `EVERY`) sowie optional einen Endzeitpunkt (Schlüsselwort `UNTIL`).

Beispiel 8.8: (Definition periodischer Konsistenzwahrung in GISpeL)
Die im vorigen Beispiel definierte Existenzabhängigkeit soll täglich überprüft werden:

```
Informix.dbin.IN_Wire.wire_id :- Sybase.dbmn.MN_Wire.wire_id
      AT 1.1.1997 EVERY day UNTIL 7.2.1997
```

8.6.1.2 Parser

Die lexikalische und syntaktische Analyse von GISpeL-Dateien erfolgt unter Verwendung der UNIX-Werkzeuge lex und yacc. Weitere Einzelheiten sind in [Mrs96] beschrieben. Der Parser produziert den C^{++}-Code von ECA-Regeln entsprechend vordefinierter Ableitungen, wobei von der Art konsistenzverletzender Ereignisse ausgegangen wird. Dabei liegen Regel-Templates zugrunde.

Existenzabhängigkeiten

Grundlage bilden GISpeL-Anweisungen in folgendem Format:

$dbs_1.db.class.\langle a_1, \ldots, a_n\rangle$:- $dbs_2.db.\langle class_1.x_1, \ldots, class_n.x_n\rangle$ WHERE P ;

$dbs_1.db.class_1.\langle a_{11}, \ldots, a_{1n}\rangle$:-: $dbs_2.db.class_2.\langle a_{21}, \ldots, a_{2n}\rangle$;

Die Konsistenz gerichteter Existenzabhängigkeiten kann durch folgende Operationen verletzt werden (vgl. Seite 91):

1. Löschen von Objekten aus *class*
2. Änderungen der Attribute $a_1, \ldots, a_n$
3. Einfügen von Objekten in $class_1, \ldots, class_n$
4. Änderungen der Attribute $x_1, \ldots, x_n$
5. Änderungen der Attribute aus P

Bei ungerichteten Existenzabhängigkeiten sind dagegen folgende Operationen kritisch:

1. Löschen von Objekten aus $class_1$ oder $class_2$
2. Einfügen von Objekten in $class_1$ oder $class_2$
3. Ändern eines der Attribute $a_{11}, \ldots, a_{1n}$ oder $a_{21}, \ldots, a_{2n}$

Für jedes dieser Ereignisse werden ECA-Regeln generiert, die bei Eintreten des Ereignisses überprüfen, ob die Integrität verletzt wurde, um diese (nach vorgegebener Strategie) wiederherzustellen. Der Regelcode besteht aus C^{++}-Anweisungen, die die Persistence-Klassen und die darauf definierten Methoden benutzen. Sie werden aus vorgefertigen Templates generiert. Für Änderungen an Schlüsselattributen werden keine ECA-Regeln erzeugt, da diese in Persistence nicht erlaubt sind.

Bei allen Regeln wird überprüft, ob tatsächlich eine Verbindung (Connection) zu den beteiligten Datenbanksystemen existiert und ob eine Transaktion begonnen wurde. Falls keine Transaktion begonnen wurde, wird eine neue gestartet, da in Persistence keine impliziten Transaktionen erlaubt sind.

Falls ein Prädikat *P* angegeben war, wird im Bedingungsteil der ECA-Regel überprüft, ob das Prädikat das vom Ereignis betroffene Attribut qualifiziert. Ist dies nicht der Fall, liegt keine Integritätsverletzung vor. Für alle eingefügten, gelöschten oder geänderten Objekte der Klassen $class_1$, . . ., $class_n$ wird überprüft, ob sie das angegebene Prädikat erfüllen. Anschließend werden im jeweils korrespondierenden DBS die aktuellen Werte der beteiligten Attribute abgefragt. Dabei muß mindestens ein Wert vorhanden sein, oder es darf kein Wert vorhanden sein - je nach Ereignis. Z.B. darf beim Löschen eines Objekts aus *class* bei gerichteter Existenzabhängigkeit kein korrespondierendes Objekt vorhanden sein, während beim Erzeugen eines neuen Objekts einer der Klassen $class_1$, . . ., $class_n$ mindestens ein korrespondierendes Objekt existieren muß.

Die Abfrage der korrespondierenden Klasse erfolgt für 3 verschiedene Fälle unterschiedlich:

1. Einfache Existenzabhängigkeit mit nur jeweils einem Attribut ohne Angabe eines Prädikats
2. Existenzabhängigkeit mit jeweils mehr als einem Attribut auf jeder Seite und/oder Angabe eines Prädikats
3. Gerichtete Existenzabhängigkeit mit mehr als einer Klasse auf der rechten Seite

Im ersten Fall wird der Wert des korrespondierenden Attributs mit der Persistence-Methode `get<`*Attributname*`>()` abgefragt. Im zweiten Fall geschieht dies mit der Persistence-Klassenmethode <*corresponding_class*>::`querySQLWhere()`, wobei auch das Prädikat mit übergeben wird. Im letzten Fall wird die `selectMany()`-Methode von Persistence aufgerufen, evtl. auch inklusive des angegebenen Prädikats.

Im Bedingungsteil wird lediglich die Existenz eines korrespondierenden Objektes getestet - nicht die Anzahl. Der Bedingungsteil liefert somit nur einen Wahrheitswert, der den Aktionsteil auslöst oder nicht.

Im Aktionsteil werden Verbindung und Transaktionsbeginn überprüft. Anschließend wird in Abhängigkeit von der in der GISpeL-Anweisung definierten Strategie entweder die aktuelle Transaktion zurückgesetzt (Blockieren), oder es werden die korrespondierenden Attribute angepaßt, die korrespondierenden Objekte gelöscht oder neue eingefügt, je nach Datenbankereignis. Je nach Strategie werden die Regeln entweder sofort nach Eintritt des Ereignisses (Blokkieren) oder unmittelbar vor dem Commit der Transaktion (Propagieren) ausgeführt. Ein sofortiges Propagieren birgt die Gefahr der Verletzung globaler Transaktionssemantik. Das Blockieren wird sofort veranlaßt, um das Problem kaskadierender Aborts zu vermeiden.

Wertabhängigkeiten

Hierbei bilden GISpeL-Anweisungen in folgendem Format die Grundlage:

$dbs_1.db.class.<a_1, \ldots, a_n>$:= $dbs_2.db.<class_1.x_1, \ldots, class_n.x_n>$ WHERE P ;

$dbs_1.db.class_1.<a_{11}, \ldots, a_{1n}>$:=: $dbs_2.db.class_2.<a_{21}, \ldots, a_{2n}>$;

Folgende Operationen sind potentiell konsistenzverletzend und müssen durch ECA-Regeln behandelt werden.

1. Einfügen neuer Objekte in eine der Klassen $class$, $class_1$, . . ., $class_n$
2. Ändern eines der Attribute a_1, ..., a_n, a_{11}, ..., a_{1n}, a_{21}, ..., a_{2n}
3. Ändern eines der Attribute des Prädikats P

Die Regel-Templates für Einfügen und Ändern können zusammengefaßt werden, da die Konsistenzüberprüfung jeweils auf gleiche Weise erfolgt. Eine Verletzung ist nur dann gegeben, wenn ein korrespondierendes Objekt existiert.

8.6.2 Editor (GISpeL-GUIDe)

Um GISpeL-Dateien zu erzeugen, genügt schon ein einfacher Editor, so daß die Dateien anschließend zur Generierung der Regeln weiterverarbeitet werden. Um diesen Prozeß weiter zu vereinfachen, wurde eine Benutzerschnittstelle, GISpeL-GUIDe, entwickelt. GUIDe ist ein Akronym und steht für **G**raphical **U**ser **I**nterface & **De**rivator. GUIDe dient einerseits dazu, GISpeL-Dateien zu editieren, zum anderen sind verschiedene Prozesse zusammengefaßt, die zum Erzeugen eines regelbasierten Vermittlersystems notwendig sind. Dazu gehören: Übersetzen von GISpeL-Dateien, Erzeugen (und auch Löschen) von ECA-Regeln, Konfiguration des Gateways sowie Aufruf des Vermittler-Generators. Geschrieben wurde das Programm mit Tcl/Tk [Ous94]. Abbildung 8.17 zeigt das Hauptfenster des GISpeL-GUIDe nach dem Laden der Beispiel-Datei `gispel3`.

Das File-Menü umfaßt die üblichen Funktionen `New`, `Load`, `Save`, `Save as...`, `Quit`. Um eine Datei zu laden, die bereits geparst wurde (d.h. aus der bereits Regeln erzeugt wurden), ist der Schalter "`Current file:`" zu aktivieren, so daß eine Dateiauswahlliste erscheint.

Parsen von GISpeL-Dateien

Um eine GISpeL-Datei parsen zu können, muß zunächst ein Persistence-Objektschema ausgewählt werden. Ein aktuelles Persistence-Objektschema ist die Grundlage dafür, daß der Schalter `Parse` aktiviert werden kann. Beim Parsen werden nach dem Übersetzen der GISpeL-Anweisungen auch die entsprechenden Stubs-Dateien mit dem ECA-Regelcode erzeugt (Aufruf der Regelableitungskomponente, siehe Abschnitt 8.6.3). Nachdem die Datei erfolgreich geparst wurde, muß MERU mit dem produzierten Quellcode neu übersetzt werden, so daß auch die Regelbibliothek aktualisiert werden kann. Dazu muß der Schalter `Compile` aktiviert werden. Da es möglich ist, ECA-Regelcode zu generieren, ohne diesen sofort zu übersetzen, bestehen manuelle Eingriffsmöglichkeiten, z.B. um noch benutzerdefinierte Erweiterungen in Bedingungs- oder Aktionsteil einer Regel zu integrieren. Die so veränderte Regel kann später mittels `Compile` übersetzt werden. Durch Betätigen des Schalters `Delete Rules` können alle Regeln aus einer GISpeL-Datei nach deren Parsen wieder aus dem Code von MERU entfernt werden.

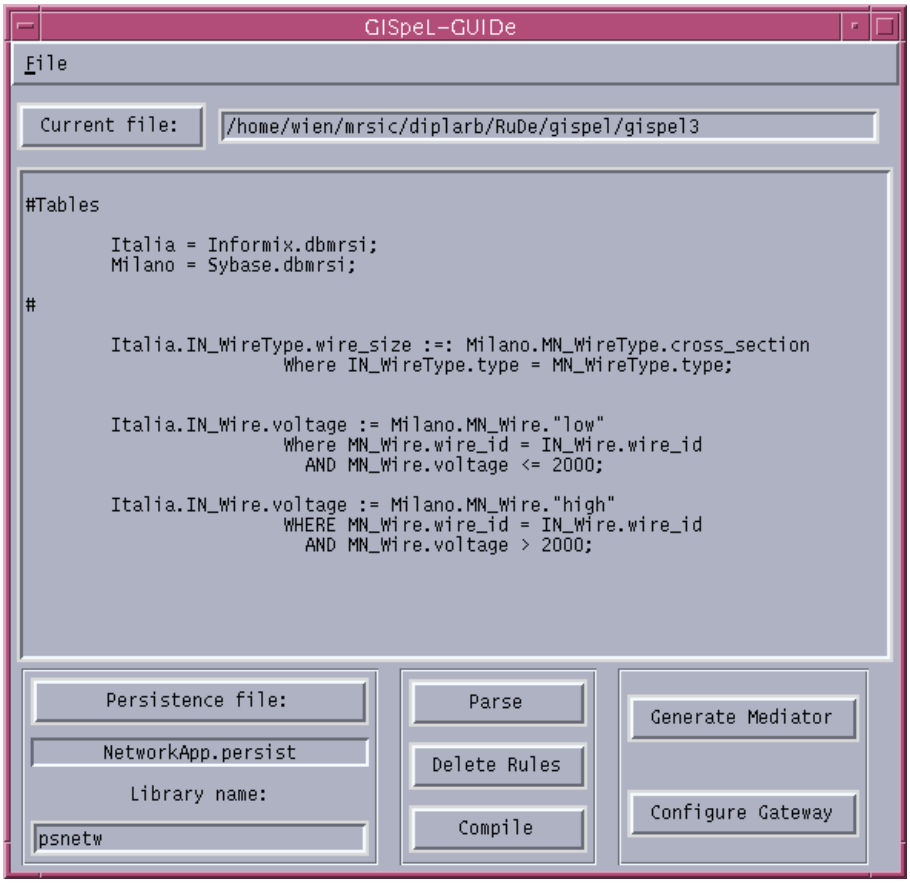


Abbildung 8.17: Hauptfenster des GISpeL-GUIDe nach dem Laden einer GISpeL-Datei

Erzeugen des Vermittlers

Um einen Vermittler zu erzeugen, wird das Programm `MediatorGenerator` aufgerufen mit den benötigten Parametern (vgl. Beschreibung auf Seite 150). Nach Aktivierung des Schalters `Generate Mediator` können in einem Dialogfenster diese Parameter eingegeben werden. Die dafür vorhandenen Eingabefelder sind bereits mit Standardwerten belegt (die überschrieben werden können):

- kompletter Pfadname des aktuellen Persistence-Objektschemas
- Name der Connection Mapping-Datei, die von Regelableitungskomponente generiert wird
- RPC-Programmnummer
- Verzeichnis, das den Code des generierten Vermittlers enthält: `./GeneratedMediator`

Konfiguration des Gateways

Bei jedem Parserdurchlauf wird durch die Regelableitungskomponente auch automatisch eine Konfigurationsdatei für das Gateway generiert, bestehend aus einer Liste der zu überwachenden Tabellen. Die so erzeugte Konfigurationsdatei muß dann noch in das Verzeichnis des Gateways kopiert oder mit einer dort bereits vorhandenen Datei vereinigt werden. Zu diesem Zweck muß der Schalter `Configure Gateway` aktiviert werden. Daraufhin können in einem Dialogfenster Quell- und Zieldatei von `gateway.config` angegeben werden.

8.6.3 Automatische Ableitung von Regeln (RuDe)

Die Regelableitungskomponente RuDe (**Ru**le **De**rivator) generiert aus einer Quelldatei, d.h. einer syntaktisch und semantisch korrekten GISpeL-Datei, den Code der ECA-Regeln sowie die erforderlichen Objekte in der Datenbank von MERU. Abbildung 8.18 veranschaulicht den Vorgang der Regelableitung mit allen daran beteiligten Programmkomponenten sowie die Datenflüsse zwischen ihnen.

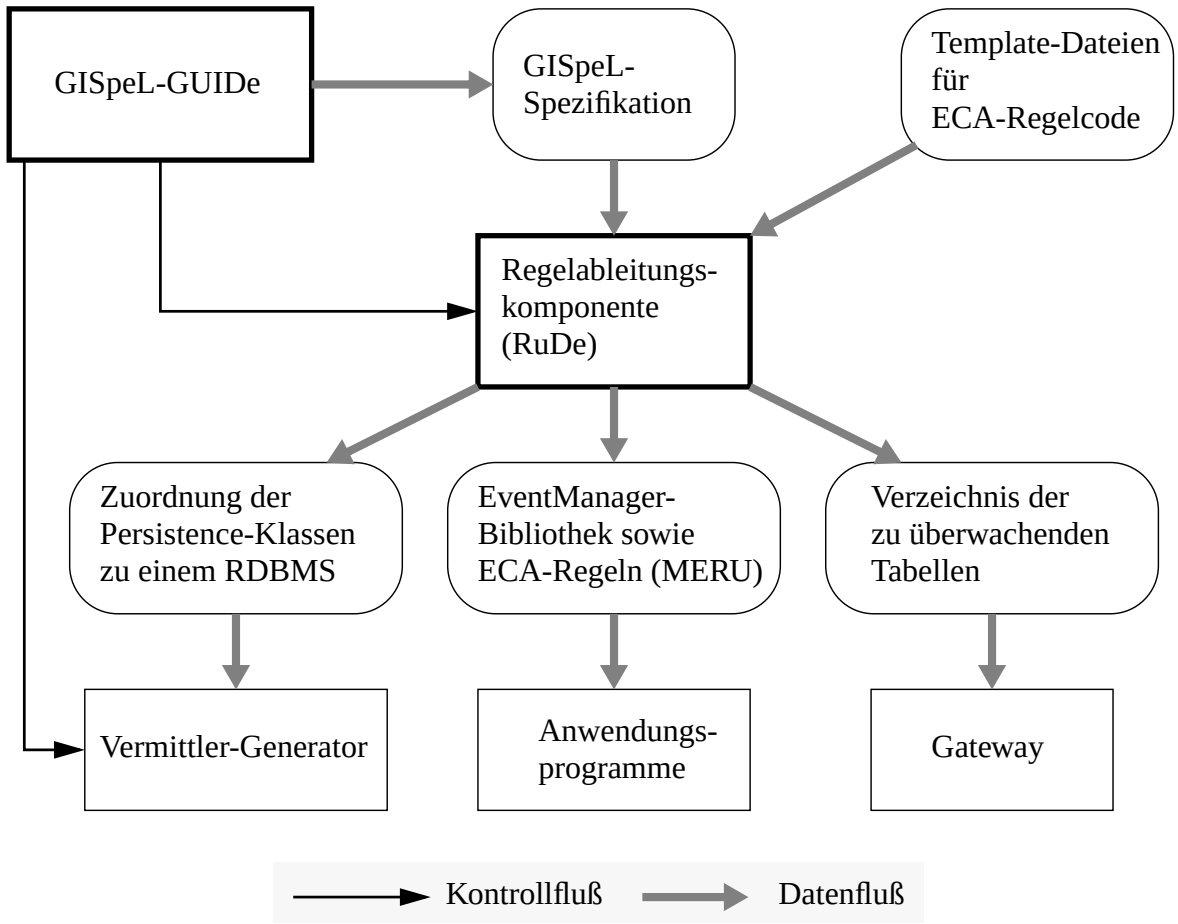


Abbildung 8.18: Regelableitungsprozeß

Die Ableitung der ECA-Regeln aus einer GISpeL-Datei geschieht in vier Phasen:

1. Parsen der GISpeL-Datei und Erstellen der Stubs-Dateien mit dem ECA-Regelcode für MERU.
2. Generieren des Standardcodes von Persistence-Klassen für die neuen Regeln.
3. Übersetzen des gesamten vorher generierten C^{++}-Codes.
4. Aktualisieren der Datenbank von MERU durch Erzeugen der benötigten Tabellen und Tupel.

Die vier Phasen bauen aufeinander auf, können aber auch getrennt voneinander aufgerufen werden (z.B. nur die 1. Phase mit einer möglichen manuellen Nachbereinigung). Dies könnte sinnvoll sein bei komplizierten Abhängigkeiten, wenn sie nicht von GISpeL unterstützt werden.

1. Parsen der GISpeL-Datei und Generierung der Stubs-Dateien

Nach dem erfolgreichen Parsen einer GISpeL-Anweisung wird die Generierung der Stubs-Dateien gestartet. Zunächst erfolgt eine semantische Prüfung der GISpeL-Anweisung (z.B. Existenz der angegebenen Namen im Objektschema). Je nach Anweisung werden eine unterschiedliche Anzahl von ECA-Regeln für den EventManager erzeugt. Jede so erzeugte ECA-Regel hat das Präfix `RuDe`. Wie die Eindeutigkeit der Regelnamen gesichert wird, ist ausführlich in [Mrs96] erläutert.

Beispiel 8.9: (Generierung von Regelnamen)
Zugrunde liegt folgende GISpeL-Anweisung:

```
Sybase.dbmn.MN_Plant.id :-  Informix.dbin.IN_Node.id
          WHERE region = 'Milano' AND function = 'p'
```

Daraus werden folgende Regelnamen erzeugt:

`RuDe_A_0_dl`	Löschen eines Objekts aus `MN_Plant`
`RuDe_A_0_irl`	Einfügen eines Objekts in `IN_Node`
`RuDe_A_0_uw1`	Ändern eines Attributwertes von `region` in `IN_Node`
`RuDe_A_0_uw2`	Ändern eines Attributwertes von `function` in `IN_Node`

Hierbei gelten folgende Namenskonventionen:
`d` = delete, `i` = insert, `u` = update, `l` = left, `r` = right, `w` = predicate (d.h. Attribut von *P*)

Die Methoden für den Bedingungs- und Aktionsteil der ECA-Regeln werden aus vordefinierten Templates generiert. Diese Templates enthalten den C^{++}-Code der jeweiligen Prozedur und darüber hinaus Platzhalter, die durch Code ersetzt werden, der erst zum Zeitpunkt des Parsens ermittelt werden kann. Dies betrifft vor allem die Namen der beteiligten Klasse und Attribute sowie die Unterstützung verschiedener Strategien bei der Ausführung des Aktionsteils. Alle Platzhalter, die in den Template-Dateien ersetzt werden sollen, sind durch zwei Tilden eingeschlossen, wobei der Text zwischen den Tilden den Platzhalter eindeutig kennzeichnet. Alle Platzhalter, die Werte repräsentieren, also z.B. `~at~` ("Ermittle Wert des aktuellen Attributs in Abhängigkeit vom Typ"), werden durch Code ersetzt, der zur Laufzeit den entsprechenden Wert ermittelt. Alle anderen Platzhalter, z.B. `~rn~` ("Regelname = Name der C^{++}-Regelklasse"), werden durch Zeichenketten ersetzt. Alle anderen Zeichenfolgen werden zeichenweise in die Zieldatei kopiert. Zusätzlich zu den oben vorgestellten Platzhaltern, die durch Code ersetzt werden, gibt es auch spezielle Zeichenfolgen in Form von C^{++}-Kommentaren, die bestimmte Codeteile ausblenden, z.B. bei sich gegenseitig ausschließenden Konsistenzwahrungsstrategien. Tabelle 8.11 gibt einen Überblick über alle Platzhalter.

Direktive	Bedeutung	Ersetzt durch ...
`~rn~`	rule name	Name der C^{++}-Klasse für die Regel
`~cn~`	class name	Name der Klasse, in der das Ereignis auftrat
`~ccn~`	corresponding class name	Name der korrespondierenden Klasse für aufgetretenes Ereignis
`~An~`	Attribute name	Name des Attributs in Abhängigkeit, großgeschrieben (für Methodenaufrufe)
`~cAn~`	corresponding Attribute name	dito, für korrespondierendes Attribut
`~at~`	attribute type	Code, der zur Laufzeit den Wert des aktuellen Attributs in Abhängigkeit vom Typ ermittelt

Tabelle 8.11: Direktiven in Template-Dateien für den ECA-Regelcode

Direktive	Bedeutung	Ersetzt durch ...
`~ga~`	get attributes	Code, der zur Laufzeit den Wert aller an der Abhängigkeit beteiligten Attribute einer Seite ermittelt
`~gao~`	get attributes old	dito, jedoch mit altem Wert des vom Ereignis betroffenen Attributs (falls Before-Hook)
`~gra~`	get replaced attributes	Attributwerte eines Prädikats, die die eigene Klasse betreffen (in Verbindung mit `~wcr~`)
`~cwc~`	corresponding where clause	WHERE-Klausel, um das korrespondierende DBS abzufragen
`~cwco~`	corresponding where clause old	dito, jedoch mit altem Wert des vom Ereignis betroffenen Attributs (falls Before-Hook)
`~wcr~`	where clause, replaced	WHERE-Klausel, die durch aktuelle Attributwerte der eigenen Klasse ersetzt ist (nur bei ungerichteten Wertabhängigk.)
`~pe~`	predicate existence dependency	Code, um im Bedingungsteil zu prüfen, ob Objekt, auf dem Ereignis stattfand, ein evtl. angegebenes Prädikat erfüllt
`~pv~`	predicate value dependency	dito, jedoch für Wertabhängigkeiten
`~ca~`	compare attributes	Code zum Vergleich von Attributen mit Konstanten, typabhängig
`~sa~`	set attributes	Code zum Setzen neuer Werte
`~sca~`	set corresponding attribute	Code zum Setzen der korrespondierenden Attribute

Tabelle 8.11: Direktiven in Template-Dateien für den ECA-Regelcode (Forts.)

2. Generierung des Persistence-Codes der neuen Regeln

Um eine komplette Neugenerierung des Persistence-Codes aus dem Objektschema (durch Aufruf des Relational Interface Generator) bei Hinzufügen neuer Regeln zu vermeiden, wurde eine andere Lösung gewählt: Mit Ausnahme der Stubs-Dateien (in denen das Verhalten repräsentiert ist) stimmen alle Regeln in ihrer Struktur überein, so daß auch der daraus generierte Code in Persistence identisch ist. Deshalb wurde dieser Code einmalig generiert und die erzeugten Dateien in Templates umgewandelt. Somit werden beim Übersetzen des Codes nur noch die neu generierten Regelklassen berücksichtigt, während bereits vorhandene Regeln unverändert bleiben.

3. Anpassung und Übersetzung des EventManagers in MERU

Vor der Übersetzung des EventManagers ist eine Anpassung seines Codes erforderlich. Dies umfaßt die Herstellung der nötigen Datenbank-Verbindungen als auch die Namen der Regelklassen, die bereits zur Übersetzungszeit bekannt sein müssen. Die erforderliche Übersetzung der Regelklassen ist im Vergleich zu den anderen Schritten relativ zeitaufwendig, da durch Persistence aus jeder Klasse fünf weitere Klassen erzeugt werden (siehe Seite 141). Deshalb sollte das Parsen von GISpeL-Anweisungen bei mehreren Dateien im ersten Schritt zusammengefaßt werden. Ist die Übersetzung des neu generierten Codes abgeschlossen, kann die erzeugte Bibliothek in Anwendungen eingebunden werden und diese somit unter Beachtung der spezifizierten GISpeL-Integritätsregeln ablaufen. Dazu müssen jedoch die notwendigen Hooks im Persistence-Objektschema vorhanden sein, außerdem müssen die Klassen durch den erweiterten Relational Interface Generator (Option `-raise`, siehe Seite 157) erzeugt werden.

4. Aktualisierung der Regel-Datenbank

Es wird eine Applikation generiert, übersetzt und gestartet, die alle Klassen und Objekte (d.h. Tabellen und Tupel) in der Regel-Datenbank erzeugt. Für eine ausführliche Beschreibung der Bedienung des Programms `RuDe` sei auf [Mrs96] verwiesen.

Sonstige Aspekte

Probleme beim Propagieren von Änderungen können dann auftreten, wenn zwei in einer Abhängigkeit stehende Objekte in derselben Transaktion geändert wurden oder eines sogar gelöscht wurde, da die Regeln erst vor dem Commit der Transaktion ausgeführt werden. Wenn Nichtschlüsselattribute zur Bestimmung der Korrespondenz dienen, kann u.U. die Korrespondenz nicht mehr rekonstruiert werden. Die Konsequenz ist, daß bei einem fehlgeschlagenen Versuch, ein korrespondierendes Tupel zu ändern, die Transaktion zurückgesetzt wird. Die Abbildung der zeitbezogenen Konstrukte von GISpeL auf ECA-Regeln mit Time-Events wurde im Rahmen dieser Arbeit nicht implementiert, da hierfür noch eine Änderung der Zeitkomponente von MERU erforderlich ist, um anwendungsunabhängig temporale ECA-Regeln auszulösen.
Bei der Definition von Abhängigkeiten in GISpeL ist die Entwicklung weiterer Werkzeuge denkbar, die eine Analyse der Korrektheit der Spezifikation ermöglichen (Konfliktfreiheit, Terminierung). Hierzu sei auch auf [Kar95] verwiesen.

8.7 Der DDL-Vermittler

Der DDL-Vermittler, der als eigenständiger Serverprozeß läuft, hat die Aufgabe, Inkonsistenzen zwischen lokalen Schemata und dem globalen Persistence-Objektschema zu behandeln. Dabei beschränken sich die Möglichkeiten auf eine Neuerzeugung des vorhandenen Objektschemas sowie eine Signalisierung der eingetretenen Veränderungen. Die bei einer Schemaevolution zu lösenden Probleme bestehen in der Anpassung der Datenbank-Instanzen sowie der Anpassung der Applikationen. Eine ausführliche Darstellung hierzu gibt die Arbeit von Schiefer [Sch93], eine dem DDL-Vermittler verwandte Arbeit ist [BIPG92].

Ein hier nicht untersuchter Aspekt der Verarbeitung von DDL-Ereignissen betrifft die Änderung lokaler Constraints (Domänenbedingungen, UNIQUE, referentielle Integrität) in einer RDB und die Behandlung dabei auftretender möglicher Konflikte mit globalen ECA-Regeln.

Die Detektion und Signalisierung von Schemamodifikationen ist eine Funktion des im vorigen Kapitel beschriebenen Gateways und wurde am Beispiel des DBMS Sybase realisiert. Allerdings sind die Möglichkeiten der Schemamodifikation in der aktuell verfügbaren Version 11 von Sybase sehr eingeschränkt (im Vergleich zu anderen RDBMS wie Oracle und Informix), so daß nur Änderungen vorgenommen werden können, die keine Auswirkungen auf existierende Tupel haben.

Betrachtet werden hierbei jedoch die globalen Konsequenzen für Objektschema und Applikation, und dementsprechend werden bei DDL-Ereignissen drei Fälle unterschieden: Ohne Auswirkungen bleibt das Anlegen von Tabellen und Sichten oder die Änderung von Default-Werten. Eine Änderung des Persistence-Schemas wird erforderlich bei Umbenennung von Spalten, Tabellen oder Sichten. Kritisch sind das Löschen von Tabellen, Sichten oder Schlüsseln, die eine manuelle Anpassung der Persistence-Applikationen nach sich ziehen.

Die Architektur des DDL-Vermittlers ist in Abbildung 8.19 dargestellt und wird nachfolgend komponentenweise erläutert: Der Operationsmanager sammelt die empfangenen DDL-Operationen unter Berücksichtigung der lokalen Transaktionssemantik. Der Schemagenerator erzeugt mit Hilfe der Änderungsoperationen und aus dem vorhandenen Schema ein neues Persistence-Schema und protokolliert die Modifikationsinformationen. Der Modifikationsmelder ist verantwortlich für die Signalisierung der Änderungen an den Schema-Administrator. Eine ausführliche Darstellung findet sich in [Rec96].

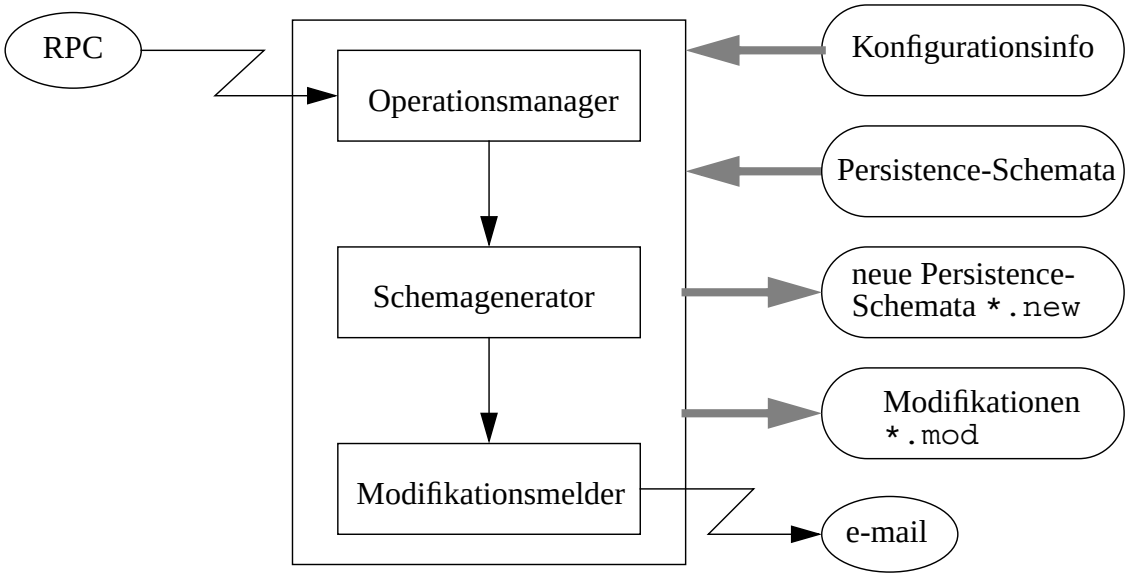


Abbildung 8.19: Architektur des DDL-Vermittlers

Konfiguration

Die Konfigurationsdatei (`persist.cfg`) beinhaltet folgende Informationen:

- Namen und Zugriffspfade der Persistence-Schemata
- verantwortliche Administratoren der Persistence-Schemata
- Namen der lokalen Datenbanken und Tabellen, die durch die Schemata beschrieben werden

Sie besteht aus einer Liste von Schemainformationen, die eine Aufzählung von Datenbanken mit zugehörigen Tabellen beinhalten. Die E-Mail-Adresse eines Administrators kann optional angegeben werden. Durch die Zuordnung von Datenbank und Tabelle zu einem Persistence-Schema wird berücksichtigt, daß eine Klasse in Persistence laufzeitdynamisch verschiedenen Datenbanken zugeordnet werden kann, somit nicht jede lokale Schemaänderung Einfluß auf die globale Applikation haben muß.

Operationsmanager

Der Operationsmanager empfängt die DDL-Operationen über die RPC-Schnittstelle und sammelt diese entsprechend der lokalen Transaktionsbefehle, die ihm ebenfalls signalisiert werden. Die eventauslösende Transaktion wird über `thread_id` und `tran_level` identifiziert, die als Parameter mit übergeben werden (vgl. auch Seite 127).

Der Ablauf bei Ankunft einer Operation ist wie folgt: Es wird die Transaktionstiefe und der Operationscode geprüft. Ist die Transaktionstiefe = 0 (d.h. Operation außerhalb einer Transaktion) wird die Operation propagiert bzw. beim Commit einer Top-Level-Transaktion die zu

dem Thread gehörenden Operationen zur Weiterverarbeitung freigegeben. Bei einem Rollback werden alle zu diesem Thread gehörenden Operationen aus der Liste gelöscht ggf. unter Berücksichtigung von Sicherungspunkten.

Schemaanpassung

Der Schemagenerator unterstützt die Anpassung des Persistence-Schemas an das geänderte lokale Schema. Vom Operationsmanager wird eine Liste von Operationen bereitgestellt, die die lokalen Schemamodifikationen beschreiben und nacheinander verarbeitet werden. Die Schemaanpassung verläuft in mehreren Schritten:

- Einlesen der Konfigurationsdatei `persist.cfg`,
- Auswertung der Operationen aus der Operationsliste und Bestimmung der zu modifizierenden Persistence-Schemata,
- Einlesen jeweils eines alten Persistence-Schemas in Variablen unter Verwendung des Schema-Analysators (siehe Abschnitt 8.3.2.2 auf Seite 153),
- Anpassung der Variableninhalte, die Inkonsistenzen zum lokalen Schema aufweisen,
- Erzeugung eines neuen Persistence-Schemas mit dem Suffix `.new`,
- Erzeugung eines Text-Files (Suffix `*.mod`) zur Beschreibung der Modifikationen im Objektschema mit: Angaben über Datum, Uhrzeit, Persistence-Schema, Name der von der Änderung betroffenen Klasse, Änderungstyp (Operationscode) und die Änderung selbst.

Der Schemagenerator-Modul erzeugt somit ein Schema, das ohne Einschränkungen vom Relational Interface Generator von Persistence weiterverarbeitet werden kann (weitere Details in [Rec96]).

Modifikationssignalisierung

Der Modifikationsmelder ist verantwortlich für die Signalisierung der Modifikationen der globalen Objektschemata an den dafür zuständigen Administrator, der im Konfigurationsfile vermerkt ist. Die Benachrichtigung erfolgt über Electronic Mail und besteht aus zwei Dateien: <*Persist-Schema*>`.new` und <*Persist-Schema*>`.mod`. Damit ist auch gesichert, daß eine sofortige Reaktion erfolgen kann. Der für das Persistence-Schema verantwortliche Administrator muß daraufhin entsprechende Schritte zur Wiederherstellung der globalen Schemakonsistenz einleiten, wie z.B. Aktualisierung und Neuübersetzung der Persistence-Applikationen mit dem veränderten Schema. Gegebenenfalls sind auch weitere Abstimmungen mit lokalen Datenbankadministratoren notwendig.

Kapitel 9

Verwandte Arbeiten

Ein Hauptproblem der semantischen Interoperabilität ist die Sicherung der globalen Integrität in heterogenen Datenbanken. Hierfür wurde bereits ein Vielzahl von Ansätzen, Architekturen und Prototypen veröffentlicht, von denen die wichtigsten nachfolgend skizziert werden sollen. Verschiedentlich wurden diese Arbeiten bei der Diskussion einzelner Teilaspekte auch schon referenziert.

Eine Reihe von Arbeiten beschäftigt sich mit dem Management interdependenter Daten, d.h. Daten, die in lose gekoppelten Systemen durch globale Constraints miteinander verbunden sind (Abschnitt 9.1). Es werden eine Reihe verschiedenartiger Ansätze deutlich, wie solche Constraints überwacht werden können. Zur Kontrolle der Regelausführung sind neuartige Transaktionsmodelle erforderlich, wie in [SRK92, ABD+93, DD95] beschrieben. Allen Ansätzen gemeinsam ist das Problem der Erkennung von Konsistenzverletzungen

In Abschnitt 9.2 werden einige Protokolle vorgestellt, deren Grundidee darauf beruht, globale Integritätsbedingungen in lokale Constraints zu transformieren, so daß die Konsistenzprüfung lokal erfolgen kann. Der Vorteil dieser Ansätze besteht darin, daß sie auch bei eingeschränkter Verfügbarkeit der Datenbankföderation funktionieren und eine höhere Effizienz aufweisen.

Eine Reihe von Arbeiten verfolgt die Idee, globale Integritätsbedingungen durch ECA-Regeln auszudrücken und diese in einem Multidatenbanksystem zu integrieren (Abschnitt 9.3). Dabei werden Applikationen unterstützt wie geographische Informationssysteme [PRV95], Fabrikplanung [SCH+96], Logistik und Workflow Management [SSMR96] oder kooperatives CAD [USR+94].

Die Anwendung verteilter Objektmanagementtechnologie zur Integration heterogener Systeme erfordert die Möglichkeit, systemübergreifend Constraints durch Anreicherung mit ECA-Regeln auszudrücken. Einige Arbeiten auf diesem Gebiet werden in Abschnitt 9.4 diskutiert, die auch Verwandtschaft zu den im vorigen Abschnitt genannten Arbeiten aufweisen.

ECA-Regeln sind ein Mittel der Wissensrepräsentation und somit auch geeignet für die explizite Darstellung der Semantik heterogener Datenquellen. Eine Reihe von Arbeiten unter der Überschrift *Mediator* widmet sich der Unterstützung semantischer Interoperabilität, wobei jedoch auch andere Darstellungsformen für größere Datenmengen (z.B. Ontologien) vorgeschlagen werden (Abschnitt 9.5). Ebenso läßt sich auch die Abschwächung von Konsistenz deklarativ beschreiben [SK93a].

Der letzte Abschnitt in diesem Kapitel geht auf Ansätze der Verteilten Künstlichen Intelligenz ein, die Konzepte liefern, die auf die Kontrolle von Interdatenbankabhängigkeiten angewandt werden können. Ein Schwerpunkt der Arbeiten hier liegt in der intelligenten Erkennung solcher Abhängigkeiten [Klu97].

9.1 Kontrolle interdependenter Daten in heterogenen Systemen

Aeolos

Karabatis führt den Begriff *Interdependentes Datenbanksystem* (IDBS) ein, das durch seine Hauptaufgabe, die Wahrung globaler Konsistenzbedingungen, definiert ist, womit es sich gegenüber Multidatenbanksystemen unterscheidet, die hauptsächlich globale Queries unterstützen. In [Kar95] wird ein Prototyp eines solchen IDBS unter dem Namen *Aeolos* präsentiert:
Die im IDBS beteiligten lokalen Datenbanksysteme sind um eine zusätzliche Ebene, das *Dependency Subsystem*, erweitert. Jedes Dependency Subsystem ist verantwortlich für die Konsistenz der interdependenten Objekte, die sich in der darunterliegenden lokalen Datenbank befinden. Die Abhängigkeiten sind als *Data Dependency Descriptors* (D^3) in einem *Interdatabase Dependency Schema* (IDS) zusammengefaßt, das über die lokalen Datenbanken verteilt ist (siehe auch Seite 38). Ein Dependency Subsystem muß auftretende Events beobachten, den Grad an Inkonsistenz für jeden D^3 im Schema berechnen und die Konsistenz der Zielobjekte innerhalb bestimmter Grenzen einhalten. Jedes Dependency Subsystem besteht aus fünf Hauptkomponenten: Der IDS Manager verwaltet die D^3s im IDS. Der Monitor ist verantwortlich für die Detektion der Events, die auf interdependenten Daten auftreten. Dabei verwendet der Monitor jedes Subsystems die Eventdetektionsfähigkeiten des darunterliegenden LDBS. Die Heterogenität der Eventdetektion bei den verschiedenen LDBS wird durch ein einheitliches Interface zwischen Monitor und zugrundeliegendem LDBS ausgeglichen, unabhängig davon, ob lokal Trigger unterstützt werden oder nicht. Der Consistency Manager berechnet den Grad an Inkonsistenz aller interdependenten Objekte (siehe hierzu auch [KRS93]) und startet Restaurationsprozeduren als Transaktionen auf den LDBS, falls Konsistenzbedingungen verletzt sind. Der Execution Agent ist verantwortlich für die Ausführung dieser Restaurationsprozeduren. Der Recovery Manager behandelt Transaktions- und Knotenfehler. Als Ausführungsmodell wird das Konzept der *Polytransaction* in [SRK92] vorgestellt; dabei handelt es sich um eine Gruppe von Transaktionen, die durch ein Update auf einem Datenobjekt erzeugt werden. In [Kar95] wird der IDBS-Ansatz aktiven Datenbanken gegenübergestellt, die aber aufgrund ihrer zentralisierten Architektur eher als ungeeignet für die Kontrolle interdependenter Daten beurteilt werden.

Quasi-Transaktionen

Arizio u.a. behandeln das Konsistenzproblem bei interdependenten Daten im Kontext von Softwaresystemen für Telekommunikationsnetze [ABD+93]. Die Autoren schlagen eine Spezifikation interdependenter Daten mit einem Ausführungsmodell vor, das auf ECA-Regeln und sogenannten *Quasi-Transaktionen* (QT), geschachtelten Transaktionen mit abgeschwächten ACID-Eigenschaften, beruht. ECA-Regeln sind erweitert um die explizite Angabe der Daten, die bei Regelausführung gelesen bzw. geschrieben werden. Der Kontrollfluß einer Quasi-Transaktion ist durch ein Script definiert, Fehler werden durch Exception Handler behandelt, die in die QTs einbezogen sind. Das Regelmodell ist angelehnt an [RSK91], erweitert aller-

dings die Restaurationsprozeduren, die nicht auf eine einzige Zieldatenbank beschränkt sind, sondern entsprechend dem Quasi-Transaktionsmodell beschrieben werden. Die Systemarchitektur kann folgendermaßen skizziert werden: Jedes Komponentendatenbanksystem umfaßt einen Regelmanager, der auf einem kommerziellen DBMS basiert. Der Regelmanager umfaßt einen Eventdetektor für temporale und Datenbank-Events sowie den Controller zur Steuerung der Regelausführung. Die globale Regelbasis ist entsprechend der Datenverteilung auf die Komponentensysteme aufgeteilt. Der Quasi-Transaktionsmanager ist eine Softwareschicht oberhalb eines kommerziellen Transaktionsmanagers, die die Quasi-Transaktionen entsprechend ihrer besonderen Semantik steuert. Für die Realisierung des Prototypen wird ENCINA^{TM} Transarc eingesetzt, der Quasi-Transaktionsmanager ist mit den ENCINA TRAN Services implementiert, so daß verteilte Transaktionen über heterogene Datenbanken ausgeführt werden können. Um die Ausführungsautonomie der Komponentensysteme zu bewahren, wird allerdings nur bei einer immediate oder deferred Kopplung der Aktion eine verteilte globale Transaktion gestartet.

Lokale Asynchrone Update-Transaktionen

Do und Drew behandeln ebenfalls das Problem der Wahrung globaler Constraints in heterogenen Datenbankumgebungen [DD95]. Dabei wird ein Kompromiß zwischen der flexiblen autonomen Ausführung existierender lokaler Applikationen und der konsistenten Wahrung von globalen Datenabhängigkeiten durch ein globales Transaktionsmanagement-System angestrebt. In ihrem MDBS-Modell werden drei Arten von Transaktionen unterschieden: Lokale Transaktionen (LTs) greifen nur auf Daten zu, die keine Beziehung zu anderen Datenbanken aufweisen. Globale Transaktionen (GTs) werden über einen globalen Transaktionsmanager ausgeführt und operieren auf Daten verschiedener Knoten. Lokale asychrone Update-Transaktionen (LAUs) ändern lokal Daten, die in irgendeiner Weise mit anderen Objekten in anderen Datenbanken in Beziehung stehen (z.B. als Replikat). Eine LAU triggert die Ausführung weiterer LAUs auf den korrespondierenden Objekten anderer Datenbanken. Somit können diese LAUs konzeptionell zu einer globalen LAU (GLAU) gruppiert werden, die für die Ausführung äquivalenter Updates auf (konzeptionellen) Replikaten verantwortlich ist. Zu jeder lokalen Datenbank gehört ein Wrapper-Prozeß, der Informationen über die LAUs und ihre zugeordneten Replikat-Objekte verwaltet und dadurch die Ausführung der LAUs kontrolliert. Es werden zwei Arten globaler Konsistenz unterschieden: Äquivalenz zwischen konzeptionell identischen Daten (*CRep Equivalence*) sowie globale Abhängigkeitsbeziehungen (*Causal Dependency*). Für die Wahrung dieser Konsistenz wird der sogenannte *Dynamic Primary Copy* (DPC) Algorithmus vorgeschlagen, dessen Details [DD95] entnommen werden können. Die Autoren skizzieren eine Implementierung des Wrapper-Prozesses, die auf aktiver Datenbanktechnologie beruht, und zeigen, wie globale Constraints durch ECA-Regeln ausgedrückt werden können.

PARDES

Etzion beschreibt in [Etz93] ein aktives semantisches Datenmodell, das die automatische Durchsetzung von Abhängigkeiten zwischen fragmentierten verteilten (möglicherweise heterogenen) Datenbanken beinhaltet. Das konzeptuelle Schema beruht auf dem semantischen aktiven Datenmodell PARDES und ist allgemeingültig für beliebige logische Datenbankmodelle anwendbar [Etz90]. In diesem Modell werden Abhängigkeiten durch Invarianten ausgedrückt, die unabhängig von physischen oder Verteilungsaspekten definiert werden. Die Abhängigkeitsdefinitionen werden dabei in PATH-Strukturen übersetzt. Dabei handelt es sich um implementationsunabhängige Wissensrepräsentationseinheiten, in denen syntaktisches und semantisches Wissen über die Abhängigkeiten gesammelt wird. Die Entwurfsentscheidungen betreffen

Schema und Implementationswerkzeug für jede lokale Datenbank sowie die Konsistenzmodi und Ausnahmebehandlungen für jede Abhängigkeit. Die Darstellung des konzeptuellen Modells wird zusammen mit diesen Entwurfsentscheidungen in ein verfeinertes Schema mit einem Ableitungsgraphen übersetzt. Dieses bildet die Grundlage für die Übersetzung in ein Ausführungsmodell durch den sogenannten *Distributed Matching Processor* (DMP). Dabei wird der Umfang der Interdatenbank-Transaktionen bestimmt, die die Konsistenz des verteilten Systems relativ zur definierten Sequenz der Invarianten wahren [Etz93].

Ein Toolkit für Constraint Management in lose gekoppelten Systemen

Chawathe, Garcia-Molina und Widom stellen in [CGW96] ein Toolkit und ein Framework für das Management von Constraints in lose gekoppelten heterogenen Informationssystemen vor. Das Framework gestattet abgeschwächte Konsistenzdefinitionen und untersucht, wie diese definiert, implementiert und genutzt werden können. Dazu werden *Interfaces* (Zugriffsmodi) definiert, die von jeder Datenbank dem Constraint Manager bereitgestellt werden. Im Framework lassen sich *Strategien* für das Constraint Management definieren, die entweder die Konsistenz wiederherstellen (*Enforcement*) oder in manchen Fällen nur deren Verletzung beobachten (*Monitoring*). Außerdem können *Garantien* für einen bestimmten Grad an Konsistenz definiert werden. Das Toolkit nutzt dieses Framework, um eine Menge von konfigurierbaren Modulen bereitzustellen, die das Management von datenbankübergreifenden Constraints ermöglichen. Die Constraint Management Architektur läßt sich folgendermaßen beschreiben: Ein verteilter Constraint Manager (CM) besteht aus einer Menge von Constraint Manager Shells (CM-Shells). Die CM-Shell interagiert mit der darunterliegenden lokalen Datenbank und kooperiert mit anderen CM-Shells bei der Integritätssicherung entsprechend der gegebenen Strategie-Spezifikation. Aus der Strategie werden Regeln abgeleitet, die von der CM-Shell ausgeführt werden. Bei der Initialisierung müssen auch die Anforderungen an die Interfaces festgelegt werden, z.B. müssen Trigger deklariert werden, um ein *Notify*-Interface (d.h. Benachrichtigung über aufgetretene Updates in einer bestimmten Zeit) zu unterstützen. Um die Heterogenität der verschiedenen Datenbanken bzw. Informationsquellen zu berücksichtigen, wird ein CM-Translator eingesetzt, der den CM-Shells die lokalen Daten durch ein CM-Interface (CMI) zur Verfügung stellt. Zur Laufzeit verarbeiten die CM-Shells die Events, die sie von den einzelnen CM-Translators erhalten. Die Events, die bei der Regelausführung produziert werden, werden entweder an den lokalen CM-Translator oder an andere CM-Shells weitergeleitet.

9.2 Protokolle für globale Integritätskontrolle

Kooperative Constraint-Protokolle

Grefen und Widom präsentieren in [GW96] kooperative Protokolle für die Integritätskontrolle in föderierten Datenbanken. Sie diskutieren dabei das Problem, daß bei Abwesenheit von globalen Transaktionen der globale Datenbankzustand unter Umständen nicht beobachtet werden kann. Da lokale DB-Zustände nicht simultan durch Applikationen gelesen werden, ist es sogar möglich, daß ein globaler Zustand beobachtet wird, der so nie existierte (Phantom-Zustand). Deswegen wird der Begriff des “ruhenden” Zustandes (*quiescent state*) eingeführt, ein Zustand, bei dem alle lokalen Update-Transaktionen beendet sind und alle getriggerten Integritätsprüfungen abgeschlossen wurden. Ein Integritäts-Protokoll ist sicher (*safe*), wenn es jede Transition in einen quiescent state, in dem ein Constraint verletzt ist, erkennt. Ein Protokoll ist genau (*accurate*), wenn es einen Alarm nur dann signalisiert, wenn ein Constraint tatsächlich verletzt ist; in vielen Fällen sind aber “pessimistische” Protokolle, die auch “falsche Alarme”

produzieren, tolerabel. Die Basis-Architektur läßt sich auf zwei Datenbanksysteme zurückführen. Jeder lokalen Datenbank ist ein Constraint-Manager zugeordnet, der die Constraints kontrolliert, die durch lokale Updates auf dieser Datenbank verletzt werden können. Der Constraint-Manager bekommt die lokalen Änderungen als Delta-Mengen mitgeteilt. Es wird davon ausgegangen, daß es keine globale Transaktionen bzw. Queries gibt. Die Protokolle werden durch vier verschiedene Dimensionen charakterisiert. Um die Genauigkeit der Protokolle zu erreichen, werden Timestamp-Mechanismen genutzt, um den Delta-Mengen bzw. Abfrageergebnissen globale Zeitstempel zuzuweisen. Ebenso läßt sich durch lokale Transaktionen in den Protokollen bestimmen, ob das Constraint auf einem Phantom-Zustand ausgewertet wurde. Sogenannte *Change Logging* Verfahren und lokale Tests ermöglichen darüber hinaus eine Erhöhung der Performance der Protokolle. Noch nicht ausreichend behandelt werden das Problem der Wiederherstellung von Constraints (*Constraint Repair*) sowie die Anwendbarkeit der Protokolle auf mehr als zwei Datenbanken.

Demarcation Protocol

Das *Demarcation Protocol*, das in [BG94] präsentiert wird, stellt gegenüber herkömmlichen Protokollen in verteilten Datenbanksystemen eine entscheidende Verbesserung dar, indem der Overhead wie starker Nachrichtenverkehr, Beschränkungen oder Sperren des Zugriffs auf Ressourcen während der Protokollausführung vermieden wird. Davon profitieren insbesondere Echtzeitsysteme sowie sehr große verteilte (ggf. heterogene) Datenbanken, deren Autonomie nicht beeinträchtigt wird. Die Idee dieses Protokolls besteht darin, explizite (globale) Konsistenzbedingungen als Korrektheitskriterium zu verwenden. Es werden Limits für lokale Änderungen festgelegt, deren Einhaltung die globale Konsistenz nicht verletzen kann (“sichere” Operationen). Die Limits müssen allerdings nicht statisch sein, sondern können sich bei Bedarf ändern. Das Demarcation Protocol selbst besteht aus zwei Operationen: eine zur Änderung des Limits sowie eine zur Bestätigung der Änderung. Zum Protokoll gehören Policies, die spezifizieren, wann eine Limitänderung beginnen soll, wie ein neues Limit berechnet wird und was zu tun ist, falls Transaktionen ein Update versuchen, das das Limit überschreitet. Die Autoren demonstrieren das Protokoll am Beispiel von Constraints, die durch lineare arithmetische Ungleichungen ausgedrückt werden, zeigen jedoch auch dessen Anwendbarkeit für andere Arten von Integritätsbedingungen (referentielle Integrität, Key Constraints, Copy Constraints).

Lokale Prüfung globaler Constraints

Gupta und Widom verfolgen ebenfalls die Idee, die Prüfung globaler Integritätsbedingungen durch lokale Verifikation zu optimieren [GW93]. Ein globales Constraint wird zunächst lokal getestet, so daß die Kosten für den Zugriff auf entfernte Daten entfallen. Die Optimierung basiert auf einem Algorithmus, der als Eingabe ein globales Constraint sowie Daten bekommt, die in die lokale Datenbank eingefügt werden sollen. Der Algorithmus produziert eine lokale Bedingung, die auf den lokalen Daten geprüft wird. Wenn diese erfüllt ist, kann bei vorheriger Einhaltung des globalen Constraints gefolgert werden, daß das globale Constraint nicht verletzt wurde. Wenn die lokale Bedingung nicht mehr erfüllt ist, dann muß eine herkömmliche globale Verifikationsprozedur gestartet werden.

9.3 Erweiterung von Multidatenbanksystemen um ECA-Regeln

Darstellung von Existenz- und Wertabhängigkeiten durch ECA-Regeln

Ceri und Widom demonstrieren die Anwendbarkeit von ECA-Regeln für die Wahrung der Konsistenz interdependenter Daten in einer relationalen Multidatenbankumgebung [CW93]. Dabei betrachten sie Existenz- und Wertabhängigkeiten zwischen semantisch verwandten Daten. Semantische Konflikte zwischen den Daten werden durch Einhaltung entsprechender Integritätsbedingungen aufgelöst. Als Voraussetzung muß jede Datenbank Produktionsregeln und persistente Queues, die zur Kommunikation dienen, unterstützen. Ceri und Widom schlagen vor, das Wissen über semantische Heterogenität durch Regeln auszudrücken. Dem Benutzer wird eine deklarative High-Level-Spezifikationssprache angeboten, um ihn von der Implementierung der Regeln zu befreien, d.h. es wird nichts darüber gesagt, wie die Konsistenzeinhaltung erzwungen wird. Stattdessen werden die Regeln automatisch aus der Spezifikation generiert. Dies wird dadurch erleichtert, daß die Spezifikationssprache auf SQL-Konstrukten beruht, womit auch die Menge der in der Regel zu generierenden Aktionen (basierend auf der DML) festgelegt ist.

RIMM

Pissinou, Raghavan und Vanapipat entwickeln ein formales Integrationsmodell für heterogene Datenbanken, in dem sie aktive und temporale Konzepte mit Multidatenbanken verbinden [PRV95, VPR95]. Dieses Framework beschreibt unter dem Namen *Reactive Integration Multidatabase Model* (RIMM) die Typen von Dynamik im Kontext von Multidatenbanken und die Transitionen, die dabei getriggert werden können. Neben interdependenten Daten wird auch die Konsistenzwahrung zwischen lokalen und globalen Objekten (*Object Relativism*) untersucht. Um die zeitliche Varianz der lokalen Datenbanken und der dazugehörigen globalen Interfaces zu erfassen (auch im Hinblick auf strukturelle Veränderungen) werden diese als temporale Objekte modelliert. Zur Umsetzung des Modells wird in [VPR95] der Aufbau eines Multidatenbanksystems mit aktiven und temporalen Eigenschaften skizziert, der auf den drei Ebenen eines Mediators [Wie92] basiert. Zwischen Datenbank- und Applikationsebene ist eine Vermittler-Schicht definiert, in der ECA-Regeln sowie eine globale Wissenbasis gehalten werden.

SIGMA$_{FDB}$

Das Projekt SIGMA$_{FDB}$ (**S***chema* **I***ntegration and* **G***lobal integrity* **M***aintenance* **A***pproach for* **F***ederated* **D***ata* **B***ases*) zielt auf die Entwicklung einer Informationsinfrastruktur als Grundlage einer einheitlichen Datenhaltung für alle Phasen der Fabrikplanung [SCH+96]. Das dafür erforderliche föderierte Informationssystem soll als Rahmensystem alle an der Fabrikplanung beteiligten Software-Werkzeuge und deren lokale Daten integrieren. Aktive Mechanismen sollen dafür angewandt werden, um die systemübergreifende Konsistenz zu gewährleisten, wozu die Einzelsysteme mit ihren unterschiedlichen Integritätssicherungsmöglichkeiten verbunden werden müssen. Erste Überlegungen, wie Regelmechanismen in föderierte Datenbanksysteme integriert werden können, sind in [CT95] beschrieben. Dort wird eine allgemeine aktive FDBMS-Architektur skizziert, bei der globale Constraints auf globale ECA-Regeln abgebildet werden. Zur Signalisierung lokaler Events an den globalen Regelmanager werden lokale Regelmanager

vorgeschlagen, wobei vorausgesetzt wird, daß auftretende Events durch die lokalen Systeme detektiert werden können.

Distributed Situation Monitoring

Das MITRE-Projekt *Distributed Situation Monitoring* (DSM) hat das Ziel, das Monitoring von verteilten Aktivitäten oder Situationen in Applikationen wie Logistik, Workflow Management und kooperatives CAD zu unterstützen [SSMR96]. Solche Applikationen müssen das Monitoring von *Standing Requests for Information* (SRI) ermöglichen, d.h. von Prädikaten, die über Sichten auf autonomen, verteilten Datenquellen auf der Ebene der Föderation definiert sind. Dabei wird die Extension des föderierten Schemas als *Situation* bezeichnet (unabhängig davon, ob sie materialisiert ist). Ein Situation Monitor läßt sich somit sehr gut für die Überwachung globaler Constraints in einem verteilten Entwurfsprozeß einsetzen. Die materialisierten Daten werden in einem Cache abgelegt, der durch ein aktives DBMS verwaltet wird. In der vorgestellten Architektur wird auch ein Metadata Repository eingeführt, um Abbildungen zwischen heterogenen Datenquellen zu erleichtern.
Eine effektive Implementierung der SRIs wird durch einen 2-Schichten-Ansatz erzielt: Dabei werden ausgewählte Updates auf den Komponentendatenbanken zu einem beobachtenden Knoten propagiert, auf dem ein aktives DBMS läuft. Die Update-Propagation basiert auf kommerziellen Replikationstechnologien - nicht auf ECA-Regeln, wobei der asynchronen Replikation Vorrang gegeben wird. Die Autoren diskutieren verschiedene Materialisierungsstrategien für die föderierte View und geben einer Lösung den Vorzug, die darin besteht, nur die Quellinformationen zu replizieren, die für die Auswertung einer Condition in einem SRI benötigt werden. Somit wird der Aufwand für die Materialisierung minimiert, unnötige Eventdetektionen und -signalisierungen werden von vornherein vermieden. SRIs werden übersetzt in ECA-Regeln auf den Replikaten der Quelldaten. Obwohl kommerzielle Komponenten wie Replikationsserver und aktive DBMS zum Einsatz kommen, sind zusätzliche Werkzeuge wie ein Monitor Generator bzw. SRI-Editor erforderlich. Dieser soll es dem Benutzer ermöglichen, für seine DSM-Applikation entsprechende SRIs im föderierten Schema zu definieren. Daraus lassen sich automatisch mit Hilfe des Repositories ECA-Regeln mit der entsprechenden Materialisierungsstrategie bzw. den benötigten Replikationsbeziehungen ableiten.

Aktives Entwurfsmanagement

Die Arbeit von Urban u.a. [USR+94] verfolgt das Ziel, Entwurfsprozesse im Maschinenbau durch interoperable Systeme zu unterstützen. Das Integrations-Framework basiert auf objektorientierten Schnittstellen (d.h. Sichten) zum Zugriff auf Entwurfsdaten von CAD-Tools und lokalen Datenbanken. Das Entwurfsmanagementsystem, *Shared Design Manager* (SDM), basiert auf einem OODBMS, in dem u.a. Metadaten der Entwurfsumgebung, Beziehungen zwischen den Komponenten, Abbildungsfunktionen usw. abgelegt sind. Für die Verifikation globaler Design Constraints (GDC) bzw. das Change Management in den Entwurfsumgebungen, die zugleich Multidatenbank-Umgebungen sind, werden aktive Regeln verwendet. Design Constraints werden deklarativ in einer Regelsprache namens ARL ausgedrückt. Konsistenzverletzungen können unterschiedlich behandelt werden: durch Zurückweisung, Propagierung der Änderung oder Signalisierung eines Alarms. Regeln in ARL werden definiert als Alarmgeber (*Alerter*), Trigger oder Integritätsregeln (*Integrity Maintenance Rules*, IMR). IMRs werden automatisch von Design Constraints abgeleitet. Die Autoren unterscheiden in Entwurfsumgebungen zwischen *soft* Constraints, die temporär verletzt sein können, und *hard* Constraints, die jederzeit erfüllt sein müssen.

Weitere Arbeiten

Huang und Liu diskutieren in [HL96] ebenfalls die Anwendung von ECA-Regeln für globale Konsistenzbedingungen in heterogenen Datenbanksystemen und betrachten besonders die Transaktionssemantik von globalen Regeln in verteilten Umgebungen. Dabei schlagen sie einen Algorithmus für die Regelaktivierung vor.
Blanco u.a. entwerfen ein aktives föderiertes System, das automatisch auf Veränderungen in lokalen Schemata reagiert, soweit diese das föderierte Schema betreffen [BIPG92]. Bei der Integration einer Komponente kann man entsprechend den Benutzeranforderungen wählen zwischen einer Standardkontrolle oder die Kontrolle durch deklarative Definition der Systemreaktionen anpassen. Beide Fälle basieren auf dem Gebrauch von ECA-Regeln, die vom System unterstützt werden. Diese aktive Komponente ist Bestandteil eines föderierten Systems mit der Aufgabe, die Schemakonsistenz zu wahren.

9.4 Verteilte Aktive Objekte

DOM

Das *Distributed Object Management* (DOM) Projekt untersucht die Anwendbarkeit verteilter Objektmanagement-Technologie für die Integration heterogener, autonomer und verteilter Systeme [MHG+92]. Die generische Architektur eines DOM-Systems weist große Verwandtschaft mit der Entwicklung von CORBA auf [OMG91]. Ein Distributed Object Manager vermittelt zwischen Clients und Ressourcen, die durch Interfaces bzw. Implementationen gekennzeichnet sind. Das gemeinsame Objektmodell in DOM ist FROOM, ein funktional objektorientiertes Modell, das auch ECA-Regeln unterstützt. Regeln und deren Bestandteile sind als Objekttypen in FROOM definiert. In [BÖH+92] wird gezeigt, wie erweiterte Transaktionsmodelle durch die Transaktionsstruktur, die Objekttypen und die Korrektheitskriterien charakterisiert werden können. Das DOM-Transaktionsmodell gestattet die Definition komplexer Transaktionen durch Spezifikation einer Menge von Abhängigkeiten zwischen flachen Transaktionen. Transaktionen werden als dynamisch erzeugte Objekte modelliert, die die grundlegenden Transaktionsoperationen unterstützen. Erweiterte systemübergreifende Transaktionen entstehen durch Komposition aus einfacheren Transaktionsobjekten, spezifiziert durch Dependency Deskriptoren (DD). Dabei werden Abhängigkeiten zwischen Transaktionsereignissen in Form von ECA-Regeln definiert. Außer für die Konstruktion erweiterter und Multidatenbank-Transaktionen können die DDs auch für die Spezifikation verschiedener Arten von Konsistenz zwischen interdependenten Daten [RSK91] angewandt werden.

NCL

In [SLY+96] wird die Informationsmodellierungssprache NCL vorgestellt, die als eine Synthese der Eigenschaften der Interface Definition Language (IDL) von CORBA [OMG91], der STEP-Beschreibungssprache EXPRESS [ISO92] sowie der Regelspezifikation in OSAM* [SDS95] angesehen werden kann. Eingebettet ist diese Arbeit in das NIIIP-Projekt, das das Ziel verfolgt, eine offene standard-basierte Informationsinfrastruktur für die Integration heterogener und verteilter Prozesse, Daten und Computersysteme in einem virtuellen Unternehmen zu entwickeln [SLA+95]. In einer NIIIP-Umgebung werden alle Ressourcen und Dienste von einer Menge von Servern bereitgestellt, die durch einen Object Request Broker (ORB) miteinander verbunden sind. Das Interface zu den Services wird durch die Sprache NCL beschrieben (NIIIP Common Language). Die heterogenen Daten und Applikationssysteme im virtuellen

Unternehmen werden als Objekte modelliert. Dabei kann es sich um aktive Objekte handeln, deren Verhalten durch Event-Condition-Action-AlternativeAction (ECAA)-Regeln in NCL beschrieben wird (als Erweiterung der Constraints in EXPRESS). Zusammen mit einer Wissensbasis lassen sich damit semantische Integritätsbedingungen, Expertenwissen, Security-Regeln, Geschäftsprozesse u.a. lokale oder globale Constraints modellieren. Im zugrundeliegenden Metamodell von NCL (basierend auf OSAM*) werden Regeln als Objekte betrachtet. Eine Schemadefinition in NCL wird in ein äquivalentes Schema in K.3 (einer Wissensbankprogrammiersprache) übersetzt. Dieses kann dann entweder in C^{++}-Code oder in C-Code mit einer IDL-Spezifikation übersetzt werden. Während des Übersetzungsprozesses wird die Semantik von Keyword Constraints (z.B. UNIQUE) und Assoziations- und Klassentypen (welche durch parametrisierbare Regeln gespeichert sind) in Regeln übersetzt, die an die zugehörigen Klassen gebunden werden. Um die Ausführung von Regeln zu ermöglichen, werden Mechanismen benötigt, die Methodenaufrufe beobachten (*Request Monitoring*) und die die Methoden ausführen, die den CAA-Teil der Regeln implementieren. Dabei lassen sich zwei Ansätze unterscheiden: der interpretativ-zentralisierte Ansatz und der compilativ-verteilte Ansatz. Im interpretativ-zentralisierten Ansatz werden die Methodenaufrufe durch einen zentralisierten Request Monitor zur Laufzeit abgefangen und die zugehörigen Regeln ermittelt. Dabei können die Regeln durch einen zentralisierten Regelprozessor interpretiert und ausgeführt werden. Im compilativ-verteilten Ansatz werden aus der Event/Trigger-Spezifikation der Regel zusätzlich before und after calls generiert, die in den Code eingefügt werden. Der CAA-Teil der Regeln wird in C^{++}-Methoden übersetzt. Request Monitoring und Ausführung kann verteilt auf mehreren Servern erfolgen.

ECA-Regeln in CORBA-Systemen

In [BKK96] beschreiben die Autoren am Beispiel eines Umweltinformationssystem, wie Benutzer eines föderierten Systems automatisch informiert werden können, wenn für sie relevante Informationen lokal neu eingetroffen sind (z.B. Luftmeßwerte, die bestimmte Grenzwerte überschreiten). Die Beobachtung solcher komplexen Situationen erfolgt in einem verteilten Informationssystem, das auf CORBA basiert (ORBeline bzw. Orbix). Zu lösende Probleme sind insbesondere die lokale Eventdetektion und die Anpassung eines ECA-Regelsystems an das CORBA-System. Lesender Zugriff auf die lokalen Datenbanken erfolgt über Wrapper, die Detektion primitiver Update-Events basiert auf der Anwendung lokaler Trigger. Die Detektion primitiver Events in CORBA geschieht über Filter, die als *pre-* oder *post*-Methoden definiert werden können. Für die Regelverarbeitung wird die Expertensystem-Shell CLIPS genutzt, ein frei verfügbares Produktionsregelsystem, mit dem auch ECA-Regeln definiert werden können, indem Events als Fakten integriert werden. Es ist erweiterbar durch benutzerdefinierte C-Funktionen und stellt in Kombination mit CORBA eine flexible Umgebung dar, in der effektiv hochsprachliche Regeln formuliert werden können, was sich auch für globale Integritätssicherung eignet.

9.5 Mediators

AMOS

AMOS (Active Mediator Object System) ist eine Architektur zum Suchen, Kombinieren, Ändern und Beobachten von Daten in einem verteilten Informationssystem [FRS93]. Zentraler Bestandteil von AMOS ist eine objektorientierte Query Language mit deklarativen Abfragemöglichkeiten (AMOSQL). Ein verteiltes AMOS ist in Entwicklung. Die Architektur umfaßt

vier Mediator-Klassen: Ein *Integrator* kombiniert heterogene Daten aus verschiedenen Datenquellen und erzeugt eine höhere objektorientierte Sicht der integrierten Daten. Ein objektorientierter Zugriff auf relationale Daten beinhaltet auch eine Generierung und Verwaltung von Objekt-Identifikatoren für jede Datenquelle. *Monitor-Modelle* umfassen die Beobachtung von Daten und das Signalisieren von relevanten Änderungen an die davon betroffenen Applikationen und erlauben somit die Kommunikation zwischen ihnen. Es können aktive CA-Regeln mittels AMOSQL ausgedrückt werden. Ein Interface zwischen den Regeln und den Applikationsprogrammen wird durch *Tracker* spezifiziert, Prozeduren einer Applikation eines AMOS-Servers, die bei einer Regel-Aktion aufgerufen werden [Ris89]. Domänenwissen und Daten, die in Applikationsprogrammen verborgen sind, sollten aus den Anwendungen extrahiert und in Mediators mit domänenspezifischen Modellen und Operatoren verwaltet werden, genannt *Domänen-Modelle*. Die vierte Klasse umfaßt *Locators*, die Eigenschaften anderer Mediators kennen und diese lokalisieren können, erforderlich in mobilen Umgebungen.

Context Mediator

Sciore u.a. schlagen eine Mediator-Architektur vor, um semantische Interoperabilität zu unterstützen [SSR94]. In dieser Architektur ist der *Context Mediator* die zentrale Komponente. Ein Context Mediator ist definiert als ein Agent, der den Austausch von Werten zwischen unterschiedlichen Komponenten-Informationssystemen steuert und Services anbietet, wie z.B. ein systemübergreifendes Attribut-Mapping, Bewertung von Eigenschaften und Konvertierung. Die Grundidee beruht darauf, die Semantik der gemeinsam genutzten Daten in Einheiten, sogenannten *Semantic Values*, zu erfassen und diese Werte allen Komponenten zugänglich zu machen. Semantische Werte können entweder explizit gespeichert oder in der Umgebung der lokalen Daten definiert sein, wo der Context Mediator bei Bedarf die Semantik der gemeinsam genutzten Daten abfragen kann. Zusätzlich wird Abbildungswissen in einer Komponente erfaßt, genannt *Shared Ontology*; eine Sammlung von Konvertierungsfunktionen befindet sich in einer Bibliothek, um Werte von einem Kontext in einen anderen zu konvertieren. Die Idee des Context Mediators wird auf ein relationales System angewandt. Dabei werden Anfragen in Standard-SQL so interpretiert, daß Konvertierungen und Veränderungen des Kontextes transparent für den Benutzer sind. In einer Erweiterung von SQL, genannt Context-SQL (C-SQL), kann explizit auf den Kontext eines semantischen Wertes zugegriffen werden, was in einem Prototypen realisiert wurde.

Context Interchange

Die Idee des Context Mediators wurde erweitert im *Context Interchange*-Framework, um große interoperable Datenbanksysteme zu konstruieren, bei denen die teilnehmenden lokalen Datenquellen (z.B. Datenbanken) und die globalen Benutzer häufig das System betreten und verlassen [GMS94]. Gegenüber der Context Mediator-Architektur werden mehrere Datenquellen bzw. -empfänger betrachtet. Der Context Interchange erlaubt es mehreren Datenquellen und -empfängern, gemeinsame Annahmen zu machen, als *Supra-Context* bezeichnet. Die Annahmen können aber auch ausschließlich für eine bestimmte Quelle oder einen Empfänger in einem *Micro-Context* repräsentiert werden. Bei globalen Anfragen können die Empfänger der Daten diese direkt über das Exportschema der Quellen abfragen, oder es gibt eine Abfragemöglichkeit über externe Views, die auf einem föderierten Schema definiert werden können (vgl. Abschnitt 4.2). Der Context Interchange-Ansatz zielt auf die explizite Repräsentation der unterschiedlichen Datensemantik und verzichtet auf eine Auflösung semantischer Konflikte vor der Integration (typisch für eng gekoppelte Systeme). Das erlaubt eine automatische Erkennung und Auflösung semantischer Konflikte durch den Context Mediator. Dies erfolgt je-

doch erst beim Zugriff auf die Daten. Ein Prototyp eines Context Interchange-Systems für die Integration mehrerer Finanzdatenbanken ist in Entwicklung.

MAC

Seligman und Kerschberg präsentieren einen Ansatz zur Konsistenzkontrolle zwischen Datenbankobjekten und Kopien dieser Objekte, die in Wissensbasen von Applikationen gecacht sind [SK93a]. Der Ansatz beruht auf einem intelligenten Interface zu aktiven Datenbanken und wird als *Mediator for Approximate Consistency* (MAC) bezeichnet. Der MAC gestattet es den Anwendungen, ihre Konsistenzanforderungen deklarativ zu beschreiben, basierend auf einer einfachen Erweiterung einer framebasierten Repräsentationssprache. Die Konsistenz zwischen der Kopie im Cache und der Primärkopie darf in kontrollierter Form verletzt sein (vergleichbar dem Quasi-Copy-Ansatz [AB89]). Aus der Deklaration werden automatisch die Interfaces und die für die Konsistenzwahrung nötigen Datenbankobjekte generiert. Der Mediator besteht aus zwei Modulen für die Kommunikation zwischen Applikation und aktivem DBMS: Der *Translator* übersetzt die deklarative Spezifikation der Konsistenzanforderungen in DDL- und DML-Anweisungen auf den Komponentendatenbanken, so z.B. um Queries abzusetzen oder neue Regeln zu definieren. Der *Mapper/Message Handler* empfängt die Benachrichtigung über relevante Datenbank-Updates und bildet sie auf die Wissensrepräsentation der Anwendung ab. Dabei sind zwei Arten von Messages zu behandeln: synchrone (als Antwort auf Queries vom Translator) oder asynchrone (resultierend aus dem Feuern von Regeln in der aktiven Datenbank). Die Autoren sehen den Ansatz als verallgemeinerbar für beliebige Komponenten einer Datenbankföderation an, die dynamisch Daten cachen müssen. Voraussetzung ist allerdings, daß alle Komponentensysteme ECA-Regeln unterstützen.

9.6 Verteilte Künstliche Intelligenz

ICIS

Verwandt mit dem Mediator-Konzept ist die Idee, Techniken aus der Verteilten Künstlichen Intelligenz (*Distributed Artificial Intelligence*), DAI, in verteilten Datenbanken anzuwenden, um intelligente und kooperative Interoperabilität zu unterstützen, insbesondere durch Multiagentensysteme (MAS). Der DAI-Ansatz beinhaltet hauptsächlich Techniken und Werkzeuge für unabhängige, autonome und intelligente Agenten, um gegenseitige Probleme kooperativ zu lösen. Um die Kooperation zwischen den Agenten zu erleichtern und deren Interaktionen besser zu koordinieren, wurden AI-Konzepte wie Contracting und Negotiation, Multiagent Planning und Case-Based Reasoning entwickelt. Die Einbeziehung dieser Konzepte aus der Künstlichen Intelligenz reichert verteilte Systeme mit Intelligenz an und führt schließlich zu einer neuen Generation verteilter intelligenter interoperabler Systeme (*Intelligent and Cooperative Information Systems*), ICIS. Eine generische ICIS-Architektur ist in [PLS92] beschrieben: Jedes Komponenten-Informationssystem ist mit einem intelligenten Informationsagenten versehen, der auf einem gemeinsamen Wissens- und Objektmodell basiert, um die Kommunikation und Kooperation zwischen verschiedenen Datenquellen zu erleichtern. Agenten werden dabei auch als aktive Objekte betrachtet, die als Reaktion auf bestimmte Nachrichten spezielle Retrieval-Operationen auf Daten bzw. Wissen oder Inferenzaktionen ausführen.

FCSI - Erkennung von Interdatenbankabhängigkeiten

Klusch beschreibt ein föderatives Zellsystem (FCSI) zur kontextbasierten Erkennung plausibler Interdatenbankabhängigkeiten [Klu97]. Die Architektur des FCSI ist entworfen als ein System von kooperativen autonomen Agenten (Zellen), von denen jeder eindeutig einem lokalen Datenbanksystem zugeordnet ist. Das FCSI hat das Ziel, eine kooperative Lösung für die Ermittlung semantisch in Beziehung stehender Informationen zu finden, wobei die Autonomie jedes einzelnen DBS berücksichtigt werden soll. Für diesen Zweck wird ein lokales terminologisches Informationsmodell (LIM) von jedem Agenten unter Verwendung von benutzerdefinierten intensionalen Skripten hauptsächlich durch Übersetzung des lokalen Datenbankschemas (bzw. einer Menge von Sichten) konstruiert. Jeder Informationsagent des FCSI kann zwei Arten von Interdatenbankabhängigkeiten erkennen: terminologische Interdatenbankabhängigkeiten (i-IDD) durch gegenseitige terminologische Klassifikation zwischen den LIMs sowie Interdatabase Schema Assertions (IDSA) durch eine regelbasierte Komposition von relevanten Views. Die IDSAs sind somit vergleichbar mit globalen Integritätsbedingungen. Die Erkennung der Abhängigkeiten erfolgt durch eine automatisierte Suche als auch durch eine lokale Verarbeitung der angeforderten Suchterme. Ein zentraler Vermittler ist dabei nicht erforderlich. Die Suche nach relevanten nicht-lokalen Schemadaten wird durch eine rationale Kooperation zwischen den Informationsagenten ausgeführt, die durch eine dezentrale utilitaristische Koalitionsbildung zustandekommt. Basierend auf der Erkennung von Interdatenbankabhängigkeiten, kann jeder Agent gerichtete intensionale Datenanfragen an die jeweiligen Mitglieder der Koalition stellen.

Kapitel 10

Zusammenfassung und Ausblick

10.1 Zusammenfassung der Ergebnisse

Ansatz

Ausgehend von der Frage nach dem Konflikt zwischen lokaler Autonomie und globaler Konsistenz in Datenbankföderationen, wurde in der vorliegenden Arbeit als konzeptioneller Beitrag ein Rahmen entwickelt, durch den sich Konsistenz und Autonomie in heterogenen Datenbanksystemen charakterisieren lassen.

In einem Verbund unterschiedlicher Applikations- und Datenbanksysteme (typischerweise Legacy-Systeme) kann es aufgrund der lokalen Autonomie keine zentrale Kontrolle über die Konsistenz geben. Von daher mußte die Definition des Begriffs Konsistenz in einer heterogenen Umgebung aus nichtkooperativen Komponenten gegenüber homogenen zentralisierten Systemen weitreichender gefaßt werden. Die Darstellung globaler Konsistenz erfolgte in drei Dimensionen:

- Definition der Konsistenzbedingungen unter Berücksichtigung verschiedener Arten von Heterogenität (mit Hilfe von Korrespondenzfunktionen)
- Definition von Kriterien zur Konsistenzabschwächung
- Definition von Kontrollabhängigkeiten

Autonomie wurde in den Kategorien Struktur, Verhalten und Kommunikation für unterschiedliche Ebenen eines Legacy-Systems beschrieben. Betrachtet werden dabei DBMS, Datenbank/ Datenbankschema und Applikationen sowie die Schnittstellen, die sie zur Verfügung stellen. Autonomie hat zugleich statische und dynamische Aspekte, die entweder zum Zeitpunkt der Integration eines lokalen Systems in einen Verbund oder später zur Laufzeit relevant sind.

Unser Ansatzpunkt bestand darin, aktive Mechanismen zur globalen Konsistenzkontrolle anzuwenden. Dafür wurden die besonderen Anforderungen an aktive Datenbanksysteme untersucht, insbesondere im Hinblick auf abgeschwächte Konsistenzbedingungen. Es ließ sich zeigen, daß ECA-Regeln ein ausdrucksmächtiges Konstrukt zur Spezifikation globaler modellinhärenter Konsistenzbedingungen darstellen. Dementsprechend wurden alle zu integrierenden lokalen Daten, die in eine Konsistenzprüfung einbezogen werden sollen, als aktive Objekte modelliert. Bei der Beschreibung der globalen Datenkonsistenz durch ECA-Regeln in den ge-

nannten drei Dimensionen ließen sich jeweils die Restriktionen lokaler Autonomie nachweisen. Anhand dessen konnte der Konflikt Konsistenz-Autonomie in einem heterogenen System qualitativ charakterisiert werden.

Umsetzung

Die Ausarbeitung des Ansatzes erfolgte durch Realisierung eines aktiven objektorientierten Vermittlersystems zur Konsistenzwahrung. In Analogie zur Verwendung des Konsistenzbegriffs zeigte sich, daß Konzepte zentralisierter aktiver Datenbanksysteme nicht in gleicher Weise auf heterogene Datenbanken anwendbar sind. Ereignisse können in den einzelnen lokalen Systemen über unterschiedliche Benutzerschnittstellen in verschiedenen Transaktionen ausgelöst werden und müssen jeweils dort erkannt werden. Dementsprechend wurde in der vorliegenden Arbeit ein besonderer Schwerpunkt auf das Eventmodell und die Eventdetektion in nichtkooperativen Systemen gelegt. Die Frage dabei lautete: Wie kann durch geringstmögliche Verletzung lokaler Autonomie ein hohes Maß an Konsistenz erzielt werden?

Am Beispiel relationaler Datenbanksysteme wurde gezeigt, wie ohne Eingriffe in bestehende Applikations- oder Datenbanksysteme dieses Problem gelöst werden kann. Die Idee bei der Entwicklung eines Datenbank-Gateways bestand darin, eine Schnittstelle zu schaffen, an der Dienste definiert werden, die vom lokalen System für die globale Konsistenzkontrolle zu erbringen sind. Dabei wurde auf die Übertragbarkeit des Ansatzes auf andere DBMS-Architekturen geachtet. Die gewählte Lösung, ein DBMS an seinem Kommando-Interface durch ein Gateway zu wrappen, stellte eine Kompromißlösung dar zwischen der Verwendung vorhandener Triggermechanismen und einer Anpassung bestehender Applikationssysteme. Die Hauptmerkmale des Gateways lassen sich folgendermaßen zusammenfassen.

- hohe Flexibilität bei der Erkennung beliebiger lokaler Ereignisse durch Analyse aller Benutzerkommandos an der Sprachschnittstelle des Datenbank-Servers
- ohne Einfluß auf: lokale Applikation, lokales DB-Schema, lokales DBMS
- Realisierung von Diensten entsprechend den Benutzeranforderungen:
  - Filterung relevanter Ereignisse
  - Signalisierung von Ereignissen an einen Vermittler
  - Protokollierung lokaler Ereignisse und ihrer Parameter

Die prototypische Realisierung eines Vermittlers basierte auf folgenden Voraussetzungen: Die lokalen Daten (in der gewählten Umgebung relationale Datenbanken) bleiben an ihren ursprünglichen Orten gespeichert, werden aber durch eine objektorientierte Sicht im Vermittlersystem repräsentiert. Auf diesen Sichten lassen sich Integritätsbedingungen als ECA-Regeln formulieren, die in einer Regelverarbeitungskomponente verwaltet werden. Prinzipiell werden gemäß dem Ausführungsmodell zwei Abläufe unterstützt:

- direkte Eventverarbeitung:
  Detektion lokaler Ereignisse, Signalisierung und Umwandlung in globale Ereignisse, die im Vermittlersystem interpretiert werden
- indirekte Eventverarbeitung:
  Detektion lokaler Ereignisse und Protokollierung der betroffenen Objekte, Verarbeitung gemäß globaler Integritätsregeln zu benutzerdefinierten Ereignissen

Das aktive Vermittlersystem umfaßt eine Reihe von Komponenten, die in ihrem Zusammenspiel die globale Konsistenzsicherung in einem System heterogener Datenbanken unterstützen. Dabei wird im einzelnen folgende Funktionalität angeboten:

- laufzeitdynamische Interpretation lokaler Datenbankereignisse als Ereignisse an einer globalen objektorientierten Schnittstelle
- Verwaltung und Ausführung datenbankübergreifender ECA-Regeln
- Methodensammlung für Kontrollabhängigkeiten (z.B. für asynchron replizierte Daten)
- Werkzeugkasten zur Spezifikation globaler Integritätsbedingungen und automatische Erzeugung eines Vermittlers
- Registrierung und Auswertung lokaler Schemaveränderungen

Schlußfolgerungen

Bei der Prototyp-Entwickung des Vermittlersystems trat in vielen Fällen der Gegensatz zwischen der lokalen Autonomie einerseits und der Wahrung globaler Konsistenz andererseits zutage. Folgende Auswirkungen ergeben sich auf die lokale Autonomie bei globaler Konsistenzsicherung:

- Definition globaler Konsistenz
  - Bereitstellung eines lokalen Schemas (globaler Schemainformationen)
  - Bereitstellung semantischer Informationen zur Überbrückung der Heterogenität
  - Einschränkung lokaler Schemaveränderungen, soweit sie Bestandteil globaler Konsistenzbedingungen sind
  - Berücksichtigung globaler Konsistenzbedingungen bei der Definition lokaler systemkontrollierter Constraints (Konflikte möglich)
- Eventverarbeitung
  - bei Protokollierung:
    Bereitstellung von Ressourcen für Aufzeichnung
    Verzögerung der Originalbefehle bei Protokollierung
    Synchronisation mit anderen lokalen Transaktionen
  - bei Signalisierung:
    Abhängigkeit des lokalen Systems vom Vermittlerprozeß
- Regelverarbeitung
  - Definition der erforderlichen lokale Zugriffsrechte für einen Vermittler
  - Synchronisation lokaler Uhren (bei zeitabhängigen Kontrollabhängigkeiten)
  - Kontrolle der Verbindung des Vermittlers zum LDBMS (Verfügbarkeit)
  - Vorschreiben bestimmter Datenbankoperationen, z.B. Überschreiben lokaler Daten
  - Verbieten lokaler Datenbankoperationen bei Konsistenzverletzung

Im Rahmen der Implementierung des Vermittlersystems wurde eine Reihe von Erkenntnissen über die Kombination von Eigenschaften von Multidatenbanksystemen mit denen aktiver DBS gesammelt. Diese sind insbesondere für die Konstruktion aktiver Multidatenbanksysteme von Interesse, deren Entwicklung zur Zeit erst am Anfang steht.

10.2 Ausblick auf künftige Arbeiten

Im Laufe der Arbeit ergaben sich eine Reihe von Fragestellungen, die sowohl konzeptionell als auch bei der Entwicklung des Prototypen nur skizziert bzw. nicht weiter verfolgt wurden, aber für weiterführende Arbeiten lohnenswert erscheinen.

Event Monitoring. Die aktive Komponente des Vermittlerystems ist erweiterbar in bezug auf komplexe Ereignisse, wobei eine verteilte Komposition angestrebt werden sollte, um Engpässe zu vermeiden. Ein wichtiger Aspekt ist die Behandlung von Ausfällen von Verbindungen zu lokalen Datenbank-Servern, wobei zwischen einer regulären Beendigung und Systemabstürzen unterschieden werden muß. Entsprechend sind geeignete Maßnahmen für den Wiederanlauf vorzusehen, so daß Integritätsregeln retroaktiv angewandt werden können.

Autonomie. Eine interessante theoretische Fragestellung ergab sich aus der quantitativen Bewertung der Autonomie von Datenbanken, gemessen am Anteil der Daten, die Bestandteil globaler Integritätsbedingungen sind. Dabei sollten zum einen die Daten selbst als auch Art und Häufigkeit der Zugriffe auf ihnen betrachtet werden.

Werkzeuge. Als unabdingbar bei der Entwicklung des Vermittlersystems erwies sich der Einsatz von Werkzeugen zur Konfigurierung und Generierung der benötigten Komponenten. Ein solcher Werkzeugkasten wurde bereits als Prototyp realisiert. Neben den Schemainformationen und Integritätsbedingungen sollten auch administrative Festlegungen getroffen werden, die die Anforderungen der Benutzer hinsichtlich des Grades benötigter globaler Datenkonsistenz geeignet widerspiegeln.

Metadaten. Die Erzeugung der globalen Sichten erfolgt zwar automatisch durch Verarbeitung der lokalen (relationalen) Schemata, eine semantische Anreicherung findet jedoch nicht statt. Wie bereits erwähnt, besitzt das Vermittlersystem keine eigene persistente Datenhaltung. Wünschenswert ist jedoch eine Verwaltung von Metadaten, insbesondere zur Darstellung von Objekt- und Wertkorrespondenzen sowie semantischen Informationen, die in lokalen Applikationen verborgen sind. Damit wäre es möglich, diese Informationen aus den ECA-Regeln zu extrahieren, so daß die Regeln ausschließlich Steuerungsfunktionen wahrnehmen gemäß den Anforderungen an die Qualität der Daten.

Verteilung. Der realisierte Prototyp demonstriert die Wirkungsmöglichkeiten eines Gateways im Zusammenspiel mit einem lokalen Server. Diese Lösung müßte im Hinblick auf eine Architektur, bestehend aus mehreren Gateways zu unterschiedlichen Datenbank-Servern, erweitert werden.
Die Ausführung von globalen ECA-Regeln sollte in vielen Fällen auf einem globalen Transaktionsmodell beruhen, was voraussetzt, daß die lokalen Transaktionsmanager an gemeinsamen Protokollen teilnehmen, so daß TP-Monitore für verteilte Verarbeitung eingesetzt werden können. Interessant erscheint auch die Realisierung der Vermittlerprozesse basierend auf verteilten Objektsystemen, insbesondere unter Nutzung der CORBA-Technologie. Kommerziell verfügbare Lösungen, wie z.B. die Integration von Persistence mit dem Object Request Broker Orbix, sind dafür als Plattform mittlerweile vorhanden.
Der hier vorgestellte Ansatz beruht darauf, daß die Verarbeitung der Regeln und der sie triggernden Ereignisse in einem Prozeß lokalisiert ist. Es wäre noch zu untersuchen, inwieweit eine dezentrale Regelmanagement-Architektur den Anforderungen autonomer Systeme besser gerecht werden kann. Hierbei können insbesondere Vorschläge, wie sie in Protokolle für die Integritätskontrolle in föderierten Datenbanksystemen eingegangen sind, aufgegriffen und weiterentwickelt werden, so daß globale Transaktionen weitgehend reduziert werden.

Die Ideen, die in dieser Dissertation entwickelt und umgesetzt wurden, ordnen sich ein in Bestrebungen, eine intelligente Informationsinfrastruktur für verteilte und datenintensive Applikationen zu entwerfen. Die Vision hierbei ist, eine semantische Zwischenschicht zwischen objektorientierten Basisplattformen (CORBA, OLE/DCOM) und komplexen Benutzerapplikationen einzuführen. Eine solche Schicht hat die Aufgabe, Services aktiver Datenbanksysteme anzubieten, ohne jedoch deren volle Funktionalität zu gebrauchen (*Unbundling*). Bestandteil einer solchen Architektur sollte ein Dienst zur Kontrolle semantischer Integritätsbedingungen sein, zu dem die vorliegende Arbeit Konzepte und Realisierungsmöglichkeiten aufgezeigt hat.

Anhang A

Grammatik von GISpeL

gispel	:=	[table_section] [rule_section]
table_section	:=	‘#T’ [table_definition]$^{+}$ ‘#’
table_definition	:=	name ‘=’ (db_name | class_name | attribute_name) ‘;’
	|	[name ‘=’] class_name LOG [name] ‘;’
db_name	:=	name ‘.’ name
class_name	:=	name ‘.’ name ‘.’ name
attribute_name	:=	name ‘.’ name ‘.’ name ‘.’ name
name	:=	(letter | ‘_’) [letter | digit | ‘_’]*
letter	:=	‘A’ | ... | ‘Z’ | ‘a’ | ... | ‘z’
digit	:=	‘0’ | ‘1’ | ... | ‘9’
rule_section	:=	[(existence_dep | value_dep) ‘;’]$^{+}$
existence_dep	:=	attributes ‘:-’ attributes_mt [predicate] [time_clause] [policies]
	|	attributes ‘:-:’ attributes [predicate] [time_clause] [policies]
value_dep	:=	attributes ‘:=’ attributes_mt [predicate] [time_clause] [policies]
	|	attributes ‘:=:’ attrs_or_value [predicate] [time_clause] [policies]
attributes	:=	attribute_name
	|	class_name ‘.’ ‘<‘ attribute_list ‘>’
attribute_list	:=	name ‘,’ name
	|	attribute_list ‘,’ name
attributes_mt	:=	attributes
	|	db_name ‘.’ ‘<‘ attribute_list2 ‘>’
attribute_list2	:=	name2 ‘,’ name2
	|	attribute_list2 ‘,’ name2
name2	:=	name ‘.’ name
attrs_or_value	:=	attributes_mt
	|	class_name ‘.’ value
value	:=	number | string
number	:=	‘0’ | ([‘-’] [digit]$^{+}$ [‘.’ [digit]$^{+}$])

string := anything quoted with ‘ or “

predicate := WHERE where_clause

where_clause := value_or_name comparison_op value_or_name
| where_clause AND where_clause
| where_clause OR where_clause
| NOT where_clause
| ‘(‘ where_clause ‘)’

value_or_name := value | name | name2

comparison_op := ‘=’ | ‘<‘ | ‘>’ | ‘!=’ | ‘<>’ | ‘<=’ | ‘>=’ | ‘!>’ | ‘!<‘

time_clause := at point_in_time
| WITHIN time_interval
| at point_in_time EVERY time_interval [UNTIL point_in_time]
| EVERY time_interval [UNTIL point_in_time]

at := AT | ‘@’

point_in_time := date | ([date ‘,’] time)

date := (‘1’ | ... | ‘31’) ‘.’ (‘1’ | ... | ‘12’) ‘.’ (‘0000’ | ... | ‘9999’)

time := (‘0’ | ... | ‘23’) [‘:’ (‘00’ | ... | ‘59’) [‘:’ (‘00’ | ... | ‘59’)]]

time_interval := time_unit | (number time_unit_s) | time

time_unit := YEAR | MONTH | DAY | HOUR | MINUTE | SECOND

time_unit_s := YEARS | MONTHS | DAYS | HOURS | MINUTES | SECONDS

policies := policy_type [policy_item]$^{+}$ [policies]

policy_type := BLOCK | PROPAGATE

policy_item := INSERTLEFT | INSERTL | INSLEFT | INSL | ILEFT | IL
| INSERTRIGHT | INSERTR | INSRIGHT | INSR | IRIGHT | IR
| UPDATELEFT | UPDATEL | UPDLEFT | UPDL | ULEFT | UL
| UPDATERIGHT | UPDATER | UPDRIGHT | UPDR | URIGHT | UR
| DELETELEFT | DELETEL | DELLEFT | DELL | DLEFT | DL
| DELETERIGHT | DELETER | DELRIGHT | DELR | DRIGHT | DR

Anhang B

Anwendungsbeispiel

Italienische Stromversorger

Für die Implementierung von globalen Integritätsregeln im Vermittlersystem wurde ein Beispiel ausgewählt, das zurückgeht auf eine Arbeit von Ceri und Widom [CW90, CW93] als Ergebnis einer Fallstudie, die mit ENEL, der Italienischen Energieversorgungsagentur, durchgeführt wurde. Diese Anwendung behandelt die Probleme globaler Datenintegrität anhand heterogener Datenbanken, die überlappende Weltausschnitte modellieren. Daten zur Beschreibung von Stromnetzen, d.h. Versorger, Infrastruktur und Verbraucher, sind mehrfach in relationalen Datenbanken gespeichert. Einige Datenbanken beschränken sich auf einzelne Regionen, andere wiederum auf große Gebiete.

Wir betrachten hierzu exemplarisch zwei Datenbanken, `MN` (*Milano Network*), und `IN` (*Italy Network*). Die Datenbank `MN` beschreibt das Stromversorgungsnetz in der Region von Mailand (Milano). Sie enthält - jeweils in einzelnen Tabellen - Daten über den Strom, der von den verschiedenen Kraftwerken (in Tabelle `MN_Plant`) produziert wird, über den Stromverbrauch bei verschiedenen Kunden (`MN_User`) und den Verlust an jedem dazwischengeschalteten Netzknoten (`MN_Node`). Alle Knoten im Netz sind durch gerichtete Leitungen (`MN_Wire`) verbunden, die jeweils entsprechend ihres Typs (`MN_WireType`) Strom in einer bestimmten Spannung und Stromstärke transportieren können.

Die Datenbank `IN` unterstützt die Kontrolle der Stromversorgungsnetzwerke in ganz Italien. Kraftwerke, Zwischenknoten und Abnehmer sind hier in einer einzigen Tabelle (`IN_Node`) repräsentiert und werden durch den Wert des Attributs `function` (‘`p`’, ‘`u`’ oder ‘`n`’) unterschieden. Zwischen Knoten sind gerichtete Verbindungen definiert (`IN_Connection`), denen Leitungen eines bestimmten Typs zugeordnet sind (`IN_Wire`, `IN_WireType`).

Die Relationenschemata beider Datenbanken sind definiert wie folgt:

- Datenbank MN:
  `MN_Plant (id, location, power)`
  `MN_User (id, location, power)`
  `MN_Node (id, location, power)`
  `MN_Wire (wire_id, from_point, to_point, type, voltage, power)`
  `MN_WireType (type, max_voltage, max_power, cross_section)`

- Datenbank IN:
  IN_Node (id, region, location, function, power)
  IN_Connection (connection_id, from_node, to_node)
  IN_Wire (wire_id, connection_id, type, voltage, power, age)
  IN_WireType (type, max_voltage, max_power, wire_size)

Das zu betrachtende Konsistenzproblem besteht in diesem Beispiel darin, daß der Anteil der Daten der nationalen Datenbank IN, der der Region Mailand zuzuordnen ist (Attribut region = "Milano") konsistent mit den Daten der regionalen Datenbank MN sein muß.

Die Datenbanktabellen wurden in Klassen eines äquivalenten Objektmodells (Objektschema-Datei NetworkApp.persist) überführt und mit Hilfe des Persistence-Objektmodelleditors erstellt. Dabei wurde in der Datenbank MN eine virtuelle Klasse MN_Point als Superklasse der drei Klassen MN_Plant, MN_User und MN_Node eingeführt. Somit besteht ein Äquivalent zur Klasse IN_Node, und eine gemeinsame Zugriffsschnittstelle für Objekte der Subklassen. Abbildung B.1 zeigt das resultierende Objektschema in OMT-Notation.

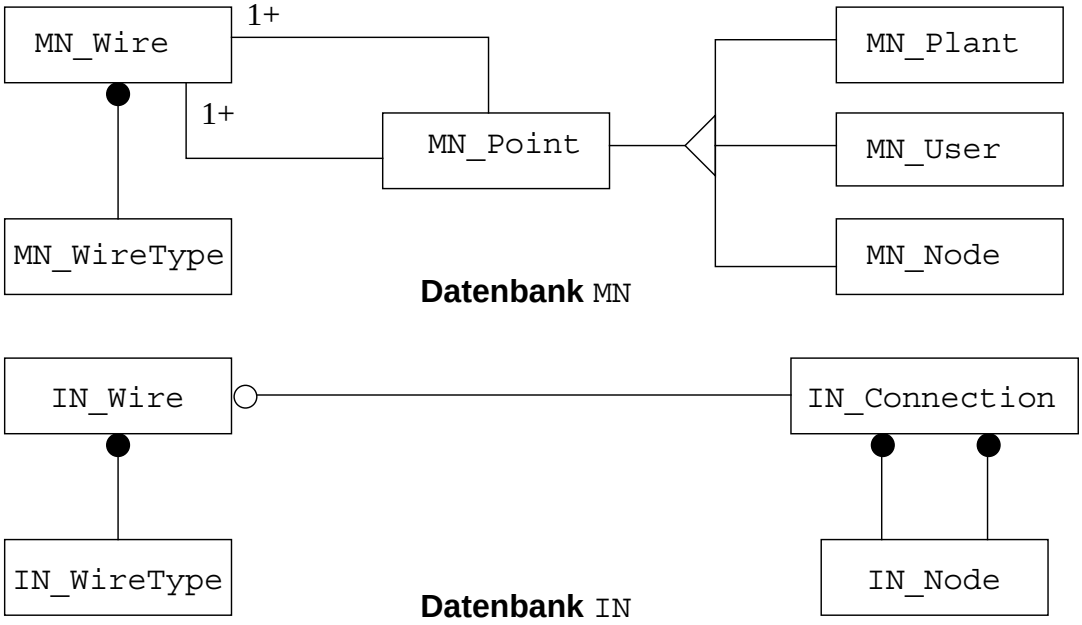


Abbildung B.1: Anwendungsbeispiel in OMT-Darstellung

Zwei lokale Testdatenbanken wurden eingerichtet (MN in Sybase, IN in Informix) und damit die aktive Funktionalität des Vermittlers getestet.

Die GISpeL-Datei ist folgendermaßen spezifiziert:

```
#Tables
        Italy = Informix.dbmrsi
        Milano = Sybase.dbmrsi;
#
        Milano.MN_Plant.id :- Italy.IN_Node.id
            Where region = 'Milano' AND function = 'p'
            Block ir ur;

        Italy.IN_Wire.voltage := Milano.MN_Wire."low"
            Where MN_Wire.wire_id = IN_Wire.wire_id
              AND MN_Wire.voltage <= 2000;
```

Interpretation

Das erste Beispiel beschreibt eine gerichtete Existenzabhängigkeit: Wenn es einen Knoten in der Datenbank `IN` gibt mit den Attributen `region = 'Milano'` und `function = 'p'` (Plant), so muß dieses Kraftwerk auch in der Datenbank `MN` existieren. Die Konsistenzverletzung kann durch Einfügen von Objekten rechterseits des Operators bewirkt werden, `ir` (Insert Dependent), oder durch Ändern von Attributen, die in Prädikat oder Identifikationsfunktion enthalten sind, `ur` (Update Dependent). Als Konsistenzwahrungsstrategie wird in diesen Fällen `Block` festgelegt, d.h. die Änderungen werden überhaupt nicht zugelassen.
Die generierten Regeln haben den Präfix `RuDe_A_0`.

Das zweite Beispiel beschreibt eine gerichtete Wertabhängigkeit: Die Beziehung besagt, daß der Wert des Attributs `voltage` eines Objekts aus `IN_Wire` 'low' sein muß, wenn es ein korrespondierendes Objekt mit gleicher `wire_id` aus `MN_Wire` gibt, für das der Wert von `MN_Wire.voltage` kleiner ist als `2000`. Es werden keine Strategien definiert, somit gilt die Standardannahme `Propagate` bei Konsistenzverletzung.
Die generierten Regeln haben den Präfix `RuDe_A_1`.

Regelcode in C++

1. Beispiel: Gerichtete Existenzabhängigkeit

```
//============================================================================
// A_0: delete left --> propagate
//============================================================================
int RuDe_A_0_dl::condition(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   int result = NO_VIOLATION;
   short newTrans = 0;

   if (aPersistenceObject == NULL)
      return result;

   MN_Point *object = (MN_Point *) aPersistenceObject;

   // Connection ueberpruefen

   PS_Connection *conn = IN_Node::getConnection();
   OPS_Connection *oconn = theconnGroup->getConnection(conn);
   newTrans = checkConn(oconn);

   // Korrespondierende Objekte abfragen

   char *whereclause = new char[1024];
   sprintf(whereclause,"id=% AND(region='Milano' AND function = 'p')",object->ge-
   tid());
   IN_Node_Cltn *corrObjCltn = IN_Node::querySQLWhere(whereclause,1);

   // Ergebnis abfragen

   if (corrObjCltn->extent() > 0) {
      result = VIOLATION
   }

   delete whereclause;
   delete corrObjCltn;

   if (newTrans)
      oconn->commitTransaction();
```

```
   return result;
}

void RuDe_A_0_dl::action(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   short newTrans = 0;

   if (aPersistenceObject == NULL)
      return;

   MN_Point *object = (MN_Point *) aPersistenceObject;

   // Connection ueberpruefen

   PS_Connection *conn = IN_Node::getConnection();
   OPS_Connection *oconn = theconnGroup->getConnection(conn);
   newTrans = checkConn(oconn);

   // propagate: Korrespondierende Objekte löschen

   char *whereclause = new char[1024];
   sprintf(whereclause,"id=% AND(region='Milano' AND function = 'p')",object->ge-
   tId());

   IN_Node_Cltn *corrObjCltn = IN_Node::querySQLWhere(whereclause,1);

   for (int i=0; i < corrObjCltn->extent(); i++) {
      (*corrObjCltn)[i]->remove();
   }

   delete whereclause;
   delete corrObjCltn;

   if (newTrans)
      oconn->commitTransaction();
   return;
}

//============================================================================
// A_0: insert right(1) --> block
//============================================================================
int RuDe_A_0_ir1::condition(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   [...]
   if (aPersistenceObject == NULL)
      return result;
   IN_Node *object = (IN_Node *) aPersistenceObject;
   IN_Node_Cltn *objCltn = IN_Node::querySQLWhere("region='Milano' AND function
   ='p'");
   if (!objCltn->contains(*object))
      return result;
   [...]
   // Korrespondierende Objekte abfragen

   char *whereclause = new char[1024];
   sprintf(whereclause, "id=%d", object->getId());
   MN_Point_Cltn *corrObjCltn = MN_Point::querySQLWhere(whereclause,1);

   // Ergebnis abfragen

   if (corrObjCltn->extent() == 0)
      result = VIOLATION;
```

```
   [...]
   return result;
}

void RuDe_A_0_ir1::action(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   [...]

   if (aPersistenceObject == NULL)
      return;

   IN_Node *object = (IN_Node *) aPersistenceObject;

   // block: Transaktion rueckgaengig machen

   PS_Connection *conn = IN_Node::getConnection();
   conn->beginTransaction();
   PS_THROW (PS_RomsError(PS_RomsError::ACTION_NOT_ALLOWED,"IN_Node"), PS_NOVALUE);
   conn->commitTransaction();

   return;
}

//============================================================================
// A_0: update predicate (1) --> block
//============================================================================
int RuDe_A_0_uw1::condition(PersistenceObject *aPersistenceObject, void *aVptr)const
{
   // analog zu RuDe_A_0_ir::condition(...)
}

void RuDe_A_0_uw1::action(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   // analog zu RuDe_A_0_ir1::action(...)
}

//============================================================================
// A_0: update predicate (2) --> block
//============================================================================
int RuDe_A_0_uw2::condition(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   // analog zu RuDe_A_0_r1::condition
}

void RuDe_A_0_uw2::action(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   // analog zu RuDe_A_0_r1::action
}
```

2. Beispiel: Gerichtete Wertabhängigkeit

```
//============================================================================
// A_1: insert left (1) --> propagate
//============================================================================
int RuDe_A_1_il::condition(PersistenceObject *aPersistenceObject, void *aVptr) const
{
[...]
   // Korrespondierende Objekte abfragen

   char *whereclause = new char[1024];
   sprintf(whereclause, "MN_Wire.wire_id=%d AND MN_Wire.voltage<=2000",
           object->getWire_id();
```

```
   MN_Wire_Cltn *corrObjCltn = MN_Wire::querySQLWhere(whereclause);

   if (corrObjCltn->extent()>0) {

      for (int i=0; i < corrObjCltn->extent(); i++) {
         // Vergleich zwischen Attributen
         MN_Wire *corrObj = (*corrObjCltn)[i];
         if (!strcmp(object->getVoltage(),"low")) break;
      }
      if (i >= corrObjCltn->extent())
         result = VIOLATION;
   }
   [...]
   return result;
}

void RuDe_A_1_il::action(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   [...]
   // propagate: Objekte anpassen
   [...]
   char *whereclause = new char[1024];
   sprintf (whereclause, "MN_Wire.wire_id=%d AND MN_Wire.voltage<=2000",
            object->getWire_id());

   MN_Wire_Cltn *corrObjCltn = MN_Wire::querySQLWhere(whereclause,1);
   MN_Wire *corrObj = (*corrObjCltn)[0];

   object->setVoltage("low");
   if (object->update() == 0) {
      // Fehler beim Update? => Transaktion zurücksetzen
      PS_Connection *conn = IN_Wire::getConnection();
      conn->beginTransaction();
      PS_THROW(PS_RomsError(PS_RomsError::ACTION_NOT_ALLOWED,"IN_Node"),PS_NOVALUE);
      conn->commitTransaction();
   }
   [...]
   return;
}

//============================================================================
// A_1: insert right --> propagate
//============================================================================
int RuDe_A_1_ir::condition(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   [...]
   // Erfuellt Objekt das Praedikat?

   MN_Wire_Cltn *objCltn = MN_Wire::querySQLWhere("MN_Wire.voltage<=2000");
   if (!objCltn->contains(*object))
      return result;
   [...]
   // Korrespondierende Objekte abfragen

   char *whereclause = new char[1024];
   sprintf(whereclause, "%d=IN_Wire.wire_id",object->getWire_id(),
           object->getVoltage());

   IN_Wire_Cltn *corrObjCltn = MN_Wire::querySQLWhere(whereclause);

   if (corrObjCltn->extent()>0) {

      for (int i=0; i < corrObjCltn->extent(); i++) {
```

```
        // Vergleich zwischen Attributen
        IN_Wire *corrObj = (*corrObjCltn)[i];
        if (strcmp(object->getVoltage(),"low")) break;
      }
      if (i >= corrObjCltn->extent())
        result = VIOLATION;
   }
   [...]
   return result;
}

void RuDe_A_1_ir::action(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   [...]
   // propagate: Objekt anpassen
   [...]
   char *whereclause = new char[1024];
   sprintf(whereclause, "%d=IN_Wire.wire_id",object->getWire_id(),
           object->getVoltage());

   IN_Wire_Cltn *corrObjCltn = MN_Wire::querySQLWhere(whereclause);

   for (int i=0; i < corrObjCltn->extent(); i++){
      IN_Wire *corrObj = (*corrObjCltn0[i];
      corrObj->setVoltage("low");
   }
   [...]
   return;
}

//============================================================================
// A_1: update left (1) --> propagate
//============================================================================
int RuDe_A_1_ul1::condition(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   // analog zu RuDe_A_1_il::condition
}

void RuDe_A_1_ul1::action(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   // analog zu RuDe_A_1_il::action
}

//============================================================================
// A_1: update predicate (1) --> propagate
//============================================================================
void RuDe_A_1_uw1::condition(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   // analog zu RuDe_A_1_ir::condition
}

void RuDe_A_1_uw1::action(PersistenceObject *aPersistenceObject, void *aVptr) const
{
   // analog zu RuDe_A_1_ir::action
```

Literaturverzeichnis

[AAK+93] J. Albert, R. Ahmed, M. Ketabchi, W. Kent, M.-C. Shan: *Automatic Importation of Relational Schemas in Pegasu*s, Proc. of the Internat. Workshop on Research Issues in Data Engineering - Interoperability in Multidatabase Systems (RIDE-IMS), Wien, 1993.

[AB89] R. Alonso, D. Barbará: *Negotiating Data Access in Federated Database Systems*, Proc. of the 5th Internat. Conference on Data Engineering, Los Angeles, 1989.

[ABD+93] R. Arizio, E. Bomitali, M.L. Demarie, A. Limongiello, P. Mussa: *Managing Interdatabase Dependencies with "Rules + Quasi-Transactions"*, Proc. of the Internat. Workshop on Research Issues in Data Engineering - Interoperability in Multidatabase Systems (RIDE-IMS), Wien, 1993.

[Adl95] N. Adly: *Management of Replicated Data in Large Scale Systems*, PhD Thesis, Corpus Christi College, University of Cambridge, 1995.

[AM88] K.R. Abbott, D. McCarthy: *Administration and Autonomy in a Replication-Transparent Distributed DBMS*, Proc. of the 14th Internat. VLDB Conference, Los Angeles, 1988.

[ANS75] X3/SPARC Study Group on Database Management Systems: *Interim Report*, FDT ACM SIGMOD Record 7(1975) 2.

[AQFG95] R.M. Alzahrani, M.A. Qutaishat, N.J. Fiddian, W.A. Gray: *Integrity Merging in an Object-Oriented Federated Database Environment*, Proc. of the 15th British National Conference on Databases (BNCOD).

[BB95] H. Branding, A. Buchmann: *On Providing Soft and Hard Real-Time Capabilities in an Active DBMS*, Proc. of the Internat. Workshop on Active and Real-Time Databases, Skovde, Schweden, 1995.

[BBC80] P. Bernstein, B. Blaustein, E. Clarke: *Fast maintenance of semantic integrity assertions using redundant aggregate data*, Proc. of the 6th Internat VLDB Conference, Montreal, 1980.

[BBKZ92] A. Buchmann, H. Branding, T. Kudrass, J. Zimmermann: *REACH: A REal-Time ACtive and Heterogeneous Mediator System*, Database Engineering 15 (1992) 1-4, S. 44-47.

[BBKZ93] H. Branding, A. Buchmann, T. Kudrass, J. Zimmermann: *Rules in an Open System: The REACH Rule System*, Proc. of the 1st Internat. Workshop on Rules in Database Systems (RIDS), Edinburgh, 1993, Springer-Verlag.

[BCN92] C. Batini, S. Ceri, S. Navathe: *Conceptual Database Design.*, Benjamin/Cummings, 1992.

[BCV86] A. Buchmann, R.S. Carrera, M.A. Vazquez-Galindo: *Handling Constraints and their Exceptions: An Attached Constraint Handler for Object-Oriented CAD Databases*, Proc. of the Internat. Workshop on Object-Oriented Database Systems, IEEE Computer Society Press, Washington, Sept. 1986.

[BD88} A. Buchmann, U. Dayal: *Constraint and Exception Handling for Design, Reliability and Maintainability*, Proc. of ASME Managing Engineering Data: Emerging Issues, 1988.

[BE95] O.A. Bukhres, A.K. Elmagarmid (Eds.): *Object-Oriented Multidatabase Systems - A Solution for Advanced Applications*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[BG94] D. Barbará, H. Garcia-Molina: *The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems*, VLDB Journal, 3(1994) 3, S. 325-353.

[BHM90] P. Bernstein, M. Hsu, B. Mann: *Implementing Recoverable Requests Using Queues*, Proc. of the ACM SIGMOD Internat. Conference on Management of Data, Atlantic City, 1990.

[BHP92] M.W. Bright, A.R. Hurson, S.H. Pakzad: *A Taxonomy and Current Issues in Multidatabase Systems*, IEEE Computer 25(1992) 3, S. 50-60.

[BIPG92] J. M. Blanco, A. Illarramendi, J.M. Perez, A. Goñi: *Making a Federated System Active*, Proc. of the 3rd DEXA Conference, Valencia, 1992, Springer-Verlag.

[Bit96] T. Bittmann: *A Rule Management System in a Mediator for Heterogeneous Databases*, Diplomarbeit, TH Darmstadt, FG DVS1, 1996.

[BKK96] G. v. Bültzingsloewen, A. Koschel, R. Kramer: *Active Information Delivery in a CORBA-based Distributed Information System*, Proc. of the 1st Internat. Conference on Cooperative Information Systems (CoopIS), Brüssel, 1996.

[BLN86] C. Batini, M. Lenzerini, S. Navathe: *A Comparative Analysis of Methodologies for Database Schema Integration*, ACM Computing Surveys 18(1986) 4, S. 323-364.

[BLT86] J. Blakeley, P. Larson, F. Tompa: *Efficiently Updating Materialized Views*, Proc. of the ACM SIGMOD Internat. Conference on Management of Data, Washington, 1986.

[BÖH+92] A. Buchmann, T. Özsu, M. Hornick, D. Georgakopoulos, F. Manola: *A Transaction Model for Active Distributed Object Systems*, in [Elm92], S. 123-158.

[Bro93] M. Brodie: *The promise of distributed computing and the challenges of legacy information systems*, in [HNS93], S. 264-302.

[BSKW91] T. Barsalou, N. Siambela, A. Keller, G. Wiederhold: *Updating Relational Databases through Object-Based Views*, Proc. of the ACM SIGMOD Internat. Conference on Management of Data, Denver, 1991.

[Buc90] A. Buchmann: *Modelling Heterogeneous Systems as a Space of Active Objects*, Proc. of the 4th Internat. Workshop on Persistent Objects, Martha's Vinyard, Sept. 1990.

[Buc94] A. Buchmann: *Active Object Systems*, in: A. Dogac, M.T. Özsu, A. Biliris, T. Sellis (Eds.): *Advances in Object-Oriented Database Systems*, Springer-Verlag, 1994.

[BZBW95] A. Buchmann, J. Zimmermann, J. Blakeley, D. Wells: *Building an Integrated Active OODBMS: Requirements, Architecture und Design Decisions*, Proc. of the 11th Internat. Conference on Data Engineering, Taipeh, Taiwan, 1995.

[Cas93] M. Castellanos: *Semiautomatic Enrichment for the Integrated Access in Interoperable Databases*, Ph.D. dissertation, Univ. Politécnica de Catalunya, Dept. LSI, Barcelona, 1993.

[Cat94] R. Cattell (Ed.): *The Object Database Standard: ODMG-93*, Morgan Kaufmann Publ., 1994.

[CB95] S. Ceri, A. Buchmann: *A Feature Benchmark of Active Database Systems*, Techn. Bericht, TH Darmstadt, FG DVS1, 1995.

[CGW96] S. Chawathe, H. Garcia-Molina, J. Widom: *A Toolkit for Constraint Management in Heterogeneous Information Systems*, Proc. of the 12th Internat. Conference on Data Engineering, New Orleans, 1996.

[CHKS91] S. Ceri, M. Houtsma, A. Keller, P. Samarati: *A Classification of Update Methods for Replicated Databases*, Technical Report STAN-CS-91-1932, Stanford University, 1991.

[CHKS95] S. Ceri, M. Houtsma, A. Keller, P. Samarati: *Independent Updates and Incremental Agreement in Replicated Databases*, Distributed and Parallel Databases 3(1995) 3.

[Chr95] Christo und Jeanne-Claude: *Wrapped Reichstag. Project for Berlin*, WWW-Page: http://www.youcan.com/youcan/christo/, 1995.

[CKSG94] M. Castellanos, T. Kudrass, F. Saltor, M. García-Solaco: *Interdatabase Existence Dependencies: a Metaclass Approach*, Proc. of the Internat. Conference on Parallel and Distributed Databases (PDIS), Austin, 1994.

[CL88] T. Connors, P. Lyngbaek: *Providing uniform access to heterogeneous information bases*, in: K.R. Dittrich (Ed.): *Advances in Object-Oriented Database Systems*, LNCS 334, Springer-Verlag, 1993, S. 162-173.

[CM91] S. Chakravarthy, D. Mishra: *An Event Specification Language (SNOOP) for Active Databases and its Detection*, TR 91-23, Univ. of Florida, Gainesville, Sept. 1991.

[CR93] P. Chrysanthis, K. Ramamritham: *Impact of Autonomy Requirements on Transactions and their Management in Heterogeneous Distributed Database Systems*, Proc. of the Internat. Workshop on Research Issues in Data Engineering - Interoperability in Multidatabase Systems (RIDE-IMS), Wien, 1993.

[Cri89] F. Cristian: *A probabilistic approach to distributed clock synchronization*, Proc. of the 9th Internat. Conference on Distributed Computing Systems, 1989.

[CSG92] M. Castellanos, F. Saltor, M. Garcia: *A Canonical Model for the Interoperability among Object Oriented and Relational Databases*, Internat. Workshop on Distributed Object Management (DOM), in [ÖDV94], S. 309-314.

[CT95] S. Conrad, C. Türker: *Active Integrity Maintenance in Federated Database Systems*, Preprint Nr. 9, Institut für Technische Informationssysteme, Universität Magdeburg, Nov. 1995.

[CW90] S. Ceri, J. Widom: *Deriving Production Rules for Constraint Maintenance*, Proc. of the 16th Internat. VLDB Conference, Brisbane, 1990.

[CW93] S. Ceri, J. Widom: *Managing Semantic Heterogeneity with Production Rules and Persistent Queues*, Proc. of the 19th Internat. VLDB Conference, Dublin, 1993.

[Dav84] S. Davidson: *Optimism and Consistency in Partitioned Distributed Database Systems*, ACM Transactions on Database Systems 9(1984) 3, S. 456-81.

[DBM88] U. Dayal, A. Buchmann, D. McCarthy: *Rules are Objects Too: A Knowledge Model for an Active Object-Oriented Database System*, Proc. of the 2nd Internat. Workshop on Object-Oriented Database Systems, Bad Muenster, Sept. 1988.

[DD95] L. Do, P. Drew: *Active Database Management of Global Data Integrity Cnstraints in Heterogeneous Database Environments*, Proc. of the 11th Internat. Conference on Data Engineering, Taipeh, Taiwan, 1995.

[DEK90] W. Du, A. Elmagarmid, W. Kim: *Effects of Local Autonomy on Heterogeneous Distributed Database Systems*, MCC Technical Report, ACT-OODS-EI-059-90, Microelectronics and Computer Technology Corp., Austin, 1990.

[DELO89] W. Du, A. Elmagarmid, Y. Leu, S. Ostermann: *Effects of Local Autonomy on Global Concurrency Control in Heterogeneous Distributed Database Systems*, Proc. of the 2nd Internat. Conference on Data and Knowledge Systems for Manufacturing and Engineering, Gaithersburg, 1988.

[DGG95] K. Dittrich, A. Geppert, S. Gatziu: *The Active Database Management Systems Manifesto*, Proc. of the 2nd Internat. Workshop on Rules in Database Systems (RIDS), Athen, Sept. 1995, Springer-Verlag.

[DH84] U. Dayal, H. Hwang: *View definition and generalization for database integration in multibase: A system for heterogeneous distributed databases*, IEEE Transactions on Software Engineering, 10(1984) 6, S. 628-644.

[DHW95] U. Dayal, E. Hanson, J. Widom: *Active Database Systems*, in: W. Kim: *Modern Database Systems*, Addison-Wesley, 1995, S. 434-456.

[DKM85] K.R. Dittrich, A.M. Kotz, J.A. Mülle: *Basismechanismen für komplexe Konsistenzprobleme in Entwurfsdatenbanken*, Datenbank-Systeme in Büro, Technik und Wissenschaft (BTW), GI-Fachtagung, Karlsruhe, 1985, Informatik-Fachberichte 94, Springer-Verlag.

[Elm92] A. Elmagarmid (Ed.): *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publ., 1992.

[EN94] R. Elmasri, S. Navathe: *Fundamentals of Database Systems*, 2nd Edition, Benjamin/Cummings, 1994.

[Ens78] P. Enslow: *What is a "Distributed" Data Processing System?*, IEEE Computer 11(1978) 1, S. 13-21.

[Etz90] O. Etzion: *PARDES - An active semantic database model*, Technical Report ISE-TR-90-1, Technion-Israel Institute of Technology, Dez. 1990.

[Etz93] O. Etzion: *Active Interdatabase Dependencies*, Information Sciences 75(1993), S. 133-163, Elsevier Science Publ. Co.

[FRS93] G. Fahl, T. Risch, M. Sköld: *AMOS - An Architecture for Active Mediators*, Next Generation Information Technologies and Systems Workshop (NGITS'93), Haifa, Israel, IDA Technical Report, LiTH-IDA-R-93-13, 1993.

[GA93] P. Grefen, P. Apers: *Integrity control in relational database systems - An overview*, Data & Knowledge Engineering 10(1993), S. 187-223.

[Gal81] H. Gallaire: *Impacts of logic on databases*, Proc. of the 7th Internat. VLDB Conference, Cannes, 1981.

[Gar83] H. Garcia-Molina: *Using semantic knowledge for transaction processing in a distributed database*, ACM Transactions on Database Systems 8(1983) 2, S. 186-213.

[GD93] S. Gatziu, K.R. Dittrich: *Events in an Active Object-Oriented Database System*, Proc. of the 1st Internat. Workshop on Rules in Database Systems (RIDS), Edinburgh, Aug. 1993, Springer-Verlag.

[GD94] S. Gatziu, K.R. Dittrich: *Detecting Composite Events in an Active Database System Using Petri Nets*, Proc. of the Internat. Workshop on Research Issues in Data Engineering - Active Database Systems (RIDE-ADB), Houston, 1994.

[GJ91] N.H. Gehani, H.V. Jagadish: *Ode as an Active Database: Constraints and Triggers*, Proc. of the 17th Internat. VLDB Conference, Barcelona, 1991.

[GJS92] N.H. Gehani, H.V. Jagadish, O. Shmueli: *Composite Event Specification in Active Databases: Model & Implementation*, Proc. of the 18th Internat. VLDB Conference, Vancouver, 1992.

[GK88] H. Garcia-Molina, B. Kogan: *Node Autonomy in Distributed Systems*, Proc. of the 1st Internat. Symposium on Databases in Parallel and Distributed Systems, IEEE CH2665, Austin, 1988, S. 158-166.

[GMB+81] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, G. Putzolu, I. Traiger: *The Recovery Manager of the System R Database Manager*, ACM Computing Surveys 13(1981) 2, S. 223-242.

[GMS94] C.H. Goh, S.E. Madnick, M.D. Siegel: *Context Interchange: Overcoming the Challenges of Large-Scale Interoperable Database Systems in a Dynamic Environment*, Proc. of the 3rd Internat. Conference on Information and Knowledge Management (CIKM), Maryland, 1994.

[GP86] V. Gligor, R. Popescu-Zeletin: *Transaction Management in distributed heterogeneous database management systems*, Information Systems 11(1986) 4, S. 287-297.

[Gra81] J. Gray: *The Transaction Concept: Virtues and Limitations*, Proc. of the 7th Internat. VLDB Conference, Cannes, 1981.

[GR93] J. Gray, A. Reuter: *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publ., 1993.

[GSC95] M. García-Solaco, F. Saltor, M. Castellanos: *Semantic Heterogeneity in Multidatabase Systems*, in [BE95], S. 129-202.

[GSF+95] G. Gardarin, S. Gannouni, B. Finance, P. Fankhauser, W. Klas, D. Pastrea, R. Legoff, A. Ramfos: *IRO-DB: A Distributed System Federating Object and Relational Databases*, in [BE95], S. 684 - 709.

[GW93] A. Gupta, J. Widom: *Local Verification of Global Integrity Constraints in Distributed Databases*, Proc. of the ACM SIGMOD Internat. Conference on Management of Data, Washington, 1993.

[GW96] P. Grefen, J. Widom: *Integrity Constraint Checking in Federated Databases*, Proc. of the 1st Internat. Conference on Cooperative Information Systems (CoopIS), Brüssel, 1996.

[Hab96a] K. Haberhauer: *Entwicklung einer temporalen Komponente in einem aktiven Vermittlersystem für heterogene Datenbanken*, Diplomarbeit, TH Darmstadt, FG DVS1, 1996.

[Hab96b] K. Haberhauer: *Asynchrone Replikation in einem aktiven Vermittlersystem*, Diplomarbeit, TH Darmstadt, FG DVS1, 1996.

[HHM+96] J.-L. Hainaut, J. Henrard, J.-M. Hick, D. Roland, V. Englebert: *Database Design Recovery*, Proc. of the 8th Internat. Conference, CAiSE'96, Heraklion, 1996, Springer-Verlag.

[HL81] M. Hammer, D. McLeod: *Database Description with SDM: A semantic database model*, ACM Transactions on Database Systems 6(1981) 3, S. 351-386.

[HL96] S.-M. Huang, J.-K. Liu: *Using active rules to maintain the data consistency in heterogeneous database systems*, Proc. of the 7th Internat. Hong Kong Computer Society Database Workshop, 1996.

[HLM88] M. Hsu, R. Ladin, D. McCarthy: *An Execution Model for Active Data Base Management Systems*, Proc. of the 3rd Internat. Conference on Data and Knowledge Bases, Jerusalem, 1988.

[HLW94] U. Hohenstein, R. Laufer, P. Weikert: *Object-Oriented Database Systems: How Much SQL Do They Understand?*, Proc. of the 5th Internat. Conference DEXA '94, Athen, 1994.

[HM85] D. Heimbigner, D. McLeod: *A Federated Architecture for Information Management*, ACM Transactions on Office Information Systems, 3(1985) 3, S. 253-278.

[HNS93] D.K. Hsiao, E.J. Neuhold, R. Sacks-Davis (Eds.): *Interoperable Database Systems (DS-5)*, Proceedings, IFIP WG 2.6 Conference on Semantics of Interoperable Database Systems, Elsevier Science Publ. B.V. , North Holland, Amsterdam, 1993.

[HTW95] W. Hahn, F. Toenniessen, A. Wittkowski: *Eine objektorientierte Zugriffsschicht zu relationalen Datenbanken*, Informatik-Spektrum, 18(1995) 3, S. 143-151.

[Hüs95] F. Hüsemann: *Konzepte der Datenintegration - Eine Bestandsaufnahme*, Techn. Bericht Math/95/19, Friedrich-Schiller-Universität Jena, Dez. 1995.

[HW93] E. Hanson, J. Widom: *An overview of production rules in database systems*, The Knowledge Engineering Review, 8(1993) 2, S. 121-143.

[HZ88] S. Heiler, S. Zdonik: *FUGUE: A Model for Engineering Information Systems and Other Baroque Applications*, Proc. of the 3rd Internat. Conference on Data and Knowledge Bases, Jerusalem, 1988.

[HZ90] S. Heiler, S. Zdonik: *Object Views: Extending the Vision*, Proc. of the 6th Internat. Conference on Data Engineering, Los Angeles, 1990.

[ILO96] o. Verf.: *ILOG World Wide Web Home Page*, http://www.ilog.fr/, 1996.

[Inf94] o. Verf.: *INFORMIX-Online Dynamic Server, Administrator's Guide*, Vol. 1, 1994.

[Inm96] W.H. Inmon: *Building the Data Warehouse*, 2nd Edition, John Wiley & Sons, 1996.

[ISO92] Subcommittee 4 of ISO Technical Committee 184: *Product Data Representation and Exchange - Part 11: The EXPRESS Language Reference Manual*, ISO Document, ISO DIS 10303-11, Aug. 1992.

[ISO95] ISO-ANSI: *Working Draft Database Language SQL (SQL / Foundation SQL3), Part 2, X3H2-94-080 and SOU-003*, 1995.

[Jab90] S. Jablonski: *Datenverwaltung in verteilten Systemen*, Informatik-Fachberichte 233, Springer-Verlag, Berlin, 1990.

[Jas97] R. Jaspert: *Ein Sybase Data Dictionary Reader für Persistence*, Studienarbeit, TH Darmstadt, FG DVS1, 1997.

[JQ92] H.V. Jagadish, X. Qian: *Integrity Maintenance in an Object-Oriented Database*, Proc. of the 18th Internat. VLDB Conference, Vancouver, 1992.

[Kar95] G. Karabatis: *Management of Interdependent Data in a Multidatabase Environment: A Polytransaction Approach*, Dissertation, University of Houston, 1995.

[KC95] J. Kiernan, M. Carey: *Extending SQL-92 for OODB Access: Design and Implementation Experience*, Proc. of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), Austin, 1995.

[KDN90] M. Kaul, K. Drosten, E. Neuhold: *ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views*, Proc. of the 6th Internat. Conference on Data Engineering, Los Angeles, 1990.

[KLB96] T. Kudrass, M. Lehmbach, A. Buchmann: *Tool-Based Re-Engineering of a Legacy MIS: An Experience Report*, Proc. of the 8th Internat. Conference, CAiSE'96, Heraklion, 1996, Springer-Verlag.

[KLB96a] T. Kudrass, A. Loew, A. Buchmann: *Active Object-relational Mediators*, Proc. of the 1st Internat. Conference on Cooperative Information Systems (CoopIS), Brüssel, 1996.

[KLM+91] A. Kemper, P.C. Lockemann, G. Moerkotte, H.-D. Walter, S.M. Lang: *Autonomy over Ubiquity: Coping with the Complexity of a Distributed World*, in H. Kangassalo (Ed.): *Entity-Relationship Approach: The Core of Conceptual Modeling*, Elsevier Science Publishers B.V., North-Holland, 1991.

[KLS92] W. Kim, Y.-J. Lee, J. Seo: *A Framework For Supporting Triggers in Object-Oriented Database Systems*, International Journal of Intelligent and Cooperative Information Systems 1(1992) 1, S. 127-143.

[Klu94] M. Klusch: *Towards a Federative Cell System FCSI for a context-based recognition of plausible Interdatabase Dependencies*, 6. GI-Workshop "Grundlagen von Datenbanken, Bad Helmstedt, 1994.

[Klu97] M. Klusch: *Rational kooperative Erkennung von Interdatenbankabhängigkeiten*, Dissertation, Uni Kiel, Institut für Informatik und Praktische Mathematik, 1997.

[KM94] A. Kemper, G. Moerkotte: *Object-Oriented Database Management - Applications in Engineering and Computer Science*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[KR87] B. Kähler, O. Risnes: *Extended Logging for Database Snapshot Refresh*, Proc. of the 13th Internat. VLDB Conference, Brighton, 1987.

[KRS93] G. Karabatis, M. Rusinkiewicz, A. Sheth: *Correctness and Enforcement of Multidatabase Interdependencies*, in: N. Adam, B. Bhargava (Eds.): *Advanced Database Systems*, LNCS 759, Springer-Verlag, 1993.

[Kra95] R. Krayer: *Entwicklung eines lokalen Datenbank-Gateways zur Unterstützung globaler Konsistenzkontrolle*, Diplomarbeit, TH Darmstadt, FG DVS1, 1995.

[KS88] H. Korth, G. Speegle: *Formal Models of Correctness without Serializability*, Proc. of the ACM SIGMOD Internat. Conference on Management of Data, Chicago, 1988.

[KS91] W. Kim, J. Seo: *Classifying Schematic and Data Heterogeneity in Multidatabase Systems*, IEEE Computer 24(1991) 12, S. 12-17.

[LA86] W. Litwin, A. Abdellatif: *Multidatabase Interoperability*, IEEE Computer 19(1986) 12, S. 10-18.

[LA87] W. Litwin, A. Abdellatif: A*n Overview of the Multi-Database Manipulation Language MDSL*, Proceedings of the IEEE 75(1987) 5, S. 621-624.

[LAC+93] M. Loomis, T. Atwood, R. Cattell, J. Duhland, G. Ferran, D. Wade: *The ODMG Object Model*, Journal of Object-Oriented Programming 6(1993) 3, S. 64-69.

[Lam78] L. Lamport: *Time, Clocks and the Ordering of Events in a Distributed System*, Communications of the ACM 21(1978) 7, S. 558-565.

[LHM+86] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, P. Wilms: *A Snapshot Differential Refresh Algorithm*, Proc. of the ACM SIGMOD Internat. Conference on Management of Data, Washington, 1986.

[Lit92] W. Litwin: *O*SQL: A Language for Multidatabase Interoperability*, in [HNS93], S. 119-137.

[LM93] Q. Li, D. McLeod: *Managing Interdependencies among Objects in Federated Databases*, in [HNS93], S. 331-347.

[LMB92] J. Levin, T. Mason, D. Brown: *lex & yacc*, O'Reilly & Associates, 1992.

[LMR90] W. Litwin, L. Mark, N. Roussopoulos: *Interoperability of Multiple Autonomous Databases*, ACM Computing Surveys 22(1990) 3, S. 267-293.

[Loe95] A. Loew: *Evaluierung des Datenbank-Integrationstools Persistence und Erprobung als aktives Vermittlersystem*, Diplomarbeit, TH Darmstadt, FG DVS1, 1995.

[LS80] B. Lindsay, P. Selinger: *Site Autonomy Issues in R*: A Distributed Database Management System*, IBM Research Report RJ2927, Sept. 1980.

[Mad95] S.E. Madnick: *From VLDB to VMLDB (Very MANY Large Data Bases): Dealing with Large-Scale Semantic Heterogeneity*, Proc. of the 21st Internat. VLDB Conference, Zürich, 1995.

[MB90] F. Manola, A. Buchmann: *A Functional/Relational Object-Oriented Model for Distributed Object Management: Preliminary Description*, TM-03331-11-90-165, GTE Laboratories, Dez. 1990.

[MHG+92] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, M. Brodie: *Distributed Object Management*, Internat. Journal of Intelligent and Cooperative Information Systems 1(1992) 1, S. 5-42.

[Mor84] M. Morgenstern: *Constraint Equations: Declarative Expression of Constraints With Automatic Enforcement*, Proc. of the 10th Internat. VLDB Conference, Singapur, 1984.

[Mrs96] M. M. Mrsic: *Ein Werkzeugkasten für globale Integritätskontrolle in heterogenen Datenbanken*, Diplomarbeit, TH Darmstadt, FG DVS1, 1996.

[Neu96] o. Verf.: *Neuron Data: Intelligent Rules Element*, WWW-Page: http://www.neurondata.com/products/ire.htm, 1996.

[ÖDV94] T. Özsu, U. Dayal, P. Valduriez (Eds.): *Distributed Object Management*, Morgan Kaufmann Publ., 1994.

[OK95] J. Orenstein, D. Kamber: *Accessing a Relational Database through an Object-Oriented Database Interface*, Proc. of the 21st Internat. VLDB Conference, Zürich, 1995.

[OMG91] Object Management Group: *The Common Object Request Broker: Architecture and Specification*, OMG Document, Revision 1.1., No. 91.12.1, Dez. 1991.

[OMG94] Object Management Group: *Common Object Services Specifications*, John Wiley, & Sons, Inc., 1994.

[Ous94] J.K. Ousterhout: *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, Mass., 1994.

[ÖV91] M.T. Özsu, P. Valduriez: *Principles of Distributed Database Systems*, Prentice Hall, 1991.

[Pap91] S. Pappe: *Datenbankzugriff in offenen Rechnernetzen*, Springer-Verlag, Reihe "Informationstechnik und Datenverarbeitung", 1991.

[PDW+93] N. Paton , O. Diaz, H. Williams, J. Campin, A. Dinn, A. Jaime: *Dimensions Of Active Behaviour*, Proc. of the 1st Internat. Workshop on Rules in Database Systems (RIDS), Edinburgh, Aug. 1993, Springer-Verlag.

[Per95a] *Persistence Reference Manual*, Release 3.0 (2 Bände), Persistence Software Inc., San Mateo, 1995.

[Per95b] *Persistence User Manual*, Release 3.0, Persistence Software Inc., San Mateo, 1995.

[PLS92] M. Papazoglou, S. Laufmann, T. Sellis: *An Organizational Framework For Cooperating Intelligent Information Systems*, Internat. Journal of Intelligent and Cooperative Information Systems 1(1992) 1, S. 169-202.

[PM88] J. Peckham, F. Maryanski: *Semantic Data Models*, ACM Computing Surveys 20(1988) 3, S. 153-189.

[PRV95] N. Pissinou, V. Raghavan, K. Vanapipat: *RIMM: A Reactive Integration Multidatabase Model*, Informatica 19(1995) 2, S. 177-193.

[PTR95] M. Papazoglou, Z. Tari, N. Russell: *Object-Oriented Technology for Interschema and Language Mappings*, in [BE95], S. 203-250.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[RC92] K. Ramamritham, P.K. Chrysanthis: *In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties*, Internat. Workshop on Distributed Object Management (DOM), in [ÖDV94], S. 212-230.

[Rec96] J. Reckziegel: *Schemakontrolle in einem aktiven Vermittlersystem für heterogene Datenbanken*, Diplomarbeit, TH Darmstadt, FG DVS1, 1996.

[Rei96] J. Reinert: *Ein Regelsystem zur Integritätssicherung in aktiven relationalen Datenbanksystemen*, Dissertation, Uni Kaiserslautern, infix-Verlag, 1996.

[Ril95] G. Riley: *What are Expert Systems, What is CLIPS*, CLIPS World Wide Web Home Page, http://www.jsc.nasa.gov/clips/CLIPS.html, 1995.

[Ris89] T. Risch: *Monitoring Database Objects*, Proc. of the 15th Internat. VLDB Conference, Amsterdam, 1989.

[RS94] A. Rosenthal, L. Seligman: *Data Integration in the Large: The Challenge of Reuse*, Proc. of the 20th Internat. VLDB Conference, Santiago, Chile, 1994.

[RSK91] M. Rusinkiewicz, A. Sheth, G. Karabatis: *Specification of Dependencies for the Management of Interdependent Data*, IEEE Computer 24(1991) 12, S. 46-53.

[Sch93] B. Schiefer: *Eine Umgebung zur Unterstützung von Schemaänderungen und Sichten in objektorientierten Datenbanksystemen*, Dissertation, Uni Karlsruhe, 1993.

[Sch96] S. Schwiderski: *Monitoring the Behaviour of Distributed Systems*, Dissertation, University of Cambridge, Technical Report No. 400, 1996.

[SCH+96] G. Saake, S. Conrad, M. Höding, S. Janssen, I. Schmitt, C. Türker: *Föderierung heterogener Datenbanksysteme und lokaler Datenhaltungskomponenten zur systemübergreifenden Integritätssicherung - Grundlagen und Ziele des Projektes* SIGMA_{FDB}, Preprint, Fakultät Maschinenbau, Universität Magdeburg, 1996.

[SDS95] S. Su, A. Doshi, L. Su: *HKBMS: An Integrated Heterogeneous Knowledge Base Management System*, in [BE95], S. 589-620.

[Ses96] R. Session: *Object Persistence*, Prentice Hall, 1996.

[SHP88] M. Stonebraker, E. Hanson, S. Potamianos: *The POSTGRES Rule Manager*, IEEE Transaction on Software Engineering 14(1988) 7, S. 897-907.

[SK93] A. Sheth, V. Kashyap: *So Far (Schematically) yet So Near (Semantically)*, (invited paper), in [HNS93], S. 283-312.

[SK93a] L. Seligman, L. Kerschberg: *Knowledge-base/Database Consistency in a Federated Multidatabase Environment*, Proc. of the Internat. Workshop on Research Issues in Data Engineering - Interoperability in Multidatabase Systems (RIDE-IMS), Wien, 1993.

[SKL89] S.Y.W. Su, V. Krishnamurthy, H. Lam: *An Object-Oriented Semantic Association Model (OSAM*)*, Chapter 17, in: S. Kumara, A. Soyster, R. Kashyap (Eds.): *Artificial Intelligence: Manufacturing Theory and Practice*, Industrial Engineering and Management Press, Norcross, Ga., 1989.

[SKM92] E. Simon, J. Kiernan, C. de Maindreville: *Implementing High Level Active Rules on top of a Relational DBMS*, Proc. of the 18th Internat. VLDB Conference, Vancouver, 1992.

[SKS91] N. Soparkar, H. Korth, A. Silberschatz: *Failure-Resilient Transaction Management in Multidatabases*, IEEE Computer 24(1991) 12, S. 28-36.

[SL90] A. Sheth, J.A. Larson: *Federated Database Systems for Managing Distributed Heterogeneous and Autonomous Databases*, ACM Computing Surveys 22(1990) 3, S. 183-236.

[SLA+95] S.Y.W. Su, H. Lam, J. Arroyo-Figueroa, T.-F. Yu, Z. Yang: *An Extensible Knowledge Base Management System for Supporting Rule-based Interoperability among Heterogeneous Systems*, Proc. of the Conference on Information and Knowledge Management (CIKM), Baltimore, 1995.

[SLY+96] S.Y.W. Su, H. Lam, T.-F. Yu, J. Arroyo-Figueroa, Z. Yang, S. Lee: *NCL: A Common Language for Achieving Rule-Based Interoperability among Heterogeneous Systems*, Journal of Intelligent Information Systems 6(1996) 2/3, S. 171-198.

[Soe96] M. Soeffky: *Replikationsmechanismen im Überblick*, Datenbank-Fokus, 9/96.

[SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, C. Mohan: *Alert: An architecture for transforming a passive DBMS into an active DBMS*, Proc. of the 17th Internat. VLDB Conference, Barcelona, 1991.

[SR90] A. Sheth, M. Rusinkiewicz: *Management of Interdependent Data: Specifying Dependency and Consistency Requirements*, Proc. of the Workshop on Management of Replicated Data, Houston, 1990.

[SRK92] A. Sheth, M. Rusinkiewicz, G. Karabatis: *Using Polytransactions to Manage Interdependent Data*, in [Elm92], S. 555-581.

[SSMR96] K. Smith, L. Seligman, D. Mattox, A. Rosenthal: *Distributed Situation Monitoring: Issues and Architectures*, Proc. of the SIGMOD Workshop on Materialized Views, Techniques and Applications, Montreal, 1996.

[SSR94] E. Sciore, M. Siegel, A. Rosenthal: *Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems*, ACM Transactions on Database Systems 19(1994) 2, S. 254-290.

[Ste90] W. R. Stevens: *UNIX Network Programming*, Englewood Cliffs, NJ, 1990.

[Sto96] M. Stonebraker: *Object-Relational Databases: The Next Great Wave*, Morgan Kaufmann Publ., 1996.

[Sub95] o. Verf.: *Subtleware Database Technology Connectivity: World Wide Web Home Page*, http://world.std.com/~subtle/info.html, Subtle Software, Inc., 1995.

[Sun87] Sun Microsystems, *RFC 1014 - XDR: External Data Representation Standard*, 1987.

[Uni95] o. Verf.: *UniSQL's Object-Relational Data Management Technology*, Enterprise Reengineering Product Profile, The Bowen Group, Ferndale, WA, 1995.

[USR+94] S. Urban, J. Shah, M. Rogers, D.K. Jeon, P. Ravi, P. Bliznakov: *A Heterogeneous, Active Database Architecture for Engineering Data Management*, Journal of Computer Integrated Manufacturing 7(1994) 5, S. 276-293.

[VA96] M. Vermeer, P. Apers: *The role of integrity constraints in database interoperation*, Proc. of the 22nd Internat. VLDB Conference, Mumbai, 1996.

[VEH92] J. Veijalainen, F. Eliassen, B. Holtkamp: *The S-transaction model*, in [Elm92], S. 467-513.

[Vei90] J. Veijalainen: *Transaction Concepts in Autonomous Database Environments*, GMD-Bericht Nr. 183, R. Oldenbourg Verlag, 1990.

[VM92] J. Veijalainen, P. Muth: *Transaction Management Issues in Autonomous Environments*, Arbeitspapiere der GMD, Nr. 691, 1992.

[Vos94] G. Vossen: *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme*, 2. Auflage, Addison Wesley Publ., 1994.

[VPR95] K. Vanapipat, N. Pissinou, V. Raghavan: *A Dynamic Framework to Actively Support Interoperability in Multidatabase Systems*, Proc. of the Internat. Workshop on Research Issues in Data Engineering - Distributed Object Management (RIDE-DOM), Taipeh, Taiwan, 1995.

[WC96] J. Widom, S. Ceri: *Active Database Systems: Triggers and Rules for Advanced Database Processing*, Morgan Kaufmann Publ., 1996.

[Wed89] H. Wedekind: *Eine logische Analyse des Verhältnisses von Anwendungs- und Datenbanksystemen*, in: Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), GI-Fachtagung, Zürich, 1989, Informatik-Fachberichte 204, Springer-Verlag.

[WF90] J. Widom, S.J. Finkelstein: *Set-Oriented Production Rules in Relational Database Systems*, Proc. of the ACM SIGMOD Internat. Conference on Management of Data, Atlantic City, 1990.

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

[Wie96] G. Wiederhold: *Value-added Mediation in Large-Scale Information Systems*, Proc. of the IFIP DS-6 Conference, Atlanta, in R. Meersman (Ed.): *Database Application Semantics*, Chapman and Hall, 1995.

[Win95] P. Winsberg: *Legacy Code: Don't Bag It, Wrap It*, IEEE Datamation 15(1995) 5, S. 36-41.

[WM89] Y.R. Wang, S.E. Madnick: *The Inter-Database Instance Identification Problem in Integrating Autonomous Systems*, Proc. of the 5th Internat. Conference on Data Engineering, Los Angeles, 1989.

[WQ87] G. Wiederhold, X. Qian: *Modeling Asynchrony in Distributed Databases*, Proc. of the 3rd Internat. Conference on Data Engineering, Los Angeles, 1987.

[XO91] X/Open DTP: *Distributed Transaction Processing: Reference Model, The XA Specification*, Reading, Berkshire, England, X/Open Ltd., 1991.

[YL92] L.-L. Yan, T.-W. Ling: *Translating Relational Schema with Constraints into OODB Schema*, in [HNS93], S. 69-86.

[ZGHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, J. Widom: *View Maintenance in a Warehousing Environment*, Proc. of the ACM SIGMOD Internat. Conference on Management of Data, San José, 1995.

[Zim97] J. Zimmermann: *Das aktive Datenbanksystem REACH*, Dissertation (in Vorbereitung), TH Darmstadt, FG DVS1, 1997.

Lebenslauf

Name:	Thomas Kudraß
Geburtstag:	16. Oktober 1965 in Weimar
Eltern:	Franz und Marianne Kudraß, geb. Scholle
Staatsangehörigkeit:	deutsch

1972-80	Polytechnische Oberschule Weimar
1980-84	Erweiterte Oberschule (Gymnasium) Weimar; Abitur
1984-85	EDV-Org.-Assistent im Robotron Optima Büromaschinenwerk Erfurt; Ausbildung zum Facharbeiter für Datenverarbeitung
1985-90	Studium in der Fachrichtung “Informationsverarbeitung” (Informatik) an der Technischen Universität Dresden; Abschluß als Dipl.-Ing. für Informationsverarbeitung
1990-91	Forschungsstudium an der Hochschule für Architektur und Bauwesen Weimar, Fakultät Informatik und Mathematik
1991	Gastforschungsaufenthalt am Forschungszentrum Informatik (FZI) an der Universität Karlsruhe, Gruppe Datenbanksysteme
1992-97	Wissenschaftlicher Mitarbeiter an der Technischen Hochschule Darmstadt, Fachbereich Informatik, FG Datenverwaltungssysteme 1
1997-	Datenbank-Spezialist beim Schweizerischen Bankverein Basel, Abteilung IT Management & Standards