Generische Dienste für datenintensive Anwendungen: Iterationsabstraktion, Integritätsüberwachung, Fehlererholung

Diplomarbeit

vorgelegt
von
Claudia Niederée

Dezember 1992

Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Str. 30
2000 Hamburg 54

Inhaltsverzeichnis

1 Einleitung und Motivation **1**

1.1 Struktur der Arbeit 2

1.2 *Add-On* versus *Built-In* 4

1.2.1 Vergleich der Ansätze 5

1.2.2 Eine Bibliothek generischer Dienste 9

1.3 Anforderungen datenintensiver Anwendungen 10

2 Beispiele neuerer Datenmodellierungskonzepte **13**

2.1 Das *Object-Relationship*-Modell (ORM) 14

2.2 Objekte, Klassen und Klassenausprägungen 15

2.2.1 Klassen im *ORM* 15

2.2.2 Andere Ansätze 16

2.3 Beziehungen zwischen Klassen 16

2.3.1 Beziehungen im *ORM* 17

2.3.2 Andere Ansätze 17

2.4 Generalisierungshierarchien und Inklusionsbeziehungen 18

2.4.1 Inklusions- und Exklusionsbeziehungen im *ORM* 18

2.4.2 Andere Ansätze 19

2.5 Integritätsbedingungen und ihre Überwachung 19

2.5.1 Allgemeine Aspekte 19

2.5.2 Integritätsbedingungen im *ORM* 22

2.5.3 Andere Ansätze 26

3 Relevante Programmiersprachenkonzepte (*P-Quest*) **29**

3.1 *Quest*-Typkonzepte 30

3.1.1 Die Ebene der Werte 31

3.1.2 Die Ebene der Typen . . . 31
3.1.3 Die Ebene der Kinds . . . 32
3.2 Funktionen in *Quest* . . . 32
3.2.1 Einfache Funktionen . . . 32
3.2.2 Rekursive Funktionen . . . 33
3.2.3 Funktionen höherer Ordnung . . . 33
3.2.4 Polymorphe Funktionen . . . 33
3.3 Datentypen und Typoperatoren . . . 35
3.3.1 Funktionstypen . . . 35
3.3.2 Rekursive Datentypen . . . 36
3.3.3 Typoperatoren . . . 36
3.3.4 Abstrakte Datentypen . . . 37
3.4 Subtypisierung und Subtyppolymorphismus . . . 38
3.5 Imperative Programmierung . . . 40
3.5.1 Veränderbare Variablen . . . 40
3.5.2 Sequenzen und Schleifen . . . 41
3.5.3 Der Datentyp *Array* . . . 41
3.6 Module und Schnittstellen . . . 42
3.7 Ausnahmebehandlung . . . 43
3.8 Persistenz . . . 44
3.8.1 Dynamische Datentypen . . . 45
3.8.2 Der Objektspeicher von *P-Quest* . . . 46

4 Eine prototypische Bibliothek generischer Dienste 47
4.1 Eine offene Architektur für datenintensive Anwendungen . . . 48
4.1.1 Die Architektur des Prototyps . . . 49
4.1.2 Abstimmung auf Benutzergruppen . . . 52
4.2 Iterationsabstraktion durch Funktionen höherer Ordnung . . . 55
4.2.1 Iterationsabstraktion in einem *Add-On*-Ansatz . . . 55
4.2.2 Höhere Iterationsabstraktion . . . 57
4.2.3 Eine sequentielle Implementierung . . . 61
4.3 Transaktionsorientierte Fehlererholung . . . 64
4.3.1 Transaktionen in einem *Add-On*-Ansatz . . . 65

4.3.2 Aufbau der Transaktionen 68
4.3.3 Fehlererholung durch Verwaltung eines *Undo-Logs* 69
4.3.4 Aufgaben der Transaktionsverwaltung 70
4.3.5 Kompensierende und geschützte Operationen 74
4.3.6 Der Enwurf von geschützten Operationen 76
4.4 Transaktionsorientierte Integritätsüberwachung 84
4.4.1 Darstellung und Test von Integritätsbedingungen 85
4.4.2 Verwaltung von Integritätsbedingungen 86
4.4.3 Übergang zur transaktionsorientierten Integritätsüberwachung 88
4.4.4 Der Entwurf konsistenzerhaltender Operationen 89
4.4.5 Zuordnung der Integritätsbedingungen 99
4.5 Weitergehende Ansätze 100
4.5.1 Ein Transaktionsgenerator 100
4.5.2 Partielle Rücksetzbarkeit von Transaktionen 102
4.5.3 Geschachtelte Transaktionen 104
4.5.4 Aktive Komponenten 108

5 Schnittstellen für ein spezielles Datenmodell 111
5.1 Architektur der Implementierung 112
5.2 Klassen 114
5.2.1 Ein Typ für Klassen 115
5.2.2 Implementierungen 118
5.2.3 Geschützte Operationen und Integritätsüberwachung 122
5.2.4 Integritätsbedingungen auf Klassen 124
5.3 Beziehungen zwischen Klassen 133
5.3.1 Inklusions- und Exklusionsbeziehungen 133
5.3.2 Objektreferenzen und referentielle Integrität 137
5.4 Die explizite Darstellung von Beziehungen 141
5.4.1 Beziehungen und ihre Elemente 142
5.4.2 Implementierungen 145
5.4.3 Geschützte Operationen und Integritätsüberwachung 147
5.4.4 Integritätsbedingungen auf Beziehungen 147
5.5 Objekte 149

6 Auswertung **153**

6.1 Erfahrungen mit *P-Quest* 153

6.1.1 Positive Erfahrungen 154

6.1.2 Einschränkungen 155

6.2 Zusammenfassung der Ergebnisse 157

6.3 Ausblick 159

Verweis auf den Anhang **161**

Literaturverzeichnis **163**

Kapitel 1

Einleitung und Motivation

Mit der Verbreitung von EDV-Systemen im allgemeinen und Datenbanksystemen im besonderen ergeben sich immer mehr Gebiete in Industrie und Forschung, in denen der Einsatz von Datenbanken möglich und erforderlich wird. Aus den neuen Anwendungsbereichen entstehen jedoch häufig auch neue Anforderungen an die Datenbanksysteme in Form von zusätzlicher Funktionalität und/oder mächtigeren Datenstrukturen. So reichen z.B. die flachen Tupel des relationalen Modells [Cod70] für viele Modellierungsaufgaben nicht aus.

Auf diese Situation wird mit der Entwicklung neuer Datenmodelle (z.B. des objekt-orientierten Modells) und mit dem Entwurf von Datenbankprogrammiersprachen, die diese Datenmodelle implementieren und die zusätzlich benötigte Funktionalität zur Verfügung stellen, reagiert. Bei den traditionellen Datenbankprogrammiersprachen wie z.B. *DBPL* [SEM88] und *Galileo* [AGO88] handelt es sich um *Built-In*-Ansätze. Das Datenmodell, die Strukturen zur Verwaltung der Massendaten und die Funktionalität zur Unterstützung datenintensiver Anwendungen sind fest in das Datenbanksystem eingebaut (*built-in*). Dadurch ist eine systematische Erweiterung und dynamische Anpassung solcher Systeme an neue Anforderungen nicht möglich. Die Entwicklung neuer Datenbankprogrammiersprachen löst also das oben angesprochene Problem zwar jeweils für eine oder mehrere spezielle Klassen neuer Anwendungen, nicht aber für den allgemeinen Fall.

Die benötigte Erweiterbarkeit kann erreicht werden, indem man die Konzepte zur Unterstützung datenintensiver Anwendungen als Dienste in Form von Bibliotheken zur Verfügung stellt (*Add-On*-Ansatz), anstatt sie fest einzubauen. *Add-On*-Ansätze sind ein neues Gebiet der aktuellen Datenbankforschung [MS92].

In dieser Arbeit soll versucht werden, einen solchen *Add-On*-Ansatz in einer strikt typisierten Umgebung zu realisieren. Dazu soll eine prototypische Bibliothek entworfen werden, die Unterstützung für datenintensive Anwendungen wie Fehlererholung, Integritätsüberwachung und Iterationsabstraktion als generische Dienste zur Verfügung stellt. Ziel ist es dabei zu untersuchen, inwieweit ein solcher Ansatz realisierbar ist und an welchen Punkten prinzipielle Probleme auftreten. Der Schwerpunkt der Arbeit liegt auf dem Entwurf der Schnittstellen der Bibliothek. Es wird aber auch eine prototypische Implementierung erstellt. Damit soll zum einen die Realisierbarkeit eines solchen Ansatzes gezeigt werden. Zum anderen soll untersucht werden, welche Anforderungen die Implementierung an die verwendete Sprache stellt. Für die Implementierung einer solchen Bibliothek generischer Dienste ist nicht jede Program-

miersprache geeignet. Es wird eine moderne Programmiersprache benötigt, die Konzepte wie Polymorphismus und Funktionen höherer Ordnung unterstützt. In dieser Arbeit wird für die Realisierung der Bibliothek die strikt typisierte, polymorphe Programmiersprache *P-Quest*, eine Erweiterung der Sprache *Quest* [Car89], verwendet.

In der durch die Dienste geschaffenen Umgebung für datenintensive Anwendungen können verschiedene Datenmodelle implementiert werden. Die Schnittstellen, die das Datenmodell realisieren, können dann ebenfalls als Teil der Bibliothek zur Verfügung gestellt werden. In der Arbeit soll exemplarisch ein Datenmodell, das neuere Datenmodellierungskonzepte beinhaltet, realisiert werden. Zu diesem Zweck wird das *Object-Relationship*-Modell [AGO91b, Rum87] gewählt.

1.1 Struktur der Arbeit

Der Rest dieses Kapitels ist wie folgt aufgebaut: In Abschnitt 1.2 wird der *Add-On*-Ansatz mit dem traditionellen *Built-In*-Ansatz verglichen. Dabei wird auch auf die Anforderungen eingegangen, die eine Bibliothek generischer Dienste bei einem *Add-On*-Ansatz zur adäquaten Unterstützung von Datenbank-Anwendungen erfüllen muß. Der letzte Abschnitt enthält eine kurze Charakterisierung datenintensiver Anwendungen. Sie ist der Arbeit [MN91] entnommen und als grobe Leitlinie für die Zusammenstellung der Bibliothek gedacht.

Die Bibliothek generischer Dienste soll eine Umgebung zur Verfügung stellen, die in ihrer Mächtigkeit den Anforderungen moderner Datenmodellierungskonzepte entspricht. In Kapitel 2 werden deshalb einige Beispiele moderner Datenmodellierungskonzepte untersucht. Der Schwerpunkt liegt dabei auf einer in [AGO91b] vorgeschlagenen Implementierung des *Object-Relationship*-Modells, da dieses Modell im Rahmen der Arbeit als *Add-On*-Ansatz in der Umgebung der generischen Dienste implementiert wird (siehe Kapitel 5).

Zur Implementierung einer solchen Bibliothek generischer Dienste wird eine moderne Programmiersprache benötigt, die Konzepte wie Polymorphismus und Funktionen höherer Ordnung zur Verfügung stellt. In dieser Arbeit wird dafür die Sprache *P-Quest* gewählt. Kapitel 3 gibt eine Einführung in die relevanten Konzepte dieser Sprache.

Kapitel 2 und 3 dienen der Einführung in für diese Arbeit relevante Modellierungs- und Programmiersprachenkonzepte, für die in der Fachliteratur keine adäquate einführende Übersicht existiert. Ein Teil dieser Konzepte sind für längerfristige Forschungsprojekte, an denen der Arbeitsbereich DBIS beteiligt ist (z.B. FIDE[1], GIF[2]) von Bedeutung. Aus diesen Gründen findet sich in der Arbeit eine etwas ausführlichere Darstellung dieser Inhalte. Leser, die bereits mit diesen Konzepten vertraut sind, können die Kapitel 2 und 3 überspringen und sich direkt Kapitel 4 und 5, dem eigentlichen Kern der Arbeit, zuwenden (siehe Abbildung 1.1).

In Kapitel 4 wird die Implementierung einer prototypischen Bibliothek generischer Dienste zur Unterstützung datenintensiver Anwendungen beschrieben. Nach einem Überblick über die Architektur werden die einzelnen realisierten Konzepte behandelt: Iterationsabstraktion, transaktionsorientierte Fehlererholung und transaktionsorientierte Integritätsüberwachung. Es handelt sich dabei um eine prototypische Implementierung, und der Schwerpunkt liegt

[1] FIDE = Formally Integrated Database Environments, ESPRIT BRA Project 3070
[2] GIF = German Israelian Foundation

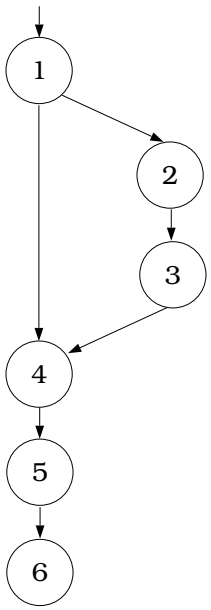

Abbildung 1.1: Vorschläge für die Lesereihenfolge

auf dem Entwurf der Schnittstellen. Neben den tatsächlich realisierten Ansätzen werden stets auch alternative Lösungsmöglichkeiten aufgezeigt. Am Ende des Kapitels wird auf weitergehende Ansätze wie z.B. Transaktionsgeneratoren und geschachtelte Transaktionen eingegangen. Diese sind zum Teil in der Bibliothek realisiert, zum Teil werden aber auch nur Anregungen zu ihrer Implementierung gegeben.

Kapitel 5 beschreibt die exemplarische Implementierung eines speziellen Datenmodells in der durch die Dienste zur Verfügung gestellten Umgebung. Dafür wird das *Object-Relationship*-Modell verwendet. Die entworfenen Schnittstellen zur Implementierung des Datenmodells werden anhand eines Beispiels veranschaulicht. Im Rahmen der Implementierung wird die Realisierbarkeit moderner Datenmodellierungskonzepte, wie Objektorientierung, Klassenkonzept und Generalisierungshierarchien, als *Add-On*-Ansatz untersucht.

Die Ergebnisse und die Erfahrungen, die bei der Implementierung der Bibliothek gemacht worden sind, werden in Kapitel 6 zusammengefaßt. Dabei wird unter anderem auf die Erfahrungen mit der Sprache *P-Quest* eingegangen. Außerdem wird ein Ausblick auf mögliche Ergänzungen und Erweiterungen des vorgestellten Ansatzes gegeben.

Die Verwendung der Schnittstellen der erstellten Bibliothek einschließlich des implementierten Datenmodells wird in Anhang A anhand eines Datenmodellierungsbeispiels veranschaulicht. Anhang B enthält ausgewählte Schnittstellen der implementierten Bibliothek generischer Dienste.

1.2 *Add-On* versus *Built-In*

Datenbankfunktionalität kann auf zwei konzeptuell verschiedene Weisen zur Verfügung gestellt werden: Sie kann in eine Programmiersprache eingebettet oder eingebaut werden (*built-in*), oder sie kann in Form von Bibliotheken zum System hinzugefügt werden (*add-on*).

Von zentraler Bedeutung im Datenbank-Bereich sind die Strukturen zur Verwaltung von Massendaten. Solche Strukturen werden in dieser Arbeit mit den üblichen Kollektionstypen wie Mengen und Listen unter dem Oberbegriff *Bulk-Typen* zusammengefaßt. Es wird dabei die Definition von Atkinson, Richard und Trinder [ART90] für Bulk-Typen zugrunde gelegt: Bulk-Typen sind homogene Strukturen von Elementen. Die Homogenität kann dabei darin bestehen, daß alle Elemente den gleichen Typ haben oder alle Subtyp eines bestimmten Typs sind. Die Ausprägungen von Bulk-Typen sind von variabler Größe und nicht in der Kardinalität beschränkt. Sie stellen meist veränderbare Werte dar, d.h., Einfügen und Löschen von Elementen ist möglich. Der Elementtyp eines Bulk-Typs ist beliebig (Typvollständigkeit).

In [MS92] werden die beiden Ansätze (*Built-In* und *Add-On*) gegenübergestellt. Der Schwerpunkt liegt dabei auf den Bulk-Typen. Die Gegenüberstellung wird hier auch auf andere Konzepte, die zur Unterstützung datenintensiver Anwendungen dienen, erweitert.

Bei einem *Built-In*-Ansatz sind die Konzepte zur Unterstützung datenintensiver Anwendungen, die von der jeweiligen Sprache bzw. dem jeweiligen System zur Verfügung gestellt werden, fest eingebaut. Typische Vertreter für *Built-In*-Ansätze sind traditionelle Datenbankprogrammiersprachen (*DBPL* [SEM88], *Galileo* [AGOO88]). Bei ihnen ist die Datenbankfunktionalität in die Programmiersprache integriert. Dabei wird meist von einer allgemeinen Programmiersprache ausgegangen, die um folgende Konzepte erweitert wird:

- einen oder mehrere Bulk-Typen, in denen die bei datenintensiven Anwendungen auftretenden Massendaten verwaltet und persistent gespeichert werden können,
- grundlegende Operationen (wie Einfügen und Löschen) auf diesen Bulk-Typen,
- deklarative Konstrukte zur Iterationsabstraktion, die die Iteration über die Elemente des (der) neu eingeführten Bulk-Typs(en) erlauben,
- und deklarative Konstrukte für den assoziativen Zugriff auf Elemente.

Ein Beispiel hierfür ist die Datenbankprogrammiersprache *DBPL* [SEM88]. Sie ist als Erweiterung der Sprache Modula-2 [Wir85] um Relationen mit Schlüssel und um deklarative Konstrukte zur Iteration und für den Zugriff auf Elemente entstanden.

Eine Datenbankprogrammiersprache ist meist in ein System eingebettet, das weitere Unterstützung für datenintensive Anwendungen zur Verfügung stellt, wie z.B. eine Transaktionsverwaltung, ein Persistenzkonzept, Unterstützung für den Mehrbenutzerbetrieb und Anfrageoptimierung.

Ein großer Nachteil der *Built-In*-Ansätze ist ihr Mangel an systematischer Erweiterbarkeit und Flexibilität. Das System kann neuen Anforderungen an seine Funktionalität nicht angepaßt werden, da die Konzepte zur Unterstützung datenintensiver Anwendungen fest eingebaut sind. Die gewünschte Erweiterbarkeit kann erreicht werden, wenn die Konzepte als Dienste in Form von Bibliotheken zur Verfügung gestellt werden (*Add-On*-Ansatz), anstatt sie fest einzubauen.

Eine solche Dienstbibliothek kann als Baukasten für Datenbanksysteme betrachtet werden. Die Schnittstellen der Bibliothek können wie Bausteine nach Bedarf zu dem System mit der benötigten Funktionalität kombiniert werden. Man erhält auf diese Weise:

Erweiterbarkeit: Die Funktionalität des Systems kann durch Hinzufügen neuer Schnittstellen erweitert werden.

Flexibilität: Die Funktionalität kann durch die Spezialisierung von Schnittstellen speziellen Anforderungen angepaßt werden.

Skalierbarkeit: Es brauchen nur die Schnittstellen in die Anwendung eingebunden zu werden, deren Funktionalität benötigt wird.

Dies setzt voraus, daß die Dienste in generischer Form angeboten werden und frei miteinander kombinierbar sind.

Im ersten Teil dieses Abschnitts werden die beiden Ansätze (*Add-On* und *Built-In*) miteinander verglichen. Bei diesem Vergleich ergeben sich Anforderungen, die eine Bibliothek generischer Dienste erfüllen sollte. Diese werden im zweiten Teil des Abschnitts zusammengefaßt. Die Implementierung einer solchen Bibliothek ist nicht in jeder Programmiersprache möglich, da sie gewisse Anforderungen an die verwendete Sprache stellt. Auf dieses Thema wird ebenfalls im zweiten Teil dieses Abschnitts eingegangen.

1.2.1 Vergleich der Ansätze

	Built-In	**Add-On**
Bulk-Typen	eingebaut, Teil des Systems	benutzerdefiniert, Teil der Bibliothek
Datenmodelle	ein festes Modell, Teil des Systems	mehrere Modelle möglich, Teil der Bibliothek
Funktionalität	modellabhängig	modellunabhängig
Erweiterbarkeit	schwierig	typsicher möglich
Wiederverwendbarkeit von Komponenten	sehr gering	hoch
Optimierung	beim Systementwurf	bei der Implementierung der Bibliothek
Skalierbarkeit	nicht möglich	einfach
Sicherheit	hoch	gering

Tabelle 1.1: *Add-On* versus *Built-In*

Die beiden Ansätze werden bezüglich der Aspekte Grobarchitektur, Realisierung von Datenmodellen, Flexibilität, Erweiterbarkeit, Effizienz, Sicherheit und Skalierbarkeit miteinander verglichen. Die Ergebnisse der Gegenüberstellung werden in Tabelle 1.1 zusammengefaßt. Auf die einzelnen Aspekte wird im folgenden genauer eingegangen.

Architektur

Abbildung 1.2 zeigt einen Vergleich zwischen den Grobarchitekturen eines *Built-In*- und eines *Add-On*-Ansatzes. Für den *Built-In* Ansatz wird das *DBPL*-System [SBK+88] zugrundegelegt. Die Darstellung der *Add-On*-Architektur orientiert sich an dem in dieser Arbeit beschriebenen Ansatz.

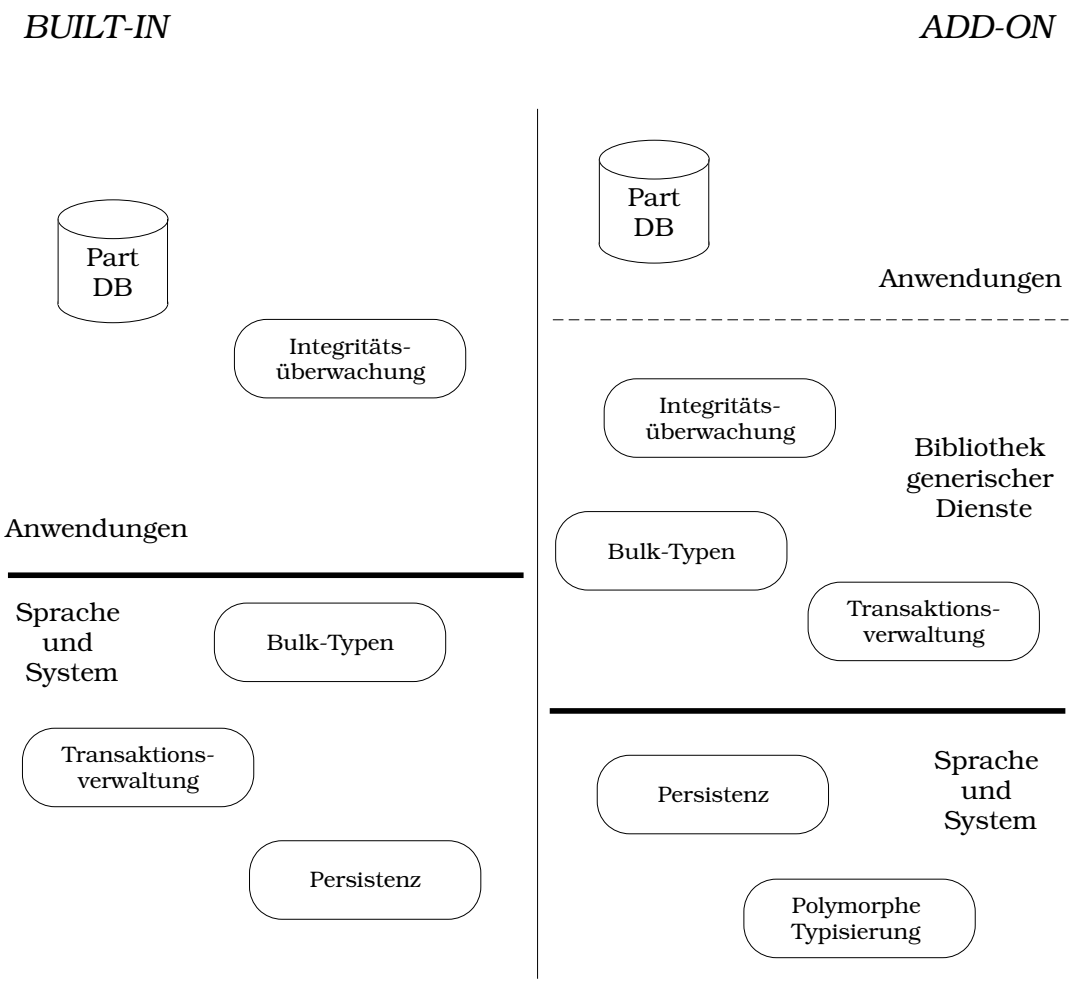

Abbildung 1.2: *Add-On* vs. *Built-In*

Bei dem *Built-In*-Ansatz gibt es eine klare Aufteilung in Sprache und System auf der einen Seite und Datenbank-Anwendungen auf der anderen Seite. Eine Reihe von Konzepten, wie z.B. bestimmte Bulk-Typen, ein Persistenzkonzept und eine Transaktionsverwaltung sind

fest in die Sprache bzw. in das System eingebaut. Die Integritätsüberwachung wird beim *DBPL*-Ansatz nicht vom System unterstützt. Es ist Teil der Anwendungsprogrammierung, die Wahrung von Integritätsbedingungen sicherzustellen.

Bei dem *Add-On*-Ansatz kann man drei Schichten unterscheiden. Die unterste Schicht wird von der Sprache und dem System gebildet. In dieser Arbeit wird die Sprache *P-Quest* verwendet. Sie ist durch die Einführung eines Persistenzkonzepts [Mü91] aus der Sprache *Quest* [Car89], einer strikt typisierten, polymorphen Programmiersprache, entstanden. Dies ist die einzige eingebaute Erweiterung der Sprache zur Unterstützung datenintensiver Anwendungen. Die zweite Schicht besteht aus einer Bibliothek generischer Dienste, die eine Umgebung für datenintensive Anwendungen zur Verfügung stellt. In dieser Umgebung können die Anwendungen erstellt werden, die die oberste Schicht bilden. Die Abgrenzung zwischen den beiden oberen Schichten ist in beiden Richtungen durchlässig:

- Schnittstellen und benutzerdefinierte Bulk-Typen, die für eine größere Benutzergemeinde von Interesse sind, können in die Dienstbibliothek aufgenommen werden.
- Schnittstellen, die als Dienste angeboten werden, können für konkrete Anwendungen spezialisiert und damit Teil der Anwendung werden.

Dies ist möglich, da die Anwendungsprogrammierung und die Programmierung der generischen Dienste im selben sprachlichen Rahmen durchgeführt werden. Die strikte Trennung zwischen der Programmierung eines Datenbanksystems und der Anwendungsprogrammierung, die bei einem *Built-In*-Ansatz vorhanden ist, fällt also weg.

Datenmodelle

Datenbankprogrammiersprachen orientieren sich mehr oder weniger strikt an einem Datenmodell, so z.B. die Sprache *Galileo* [AGOO88] am objekt-orientierten Modell und die Sprachen *Modula/R* [KMP+83] und *DBPL* [SEM88] am relationalen Modell. Bei einem *Add-On*-Ansatz hingegen ist die Implementierung eines bestimmten Datenmodells Teil der generischen Bibliothek. Ein Datenmodell (bzw. seine Implementierung) wird in Form einer oder mehrer Schnittstellen zur Verfügung gestellt. Wenn die Dienste wie z.B. Transaktionsverwaltung und Integritätsüberwachung modellunabhängig implementiert sind, können in der durch die Dienste zur Verfügung gestellten Umgebung mehrere Datenmodelle implementiert werden. Dies ermöglicht insbesondere auch den Übergang zwischen verschiedenen Datenmodellen.

Flexibilität und Erweiterbarkeit

Bei einem *Built-In*-Ansatz muß zum Zeitpunkt des Sprachentwurfs eine Entscheidung über benötigte Strukturen für die Anwendungsprogrammierung wie z.B. Bulk-Typen getroffen werden. Es ist jedoch problematisch, wenn nicht sogar unmöglich, die diversen Anwendungen einer Datenbankprogrammiersprache bereits zur Entwurfszeit vorauszusehen, insbesondere, da sich die Art der Anwendungen mit der Zeit ändern kann und immer neue Anwendungen mit zusätzlichen Anforderungen an die Sprache hinzukommen. Dies erfordert eine systematische Erweiterbarkeit, wie sie ein *Built-In*-Ansatz wegen der fest eingebauten Konzepte nicht zur Verfügung stellen kann. Erweiterungen einer Datenbankprogrammiersprache oder eines

Datenbanksystems erfordern ein Vordringen in die "Tiefen" des Systems und können deshalb nur von Systemexperten durchgeführt werden. Sie sind meist mit erheblichem Zeitaufwand verbunden.

Erweiterbare Datenbanksysteme (wie z.B. *EXODUS* [CDF+86], *Starbust* [LH90] oder *Iris* [WLH90]) erlauben die Erweiterung eines Datenbanksystems um neue Konzepte. Für die Implementation von Erweiterungen muß man aber auf eine systemnähere Programmiersprache wie z.B. *C* oder *C++* zurückgreifen.

Im Gegensatz dazu kann man bei einem *Add-On*-Ansatz Erweiterungen in der selben Sprache durchführen, in der auch Anwendungsprogramme geschrieben werden. Durch Einfügen neuer Schnittstellen in die Bibliothek generischer Dienste oder durch die Modifikation oder Spezialisierung bereits existierender Schnittstellen können die Dienste speziellen Anforderungen angepaßt und auf neue Anforderungen eingestellt werden.

Um die Wiederverwendbarkeit von Komponenten sicherzustellen ist es wichtig, daß die Dienste in generischer Form zur Verfügung gestellt werden und gut miteinander kombinierbar sind. Dies erleichtert die Anpassung der Bibliothek an neue Anforderungen, da es genügt, die betroffenen Teile zu modifizieren.

Eine noch größere Flexibilität des Ansatzes erhält man, wenn man zuläßt, daß auch externe Dienste wie z.B. C-Programme in die Bibliothek eingebunden werden können. Auf diese Weise erhält man eine Architektur, die auch für externe Dienste offen ist.

Effizienz und Sicherheit

Wie in [MS92] für das *DBPL*-System gezeigt wird, sind Datenbanksysteme häufig sehr umfangreich. Dem Benutzer wird aber nur eine sehr schmale, wohldefinierte Schnittstelle zur Verfügung gestellt. Dies erhöht die Sicherheit des Systems, da durch den eingeschränkten Zugriff auf das System dieses auch weitgehend vor unzulässiger Benutzung und Veränderung geschützt werden kann. Dem Anwender bleibt durch diese Einschränkung aber auch (unnötigerweise) der Zugriff auf viele Komponenten verwehrt, was die Flexibilität des Systems verringert.

Die hohe Flexibilität und Erweiterbarkeit der Bibliothek bei einem *Add-On*-Ansatz bedingt eine geringere Sicherheit des auf diese Weise realisierten Datenbanksystems. Sicherheit vor unzulässiger Benutzung angebotener Strukturen kann jedoch durch die Verwendung von abstrakten Datentypen erreicht werden.

Da bei datenintensiven Anwendungen mit sehr großen Datenmengen gearbeitet wird, spielt die Optimierung eine wichtige Rolle, um die Effizienz des Systems sicherzustellen. Bei einem *Built-In*-Ansatz werden alle Funktionalitäten bereits zum Zeitpunkt des Systementwurfs festgelegt. Optimierungen können also bereits zu diesem Zeitpunkt vorgesehen werden. Da die Konzepte fest eingebaut sind und nicht verändert werden, ist es gerechtfertigt, einen gewissen Aufwand in die Entwicklung und Implementierung geeigneter Optimierungsstrategien zu investieren. Welche Optimierungstrategien geeignet sind, hängt allerdings in vielen Fällen auch von der Art der Anwendungen ab, für die die Sprache verwendet wird. Auch an dieser Stelle tritt also wieder die Forderung auf, das System (hier insbesondere die Optimierungsstrategien) sich ändernden Anforderungen anpassen zu können.

Bei einem *Add-On*-Ansatz werden die Dienste von einer Bibliothek angeboten, die sich mit der Zeit verändern kann. Um die Effizienz sicherzustellen, muß dafür gesorgt werden, daß die Bibliothek effizient implementiert wird. Geeignete Optimierungen müssen vom Programmierer der Bibliothek berücksichtigt werden. Dieser Ansatz hat jedoch den Vorteil, daß nicht schon beim Entwurf der Bibliothek die Optimierungsstrategien festgeschrieben werden müssen. Die verwendeten Optimierungsstrategien können vielmehr dynamisch den Erfahrungen und neuen Anforderungen angepaßt werden, indem zu den bestehenden Schnittstellen Module mit neuen Strategien implementiert werden.

Skalierbarkeit

Datenbanksysteme, die auf einem *Built-In*-Ansatz beruhen, sind, wie bereits erwähnt, umfangreich und beinhalten ein Paket von Funktionalitäten wie z.B. Unterstützung der Nebenläufigkeit für den Mehrbenutzerbetrieb, Optimierung, persistente Speicherung. Solche Systeme lassen sich jedoch nicht geeignet skalieren: Es ist nicht möglich, auf einzelne dieser Komponenten zu verzichten, wenn sie nicht benötigt werden. Das System kann nur in seiner ganzen Mächtigkeit verwendet werden.

Bei der Bibliothek generischer Dienste ist es möglich, daß zu einer Schnittstelle mehrere Implementierungen existieren. Diese können sich in der Mächtigkeit unterscheiden, z.B. einfache und optimierte Implementierungen und solche, die auch Aspekte der Nebenläufigkeit beachten. Der Benutzer kann zwischen den verschiedenen Implementierungen auswählen und ein System mit der von ihm benötigten Funktionalität und Mächtigkeit zusammenstellen (Skalierbarkeit). Er kann auch Komponenten des Systems ausblenden, indem er die entsprechenden Schnittstellen und Module nicht mit in seine Anwendung einbindet.

1.2.2 Eine Bibliothek generischer Dienste

Bei einem *Add-On*-Ansatz wird die Unterstützung für datenintensive Anwendungen durch eine Bibliothek generischer Dienste zur Verfügung gestellt. In diesem Abschnitt soll diskutiert werden, welche Anforderungen an eine solche Bibliothek zu stellen sind und welche Anforderungen die Implementierung einer solchen Bibliothek an die verwendete Sprache stellt.

Anforderungen an die Bibliothek

Aus den Betrachtungen des vorigen Abschnitts ergibt sich, daß die Dienste, die die Bibliothek anbietet, folgende Anforderungen erfüllen sollten: Sie sollten nicht an bestimmte Bulk-Typen gebunden sein, sondern vielmehr modellunabhängig, um die Implementierung neuer Datenmodelle in der gleichen Umgebung zu erlauben und eine hohe Wiederverwendbarkeit von Komponenten sicherzustellen. Außerdem sollten die Dienste gut miteinander kombinierbar und möglichst generisch sein, damit sie einfach an verschiedene Situationen angepaßt werden können.

Die konkrete Zusammenstellung der Dienste einer Bibliothek sollte sich an den Anforderungen der Anwendungen orientieren und kann sich mit der Zeit ändern. Es lassen sich jedoch bestimmte Klassen von Diensten bestimmen, die eine solche Bibliothek unterstützen sollte.

Zum einen werden Dienste benötigt, die eine Umgebung für datenintensive Anwendungen schaffen, wie z.B. Transaktionsverwaltung und Integritätsüberwachung. Zum anderen sollte sie aber auch Schnittstellen für grundlegende Bulk-Typen wie Listen und Mengen und für spezielle Bulk-Typen wie Klassen und Relationen zur Verfügung stellen. Diese können Teil der Implementation bestimmter Datenmodelle sein.

Anforderungen an die Sprache

Die Implementation einer Bibliothek generischer Dienste ist nicht in jeder Programmiersprache möglich. Es wird dafür eine ausreichend mächtige Sprache mit einem Persistenzkonzept benötigt: Die Sprache muß gewisse Konzepte zur Verfügung stellen, die die Definition generischer Dienste erlaubt. Ein wichtiger Aspekt ist hierbei die Möglichkeit, neue Bulk-Typen zu definieren. Ausgehend von einer typisierten Umgebung erfordert die Definition von Bulk-Typen mit beliebigem Elementtyp, daß die Sprache parametrischen Polymorphismus unterstützt. Um außerdem eine korrekte Verwendung des Bulk-Typs zu gewährleisten, sollte er als abstrakter Datentyp definiert werden können. Die Sprache sollte also auch Datenabstraktion als Konzept zur Verfügung stellen.

Funktionen höherer Ordnung ermöglichen es, Dienste in generischer, allgemeiner Form zur Verfügung zu stellen, da von anwendungsabhängigen Details abstrahiert werden kann. Diese können für die jeweilige Anwendung durch die als Parameter übergebene Funktion festgelegt werden.

Eine generelle Voraussetzung für die Erstellung von Bibliotheken ist das Vorhandensein eines Modulkonzepts. Dies ermöglicht den systematischen Aufbau aber auch die Erweiterbarkeit einer Bibliothek.

In dieser Arbeit wird zur Implementierung der Bibliothek generischer Dienste die Sprache *P-Quest* verwendet. Sie bietet neben parametrischem Polymorphismus und Funktionen höherer Ordnung noch eine Reihe weiterer Konzepte an. Eine kurze Einführung in die Sprache findet sich in Kapitel 3. Die Erfahrungen mit der Sprache *P-Quest* bei der Implementierung der Bibliothek werden im Abschnitt 6.1 zusammengefaßt.

1.3 Anforderungen datenintensiver Anwendungen

Bei der Zusammenstellung einer Bibliothek generischer Dienste ist zu beachten, daß auch solche Dienste ausgewählt werden, die typischerweise von datenintensiven Anwendungen benötigt werden. In [MN91] wird versucht, eine Charakterisierung datenintensiver Anwendungen zu erstellen. Dazu wird eine Sammlung repräsentativer Beispiele datenintensiver Anwendungen aus der Standardliteratur ([AB87], [SS77], [BDRZ83], [Deu90], [AGOO88], [BR84b]) untersucht, um typische Problemmuster zu isolieren und die verschiedenen vorgestellten Formalismen für deren Darstellung zu vergleichen. Die Ergebnisse dieser Studienarbeit werden in diesem Abschnitt zusammengefaßt und nachfolgend als Grundlage für den Entwurf einer Dienstbibliothek verwendet.

Persistenz und Massendaten

Es ist Aufgabe einer Datenbank, Massendaten persistent zu speichern. Strukturen zur Verwaltung und persistenten Speicherung großer Datenmengen spielen deshalb eine zentrale Rolle im Bereich der datenintensiven Anwendungen [ABM88, AB87]. Ein Datenbanksystem muß mindestens eine solche Struktur unterstützen.

Um die Flexibilität zu erhöhen, sollte das angebotene Persistenzkonzept möglichst orthogonal zum Typsystem sein, sodaß beliebig strukturierte Daten persistent gespeichert werden können.

Identifikation

Die Objekte bzw. Elemente einer Datenbank müssen eindeutig identifizierbar sein. Dabei kann auch bei objekt-orientierten Ansätzen nicht auf eine wertbasierte Identifizierung z.B. durch Schlüsselwerte verzichtet werden, da die Objekte durch ihre Identität[3] zwar für das System voneinander unterscheidbar sind, der Benutzer aber oft eine Identifizierung über Werte, wie z.B. durch Namen benötigt. Ein weiteres Problem der Identifizierung ergibt sich im Zusammenhang mit der Generalisierung: Ein Objekt muß auf verschiedenen Stufen der Generalisierungshierarchie als identisch erkennbar sein.

Abstraktionsmechanismen

Datenintensive Anwendungen stellen den Benutzer vor das Problem, große Mengen von Information strukturieren zu müssen. Abstraktionsmechanismen sind in dieser Situation sehr nützlich. Sie ermöglichen es, zunächst von (unwichtigen) Details zu abstrahieren und sich auf die Grobstruktur der Information zu konzentrieren. Die Standardabstraktionsmechanismen sind Klassifizierung, Aggregation und Generalisierung. Die Abstraktionsmechanismen sind unabhängig voneinander. Die zugehörigen Hierarchien sollen deshalb auch unabhängig voneinander und in beliebiger Reihenfolge aufgebaut werden können.

Typvollständigkeit und rekursive Strukturen

Um ein möglichst breites Spektrum an Anwendungen adäquat darstellen zu können, sind Bemühungen in Richtung Typvollständigkeit von großem Nutzen. Dazu gehören die freie Kombinierbarkeit von Typkonstruktoren, die Unterstützung verschiedener Konstrukte zur Darstellung von Kollektionen, wie z.B. Listen (geordnet, Duplikate möglich) und Mengen (ungeordnet, Duplikateliminierung), und die Kombination des objekt-orientierten Ansatzes mit dem klassischen Ansatz: Neben Objekten und Klassen sind auch Werte und Typen zugelassen.

Beim Auftreten von rekursiven Strukturen genügt es nicht, Konstrukte zu deren adäquaten Darstellung zur Verfügung zu stellen. Es sollte auch Unterstützung für deren Verwaltung (z.B. Sicherstellen von Zykelfreiheit, Traversieren usw.) angeboten werden.

[3] Beim objektorientierten Ansatz erhält jedes Objekt bei seiner Erzeugung vom System eine systemweit eindeutige Identität zugeordnet [KC86].

Dynamische Aspekte und Transaktionen

Objekte der Datenbank werden erst durch ihre statischen Eigenschaften **und** ihr Verhalten vollständig beschrieben. Es sollte deshalb neben dem Schemaentwurf auch der Entwurf der grundlegenden Operationen (Einfügen, Löschen und Ändern) zu jedem Objekt unterstützt werden.

Nicht jede logische Operation kann durch eine atomare physikalische Operation realisiert werden. Es wird deshalb ein Transaktionskonzept benötigt, um die Konsistenz der Datenbank auch beim Auftreten von Fehlern zu bewahren (keine Effekte von unvollständig ausgeführten Operationen).

Integritätsbedingungen

Integritätsbedingungen ergänzen die Beschreibung einer Anwendung, die durch ein Datenbankschema gegeben ist. Sie können verschiedene Granularität besitzen: Sie können sich auf einzelne Attribute, auf ganze Objekte oder auch auf ganze Mengen von Objekten beziehen. Da bestimmte Integritätsbedingungen nur von bestimmten Operationen verletzt werden können, bietet es sich an, die Integritätsbedingungen an Operationen zu binden. Die Reaktion auf die Verletzung einer Integritätsbedingung kann verschieden sein (z.B. Abbruch, Fehlermeldung, korrigierende Operationen).

Kapitel 2

Beispiele neuerer Datenmodellierungskonzepte

Der objekt-orientierte Ansatz findet als neues Programmier- und Modellierungsparadigma sowohl im Bereich der Programmiersprachen (*C++* [GOP90], *Eiffel* [Mey88], *Smalltalk* [GR83]) als auch im Bereich der Datenbankmodellierung und -programmierung [Hug91] eine breite Akzeptanz. Es existiert eine Reihe von Datenbankprogrammiersprachen und -systemen (O_2 [LRV88], *GemStone* [MJAP86], *Orion* [Hug91], OODAPLEX [Day89], *OPAL* [CM84]), denen dieser Ansatz zugrunde liegt. Ein besonders ausgereifter Ansatz im Bereich der Datenbankprogrammierung ist dabei die von Albano, Ghelli und Orsini in [AGO91b, AGO91a] vorgeschlagene Implementierung des *Object-Relationship*-Modells in einer streng typisierten Umgebung. Das *Object-Relationship*-Modell ist ein objekt-orientiertes Modell, das eine explizite Darstellung von Beziehungen unterstützt [Rum87]. Die vorgeschlagene Datenbankprogrammiersprache beinhaltet neben einem Konstrukt zur expliziten Darstellung von Beziehungen auch die Überwachung bestimmter Integritätsbedingungen auf Klassen und Beziehungen. Sie wird der Kürze halber im folgenden nur noch mit *ORM* bezeichnet.

Da es sich bei den objekt-orientierten Datenbanksprachen und -systemen um *Built-In*-Ansätze handelt, stellt sich das in Abschnitt 1.2 beschriebene Problem des Mangels an systematischer Erweiterbarkeit und Flexibilität. Auf neue Anforderungen wird mit der Entwicklung neuer Systeme und Sprachen reagiert, da eine Anpassung und Erweiterung bestehender Systeme nicht oder nur mit erheblichem Aufwand möglich ist. Die Autoren des *ORM* schlagen deshalb vor, zur Erhöhung der Flexibilität nicht die gesamte Funktionalität als reinen *Built-In*-Ansatz zu implementieren, also fest einzubauen. Es wird vielmehr von einem kleinen, kompakten Sprachkern ausgegangen, der gewisse Grundfunktionalitäten zur Verfügung stellt. Die spezielleren Konzepte werden konzeptuell auf Konzepte dieses Sprachkerns abgebildet.

Es ist in diesem Zusammenhang noch einmal darauf hinzuweisen, daß das *ORM* bis jetzt nur als Vorschlag für eine Datenbankprogrammiersprache existiert und noch nicht implementiert ist. In dieser Arbeit wird ein in Bezug auf die Funktionalität ähnlich mächtiger *Add-On*-Ansatz vorgestellt (siehe Kapitel 4 und 5), bei dem die Unterstützung für datenintensive Anwendungen in Form von einer Bibliothek generischer Dienste zur Verfügung gestellt wird. Die Abbildung auf grundlegende Konzepte der Sprache wird dabei systemtechnisch implementiert. Es wird versucht, mit den Konzepten auszukommen, die eine moderne Programmiersprache

mit einem mächtigen Typsystem zur Verfügung stellt (Polymorphismus, Ausnahmebehandlung etc.).

Beim Entwurf des *ORM* wurde sich an den Erfahrungen mit der Datenbanksprache *Galileo* [AGO89] und an der Sprache *Quest* [Car89] orientiert. Eine um ein Persistenzkonzept erweiterte Version *P-Quest* der Sprache *Quest* wird für die Implementierung der generischen Dienste in dieser Arbeit verwendet[1]. Aus diesem Grund eignet sich das *ORM* gut für einen Vergleich mit dem hier vorgestellten Ansatz einer Bibliothek generischer Dienste: Es wird beim Aufbau einer Umgebung für datenintensive Anwendungen von ähnlichen Grundbausteinen ausgegangen.

Im Rest des Kapitels werden folgende Aspekte der Datenmodellierung genauer untersucht: Der Zusammenhang zwischen Objekten, Klassen und Klassenausprägungen (Abschnitt 2.2), Beziehungen zwischen Klassen und ihre Darstellung (Abschnitt 2.3), der Aufbau von Generalisierungshierarchien auf Klassen (Abschnitt 2.4) und die Definition von Integritätsbedingungen und ihre Überwachung (Abschnitt 2.5). Zu jedem der Aspekte wird nach einer allgemeinen Einleitung die im *ORM* vorgeschlagene Lösung vorgestellt. Im zweiten Teil jedes Abschnitts werden Beispiele für alternative Ansätze beschrieben. In Abschnitt 2.1 wird zunächst genauer auf einige grundlegende Konzepte des *ORM* eingegangen.

2.1 Das *Object-Relationship*-Modell (ORM)

Die in [AGO91b] vorgeschlagene Datenbanksprache (*ORM*) besteht aus zwei Ebenen. Die untere Ebene wird von einem kleinen, kompakten Sprachkern gebildet, der nur einige wenige datenbankspezifische Erweiterungen anbietet. Die zweite Ebene ist die eigentliche Sprache, die dem Benutzer zur Verfügung steht. Sie bietet eine Reihe weiterer und speziellerer Konzepte für den Einsatz im Datenbankbereich an. Die Konzepte dieser Ebene werden auf Konzepte des Sprachkerns abgebildet.

Der Sprachkern von *ORM* bietet neben den üblichen Konzepten einer Programmiersprache folgende datenbankspezifischen Erweiterungen an: Im Mittelpunkt steht ein Konstrukt zur Definition von Beziehungen (*Associations*). Bei der Definition können eine Signatur und Schlüssel für die Beziehung festgelegt werden. Weiterhin existiert ein allgemeiner Mechanismus zur Überwachung/Wahrung von Integritätsbedingungen auf Beziehungen. Er beruht darauf, daß Operationen festgelegt werden können, die vor dem Einfügen in die bzw. Löschen aus der Beziehung ausgeführt werden. Auf diesen Mechanismus wird in Abschnitt 2.5.2 genauer eingegangen.

Der Sprachkern wird ergänzt durch ein Transaktionskonzept: Durch Voranstellen des Schlüsselworts *atomic* wird eine Anweisungsfolge atomar ausgeführt. Falls es während der Ausführung zu einem Fehler kommt, wird die Datenbank automatisch auf den Zustand vor der Transaktion zurückgestzt.

Die zweite Ebene der Sprache bietet eine Reihe weiterer benutzerfreundlicher Konzepte an. Darunter befinden sich unter anderem Konzepte zur Definition von Klassen und zur Definition von n-ären Beziehungen. Sowohl bei der Definition von Klassen als auch bei der Definition von Beziehungen können spezielle Integritätsbedingungen angegeben werden.

[1] Eine Einführung in die Sprache und weiterführende Literatur werden in Kapitel 3 vorgestellt.

Auf den Typen der Sprache ist eine Inklusionshierarchie definiert: Wenn ein Typ B Subtyp eines anderen Typs A ist, so enthält A jedes Element (jeden Wert) des Typs B. Ein weiterer wichtiger Begriff für den Vergleich von Typen ist die *Kompatibilität*, die wie folgt definiert wird: Zwei Typen sind *kompatibel*, wenn ein Typ V existiert, der Subtyp beider Typen ist.

In der vollen Sprache gibt es einen Typ *Seq* für Sequenzen. Für Werte dieses Typs ist ein Satz von Operatoren definiert, der den üblichen relationalen Operatoren ähnelt. Der Typ von Klassen und der Typ von Beziehungen sind Subtypen des Typs *Seq*. Somit können die auf diesem Typ definierten Operatoren auch auf Klassen und Beziehungen angewandt werden.

2.2 Objekte, Klassen und Klassenausprägungen

In diesem Abschnitt wird an einigen Beispielen untersucht, welche Rolle das Konzept der Klassen in verschiedenen Ansätzen spielt und in welcher Beziehung es zu dem Konzept der Objekte steht.

Bei einer Reihe von objekt-orientierten Ansätzen hat die Definition einer Klasse hauptsächlich die Funktion, die Struktur und die Methoden für ihre Objekte festzulegen. In diesen Fällen sind das Konzept der Klassen und das der Objekte untrennbar miteinander verbunden. Es gibt aber auch Ansätze, bei denen das Gewicht mehr auf der Verwaltung der Klassenausprägungen, d.h. der Kollektion der Elemente, die zu der Klasse gehören, liegt. Bei einigen dieser Ansätze werden Objekte bei der Erzeugung automatisch in die Klassenausprägung eingefügt. Andere stellen Funktionen zum expliziten Einfügen in die und zum Löschen aus den Ausprägungen zur Verfügung.

Im folgenden wird das Konzept der Klassen im *ORM*, im objekt-orientierten Datenmodell O_2 und in der Datenbankprogrammiersprache *Galileo* untersucht.

2.2.1 Klassen im *ORM*

Im *ORM* ist eine Klasse eine geordnete Menge unterschiedlicher Elemente gleichen Typs. Die Betonung liegt also auf dem Aspekt der Klassenausprägung. Es gibt Funktionen für das Einfügen in und das Löschen von Elementen aus der Klasse. Die Sprache bietet eine eigene Operation zur Definition neuer Klassen an. Bei der Definition einer Klasse wird ihr Elementtyp festgelegt und es können Inklusions- und Exklusionsbeziehungen und Eindeutigkeitsbedingungen (Schlüssel) für die Klasse angegeben werden. Außerdem kann man Operationen definieren, die vor dem Einfügen bzw. Löschen eines Elements ausgeführt werden. Auf diesen Mechanismus und auf die Integritätsbedingungen wird in Abschnitt 2.5.2 genauer eingegangen.

Im Sprachkern existiert kein eigenes Konzept für die Klassen. Klassen werden auf unäre Beziehungen abgebildet.

Klassausprägungen sind bei diesem Ansatz nicht notwendigerweise Kollektionen von Objekten. Wenn eine Klasse Objekte als Elemente enthalten soll, muß ein Objekttyp als Elementtyp angegeben werden[2].

[2]Der Aspekt der Objekte im *ORM* wird in [AGO91a] behandelt

2.2.2 Andere Ansätze

Auch im O_2-Datenmodell [LRV88, LR89] gibt es ein spezielles Konstrukt zur Erzeugung von Klassen. Als Elementtyp kann ein beliebiger O_2-Typ gewählt werden. Zur Erzeugung neuer Objekte einer Klasse gibt es einen Operator *new*, der ein Objekt mit einer eindeutigen Identität erzeugt. Der Benutzer entscheidet bei der Definition der Klasse, ob eine Menge mit dem Namen der Klasse angelegt werden soll, in die Objekte der Klasse bei ihrer Erzeugung automatisch eingefügt werden. Das hat vor allen Dingen Auswirkungen auf die Persistenz der Objekte. Neben einem Elementtyp wird einer Klasse auch noch eine Reihe von Methoden zugeordnet. Da O_2 ein objekt-orientiertes Datenmodell ist, können die Werte der Objekte einer Klasse nur über die Methoden ihrer Klasse zugegriffen und verändert werden.

Bei diesem Ansatz dient die Klasse hauptsächlich der Festlegung der Struktur und der Methoden ihrer Objekte. Es kann aber optional auch eine Klassenausprägung erzeugt werden. Die Konzepte von Objekt und Klasse sind fest miteinander verbunden.

Die Programmiersprache *Galileo* [AGOO88] stellt ein spezielles Konstrukt zur Erzeugung von Klassen zur Verfügung. Bei der Erzeugung einer Klasse wird ein neuer abstrakter Datentyp definiert, der als Elementtyp der Klasse dient. Eine Klasse steht zu jedem Zeitpunkt für die Kollektion der Elemente, die zu ihr gehören. Der Begriff der Klasse und der Klassenausprägung sind hier also fest miteinander verbunden. Bei der Erzeugung einer Klasse wird eine *make*-Befehl zur Verfügung gestellt, mit dem neue Objekte der Klasse erzeugt werden können. Der Befehl fügt das neue Objekt automatisch in die Klassenausprägung ein.

2.3 Beziehungen zwischen Klassen

Beziehungen zwischen Klassen oder genauer gesagt Beziehungen zwischen Elementen von Klassen werden in objekt-orientierten Ansätzen häufig dadurch dargestellt, daß Objekte Verweise auf andere Objekte enthalten. Im Zusammenhang mit diesre Darstellung wird im folgenden von *Objektreferenzen* gesprochen. Eine mögliche Realisierung für Objektreferenzen ist es, für Attribute des Objekttyps Klassen als Domänen zuzulassen. Auf diese Weise genügt das üblichen Konzepts der Aggregation zur ihrer Darstellung. Es wird kein eigenes Konzept benötigt.

In einigen Ansätzen werden Beziehungen zwischen Klassen als eigenes semantisches Konzept betrachtet und durch ein eigenes Konstrukt repräsentiert. In [Rum87] und [AGO91b] werden Gründe für die Einführung eines solchen speziellen Konstrukts angegeben:

- Beziehungen sind ein höheres Konzept vergleichbar mit Generalisierung und Klassifizierung. Die Implementierung sollte deshalb Sache des DBMS sein und nicht vom Programmierer abhängen.
- Eine Beziehung zwischen zwei oder mehr Klassen ist ein symmetrisches Konzept. Bei der Darstellung durch ein komplexes Objekt wird diese Symmetrie zugunsten einer Richtung aufgegeben.
- Wenn die Beziehung dabei trotzdem in beiden Richtungen dargestellt wird, führt dies zu einer Aufteilung der Semantik der Beziehung auf verschiedene Objekte. Es kann da-

durch außerdem zur Speicherung redundanter Information und zu Konsistenzproblemen kommen.

- Die Definition der an der Beziehung beteiligten Klassen sollte unabhängig von der Definition der Beziehung sein. Dies ist bei der Darstellung von Beziehungen durch komplexe Objekte in der Regel nicht der Fall. Man muß nämlich bei der Angabe des Elementtyps einer Klasse bereits die Bezüge auf andere Objekte berücksichtigen.

- Die Einführung eines eigenen Konstrukts ermöglicht die Definition von Operationen auf der Relation als Ganzes.

- Einer Beziehung sind evtl. zusätzliche Attribute zugeordnet. Außerdem gibt es nicht nur binäre Beziehungen. Diese beiden Aspekte lassen sich bei der Darstellung durch komplexe Objekte nicht adäquat repräsentieren.

2.3.1 Beziehungen im *ORM*

Im *ORM* existiert ein eigenes Konstrukt zur Darstellung n-ärer Beziehungen. Bei der Definition einer solchen Beziehung wird ein Elementtyp für die Beziehung festgelegt. Für beliebig viele der Komponenten des Elementtyps, also z.B. für eine Komponente *compA* kann man festlegen, daß sie nur Werte annehmen kann, die in einer bestimmten Klasse *A* enthalten sind. Macht man eine solche Angabe für *n* Komponenten des Elementtyps, so erhält man eine n-äre Beziehung. Für diese Komponenten wird also eine Art referentielle Integrität gefordert. Dabei besteht die Auswahl zwischen drei Varianten, die sich in der Reaktion auf die Verletzung der Bedingung unterscheiden. Der Typ von *compA* und der Elementtyp der Klasse *A* müssen dazu in einer bestimmten Beziehung zueinander stehen. Die Art der geforderten Beziehung hängt von der gewählten Variante ab. Auf dieses Thema wird in Abschnitt 2.5.2 genauer eingegangen. Auf Beziehungen sind wie bei den Klassen im *ORM* Einfüge- und Löschoperationen möglich. Außerdem ist für Beziehungen sowohl die Festlegung von Inklusions- und Exklusionsbeziehungen zu anderen Beziehungen als auch die Angabe weiterer Integritätsbedingungen möglich (siehe Abschnitt 2.4.1 und 2.5.2)

2.3.2 Andere Ansätze

In O_2 werden Beziehungen zwischen Klassen durch Objektreferenzen dargestellt. Zur Realisierung wird dabei die Aggregation verwendet: Eine Klasse hat einen Elementtyp. Es besteht die Möglichkeit, Klassen als Domänen für Komponenten dieses Elementtyps anzugeben.

In *Galileo* wird, wie bereits im Abschnitt über die Klassen erwähnt, zusammen mit der Klasse ein abstrakter Datentyp definiert. Alle Werte dieses Typs, die erzeugt werden, sind auch Objekte der Klasse und somit in der Klassenausprägung enthalten. Wenn man diesen ADT als Domäne eines Attributs angibt, kann das Attribut also nur Werte der Klasse annehmen. Auf diese Weise können in *Galileo* Referenzen auf Objekte realisiert werden, ohne ein spezielles Konstrukt dafür einzuführen.

Wenn beide Richtungen der Beziehung dargestellt werden sollen, kann man dafür das Konzept der *derived attributes* verwenden. Dabei wird bei einem Attribut statt des eigentlichen Werts

nur eine Vorschrift zu seiner Berechnung angegeben. Dies vermeidet die Speicherung von redundanter Information und die damit verbundene Gefahr von Inkonsistenzen.

2.4 Generalisierungshierarchien und Inklusionsbeziehungen

Für den Aufbau von Generalisierungshierarchien gibt es verschiedene Konzepte. Den meisten typisierten Ansätzen gemeinsam ist jedoch die Forderung, daß der Elementtyp einer Subklasse in Subtypbeziehung zum Elementtyp der Superklasse stehen soll. Dies ermöglicht insbesondere die Anwendung von Methoden einer Superklasse auf Elemente ihrer Subklassen.

In vielen objekt-orientierten Ansätzen liegt der Schwerpunkt beim Aufbau von Generalisierungshierarchien auf der Vererbung: Eine Subklasse erbt die Attribute (und Methoden) ihrer Superklasse oder Superklassen (bei Mehrfachvererbung). Für die Subklasse müssen nur noch zusätzliche und redefinierte Attribute angegeben werden und/oder ererbte redefiniert. Diese Vorgehensweise stellt sicher, daß der Elementtyp der Subklasse ein Subtyp des Elementtyps der Superklasse ist. Sie erfordert jedoch ein eingebautes Konzept der Vererbung.

Einige Ansätze kommen beim Aufbau von Generalisierungshierarchien ohne ein solches Konzept der Vererbung aus: Bei der Definition einer Subklasse wird deren voller Elementtyp angegeben. Das System übernimmt dann den Test, daß die Elementtypen von Sub- und Superklasse in Subtypbeziehung stehen.

Ein anderer Aspekt von Subklassenbeziehungen ist die Inklusionsbeziehung zwischen den Klassenausprägungen: Jedes Element einer Subklasse sollte auch in all seinen Superklassen enthalten sein. Neben Inklusionsbeziehungen gibt es auch Exklusionsbeziehungen zwischen Klassen, d.h., die Klassen haben keine gemeinsamen Elemente.

2.4.1 Inklusions- und Exklusionsbeziehungen im *ORM*

Bei der Definition einer Klasse A können Klassen $C_1 \ldots C_n$ angegeben werden, die direkte Superklassen von A sein sollen. Es muß gelten, daß die Elementtypen der angegebenen Klassen alle Supertypen des Elementtyps der neu definierten Klasse sind. Die Konsequenz einer solchen Angabe ist eine Inklusionsbeziehung zwischen der Klassenausprägung von A und den Klassenausprägungen der Klassen $C_1 \ldots C_n$, d.h., alle Elemente aus A sind auch in jeder der Klassen C_i enthalten.

Inklusionsbeziehungen werden im Sprachkern durch den Test geeigneter Integritätsbedingungen (Inklusionsbedingung) und die Ausführung von Triggerprozeduren im Fall der Verletzung einer Bedingung realisiert. In Abschnitt 2.5.2 wird genauer auf Inklusionsbedingungen und ihre Wahrung eingegangen.

Neben Superklassen können zu einer Klasse A bei ihrer Definition auch Klassen $B_1 \ldots B_m$ angegeben werden, mit denen sie in Exklusionsbeziehung steht. Klasse A darf dann mit keiner dieser Klassen gemeinsame Elemente haben. Für die Elementtypen der Klassen B_i muß gelten, daß sie alle kompatibel mit dem Elementtyp der Klasse A sind. Die Exklusionsbeziehung zwischen den Klassen wird durch den Test geeigneter Integritätsbedingungen (Exklusionsbedingungen) beim Einfügen in die Klassen A und B_i ($1 \leq \mathrm{i} \leq \mathrm{n}$) sichergestellt. Die Form dieser Bedingung wird in Abschnitt 2.5.2 behandelt.

Inklusions- und Exklusionsbeziehungen können auch bei der Definition von Beziehungen angegeben werden. Sie sind entsprechend realisiert.

2.4.2 Andere Ansätze

Wie bereits erwähnt steht bei den Klassen in O_2 die Rolle als Träger der Struktur und der Methoden im Mittelpunkt. Beim Aufbau von Generalisierungshierarchien wird das eingebaute Konzept der Vererbung verwendet. Man kann eine Klasse A als Subklasse einer oder mehrer anderer Klassen $C_1 \ldots C_n$ definieren. A erbt dann die Attribute und Methoden der Klassen C_i. Es können für A auch noch zusätzliche Attribute definiert oder ererbte redefiniert werden.

In *Galileo* kann eine Klasse B als Subklasse einer bereits existierenden Klasse A definiert werden. Die Klasse B erbt dann alle Attribute der Klasse A. Es können für die Klasse B auch noch zusätzliche Attribute definiert werden oder ererbte redefiniert.

Wie bereits erwähnt, wird durch die Definition einer Klasse eine Funktion zur Erzeugung neuer Objekte dieser Klasse generiert, die die Objekte auch gleich in die Klassenausprägung einfügt. Bei einer Subklasse B wird ein Element durch diese Operation nicht nur in die Ausprägung der Klasse B, sondern auch in die der Superklasse A eingefügt. Dadurch wird die Inklusionsbeziehung zwischen den Ausprägungen der Klassen A und B sichergestellt.

2.5 Integritätsbedingungen und ihre Überwachung

In einer Datenbank können Zusammenhänge zwischen Daten und Einschränkungen auf Daten, die nicht durch den Schemaentwurf erfaßt werden, durch Integritätsbedingungen ausgedrückt werden. Wenn alle Integritätsbedingungen erfüllt sind, ist die Datenbank in einem konsistenten Zustand. Ziel der Integritätsüberwachung ist es, die Datenbank in einem konsistenten Zustand zu halten. Sie muß also potentielle oder bereits entstandene Verletzungen der Integrität erkennen und geeignet darauf reagieren.

2.5.1 Allgemeine Aspekte

In diesem Abschnitt wird auf einige allgemeine Aspekte von Integritätsbedingungen und ihrer Überwachung eingegangen. Zunächst wird versucht, das breite Spektrum der existierenden Integritätsbedingungen zu strukturieren. Dazu werden im ersten Teil des Abschnitts mögliche Klassifizierungen von Integritätsbedingungen vorgestellt. Im zweiten Teil wird genauer auf einige Arten von Integritätsbedingungen, die in vielen Modellen eine Rolle spielen, eingegangen. Der letzten Teil des Abschnitts stellt verschiedene Methoden vor, auf Integritätsverletzungen zu reagieren.

Klassifizierung von Integritätsbedingungen

Integritätsbedingungen lassen sich nach den verschiedensten Kriterien in Klassen einteilen. In [WSK83], [Kre88], und [BDRZ83] werden z.B. folgende Kriterien für die Klassifizierung von Integritätsbedingungen angegeben:

Anlaß des Tests:

- **Operationsgebundene Bedingungen:** Die meisten Integritätsbedingungen können nur von bestimmmten Operationen verletzt werden. Um unnötige Tests zu verhindern, kann man sie an Operationen binden. Sie werden dann nur getestet, wenn die zugehörige Operation ausgeführt wird. Die an Operationen gebundenen Bedingungen lassen sich weiter in **Vor-** und **Nachbedingungen** unterteilen, je nachdem, ob sie vor oder nach der Operation getestet werden.
- **Operationsunabhängige Bedingungen:** Die operationsunabhängigen Bedingungen sind nicht an bestimmte Operationen gebunden, d.h., sie können von jeder Operation verletzt werden und müssen deshalb zumindest am Ende jeder Transaktion getestet werden.

Zeitpunkt der Überprüfbarkeit:

- **Verzögerbare Bedingungen** sind solche Bedingungen, deren Test auf das Ende der Transaktion verschoben werden kann. Da die Konsistenz der Datenbank und die Erfüllung aller Bedingungen erst für das Ende der Transaktion gefordert wird, sind alle Bedingungen verzögerbar.
- **Unmittelbare Bedingungen** sind Bedingungen, die vor bzw. direkt nach der Operation getestet werden können, da ihre Verletzung in der Regel in der Transaktion nicht mehr korrigiert werden kann. Ein Beispiel hierfür sind Eindeutigkeitsbedingungen. Im Gegensatz dazu gibt es Bedingungen, deren zeitweise Verletzung in Kauf genommen werden muß und die deshalb erst am Ende der Transaktion getestet werden können. Solche Bedingungen werden im folgenden als **verzögerte** Bedingungen bezeichnet. (Ein Beipiel für diese Gruppe ist die Bedingung: "Zu jedem Element in Menge A existiert genau ein Element in Menge B").

Berücksichtigung dynamischer Aspekte:

- **Zustandsbedingungen** beziehen sich auf den statischen Zustand der Datenbank zum Zeitpunkt des Tests. Dynamische Aspekte werden hierbei nicht berücksichtigt.
- **Übergangsbedingungen** berücksichtigen auch den dynamischen Aspekt von Datenbanken. Sie führen Beschränkungen für die zulässigen Zustandsübergänge ein. Um solche Bedingungen formulieren zu können, braucht man eine Möglichkeit, gleichzeitig auf den Zustand von Variablen vor und nach einer Operation zuzugreifen, d.h. der Zustand muß vor Ausführung der Operation in irgendeiner Form gespeichert werden.

Menge der betroffenen Objekte:

- **Einzelne Attribute**
- **Einzelne Objekte (Tupel)**
- **Mehrere Objekte in einer Klasse (Tupel in einer Relation)**
- **Objekte in verschiedenen Klassen (Tupel in verschiedenen Relationen)**

Art der Reaktion auf Verletzungen:

- **Starke Bedingungen:** Die Verletzung von starken Bedingungen führt auf jeden Fall zu einem Abbruch der Transaktion.
- **Schwache Bedingungen:** Bei der Verletzung einer schwachen Bedingung erhält der Benutzer eine Warnung und kann selbst entscheiden, wie weiter verfahren wird.
- **Getriggerte und selbstkorrigierende Bedingungen:** Bei getriggerten Bedingungen wird zu einer Bedingung eine Prozedur angegeben, die im Fall der Verletzung der zugehörigen Bedingung ausgeführt wird (Triggerprozedur). Aufgabe der Prozedur ist es, die Inkonsistenzen zu korrigieren oder eine andere geeignete Operation auszuführen. Selbstkorrigierende Bedingungen sind ein Sonderfall der getriggerten Bedingungen. Bei ihnen wird nach dem Ausführen der Prozedur noch einmal die Bedingung getestet, um zu überprüfen, ob die Verletzung tatsächlich behoben wurde.

Spezielle Integritätsbedingungen

Die Bedeutung einer Integritätsbedingung kann immer nur in Bezug auf ein bestimmtes Datenmodell angegeben werden, da Integritätsbedingungen in unterschiedlichen Modellen verschieden wichtig sind. Es gibt jedoch eine Reihe von speziellen Integritätsbedingungen, die in sehr vielen Modellen und Datenbanksprachen eine wichtige Rolle spielen:

Eindeutigkeits- und Schlüsselbedingungen: Eindeutigkeits- und Schlüsselbedingungen legen die Eindeutigkeit von Elementen innerhalb einer Kollektion (Klasse, Relation) bezüglich einer oder mehrerer Komponenten ihres Typs fest. Diese Art Bedingung findet sich sowohl in relationalen Ansätzen (*DBPL* [MS90], *SQL* [Dat89, SQL87]) als auch in objekt-orientierten Ansätzen wie z.B. in *Galileo* oder im *ORM*. Bei relationalen Ansätzen kommt zu der Schlüsselbedingungen meist noch der Aspekt des Schlüsselzugriffs hinzu, bei dem ein effizienter Zugriff auf ein Element über seinen Schlüssel unterstützt wird. Bei objekt-orientierten Ansätzen kann auf Objekte über ihre (systemvergebene) eindeutige Identität zugegriffen werden. Es ist jedoch manchmal auch ein wertorientierter Zugriff, wie ihn der Schlüsselzugriff darstellt, auf Objekte wünschenswert.

Referentielle Integrität: Im relationalen Modell sind die Elemente der Relationen flache Tupel. Um Beziehungen zwischen Elementen darzustellen, werden Schlüssel oder andere identifizierende Attribute als Referenz auf ein Element gespeichert. Es muß sichergestellt werden, daß die referentielle Integrität gewahrt wird ([Dat81, Mar90]), d.h., daß zu der gespeicherten Referenz auch ein Element in der entsprechenden Relation existiert.

Bei objekt-orientierten Ansätzen tritt das Problem der referentiellen Integrität in dieser Form nicht auf, da direkte Bezüge (Zeiger) auf die Objekte gespeichert werden und keine Referenzen. Ein der referentiellen Integrität jedoch verwandtes Problem, ist hierbei die Frage, was mit den Bezügen auf ein Objekt geschieht, wenn es gelöscht wird bzw., ob ein Objekt, auf das noch Bezüge existieren, gelöscht werden darf.

Falls, wie z.B. im *ORM*, Löschoperationen auf Klassen erlaubt sind, muß sichergestellt werden, daß kein Objekt aus einer Klasse gelöscht wird, auf das noch Verweise existieren.

Das Problem ist der referentiellen Integrität im relationalen Fall sehr ähnlich und wird deshalb im folgenden als spezielle Form der referentiellen Integrität behandelt.

Kardinalitätsbeschränkungen Kardinalitätsbeschränkungen werden meist für Beziehungen zwischen Objekten angegeben. Es wird eine untere und/oder obere Schranke für die Anzahl der Elemente einer Klasse (Relation), die mit einem Element einer anderen Klasse (Relation) in Beziehung stehen dürfen, angegeben. Ein wichtiges Beispiel für die Angabe einer unteren Schranke ist die Totalität (Surjektivität) einer Beziehung bezüglich einer Klasse (Relation) A. Sie legt fest, daß jedes Element aus A an der Beziehung teilnehmen muß (untere Schranke = 1).

Einschränkungen von Domänen Die Angabe eines Typs für ein Attribut stellt bereits eine Einschränkung seiner Domäne, also der Werte, die es annehmen kann, dar. Häufig soll aber die Domäne noch weiter eingeschränkt werden, als dies durch Angabe von Typen möglich ist. Dies kann durch Integritätsbedingungen geschehen, die Prädikate angeben, die zulässige Werte des Attibuts erfüllen müssen.

Integritätests und ihre Konsequenzen

Wenn eine Operation eine Integritätsbedingung der Datenbank verletzt, gibt es mehrere, konzeptuell verschiedene Möglichkeiten damit umzugehen. Zum einen kann mit einem Abbruch reagiert werden. Da die Konsistenz der Datenbank gewahrt bleiben soll, wird in diesem Fall evtl. ein Zurücksetzen einer oder mehrerer Operationen notwendig. Bei einem transaktionsorientierten Ansatz werden meist die Effekte der gesamten Transaktion auf die Datenbank rückgängig gemacht.

Ein anderer Ansatz für das Vorgehen bei Integritätsverletzungen ist es, für diesen Fall spezifizierte Operationen durchzuführen, die die Verletzung der Integrität korrigieren. Dieser Ansatz kann so realisiert sein, daß ein fester Satz (wichtiger) Integritätsbedingungen mit vordefinierten Operationen für die Korrektur angeboten wird. In anderen Systemen ist es Sache des Benutzers, geeignete Prozeduren zur Korrektur von Integritätsverletzungen zu definieren.

Ein allgemeines Problem mit dem Ansatz der Ausführung von weiteren Aktionen zur Wahrung bzw. Wiederherstellung von Integritätsbedingungen ist, daß die angestoßene Operation wiederum zu neuen Integritätsverletzungen führen kann, die korrigiert werden müssen. Dieses Anstoßen weiterer Aktionen kann sich über mehrere Stufen fortsetzen. Es besteht die Gefahr, daß der Benutzer durch eine von ihm (evtl. fehlerhaft) definierte Operation eine Reihe von Aktionen auf der Datenbank auslöst, die er erstens nicht mehr überschaut und zweitens vielleicht auch garnicht beabsichtigt hat. Ein mehr technisches Problem dieses Ansatzes ist die Handhabung von Konfliktsituationen (siehe [Mar90]) und die Wahrung der Endlichkeit der Ausführung durch Vermeidung von Zykeln. Eine mögliche Hilfe stellt die graphische Anzeige und Kontrolle der resultierenden Propagierungen dar, wie sie z.B. vom System *Gambit* [BDRZ83] angeboten wird.

2.5.2 Integritätsbedingungen im *ORM*

Bei der Definition von Klassen und Beziehungen können beim *ORM* Integritätsbedingungen angegeben werden. Dabei stehen Eindeutigkeits- und Schlüsselbedingungen, Inklusions- und

Exklusionsbedingungen, referentielle Integritätsbedingungen, Surjektivitätsbedingungen (Totalität) und Unveränderlichkeitsbedingungen zur Verfügung. Weiterhin gibt es die Möglichkeit, Operationen anzugeben, die vor dem Einfügen bzw. dem Löschen eines Elements aus der Klasse (Beziehung) ausgeführt werden. Mit *self* kann dabei auf die zu definierende Klasse (Beziehung) und mit *this* auf das einzufügende bzw. zu löschende Element zugegriffen werden. Der Test von Integritätsbedingungen ist eine der möglichen Operationen.

Die Definitionen von Klassen und Beziehungen werden auf Operationen eines kleinen kompakten Sprachkerns abgebildet. Dieser beinhaltet eine grundlegende Funktion zur Definition von Beziehungen, die nur die Angabe von Elementtyp und Schlüsseln ermöglicht. Zur Verwaltung der Integritätsbedingungen stellt der Sprachkern folgenden Mechanismus zur Verfügung: Zu jeder Beziehung werden zwei Listen verwaltet, von denen eine zu der Einfügeoperation in die Beziehung gehört und die andere zur Löschoperation. Die Listen enthalten Operationen, die vor der Einfüge- bzw. Löschoperation ausgeführt werden. Durch die Funktionen *beforeInsert* und *beforeRemove* können neue Elemente in diese Listen eingefügt werden.

Der Test einer Integritätsbedingung wird, wie oben bereits erwähnt, als eine spezielle Operation betrachtet, die aus dem Schlüsselwort *assert* gefolgt von der zu testenden Bedingung besteht. Durch Voranstellen des Schlüsselworts *defer* wird der Test der Bedingung bis ans Ende der Transaktion verzögert. Mit Hilfe dieses Konstrukts kann auch die Ausführung anderer Operationen auf das Ende der Transaktion verschoben werden.

Zur Formulierung von Übergangsbedingungen kann der Operator *old* auf Variablen, Beziehungen und Klassen angewandt werden. Er liefert den Wert der Variablen (Klasse, Beziehung) zurück, so wie er zu Beginn der Transaktion gewesen ist.

Integritätsbedingungen für die Operationen auf Klassen werden auf die gleiche Weise verwaltet, da Klassen im Sprachkern als unäre Beziehungen behandelt werden.

Für die von diesem Ansatz unterstützten Integritätsbedingungen gilt also:

- Bedingungen sind an Operationen gebunden.
- Nichtverzögerte Bedingungen sind Vorbedingungen.
- Der verzögerte Test von Integritätsbedingungen ist möglich.
- Es können Triggerprozeduren angegeben werden.
- Die Formulierung von Übergangsbedingungen ist möglich.

Die speziellen Integritätsbedingungen werden auf geeignete Operationen abgebildet und in die entsprechenden Listen eingefügt, sodaß sie von dem im Sprachkern enthaltenen allgemeinen Mechanismus verwaltet werden können. Im folgenden werden die einzelnen speziellen Bedingungen und ihre Übersetzung in Aufrufe von Funktionen des Sprachkerns beschrieben.

Eindeutigkeits- und Schlüsselbedingungen

Sowohl bei der Definition einer Klasse als auch bei der einer Beziehung können ein oder mehrere alternative Schlüssel angegeben werden. Die Eindeutigkeit der Elemente bezüglich der angegebenen Schlüssel wird von dem System sichergestellt. Schlüsselbedingungen gehören zum Sprachkern.

Inklusionsbedingungen

Bei der Definition einer Klasse A kann man eine oder mehrere Klassen $C_1 \ldots C_n$ als direkte Superklassen angeben. Zwischen den Klassen $C_1 \ldots C_n$ und der Klasse A besteht dann eine Inklusionsbeziehung. Dieser Zusammenhang wird, wie bereits erwähnt, durch Test und Wahrung der Inklusionsbedingung realisiert. Das System stellt die Inklusionsbedingung sicher, indem vor dem Einfügen eines Elements in eine Klasse, das Element vorher in jede ihrer Superklassen eingefügt und vor dem Löschen eines Elements aus einer Klasse es zunächst aus all ihren Subklassen entfernt wird, in denen es enthalten ist.

Die Angabe von Inklusionsbedingungen ist auch für Beziehungen möglich, d.h., man kann Generalisierungshierarchien auf Beziehungen aufbauen.

Exklusionsbedingungen

Man kann bei der Definition einer Klasse A nicht nur Klassen angeben, in denen sie enthalten ist, sondern auch Klassen $B_1 \ldots B_m$, mit denen sie keine gemeinsamen Elemente haben soll (Exklusionsbedingungen). Exklusionsbedingungen können wie Eindeutigkeitsbedingungen nur durch Einfügeoperationen verletzt werden. Um die Exklusion zu gewährleisten, werden folgende Integritätsbedingungen getestet: Vor dem Einfügen eines Elements in Klasse A wird getestet, daß es nicht bereits in einer der angegeben Klassen enthalten ist. Vor dem Einfügen eines Elements in eine der Klassen B_i wird überprüft, ob das Element bereits in Klasse A enthalten ist.

Referentielle Integritätsbedingungen

Referentielle Integritätsbedingungen werden bei der Definition von Beziehungen angegeben. Sie stellen eine Verbindung zwischen einer Komponente *componentA* des Elementtyps der definierten Beziehung und einer Klasse A her.

Die referentielle Integrität muß beim Einfügen eines Elements in die Beziehung und beim Löschen von Elementen aus der Klasse getestet werden. Beim Einfügen wird getestet, ob der Wert der Komponente *componentA* des einzufügenden Elements r in der Klasse A enthalten ist, und beim Löschen, daß in der Beziehung kein Element mehr existiert, das sich auf das zu löschende Element bezieht.

Es gibt bei diesem Ansatz drei Formen der referentiellen Integritätsbedingungen, die sich durch die Reaktion auf die Verletzung der Bedingung unterscheiden:

in **A:** Der Versuch, ein Element r in die Beziehung einzufügen, zu dem kein Element in der Klasse A existiert, und der Versuch, ein Element aus der Klasse A zu löschen, zu dem noch ein Element in der Beziehung existiert, führen beide zu einem Abbruch.

are **A:** Die Komponente *componentA* des Elements r wird automatisch in die Klasse A eingefügt, falls dieses Element dort noch nicht vorhanden ist. Falls ein Element aus Klasse A gelöscht wird, zu dem noch Elemente in der Beziehung existieren, so werden diese automatisch gelöscht.

owned_by **A:** Mit dieser Option ordnet man die Beziehung der Klasse A zu[3]. Die Option stellt einen Mittelweg zwischen den ersten beiden dar: Beim Einfügen von Elementen in die Beziehung wird wie bei der Option *in* verfahren und beim Löschen wie bei der Option *are*.

Für die Option *are* muß der Typ der Komponente *componentA* ein Subtyp des Elementtyps der Klasse A sein, für die Optionen *in* und *owned_by* müssen diese Typen miteinander kompatibel sein.

Surjektivität

Für eine Beziehung kann festgelegt werden, daß sie bezüglich einer oder mehrer Klassen surjektiv ist, d.h., jedes Element der Klasse(n) muß an der Beziehung teilnehmen. Wie durch die Angabe einer referentiellen Integritätsbedingung wird auch hier ein Zusammenhang zwischen einer Komponente *componentA* des Elementtyps der Beziehung und einer Klasse A hergestellt. Der Typ der Komponenten und der Elementtyp der Klasse A müssen kompatibel sein.

Die Surjektivität einer Beziehung bezüglich einer Klasse kann beim Einfügen von Elementen in die Klasse und beim Löschen von Elementen aus der Beziehung verletzt werden. Beim Einfügen eines Elements in die Klasse muß sichergestellt werden, daß innerhalb der gleichen Transaktion ein zugehöriges Element in die Beziehung eingefügt wird. Wenn das letzte Element, das sich auf ein Element a der Klasse bezieht, gelöscht wird, so muß sichergestellt werden, daß innerhalb der gleichen Transaktion auch a gelöscht wird.

Es wird von folgender Reihenfolge auf den Operationen ausgegangen: Ein Element a wird in die Klasse eingefügt, bevor Elemente, die sich auf a beziehen, in die Beziehung eingefügt werden. Beim Löschen wird von der umgekehrten Reihenfolge ausgegangen. Mit dieser Annahme können referentielle Integritätsbedingungen sofort getestet werden, während der Test der Surjektivität bis zum Ende der Transaktion zurückgestellt werden muß.

Es gibt zwei Formen der Surjektivitätsbedingung, die sich wierderum durch die Reaktion auf eine Verletzung der Bedingung unterscheiden:

owns **A:** Diese Option ordnet die Klasse A der Beziehung zu. Wenn das letzte Element der Beziehung, das zu einem Element a aus A gehört, gelöscht wird, so wird automatisch auch das Element a gelöscht. Wenn zu einem neu in die Klasse eingefügten Element innerhalb der gleichen Transaktion kein zugehöriges Element in die Beziehung eingefügt wird, so führt dies zu einem Abbruch.

onto **A:** Bei Angabe dieser Option werden keine automatischen Löschoperationen durchgeführt. Beide Fälle führen zu einem Abbruch.

Unveränderlichkeitsbedingungen

Es ist weiterhin möglich festzulegen, daß eine Beziehung bezüglich einer oder mehrerer Klassen unveränderlich ist. Unveränderlichkeit bezüglich einer Klasse A bedeutet, daß alle Elemente

[3]Die Beziehung gehört der Klasse A

der Beziehung, die sich auf ein Element a der Klasse A beziehen, innerhalb der gleichen Transaktion eingefügt werden wie dieses Element. Außerdem können sie erst in der Transaktion wieder gelöscht werden, in der auch das Element a aus der Klasse gelöscht wird. Für die Unveränderlichkeit bezüglich mehrerer Klassen $A_1 \ldots A_n$ wird diese Forderung entsprechend erweitert: Alle Elemente der Beziehung, die sich auf die Elemente $a_1 \ldots a_n$ aus den jeweiligen Klassen beziehen, müssen innerhalb der gleichen Transaktion wie das letzte dieser Elemente eingefügt werden. Ein Löschen dieser Elemente der Beziehung ist erst innerhalb der Transaktion zulässig, in der mindestens eines der Elemente a_i gelöscht wird.

Eine Umsetzung dieser Bedingung ist möglich, da der Operator *old* einen Zugriff auf den Zustand der jeweiligen Klassen vor der aktuellen Transaktion erlaubt. Dadurch kann überprüft werden, ob ein Element erst in dieser Transaktion oder schon vorher eingefügt worden ist.

Für die Unveränderlichkeit bezüglich einer Klasse A müssen der Typ der Komponente der Beziehunng, die sich auf die sich auf die Klasse bezieht, und der Elementtyp von A kompatibel sein. Bei der Unveränderlichkeit bezüglich mehrer Klassen müssen entsprechend die Typen der jeweiligen Komponenten mit den Elementtypen der zugehörigen Klassen kompatibel sein.

2.5.3 Andere Ansätze

Integritätsbedingungen in *Gambit*

In *Gambit* [BDRZ83] wird die Definition des Schemas und der Integritätsbedingungen durch ein grafisches System erleichtert. Integritätsbedingungen werden stets an eine oder mehrere Operationen gebunden: Man kann bei der Definition der Bedingung festlegen, bei welchen Operationen sie getestet werden soll. Außerdem hat man die Möglichkeit die Bedingungen in starke, schwache und getriggerte Bedingungen zu unterteilen. Im Falle der starken bzw. schwachen Bedingungen kann der Benutzer einen Text für die Fehlermeldung bzw. die Warnung festlegen, der im Falle einer Verletzung ausgegeben wird. Bei getriggerten Bedingungen kann er eine Triggerprozedur angeben, die im Fall der Verletzung der Bedingung ausgeführt wird.

Die eigentlichen Integritätsbedingungen werden in der Sprache Modula/R [KMP+83] formuliert, auf die auch die ganze mit *Gambit* erstellte Schemadefinition abgebildet wird. Es besteht dabei durch Voranstellen von *old* bzw. *new* die Möglichkeit, sowohl auf die Werte vor als auch auf die Werte nach Ausführung der Operation zuzugreifen. Es können also auch Übergangsbedingungen formuliert werden. Für die Formulierung von Eindeutigkeitsbedingungen gibt es einen vordefinierten Operator *UNIQUE* mit dem ein Attribut oder auch eine Kombination von Attributen als eindeutig innerhalb ihrer Relation festgelegt werden können.

Weiterhin können bei der grafischen Schemadefinition Kardinalitätsbeschränkungen für Beziehungen angegeben werden. Dabei ist sowohl die Angabe von oberen als auch von unteren Schranken möglich. Die aus diesen Angaben resultierenden Propagierungen von Einfüge- und Löschoperationen kann sich der Benutzer des Systems grafisch anzeigen lassen. Er hat dabei die Möglichkeit gewisse Propagierungen zu verbieten.

Gambit erzeugt bei der Abbildung der Schemadefinition in Modula/R-Code zu jeder Relation die grundlegenden Transaktionen zum Einfügen, Löschen und Ändern. Dabei werden die Tests der definierten Integritätsbedingungen an geeigneter Stelle in den Code eingefügt.

Integritätsbedingungen in *Galileo*

In *Galileo* wird bei der Definition von abstrakten Datentypen automatisch eine Funktion generiert, mit der neue Werte des Typs erzeugt werden können. Bei der Definition von abstrakten Datentypen besteht die Möglichkeit, Integritätsbedingungen anzugeben. Ein Test dieser Integritätsbedingungen findet bei der Erzeugung von neuen Werten des abstrakten Datentyps statt. Falls eine der Bedingungen verletzt ist, führt dies zu einem Abbruch mit einer Fehlermeldung, deren Text ebenfalls bei der Definition des abstrakten Datentyps festgelegt wird. Da die Definition von Klassen in *Galileo* auf der Definition eines abstrakten Datentyps beruht, kann man auch für Klassen Integritätsbedingungen angeben, die beim Erzeugen von neuen Objekten der Klasse getestet werden.

Selbstkorrektur in aktiven Datenbanken

Ceri und Widom schlagen in [CW90] vor, das Problem der Wahrung bzw. Wiederherstellung von Integritätsbedingung im Rahmen der Einführung von allgemeinen Regeln in ein Datenbanksystem ([MD89, Mor83]) zu lösen. Es wird dabei von einer *SQL*-ähnlichen Sprache ausgegangen. Die Erweiterung zu einer aktiven Datenbank durch die Einführung von Regeln wird in [WF90] beschrieben, die Verwendung der Regeln zur Integritätsüberwachung in [CW90]. Es handelt sich dabei um einen transaktionsorientierten Ansatz.

Zur Definition einer Regel gehört die Angabe von Operationen, die die Regel auslösen können. Weiterhin wird eine Bedingung festgelegt, und eine beliebig komplexe Aktion, die ausgeführt wird, falls die Bedingung erfüllt ist. Es wird ein Protokoll über die während der Transaktion durchgeführten Operationen geführt. Am Ende der Transaktion werden anhand des Protokolls alle Regeln herausgesucht, die ausgelöst werden können. Aus dieser Menge wird eine Regel, deren Bedingung erfüllt ist, ausgesucht[4] und ausgeführt. Bei der Wahl der nächsten Regel werden die von dieser Regel ausgeführten Operationen mitberücksichtigt. Die Regeln bilden einen Graphen. Um die Endlichkeit der Ausführung zu gewährleisten, ist das Auftreten von Zykeln zu vermeiden. Auf dieses Thema wird in [CW90] eingegangen.

Indem man als Bedingung der Regel die zu wahrende Integritätsbedingung (in negierter Form) wählt, kann man diesen Ansatz zur Integritätsüberwachung verwenden. Als Operationen, die die Regel auslösen können, werden die Operationen angegeben, die die Integritätsbedingung verletzen können. Die ausgelöste Aktion hat die Aufgabe, die Datenbank so zu modifizieren, daß nach ihrer Ausführung die auslösende Integritätsbedingung wieder erfüllt ist. Da aber die Aktion in der Regel wiederum aus Operationen auf der Datenbank besteht, können durch ihre Ausführung neue Integritätsbedingungen verletzt werden, die durch den Aufruf weiterer Aktionen wiederhergestellt werden müssen.

Kaskadierte Löschoperationen in *SQL2*

SQL2, dessen wichtigste Erweiterungen gegenüber *SQL* in [Sha90] vorgestellt werden, ist ein Beispiel für einen Ansatz mit vordefinierten Aktionen zur Korrektur von Integritätsverletzungen. Zu einem Attribut einer Tabelle kann angegeben werden, daß es sich auf ein anderes

[4] Auf mögliche Auswahlkriterien wird in den oben erwähnten Texten eingegangen

Element bezieht, also eine Referenz ist. An dieser Stelle kann der Benutzer außerdem festlegen, wie verfahren werden soll, falls das Bezugsobjekt gelöscht wird (Verletzung der referentiellen Integrität). Er hat dabei die Auswahl zwischen drei Alternativen: (a) Das Objekt, das den Bezug enthält, wird ebenfalls gelöscht (kaskadiertes Löschen). Daraus können evtl. weitere Löschoperationen resultieren, wenn auch auf dieses Objekt noch Bezüge existieren. (b) Der Bezug wird auf einen Nullwert gesetzt. Macht der Benutzer keine Angaben, so ist (c) ein Löschen des Bezugsobjektes verboten.

Kapitel 3

Relevante Programmiersprachenkonzepte (P-Quest)

Quest[1] ist eine strikt typisierte, polymorphe, funktionale Programmiersprache, in der jedoch auch imperative Eigenschaften zu finden sind. Die Typüberprüfung findet statisch zur Übersetzungszeit statt.

Die Sprache *Quest* wurde von Luca Cardelli bei DEC SRC in Palo Alto, USA [Car89] entwickelt. Im Rahmen einer Diplomarbeit an der Universität Hamburg [Mü91] wurde die Sprache durch Hinzufügen eines Persistenzkonzeptes zu *P-Quest* erweitert. In diesem Kapitel werden zunächst die für diese Arbeit relevanten Konzepte der Sprache *Quest* kurz eingeführt und danach das Persistenzkonzept des *P-Quest*-Systems erläutert. Eine vollständige Einführung in die Sprache *Quest* findet sich in [Car90].

Das *Quest*-System besteht aus mehreren Komponenten. Die Komponenten sind eine interaktive Programmierumgebung, eine Reihe von Bibliotheken (z.Zt. Standard-, Bulk Data Type- und Grafik-Bibliothek), sowie eine in Modula-3 [CDG+88] realisierte Abstrakte Maschine. Die Abstrakte Maschine interpretiert den vom *Quest*-Compiler erzeugten Bytecode.

Mit *Quest* werden viele neuere Programmierkonzepte realisiert, die in den nachfolgenden Abschnitten anhand von Beispielen vorgestellt werden sollen.

Die Konzepte sind im einzelnen:

- Erweitertes Typkonzept (Werte, Typen, Kinds)
- Funktionen höherer Ordnung
- Subtypisierung
- Parametrischer und Subtyppolymorphismus
- Abstrakte Datentypen

[1] *Quest* steht für *Quantifiers and Subtypes*.

- Module und Schnittstellen
- Funktionen und Module als Objekte *erster Klasse*
- Ausnahmebehandlung

3.1 *Quest*-Typkonzepte

Traditionelle Progammiersprachen unterscheiden nur zwischen *Werten* und *Typen*. Werte stellen die Ebene 0 und Typen die Ebene 1 dar. Wie in der Abbildung 3.1 zu sehen ist, gibt es in *Quest* noch eine weitere Ebene (Ebene 2), die Ebene der *Kinds*. Kinds klassifizieren Typen in der gleichen Weise, wie Typen Werte. Die Ebene der Kinds wird benötigt, um die in *Quest* ungewöhnlich reiche Typebene zu strukturieren. Die drei Ebenen werden in den nachfolgenden Abschnitten anhand von Beispielen beschrieben.

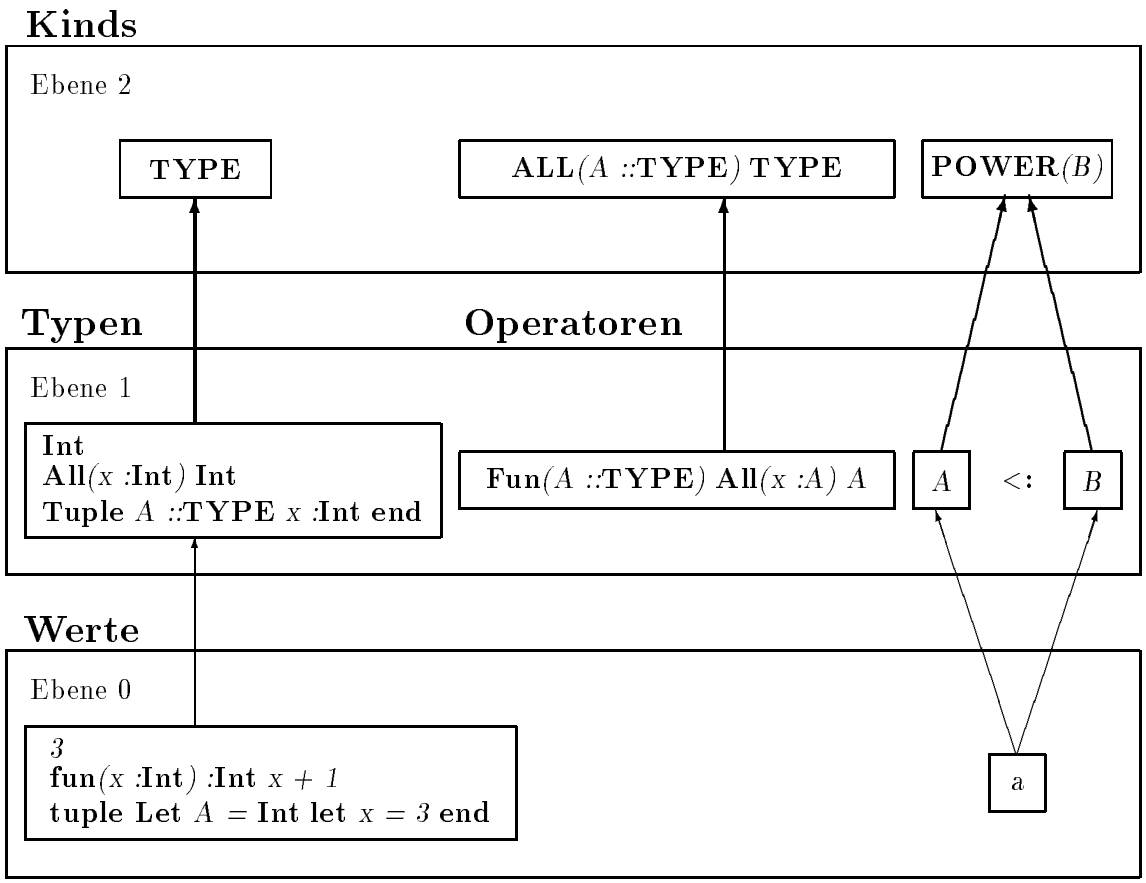

Abbildung 3.1: Das Typkonzept von Quest

Abbildung 3.1 zeigt nicht nur die im Typkonzept von *Quest* vorhandenen drei Ebenen, sondern

sie verdeutlicht auch die in Abschnitt 3.1.3 aufgeführte Typisierung von Operatoren in der Kindebene und die in Abschnitt 3.4 beschriebene Subtypisierung.

Bezüglich der Groß- und Kleinschreibung ist folgendes zu beachten: Schlüssselworte im Zusammenhang mit Werten werden klein geschrieben, Schlüsselworte im Zusammenhang mit Typen beginnen mit einem Großbuchstaben und Schlüsselworte im Zusammenhang mit Kinds werden ganz in Versalien geschrieben. In dieser Arbeit wird zur besseren Unterscheidung zwischen den Ebenen die Konvention von Cardelli übernommen, diese Regel auch auf die Schreibweise von Bezeichnern für Objekte in der jeweiligen Ebene anzuwenden.

3.1.1 Die Ebene der Werte

Zur Ebene der Werte gehören Ausprägungen der in *Quest* angebotenen Basistypen, strukturierte Werte, wie Tupel und Optionen, aber auch Funktionen und polymorphe Funktionen.

let *x = 3*
let *sven =* **tuple let** *name = "sven"* **let** *age = 25* **end**
let *name =* **fun***(p :Person)* **:String** *p.name*
let *identity =* **fun***(A ::***TYPE** a *:A) :A* a

Ein Wert wird mit dem Schlüsselwort *let* an eine Variable gebunden. Im ersten Beispiel wird eine Ausprägung des Basistyps *Integer* an die Variable *x* gebunden. Im zweiten Beispiel wird ein strukturierter Wert, und zwar ein *tuple*, an die Variable *sven* gebunden. In den folgenden Beispielen werden eine Funktion und eine polymorphe Funktion definiert. Wie am vierten Beispiel zu sehen ist, können bereits auf dieser Ebene Typen auftreten, und zwar z.B. als Komponenten von Tupeln und als Parameter von polymorphen Funktionen.

3.1.2 Die Ebene der Typen

Die Typebene enthält Typen und Typoperatoren. Zu den Typen gehören neben den Basistypen wie z.B. *Int*, *Real* und *Ok* [2], strukturierte Typen, wie z.B. Tupeltypen und Funktionstypen. Typoperatoren (Typfunktionen) sind Funktionen, die es ermöglichen, Typen auf Typen abzubilden. Objekte der Typebene können durch das Vorstellen des Schlüsselworts *Let* an Namen gebunden werden. Es folgen einige Beispiele für die Definition und Benennung von Typen.

Let *Number =* **Int**
Let *Person =* **Tuple** *name* **:String** *age* **:Int end**
Let *Name =* **All***(:Person)* **String**
Let *Identity =* **Fun***(A ::***TYPE***) ::***TYPE** *A*

Die Beispiele korrespondieren mit den Beispielen auf der Ebene der Werte. Das erste Beispiel führt einen neuen Namen für den Typ der ganzen Zahlen ein. *Person* ist der Typ der Variablen *sven* und *Name* der Typ der Funktion *name*. Das vierte Beispiel definiert einen Typoperator. Es handelt sich dabei um eine Identitätsfunktion auf der Typebene.

[2] *Ok* ist ein Typ, der nur die Konstante *ok* beinhaltet. Entspricht *void* in C.

3.1.3 Die Ebene der Kinds

Kinds sind die Typen von Typen. Sie dienen zur Strukturierung der Menge der Typen. Der allgemeinste Kind ist *TYPE*; das ist der Kind, der alle Typen enthält. Weitere Kinds können vom Benutzer definiert werden. Die Definition von Kinds geschieht durch Voranstellen des Schlüsselwortes *DEF*.

DEF *TYPOP* = **ALL**(*A* ::**TYPE**) **TYPE**
DEF *PERSON* = **POWER**(**Tuple** *name* :**String end**)

Im ersten Beispiel wird der Kind der einstelligen Typoperatoren definiert und an den Namen *TYPOP* gebunden. Das Schlüsselwort *ALL* dient zur Definition von Typen von Typoperatoren. Der im zweiten Beispiel definierte Kind *PERSON* repräsentiert alle Tupeltypen, deren erste Komponente den Attributnamen *name* hat und vom Typ *String* ist, die aber eventuell weitere Komponenten besitzen. Wenn *T* ein Typ ist, dann ist *POWER(T)* ein Kind, und zwar die Menge aller Subtypen von *T*.

3.2 Funktionen in *Quest*

Neben den aus anderen Programmiersprachen bekannten *einfachen* und *rekursiven* Funktionen gibt es in *Quest* zusätzlich Funktionen *höherer Ordnung* und *polymorphe* Funktionen. Auf diese Arten von Funktionen und ihre Definition in *Quest* wird im folgenden genauer eingegangen.

3.2.1 Einfache Funktionen

Da das Konzept der einfachen Funktionen aus anderen Programmiersprachen hinreichend bekannt ist, soll es hier genügen, ihre Syntax in *Quest* einzuführen. Diese unterscheidet sich allerdings ein wenig von der in traditionellen Programmiersprachen üblichen. Funktionen werden durch das Schlüsselwort *fun* eingeleitet, und die einzelnen Parameter werden nicht durch Kommata, sondern durch Leerzeichen voneinander getrennt, wie in den beiden folgenden Beispielen zu sehen ist:

let *succ* = **fun**(*a* :**Int**) :**Int** *a + 1*
let *add* = **fun** (*a* :**Real** *b* :**Real**) :**Real** *a ++ b*

Die erste Funktion *succ* erwartet einen Parameter vom Typ *Int* und liefert als Ergebnis auch einen Wert vom Typ *Int* zurück. Sie berechnet für einen ganzzahligen Eingabewert den direkten Nachfolger. Die zweite Funktion *add* addiert zwei reelle Zahlen. Sie erwartet zwei Parameter vom Typ *Real* und liefert einen Wert vom Typ *Real* zurück. Die Syntax von *Quest* läßt auch folgende verkürzte Schreibweisen zu:

let *succ*(a :*Int*) :*Int* = a + *1*
let *add*(a, *b* :*Real*) :*Real* = a ++ *b*

In *Quest* gibt es keine Operatorüberladungen. Deshalb gibt es je nach Parametertytp unterschiedliche Additionsoperatoren. Während zwei ganze Zahlen mit einem Pluszeichen (+) addiert werden, benutzt man für die Addition zweier reeller Zahlen ein doppeltes Pluszeichen (++).

3.2.2 Rekursive Funktionen

Rekursive Funktionen werden durch das Schlüsselwort *rec* gekennzeichnet. Ein Beispiel für rekursive Funktionen ist die bekannte Fakultätsberechnung:

```
let rec fac(n :Int) :Int =
  if n is 0
    then 1
    else n * fac(n - 1)
  end
```

Bei diesem Beispiel ist außerdem zu beachten, daß der Test auf Gleichheit zweier Werte nicht durch das Gleichheitszeichen, sondern durch den Operator *is* geschieht. Das dient der Vermeidung von Operatorüberladungen [3]. Der Operator *is* testet einfache Werte, wie z.B. Zahlen auf Gleichheit, strukturierte Werte hingegen, wie z.B. Tupel, auf Identität.

3.2.3 Funktionen höherer Ordnung

Funktionen höherer Ordnung sind Funktionen, denen Funktionen als Parameter übergeben werden bzw. die Funktionen als Ergebnis zurückliefern können.

Ein Beispiel für Funktionen höherer Ordnung ist die Funktion *double*.

```
let double(f(a :Int) :Int a :Int) :Int = f(f(a))
```

Sie erhält als ersten Parameter eine Funktion, die ganzzahlige Werte auf ganzahlige Werte abbildet, und als zweiten Parameter einen ganzzahligen Wert übergeben. *double* wendet die übergebene Funktion zweimal hintereinander auf den übergebenen Wert an.

Ein Aufruf der Funktion *double* mit der Funktion *succ* und der Zahl 4 als Parameter (*double(succ 4)*) zum Beispiel, berechnet den Nachfolger vom Nachfolger von vier, also die Zahl Sechs.

3.2.4 Polymorphe Funktionen

Eine Funktion wird zu einer polymorphen oder generischen Funktion, indem man ihre Signatur durch einen oder mehrere Typparameter erweitert (*parametrischer Polymorphismus*). Die Typparameter werden beim Aufruf durch Typen instanziiert.

Ein einfaches Beispiel für ein polymorphe Funktion ist eine polymorphe Identitätsfunktion. Sie soll hier in zwei Versionen betrachtet werden:

[3] Das Gleichheitszeichen dient der Bindung an eine Variable.

let *identity1(A ::***TYPE***)(a :A) :A = a*
let *identity2(A ::***TYPE** *a :A) :A = a*

Die Versionen unterscheiden sich dadurch, daß die erste in zwei Stufen parametrisiert werden kann, während die zweite in einer Stufe parametrisiert werden muß.

Die erste Version kann in der ersten Stufe nur mit einem Typ T parametrisiert werden. Man erhält dann eine Identitätsfunktion auf dem Typ T, was hier am Beispiel des Typs *Integer* gezeigt wird:

let *intId = identity1(:***Int***)*

Die zweite Version muß sofort mit einem Typ T und mit einem Wert t dieses Typs parametrisiert werden und liefert als Ergebnis t zurück. In der Regel kann dabei jedoch die Angabe des Typs weggelassen werden, da dieser vom *Quest*-System durch Typinferenz aus dem übergebenen Wert hergeleitet werden kann.

Folgende Aufrufe der definierten Funktionen liefern alle das gleiche Ergebnis und zeigen noch einmal die verschiedenen Parametrisierungen der beiden Versionen der polymorphen Identitätsfunktion:

intId(4)
*identity1(:***Int***)(4)*
*identity2(:***Int** *4)*
identity2(4)

Für die Instanziierung von Typparametern sind nicht nur die Basistypen, sondern auch beliebige benutzerdefinierte Typen zulässig.

Der große Vorteil von polymorphen Funktionen ist, daß man eine Funktion schreiben kann, die auf allen Typen arbeitet, anstatt für jeden Typ eine eigene Version dieser Funktion zu schreiben. Sie eignen sich also besonders zur Beschreibung von Kontrollflüssen, die typunabhängig sind.

Eine besondere Stärke von *Quest* liegt in der Möglichkeit, das Konzept der polymorphen und der höheren Funktionen miteinander zu kombinieren. Dies soll am Beispiel einer polymorphen Sortierfunktion veranschaulicht werden. Das eigentliche Sortieren, d.h. das Umordnen der Elemente ist typunabhängig, kann also im Rahmen einer polymorphen Funktion beschrieben werden. Das Vergleichen der Elemente jedoch, das beim Sortieren durchgeführt werden muß, ist typabhängig. Diese Aufgabe kann gelöst werden, indem man der polymorphen Sortierfunktion eine Funktion als Parameter übergibt, deren Aufruf den Vergleich von zwei Elementen durchführt.

Die Signatur einer polymorphen Sortierfunktion, die Felder von Elementen sortiert, könnte dann wie folgt aussehen:

let *sort(A ::***TYPE** *order(a,b :A) :***Bool** *a :***Array***(A)) :***Array***(A) = ...*

Um ein konkretes Feld zu sortieren, muß man noch eine Vergleichsfunktion auf dem Elementtyp schreiben. Mit dieser Funktion legt man das Sortierkriterium und die Sortierreihenfolge fest. Als Beispiel sei hier die Sortierung von Personen nach ihrem Alter gewählt:

let *older(a, b :Person)* :**Bool** = a.age > b.age

Der folgende Aufruf der Funktion *sort* sortiert dann ein Feld von Personen, und zwar nach ihrem Alter.

sort(:Person older personArray)

3.3 Datentypen und Typoperatoren

Die Sprache *Quest* enthält neben den Basistypen *Ok*, *Bool*, *Char String*, *Int* und *Real* die Typkonstruktoren *Tuple*, *Option*, *Array* und *Exception*[4]. Dabei entspricht der Typ *Tuple* dem Record traditioneller Programmiersprachen und der Typ *Option* dem varianten Record. Definitionen von Tupel- bzw. Optionstypen und zugehöriger Wert sehen wie folgt aus:

```
Let Point = Tuple x :Int y :Int end
let point = tuple let x = 5 let y = 7 end

Let Figure =
  Option
    circle with center :Point radius :Int end
    triangle with point1, point2, point3 :Point  end
    rectangle with point1, point2, point3 :Point end
  end
let circle =
  option circle of Figure with let center = point let radius = 1 end
```

Der Zugriff auf Komponenten eines Tupels geschieht mit Hilfe der Punktnotation. Optionen werden im allgemeinen mit einer *case*-Anweisung inspiziert.

Soviel zu den grundlegenden Typen in *Quest*. Im folgenden wird auf die spezielleren Typen und auf die Typoperatoren eingegangen.

3.3.1 Funktionstypen

In *Quest* ist jeder Funktion ein Typ zugeordnet. Dies gilt auch für die polymorphen Funktionen und für die Funktionen höherer Ordnung. Funktionstypen werden mit dem Schlüsselwort *All* eingeleitet. Zur Veranschaulichung sind anschließend die Typen einiger der im Abschnitt 3.2 definierten Funktionen aufgeführt:

succ, fac :**All**(:**Int**) **Int**
add :**All**(:**Real** :**Real**) **Real**
double :**All**(:**All**(:**Int**) **Int** :**Int**) **Int**
identity1 :**All**(*A* ::**TYPE**) **All**(:*A*) *A*
identity2 :**All**(*A* ::**TYPE** :*A*) *A*
sort :**All**(*A* ::**TYPE** :**All**(:*A* :*A*) **Bool** :**Array**(*A*)) **Array**(*A*)

[4] Auf die Typkonstruktoren *Array* und *Exception* wird in späteren Abschnitten genauer eingegangen.

3.3.2 Rekursive Datentypen

Rekursive Datenstrukturen wie Listen, Mengen und Bäume spielen eine wichtige Rolle in der Informatik. *Quest* bietet deshalb auch eine Möglichkeit zur Definition von rekursiven Datentypen, mit denen rekursive Datenstrukturen auf einfache Weise realisiert werden können. Als Beispiel sei hier die Definition einer Liste für ganze Zahlen gezeigt:

```
Let Rec IntegerList ::TYPE =
  Option
    nil
    cons with head :Int tail :IntegerList end
  end
```

Die Definition von rekursiven Typen wird durch *Rec* eingeleitet.

An diesem Beispiel läßt sich die Einführung von Typoperatoren motivieren, die im nächsten Abschnitt beschrieben werden. Ähnlich wie beim Übergang zu polymorphen Funktionen ist es auch hier wünschenswert, gemeinsame, typunabhängige Muster herauszukristallisieren und für diese Muster generischen Code zu schreiben, der für die einzelnen Typen instanziiert werden kann. Im konkreten Fall wäre das so etwas wie eine polymorphe Liste, die mit einem Typ E instanziert eine Liste über Elementen vom Typ E ergibt.

3.3.3 Typoperatoren

Typoperatoren sind Funktionen, die Typen auf Typen abbilden. Sie ermöglichen es, Typdeklarationen zu parametrisieren.

Ein einfaches Beispiel für einen Typoperator ist eine der weiter oben eingeführten Identitätsfunktion entsprechende Funktion auf der Typebenen:

```
Let Identity = Fun(E ::TYPE) ::TYPE E
```

oder kürzer (ähnlich wie bei den Funktionen):

```
Let Identity(E ::TYPE) ::TYPE = E
```

Der Operator *Identity* bildet den als Parameter übergebenen Typ auf sich selbst ab.

Wie im vorigen Abschnitt bereits erwähnt, soll jetzt ein Typoperator *List* betrachtet werden, der einen Typ E auf den Typ einer Liste mit Elementen des Typs E abbildet. Dazu wird in die Listendefinition des vorigen Abschnittes ein Typparameter eingeführt.

```
Let List(E ::TYPE) ::TYPE =
  Rec(L)
    Option
      nil
      cons with head :E tail :L end
    end
```

Der Typoperator *List* bildet den Typ E auf einen rekursiven Optionstyp L ab, wobei die erste Komponente vom Typ E und die zweite vom Typ L ist [5]. Die notwendigen Listenoperationen können dann durch polymorphe Funktionen implementiert werden.

```
let new(E ::TYPE) :List(E) =
  option nil of List(E) end

let cons(E ::TYPE head :E tail :List(E)) :List(E) =
  option cons of List(E) with head tail end
```

Mit der polymorphen Funktion *new* können leere Listen beliebigen Typs erzeugt werden, und über die Funktion *cons* können Elemente an den Listenanfang angefügt werden.

Die Konzepte der polymorphen Funktionen und der Typoperatoren ermöglichen die Verwendung von generischem Code sowohl bei der Beschreibung von Strukturen als auch bei der Beschreibung von Verhalten.

3.3.4 Abstrakte Datentypen

Ein Abstrakter Datentyp (ADT) besteht aus einem Datentyp und einem Satz von Operationen auf diesem Datentyp. Dabei wird dem Benutzer nur der Name des Typs und die Signaturen und Namen der Operationen zur Verfügung gestellt. Ein ADT verbirgt die Implementation des Typs und der Operationen nach außen. Die zur Verfügung gestellten Operationen sind die einzig möglichen und zulässigen auf dem abstrakten Datentyp. Dadurch werden die Elemente des ADTs vor unzulässiger Benutzung geschützt.

Da die Implementation verborgen bleibt, kann sie nachträglich geändert werden, ohne dadurch Programme zu invalidieren, die den ADT benutzen. Auch kann, falls mehrere Implementationen eines ADT zur Verfügung stehen, zwischen diesen dynamisch gewählt werden.

Als Beispiel sei hier ein allgemeiner, funktionaler Kellerspeicher mit Operationen auf demselben vorgestellt. Der Kellerspeicher wird in Form eines polymorphen, abstrakten Datentyps deklariert.

```
Let FunStack =
  Tuple
    T ::ALL(E ::TYPE) TYPE
    new(E ::TYPE) :T(E)
    empty(E ::TYPE stack :T(E)) :Bool
    push(E ::TYPE element :E stack :T(E)) :T(E)
    pop(E ::TYPE stack :T(E)) :T(E)
    top(E ::TYPE stack :T(E)) :E
  end
```

[5] Hierbei ist zu beachten, daß der Operator *List* einen Typ E auf einen rekursiven Typ abbildet. Der eigentliche Typoperator ist jedoch nicht rekursiv. Rekursive Typoperatoren sind in *Quest* nicht zulässig, da sie zu Entscheidbarkeitsproblemen führen.

Zu dieser Schnittstelle wird im folgenden eine auf dem Modul *list*[6] basierende Implementation vorgestellt.

```
let listStack :FunStack =
  tuple
    Let T(E ::TYPE) ::TYPE = list.T(E)
    let new(E ::TYPE) :T(E) = list.new(:E)
    let empty(E ::TYPE stack :T(E)) :Bool = list.empty(stack)
    let push(E ::TYPE element :E stack :T(E)) :T(E) =
      list.cons(element stack)
    let pop(E ::TYPE stack :T(E)) :T(E) = list.tail(stack)
    let top(E ::TYPE stack :T(E)) :E = list.head(stack)
  end
```

Man sieht, daß in diesem Beispiel die Operationen *new*, *empty*, *push*, *pop* und *top* als polymorphe Funktionen implementiert wurden, sodaß Kellerspeicher für beliebige Datentypen erzeugt werden können.

3.4 Subtypisierung und Subtyppolymorphismus

In *Quest* gibt es eine Subtypbeziehung (<:) zwischen Typen. Sie ist wie folgt definiert: Wenn x vom Typ A ist und es gilt $A <: B$ (A ist Subtyp von B), dann ist x auch vom Typ B. Die Intuition hinter dieser Subtypisierung ist Mengeninklusion.

Die Menge aller Subtypen eines Typs T wird mit *POWER(T)* bezeichnet. $A <: B$ hat damit die gleiche Bedeutung wie *A ::POWER(B)* und wird als abkürzende Schreibweise für diesen Ausdruck betrachtet.

Zwischen den Basistypen gibt es keine Subtypbeziehungen außer den trivialen. Für strukturierte Typen sind die Subtypbeziehungen induktiv definiert. Dazu gibt es für alle Typoperatoren Subtypisierungsregeln. Aus Platzgründen wird an dieser Stelle jedoch nur die Subtypisierung bei Tupel- und Optionstypen betrachtet:

Tupel: Ein Tupeltyp A ist Subtyp eines Tupeltyps B, wenn A mindestens alle Attribute von B enthält oder genauer, wenn für alle Komponenten von B gilt: Die Namen der i-ten Komponente in A und B stimmen überein und der Typ der i-ten Komponente in A muß ein Subtyp der i-ten Komponente in B sein. Da die Komponenten von Tupeln geordnet sind, kommt es dabei auf die Reihenfolge der Komponenten an. So ist in folgendem Beispiel der Typ *Employee* ein Subtyp des Typs *Person*, der Typ *Student* jedoch nicht:

```
Let Person = Tuple name :String age :Int end
Let Employee = Tuple name :String age :Int company :String end
Let Student = Tuple name :String matrNr :Int age :Int end
```

[6] *list* ist ein Modul der Standardbibliothek, das eine polymorphe Liste mit dazugehörigen Operationen anbietet.

Optionen: Ein Optionstyp A ist Subtyp eines Optionstyps B, wenn B mindestens alle Varianten von A enthält oder genauer, wenn für alle Komponenten von A gilt: Die Namen der i-ten Komponente in A und B stimmen überein und der Typ der i-ten Komponente in A muß ein Subtyp der i-ten Komponente in B sein (im Sinne der Subtypisierung bei Tupeln). Auch die Komponenten von Optionen sind geordnet. Es kommt also auch hier auf die Reihenfolge der Komponenten an.

Der Typ *WeekDay* in folgendem Beispiel ist ein Subtyp vom Typ *Day*. Es gilt die Beziehung *WeekDay* **<:** *Day*.

```
Let Day =
  Option mon tue wed thu fri sat sun end

Let WeekDay =
  Option mon tue wed thu fri end
```

Ein großer Vorteil der Subtypisierung ist die Tatsache, daß Funktionen, die auf einem Typ arbeiten, auch Werte jedes beliebigen Subtyps dieses Typs akzeptieren. Subtypisierung erleichtert somit die nachträgliche Erweiterung von Programmen, insbesondere die Erweiterung von Datenstrukturen um neue Komponenten. Funktionen, die für den ursprünglichen Typ geschrieben worden sind, arbeiten dann auch auf Werten des neuen Typs, weil sie als Instanzen des alten Typs erkannt werden.

Folgendes Beispiel zeigt die Anwendung, aber auch die Begrenzungen dieses Konzeptes. Seien folgende Tupel definiert:

```
let florian :Person =
  tuple
    let name = "Florian"
    let age = 27
  end
```

```
let gabi :Employee =
  tuple
    let name = "Gabi"
    let age = 29
    let company = "ibm"
  end
```

Es wird jetzt eine Funktion *show* betrachtet, die den Inhalt eines Personentupels anzeigen kann.

```
let show(p :Person) :Person = p
show(florian)
tuple name = "Florian" age = 27 end :Person
```

Da der Typ *Employee* ein Subtyp des Typs *Person* ist, kann man dieser Funktion auch ein Angestelltentupel als Parameter übergeben.

```
show(gabi)
tuple name = "Gabi" age = 29 end :Person
```

Allerdings ist der Rückgabewert der Funktion vom Typ *Person*. Es geht also Typinformation verloren, was zur Folge hat, daß die Firma von Gabi nicht ausgegeben wird. Damit auch dieses Attribut bei der Ausgabe berücksichtigt wird, ist es notwendig, die Funktion *show* als polymorphe Funktion zu implementieren.

let *show2(A <:Person a :A) :A = a*

show2(gabi)
tuple *name = "Gabi" age = 29 company = "ibm"* **end** *:Employee*

Da genügend Typinformation in die Funktionsdefinition integriert wurde, werden nun alle Attribute ausgegeben. Durch die Angabe von $A <:$ *Person* wird die Funktion *show* zu einer polymorphen Funktion, wobei jedoch der Polymorphismus im Gegensatz zum parametrischen Polymorphismus auf Subtypen des Typs *Person* beschränkt wird. Diesen auf Subtypen eines Typs eingeschränkten Polymorphismus bezeichnet man als *Subtyppolymorphismus*.

3.5 Imperative Programmierung

Die imperative Programmierung basiert auf veränderbaren Werten in einem globalem Speicher. Der Kontrollfluß wird durch Konstrukte für Sequenzen und Schleifen gesteuert.

3.5.1 Veränderbare Variablen

Die Bindung eines Namens an einen Wert durch das *let*-Konstrukt entspricht der Definition einer Konstanten, da der dem Namen zugewiesene Wert nicht durch eine Zuweisung modifiziert werden kann. Wird an einen bereits existierenden Namen mit *let* ein neuer Wert gebunden, so bewirkt dies die Erzeugung einer neuen Variablen (eigentlich Konstanten).

Soll eine Variable modifizierbar sein, so muß sie bei der Bindung explizit als modifizierbar gekennzeichnet werden. Dies geschieht durch Angabe des Schlüsselworts *var* bei der Definition.

let *a = 7*
let var *b = 9*
b := b + 4

Nur für die Variable *b* ist eine Zuweisung möglich. Eine Zuweisung an die Variable a führt bei der Übersetzung zu einem Typfehler, da modifizierbare Werte nicht vom Typ T, sondern von einem speziellen Typ *Var(T)* sind.

Es ist zu beachten, daß in der gegenwärtigen Implementation von *Quest* für beliebige Typen *T* weder globale Variablen vom Typ *Var(T)* innerhalb von Funktionen verändert werden können, noch Werte vom Typ *Var(T)* als Parameter von Funktionen zugelassen sind. In beiden Fällen, muß vorher eine Einbettung der modifizierbaren Größe in eine Struktur, z.B. in ein Tupel, durchgeführt werden [7].

[7] Ein Beispiel hierfür findet sich bei der Definition des modifizierbaren Kellerspeichers in Abschnitt 3.6.

3.5.2 Sequenzen und Schleifen

Eine Sequenz ist eine Zusammenfassung von Ausdrücken. Der Typ einer Sequenz ist der Typ des letzten Ausdrucks in der Sequenz. Sequenzen werden durch die Schlüsselworte *begin* und *end* zu Blöcken zusammengefaßt.

Schleifen fassen Folgen von Ausdrücken zum Zweck der Iteration zusammen. Der Typ einer solchen Schleife ist *Ok*. Die allgemeinste Form ist die *loop*-Schleife. Zusätzlich gibt es noch zwei speziellere Formen von Schleifen, die *while*-Schleifen und die *for*-Schleifen. Als Beispiel für Sequenzen und Schleifen sei hier eine Funktion angegeben, die den größten gemeinsamen Teiler von zwei ganzen Zahlen bestimmt.

```
let gcd(n,m :Int) :Int =
  begin
    let var vn = n
    and var vm = m
    while vn isnot vm do
      if vn > vm then vn := vn - vm end
      if vn < vm then vm := vm - vn end
    end
    vn
  end
```

Typ und Wert der Sequenz ergeben sich aus ihrem letzen Ausdruck, in diesem Fall *vn*.

3.5.3 Der Datentyp *Array*

Ein Feld (*Array*) ist eine durch nicht negative ganze Zahlen indizierte Sequenz von Elementen gleichen Typs. Die Größe des Felds wird bei der Erzeugung festgelegt und kann danach nicht mehr geändert werden. Die Elemente des Felds hingegen stellen modifizierbare Werte dar.

```
let a:Array(Int) = array of 0 1 2 3 4 5 end
let b:Array(Bool) = array of (5 false)
b[0] := {a[0] is 0}
```

Es gibt zwei Möglichkeiten, ein Feld zu initialisieren: Beim sechselementigen Feld a sind die Elemente einzeln aufgeführt und in *array of* und *end* eingeschlossen. Die zweite Möglichkeit ist die Angabe der Anzahl der Elemente und des Initialisierungswertes, so geschehen beim Feld *b*. Die einzelnen Elemente können, wie aus anderen Programmiersprachen bekannt, durch einem Wert in eckigen Klammern indiziert werden.

Mit Hilfe der *for*-Schleife kann eine Summenfunktion für beliebig große Felder ganzer Zahlen definiert werden.

```
let sum(a :Array(Int)) :Int =
  begin
    let var total = 0
```

```
    for i = 0 upto extent(a) - 1 do
      total := total + a[i]
    end
    total
  end
```

Mit dem Operator *extent* kann die Größe von Feldern ermittelt werden. Die Schleifenvariable *i* muß nicht deklariert werden.

Sind Felder Parameter von Funktionen, so kann eine abkürzende Schreibweise benutzt werden.

sum **of** *1 2 3 4* **end**	statt *sum*(**array of** *1 2 3 4* **end**)
sum **of** *(5 2)*	statt *sum*(**array of** *(5 2)*)

3.6 Module und Schnittstellen

Die Modularisierung ist nach den Funktionen die wichtigste Strukturierungsmöglichkeit moderner Programmiersprachen. *Quest* bietet ähnlich wie Modula-2 [Wir85] die Möglichkeit, große Programme in Schnittstellenmodule (*interfaces*) und Implementationsmodule (*modules*) zu unterteilen. In den Schnittstellenmodulen werden die Bezeichner und Typen der Variablen, die Bezeichner und Kinds der abstrakten Datentypen und die Bezeichner und Signaturen der Funktionen und Typoperatoren, die von den Moduln implementiert und exportiert werden, deklariert.

Zusätzlich kann ein Schnittstellenmodul Typdefinitionen enthalten. Diese werden nicht wie normale Typdefinitionen mit *Let*, sondern mit dem Schlüsselwort *Def* eingeleitet. Die Definition dieser Typen ist, anders als die abstrakten Datentypen, für alle Benutzer der Schnittstelle sichtbar.

In *Quest* können zu einem Schnittstellenmodul im Gegensatz zu Modula-2 beliebig viele Implementationsmodule existieren. Module sind, wie Funktionen, Objekte *erster Klasse*, was bedeutet, daß sie an Bezeichner gebunden und als Parameter an Funktionen übergeben werden können.

Schnittstellen- und Implementationsmodule müssen auf dem *Top-Level* der interaktiven Programmierumgebung definiert werden, wobei dadurch implizit externe Dateien, die den Schnittstellen- beziehungsweise Modulcode enthalten, erzeugt werden.

Beide Modularten können ihrerseits von anderen Modulen importieren. Dies geschieht mittels des Schlüsselwortes *import*. Als Beispiel wird noch einmal ein Kellerspeicher betrachtet. Im Gegensatz zu dem in Abschnitt 3.3.4 vorgestellten wurde hier aber ein modifizierbarer Kellerspeicher gewählt.

```
interface Stack
export
  T(E ::TYPE) ::TYPE
  new(E ::TYPE) :T(E)
```

```
    empty(E ::TYPE stack :T(E)) :Bool
    push(E ::TYPE element :E stack :T(E)) :Ok
    pop(E ::TYPE stack :T(E)) :Ok
    top(E ::TYPE stack :T(E)) :E
end
```

Ein zugehöriges Implementationsmodul, das wie die in Abschnitt 3.3.4 vorgestellte Implementation eine Realisierung des Stacks basierend auf Listen verwendet, könnte dann wie folgt aussehen:

```
module stack :Stack
import list:List
export
  Let T(E ::TYPE) ::TYPE = Tuple var l :list.T(E) end
  let new(E ::TYPE) :T(E) =
    tuple let var l = list.new(:E) end
  let empty(E ::TYPE stack :T(E)) :Bool = list.empty(stack.l)
  let push(E ::TYPE element :E stack :T(E)) :Ok =
    stack.l := list.cons(element stack.l)
  let pop(E ::TYPE stack :T(E)) :Ok =
    stack.l := list.tail(stack.l)
  let top(E ::TYPE stack :T(E)):E = list.head(stack.l)
end
```

Das Modul kann seinerseits von anderen Modulen importiert werden. Nach einem Import in ein Programm oder Modul kann mit der Punktnotation auf die Modulkomponenten zugegriffen werden.

```
module main: Main
import stack :Stack print :Print
export
  let s = stack.new(:Int)
  stack.push(7 s)
  if not stack.empty(s) then
    print.Int(stack.top(s))
  end
end;
```

3.7 Ausnahmebehandlung

Wie die Modularisierung dient auch die Ausnahmebehandlung (*Exception handling*) der Strukturierung großer Programme. Sie ist ein Kontrollflußmechanismus, der sich in *Quest* aber problemlos in das Typkonzept einfügt.

Die Ausnahmebehandlung ist ein wichtiges Konzept in einer Programmiersprache, weil bereits einfache Funktionen, wie z.B das Teilen durch Null, Fehlersituationen erzeugen, die

abgefangen werden müssen. Neben diesen Standardausnahmen unterstützt *Quest* auch benutzerdefinierte Ausnahmen.

Dafür gibt es einen speziellen Typoperator *Exception*, mit dem man verschiedene Typen für Ausnahmen definieren kann. Eine Ausnahme vom Typ *Exception(T)* kann, wenn sie ausgelöst wird, einen Wert vom Typ T als Parameter erhalten.

Ähnlich wie beim Typ *Tuple* werden Werte vom Typ *Exception(T)* durch ein Schlüsselwort, und zwar *exception*, eingeleitet und durch *end* abgeschlossen:

```
let exc1 : Exception(Int) =
    exception integerException :Int  end
let exc2 : Exception(String) =
    exception stringException :String  end
```

Eine Ausnahme wird mit dem *raise*-Befehl ausgelöst. Dabei kann, wie oben bereits erwähnt, ein Wert vom entsprechenden Typ als Parameter übergeben werden.

```
raise exc1 with 3 end
```

Weiterhin bietet *Quest* die Möglichkeit Ausnahmen abzufangen. Dazu wird das *try*-Konstrukt verwendet:

```
try
    ...
    raise exc1 with 55 end
    ...
    when exc1 with x then x + 5
    when exc2 then 1
    else 0
end
```

Die einzelnen Zweige des *try*-Konstrukts dienen dazu, spezielle Ausnahmen, hier *exc1* und *exc2*, zu behandeln. Für die jeweilige Ausnahme wird die genannte Operation ausgeführt. Der mit der Ausnahme erzeugte Wert kann hierbei verwendet werden. Ausnahmen, die in keinem Zweig auftreten, werden vom *else*-Zweig behandelt.

Falls kein solcher existiert, wird die Ausnahme weiter propagiert, bis sie auf ein passendes *try*-Konstrukt trifft oder der Top-Level erreicht ist. In diesem Fall wird die Ausnahme nach außen hin sichtbar.

3.8 Persistenz

Es gibt in der Sprache *P-Quest* zwei Ansätze, um Daten persistent zu machen.

Dynamische Datentypen Das Modul *Dynamic* bietet Funktionen an, mit denen Daten mit ihrer Typinformation auf Dateien geschrieben werden können.

Objektspeicher Beim Übergang von *Quest* zu *P-Quest* wurde das System um das Konzept eines Objektspeicher erweitert, der es ermöglicht, Daten vollständig transparent persistent zu speichern.

3.8.1 Dynamische Datentypen

Die *Quest* Standardbibliothek enthält ein Modul *Dynamic*, das den abstrakten Datentyp *dynamic.T* sowie eine Menge von Funktionen auf diesem Datentyp zur Verfügung stellt. Der Typ *dynamic.T* und die dazugehörigen Operationen können dazu verwendet werden, Daten ohne Verlust von Typinformation persistent zu speichern.

Der Typ *dynamic.T* kann als ein Tupel interpretiert werden, bei dem in der ersten Komponente der Typ eines Datums und in der zweiten der Wert gespeichert wird. Nachfolgendes Beispiel demonstriert, wie Daten in eine externe Datei exportiert werden können. (Die Funktionen der Module *writer* und *reader* werden dabei zum Erzeugen bzw. Öffnen einer Datei benötigt.)

```
let wr = writer.file("Test.qst")
let dyn = dynamic.new(:Int 3)
dynamic.extern(wr dyn dynamic.portable)
writer.close(wr)
```

Der Parameter *dynamic.portable* legt die Formatierung der exportierten Daten fest.

In einer nachfolgenden Sitzung kann dieses Datum wie folgt in die interaktive Umgebung importiert und dort verwendet werden.

```
let addone(x :Int) = x + 1
let rd = reader.file("Test.qst")
let dyn = dynamic.intern(rd dynamic.portable)
reader.close(rd)
addone(dynamic.be(:Int dyn))
```

Über die Funktion *dynamic.be* wird eine Typüberprüfung initiiert. Stimmt der Typ, der der Funktion *dynamic.be* als erster Parameter übergeben wird, nicht mit dem in dem Bezeichner *rd* gespeicherten Typ überein, kommt es zu einem Laufzeitfehler. Im anderen Fall wird der Wert des Bezeichners *dyn* an den Aufrufer übergeben.

Darüberhinaus können diese Funktionen auch dazu benutzt werden, übersetzte Funktionen oder *Quest*-Programme persistent zu speichern, was im nachfolgenden Beispiel demonstriert wird.

```
let wr = writer.file("sqr.qst")
let hello() :Ok = print("Hello world")
dynamic.extern(wr dynamic.new(:All() :Ok hello) dynamic.portable)
writer.close(wr)
```

3.8.2 Der Objektspeicher von *P-Quest*

In *P-Quest* gibt es keine Unterscheidung zwischen persistenten und temporären Daten. Jedes Objekt kann persistent gespeichert werden. Das Kriterium für die Persistenz ist die Erreichbarkeit vom Hauptprogramm aus. Bei diesem Hauptprogramm handelt es sich meistens um die interaktive Compilerumgebung (Es sind jedoch auch Applikationsprogramme möglich.). Wenn das Hauptprogramm die Compilerumgebung ist, können alle benannten Top-Level-Objekte und alle von diesen aus (auch transitiv) erreichbaren Objekte persistent gemacht werden. Mögliche Top-Level-Objekte sind Werte, Typen, Kinds, aber auch ganze Schnittstellen und Module. Um diese Objekte persistent zu machen, muß eine explizite Stabilisierung des Objektspeichers durchgeführt werden.

Die Funktion

store.stabilise()

stabilisiert den Speicherinhalt und

store.halt()

stabilisiert den Speicherinhalt und beendet sofort die Programmausführung.

Die Befehle können auch dazu benutzt werden, um in Programmen Sicherungspunkte (checkpoints) anzulegen. Sie bewirken, daß die vom Programm aus zum Zeitpunkt des Aufrufs erreichbaren Objekte persistent gemacht werden. Wenn das *P-Quest*-System durch Control-D verlassen wird oder es zu einem Absturz des Systems kommt, werden alle Änderungen, die seit dem letzten Stabilisieren an Objekten des persistenten Speicher durchgeführt worden sind, rückgängig gemacht.

Der Aufruf des System durch den Befehl *PQuestRecover* bewirkt dann eine Wiederaufnahme der Ausführung des Programms beim nächstem Befehl hinter dem letzten *store.stabilise* bzw. *store.halt*. Bei einen Aufruf von *PQuest* wird das Hauptprogramm neu gestartet.

Es existiert keine Operation für das explizite Löschen von Objekten aus dem persistenten Speicher. Stattdessen gibt es eine Funktion, die alle Objekte ermittelt, die nicht mehr erreichbar sind, und den von ihnen belegten Speicherplatz wieder freigibt (*Garbage Collection*).

Dieser Vorgang kann zu jedem Zeitpunkt vom Benutzer durch einen expliziten Aufruf folgender Funktion initialisiert werden.

store.garbageCollect()

Außerdem wird der Vorgang automatisch gestartet, wenn es zu einem Überlauf des persistenten Speichers kommt.

Kapitel 4

Eine prototypische Bibliothek generischer Dienste

Die meisten traditionellen Systeme für datenintensive Anwendungen sind *Built-In* Ansätze (vgl. Abschnitt 1.2). Es liegt ihnen ein festes Datenmodell zugrunde Zur Realisierung dieses Daternmodells sind eine oder mehrere Arten von Bulk-Typen (z.B. Relationen, Klassen) zur Speicherung von Massendaten ebenso wie eine Reihe von Konzepten zur Unterstützung datenintensiver Anwendungen fest eingebaut.

In diesem Kapitel wird ein *Add-On*-Ansatz zur Unterstützung datenintensiver Anwendungen vorgestellt. Die Unterstützung für datenintensive Anwendungen wird dabei durch eine Dienstbibliothek realisiert. In diesem Kapitel wird die Implementierung einer prototypischen Bibliothek generischer Dienste beschrieben, die eine Umgebung für datenintensive Anwendungen zur Verfügung stellt. Der Schwerpunkt liegt dabei auf dem Entwurf der Schnittstellen.

Das Kapitel ist wie folgt strukturiert: Im ersten Abschnitt wird auf die Architektur der prototypishen Implementierung eingegangen. In den weiteren Abschnitten werden die in der Bibliothek realisierten Dienste beschrieben: Thema des Abschnitt 4.2 ist das Konzept der Iterationsabstraktion und eine mögliche mögliche, generische Realisierung in einem *Add-On*-Ansatz. Es werden Funktionen zur Iterationsabstraktion vorgestellt, die z.B. die Formulierung einer Anfragesprache für beliebige Bulk-Typen erlaubt. Abschnitt 4.3 behandelt einen Ansatz zur transaktionsorientierten Fehlererholung und dessen Einbettung in eine allgemeine Transaktionsverwaltung. In Abschnitt 4.4 wird ein dynamischer Ansatz zur transaktionsorientierte Integritätsüberwachung vorgestellt, der auf einer Verwaltung der Integritätsbedingungen in Kollektionen beruht. Der letzte Abschnitt dieses Kapitels geht auf weitergehende Ansätze im Bereich der Transaktionsverwaltung, Fehlererholung und Integritätsüberwachnubg ein. Dabei wird sowohl die Implementierung eines Transaktionsgenerators als auch die Realisierung von geschachtelten Transaktionen untersucht.

4.1 Eine offene Architektur für datenintensive Anwendungen

In Abschnitt 1.2 werden Architekturen von *Add-On-* und *Built-In-*Ansätzen miteinander verglichen. Dabei zeigt sich, daß sich *Add-On-*Architekturen durch höherer Flexibilität und Erweiterbarkeit auszeichnen. Die hier vorgestellte Archtiktur ist in mehreren Hinsichten offen. Sie ist offen für:

Erweiterungen: Es wird eine prototypische Bibliothek vorgestellt, die für datenintensive Anwendungen benötigte Dienste in generischer Form zur Verfügung stellt. Beispiele für solche Dienste sind Transaktionsverwaltung und Integritätsüberwachung. Keiner der Dienste ist eingebaut und damit als unveränderlich festgeschrieben. Die Dienste können den Bedürfnissen des Benutzers durch Veränderung von Modulen der Bibliothek oder durch Hinzufügen neuer Module und Schnittstellen angepaßt werden.

Neue Datenmodelle: Bei dem hier gewählten Ansatz sind Datenmodelle nicht fest eingebaut, sondern werden in Form von Schnittstellen zur Verfügung gestellt. Der Benutzer kann also neue Datenmodelle implementieren, indem er eine oder mehrere Schnittstellen und die zugehörigen Implementierungen entwirft. Da die Bibliothek die Dienste in modellunabhängiger Form zur Verfügung stellt, kann er dabei die Dienste der Bibliothek verwenden. Außerdem wird er durch die sprachliche Mächtigkeit der Sprache *P-Quest* unterstützt.

Bei der Entwicklung neuer Datenmodelle stehen neue, effizientere Technologien und mächtige Datenmodelle den bereits existierenden Anwendungen mit z.T. erheblichen Datenbestäden gegenüber. Wenn aus Gründen der Effizienz oder wegen neu aufgetretener Anforderungen ein Übergang zu einem neuen Datenmodell notwendig wird, stellt dies in der Regel ein großes Problem dar (vgl. [Bro92, Kis92]), besonders wenn bereits sehr große Datenbestände existieren und längere Ausfälle der Datenbank aufgrund der Umstellung nicht in Kauf genommen werden können. Der hier gewählte Ansatz bietet einen einheitlichen Rahmen für solche Übergänge. Das neue Datenmodell kann in der selben Umgebung implementiert werden wie das alte. Dadurch ist ein expliziter Zugriff auf die alte Datenbank möglich. Es können Funktionen für die Umwandlung der Datenbestände geschrieben werden oder auch für eine Übergangszeit mit einer Kombination aus beiden Datenmodellen gearbeitet werden.

Externe Dienste: Es besteht die Möglichkeit, die Funktionalität von Schnittstellen der Bibliothek durch externe Module wie z.B. C-Programme oder gemeinsam genutzte Bibliotheken (*Shared Libraries*) zu realisieren. Dies ist besonders in Hinblick auf eine mögliche Erweiterung in Richtung verteilter Umgebungen von Interesse (Zugriff auf externe Dienstbringer).

Der Rest des Abschnitts ist wie folgt aufgebaut: Der erste Teil gibt einen Überblick über die Architektur des Prototyps. Im zweiten Teil des Abschnitts wird eine Abstimmung der Schnittstellen auf die verschiedenen Benutzergruppen diskutiert.

4.1.1 Die Architektur des Prototyps

Ziel des Prototyps ist es, einen Rahmen für die Implementierung von Datenmodellen unter Berücksichtigung von Integritätsüberwachung und Fehlererholung zu schaffen. Von zentraler Bedeutung in einem Datenmodell sind die Bulk-Typen. Deshalb liegt in der Arbeit die Betonung darauf, eine Umgebung zur Verfügung zu stellen, in der der Entwurf von Bulk-Typen mit transaktionsorientierter Fehlererholung und Integritätsüberwachung einfach möglich ist. Außerdem soll die Definition einer Anfragesprache auf dem Bulk-Typ durch die Dienste erleichtert werden. Die Umgebung, die der Prototyp zur Verfügung stellt, kann aber auch zur Definition einzelner Objekte mit Fehlererholung und Integritätsüberwachung verwendet werden. Ein weiterer Schwerpunkt der zur Verfügung gestellten Dienste ist die Unterstützung der Definition konsistenzerhaltender Transaktionen mit Fehlererholung. Zu diesem Zweck wird unter anderem auch ein Transaktionsgenerator zur Verfügung gestellt.

Die Arbeit liefert zwei Formen der Unterstützung: Zum einen werden generische Dienste in Form von Schnittstellen angeboten. Die Funktionen der Schnittstellen können beim Entwurf neuer Bulk-Typen und bei der Erstellung von Transaktionen verwendet werden. Zum anderen werden Anleitungen und Beispiele dafür gegeben, wie z.B. beim Entwurf neuer Bulk-Typen vorgegangen werden kann. Abbildung 4.1 gibt einen groben Überblick über die im Prototyp

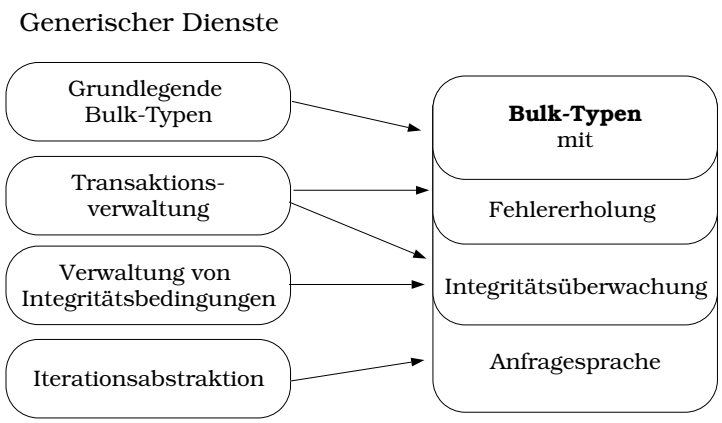

Abbildung 4.1: Die generischen Dienste

enthaltenen generischen Dienste und ihre Verwendung zur Definition eines Bulk-Typs mit Fehlererholung, Integritätsüberwachung und einer Anfragesprache. Bei der Definition eines neuen Bulk-Typs wird in der Regel auf einen bereits existierenden, grundlegenden Bulk-Typ (wie z.B. Listen, Mengen) zurückgegriffen, dessen Funktionalität in Form von einer Schnittstelle zur Verfügung steht. Die Fehlererholung im Falle eines Abbruchs einer Transaktion wird von der Transaktionsverwaltung unterstützt. Diese liefert auch Unterstützung bei der Integritätsüberwachung, und zwar beim Test der verzögerten Integritätsbedingungen. Der Dienst, der die Verwaltung von Integritätsbedingungen zur Verfügung stellt, ermöglicht eine einfache Implementierung einer Integritätsüberwachung. Die Funktionen zur Iterationsabstraktion erlauben die Definition einer Anfragesprache für einen beliebigen Bulk-Typ.

Abbildung 4.2 gibt einen Überblick über die Schnittstellen des Prototyps, die diese Dien-

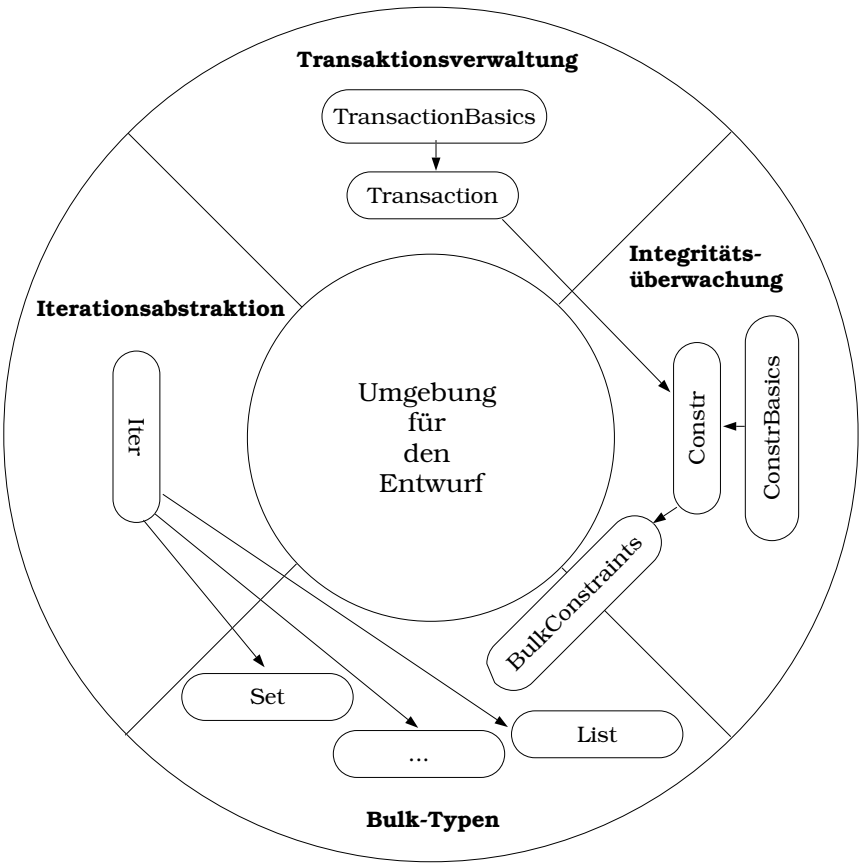

Abbildung 4.2: Die Schnittstellen des Prototyps

ste zur Verfügung stellen. Dabei sind Kästen mit abgerundeten Ecken die Schnittstellen des Prototyps und die Pfeile stellen die wichtigsten Modulabhängigkeiten dar. Im folgenden wird kurz auf die Schnittstellen eingegangen. Zu jeder Schnittstelle wird dabei angegeben, in welchem Teil der Arbeit sich eine genauere Beschreibung ihrer Funktionalität und der implementierten Konzepte findet. Zur Benennung der Schnittstellen ist folgendes zu bemerken: Die Schnittstellen, deren Namen auf *Basics* enden, z.B. *TransactionBasics*, stellen die grundlegendere Funktionalität zur Verfügung. Sie sind meist weniger komfortabel und sicher, dafür aber flexibler (siehe dazu auch Abschnitt 4.1.2). Die Schnittstellen gleichen Namens ohne den Namenszusatz *Basics* bieten in der Regel speziellere Funktionen an. Ihre Implementierung beruht meist auf den Funktionen der darunterliegenden *Basics*-Schnittstelle.

Grundlegende Bulk-Typen

In der Standardumgebung von *P-Quest* existiert bereits eine Bibliothek von Schnittstellen für die grundlegenden Bulk-Typen, wie Listen und Mengen. Die Schnittstellen beinhalten unter anderem einen Typoperator, der einen Typ E auf den Typ der jeweiligen Kollektion mit Elementtyp E abbildet, und eine Operation zum Erzeugen von Kollektionen. Weiterhin liefert

die Schnittstelle Operationen zum Einfügen und Löschen von Elementen. Diese Operationen sind für verschiedene Arten von Bulk-Typen unterschiedlich, da sie die spezielle Semantik des jeweiligen Bulk-Typs beinhalten. Ein Beispiel dafür ist das Einfügen an einer festgelegten Stelle einer Liste. Außerdem enthält jede der Schnittstellen eine Funktion *elements* zur Erzeugung einer Iteration über die jeweilige Kollektion[1].

Iterationsabstraktion

Die Schnittstelle *Iter* stellt Funktionen zur Iterationsabstraktion zur Verfügung. Eine Beschreibung findet sich in Abschnitt 4.2.2. Die Funktionen dieser Schnittstelle ermöglichen unter anderem die Definition einer Anfragesprache für beliebige Bulk-Typen. Sie bilden außerdem eine Grundlage des Systems und werden an vielen Stellen von anderen Implementierungen verwendet.

Transaktionsverwaltung

Die Schnittstelle *TransactionBasics* stellt grundlegende Funktionen zur Transaktionsverwaltung zur Verfügung, unter anderem Funktionen zum Starten, Beenden und Abbrechen einer Transaktion. Die vorgestellte Transaktionsverwaltung unterstützt Fehlererholung, Integritätsüberwachung und geschachtelte Transaktionen. Dabei ist ein Konzept zur Fehlererholung implementiert, das darauf beruht, daß Operationen im Falle des Abbruchs oder Zurücksetzens einer Transaktion rückgängig gemacht werden (siehe Abschnitt 4.3.3). Die Integritätsüberwachung wird durch den Test verzögerter Integritätsbedingungen am Ende der Transaktion unterstützt (siehe Abschnitt 4.4.3). Die Transaktionsverwaltung erlaubt außerdem die Definition von geschachtelten Transaktionen. Dieses Thema wird in Abschnitt 4.5.3 behandelt.

Über der Schnittstelle *TransactionBasics* existiert noch eine Schnittstelle *Transaction*. Diese stellt einen Transaktionsgenerator, mit dem man aus Funktionen automatisch Transaktionen erzeugen kann, zur Verfügung (siehe Abschnitte 4.5.1 und 4.5.3).

Integritätsüberwachung

Grundlage der Integritätsüberwachung ist bei dem im Prototyp gewählten Ansatz die Verwaltung von an Operationen gebundenen Integritätsbedingungen. Es handelt sich dabei um einen dynamischen Ansatz: Jeder Operation ist eine veränderbare Kollektion von Integritätsbedingungen zugeordnet. Es können zu jeder Zeit Integritätsbedingungen in Kollektionen eingefügt und aus den Kollektionen gelöscht werden.

Die Schnitstelle *ConstrBasics* stellt den Dienst der Verwaltung von Kollektionen von Integritätsbedingungen zur Verfügung: Funktionen zum Einfügen und Löschen von Integritätsbedingungen und zum Test von Kollektionen von Integritätsbedingungen (siehe Abschnitt 4.4.2). Die darüberliegende Schnittstelle (*Constr*) bietet Funktionen zur Verwaltung von unmittelbar und verzögert zu testenden Bedingungen unter Berücksichtigung der transaktionsorientierten Integritätsüberwachnung an. Auf diese Schnittstelle wird im Abschnitt 4.4.3

[1] Auf diesen Ansatz wird in Abschnitt 4.2.1 genauer eingegangen.

eingegangen. Ein Teil der Integritätsüberwachung wird wie oben bereits erwähnt auch von der Transaktionsverwaltung übernommen. Speziell für die Verwaltung von Integritätsbedingungen bei Bulk-Typen existiert eine Schnittstelle *BulkConstraints*. Sie wird in Abschnitt 4.4.4 beschrieben.

Anleitungen und Beispiele

Neben den zur Verfügung gestellten Diensten werden auch Anleitungen dafür gegeben, wie man die Dienste für die Implementierung von Datenmodellen und Bulk-Typen nutzen kann. So enthält z.B. Abschnitt 4.3.6 Regeln für den Entwurf von sog. *geschützten* Operationen. Das sind Operationen, die die Transaktionsverwaltung im Falle des Abbruchs einer Transaktion automatisch zurücksetzt. Weiterhin werden in Abschnitt 4.4.4 Regeln für den Entwurf von konsistenzerhaltenden Operationen vorgestellt.

In Abschnitt 5 wird versucht, das *Object-Relationship*-Modell in der durch die Dienste zur Verfügung gestellten Umgebung zu implementieren[2]. Anhand dieses Modells soll die Eignung der Dienste des Prototyps an einem konkreten Beispiel getestet werden. Dabei wird unter anderem die Realisierung spezieller Integritätsbedingungen des Modells in dem allgemeinen zur Integritätsüberwachung geschaffenen Rahmen betrachtet.

Ein weiterer Zweck der Implementierung dieses Modells ist die Untersuchung der Frage, inwieweit man neuere Konzepte der Datenmodellierung (z.B. Objekt-Orientierung, Klassen, Beziehungen zwischen Klassen und Generalisierungshierarchien) in einem *Add-On*-Ansatz implementieren kann und welche Probleme dabei auftreten. Das gewählte Modell stellt jedoch nur ein Beispiel dar. Es ist denkbar, auch andere Datenmodelle in der geschaffenen Umgebung zu implementieren.

4.1.2 Abstimmung auf Benutzergruppen

Bei datenintensiven Anwendungen kann man eine Reihe von Benutzergruppen unterscheiden, die in Bezug auf Flexibilität, Sicherheit und die benötigten Dienste verschiedene Anforderungen an das System, d.h. an die von ihnen benutzten Schittstellen, haben. Es ist deshalb sinnvoll, für verschiedenen Benutzergruppen speziell auf deren Bedürfnisse abgestimmte Schnittstellen zur Verfügung zu stellen. Hier werden die Benutzer in die vier folgenden Gruppen eingeteilt: Endbenutzer, Anwendungsprogrammierer, Datenbankadministratoren und Designer neuer Datenmodelle.

Ein *Add-On* Ansatz zeichnet sich durch seine hohe Flexibilität aus (vgl. Abschnitt 1.2.1). Um die Sicherheit des Systems zu gewährleisten, ist es dabei wichtig, nicht jedem Benutzer jede Art der Änderung zu gewähren. Änderungsrechte werden im allgemeinen abhängig von der Zugehörigkeit zur Benutzergruppe vergeben. So hat z.B. ein Datenbankadministrator in der Regel mehr Rechte als ein Endbenutzer. Auch aus diesem Grund sollte es verschiedene Schnittstellen für verschiedene Benutzergruppen geben. Jeder Benutzergruppe sollte die von ihr benötigte Funktionalität zur Verfügung stehen, zugleich aber auch ein Schutz vor absichtlicher und versehentlicher unerlaubter Veränderung der Datenbank gewährleistet

[2]Bei dem *Object-Relationship*-Modell handelt es sich um ein objekt-orientiertes Modell mit einer expliziten Darstellung von Beziehungen [AGO91b, Rum87]

sein. Allgemein kann man die Regel aufstellen, daß sich Sicherheit und Benutzerfreundlichkeit der Schnittstellen entgegengesetzt zur Flexibilität und Modellunabhängigkeit der Schnittstellen entwickeln. Im folgenden wird kurz auf die Bedürfnisse der einzelnen Benutzergruppen eingegangen:

Endbenutzer: Der Endbenutzer greift in der Regel nicht direkt, sondern nur über Anwendungsprogramme (Transaktionen) auf eine Datenbank zu. Bei den Schnittstellen für Endbenutzer sind Benutzerfreundlichkeit, einfache Bedienung und Sicherheit besonders wichtig. Die Schnittstellen für den Benutzer sind nicht Teil der Bibliothek der generischen Dienste, sondern werden vom Anwendungsprogrammierer geschrieben. Es ist zu beachten, daß die Datenbank evtl. von einer großen Anzahl von Benutzern verwendet wird, die nichts voneinander wissen. In diesem Umfeld ist die Gewährleistung der Sicherheit vor unsachgemäßer Benutzung besonders wichtig. Zur Steigerung der Benutzerfreundlichkeit auch für Laien können grafische Schnittstellen eingesetzt werden.

Anwendungsprogrammierer: Der Anwendungsprogrammierer schreibt Anwendungsprogramme für die Benutzung einer Datenbank. Er arbeitet mit einem festen Schema und auch mit einem festen Datenmodell. Seine Programme führen Datenmanipulationen (Einfügen, Ändern, Löschen von Datensätzen) durch, stellen Anfragen an die Datenbank und verarbeiten Anfrageergebnisse. Die Schnittstellen für den Anwendungsprogrammierer sollten benutzerfreundlich und sicher sein, aber auch die notwendige Funktionalität zur Formulierung komplexer Anfragen und zur Erzeugung von Transaktionen zur Verfügung stellen. Modellunabhängigkeit ist auf dieser Ebene nicht erforderlich.

Datenbankadministrator: In der Regel arbeitet der Datenbankadministrator mit einem festen Datenmodell. Seine Aufgabe ist die Definition neuer Datenbanken, die Wartung bestehender Datenbanken und ihre Anpassung an neue Gegebenheiten in der modellierten Anwendung. Es müssen ihm also Funktionen zur Schemadefinition und zur Schemamodifikation (Einfügen neuer Klassen, Hinzufügen neuer Attribute u.ä.) zur Verfügung stehen. Außerdem sollte er die Möglichkeit haben, Integritätsbedingungen einzufügen und zu löschen. Zu Administrationszwecken sollte er Zugriff auf Metadaten wie z.B. die Menge aller definierter Klassen haben.

Bei diesen Schnittstellen ist hohe Flexibilität wichtig, sodaß alle notwendige Funktionalität zur Verfügung steht. Der Datenbankadministrator kann Änderungen mit sehr weitreichenden Folgen durchführen.

Designer: Aufgabe des Designers ist die Implementierung neuer Datenmodelle in der Umgebung der generischen Bibliothek. Für diese Aufgabe ist es wichtig, daß die Dienste in modellunabhängiger Form angeboten werden. Der Designer benötigt ein breites Spektrum an Funktionalität, für das in der Regel eine große Anzahl von Funktionen oder auch Funktionen mit vielen Parametern in Kauf genommen werden müssen. Dies führt zu einer Abnahme der Benutzerfreundlichkeit. Wünschenswert sind auch Anleitungen für die Vorgehensweise beim Entwurf neuer Datenmodelle, wie sie etwa in Abschnitten 4.3.6 und 4.4.4 in Form von Regeln und im Abschnitt 5 in Form einer Beschreibung der exemplarischen Implementierung eines Datenmodells zur Verfügung gestellt werden.

Die Betonung dieses Ansatzes soll jedoch nicht auf der strikten Aufteilung in die einzelnen Benutzergruppen liegen. Ein Add-On Ansatz zeichnet sich vielmehr dadurch aus, daß die

Übergänge zwischen den Benutzergruppen fließend sind: Alle Benutzergruppen arbeiten in einem homogenen sprachlichen Rahmen.

4.2 Iterationsabstraktion durch Funktionen höherer Ordnung

4.2.1 Iterationsabstraktion in einem *Add-On*-Ansatz

Die meisten der existierenden Datenbankprogrammiersprachen, wie z.B. DBPL [SEM88] und Galileo [AGOO88] sind *Bulit-In* Ansätze (vgl Abschnitt 1.2). Sie verfügen über einen festen Satz von vordefinierten Bulk-Typen zur Realisierung der Datenbanken, wie z.B. Relationen in DBPL oder Klassen in Galileo. Außerdem existiert meist eine Reihe von Operatoren und Konstrukten auf diesen Bulk-Typen, die typische Anforderungen datenintensiver Anwendungen wie die Definition von Anfragen und Sichten unterstützen. Diese haben in der Regel deklarativen Charakter, d.h., es wird von den Details der Iteration über die Elemente und der Implementierung abstrahiert.

Bei einem *Add-On*-Ansatz sind die Bulk-Typen nicht eingebaut, sondern werden als Teil einer Bibliothek angeboten. Weiterhin besteht die Möglichkeit, daß der Benutzer neue Bulk-Typen definiert, die seinen speziellen Anforderungen angepaßt sind. Wie beim *Built-In*-Ansatz braucht man auch hier Unterstützung für typische DB-Aufgaben. Man benötigt also Dienste, die z.B. Konstrukte zur Iterationsabstraktion und zur deklarativen Formulierung von Anfragen anbieten. Diese Dienste sollen folgende Anforderungen erfüllen:

- **Geringer Aufwand:** Die Dienste sollen für jeden Bulk-Typ zur Verfügung stehen. Der Aufwand bei der Definition eines neuen Bulk-Typs soll möglichst gering sein.
- **Abstraktion:** Die angebotenen Dienste sollen möglichst einheitlich sein und abstrahieren
  - vom Elementtyp der Kollektion
  - von der Struktur der Kollektion
  - und von Details der Anwendung

  Diese Abstraktion dient der Reduktion der Anzahl der benötigten Funktionen, der Übersichtlichkeit und durch einheitliche Schnittstellen der Benutzerfreundlichkeit.
- **Kombinierbarkeit:** Es soll möglich sein, verschieden strukturierte Kollektionen zu kombinieren, z.B., indem man eine Menge und eine Liste von Elementen gleichen Typs miteinander verknüpft.
- **Erweiterbarkeit:** Die Dienste sollen erweiterbar sein, wenn sich herausstellt, daß noch weitere häufig benötigte Dienste existieren.

Abbildung 4.3 zeigt einen Ansatz zur Lösung dieser Aufgabe. Im Mittelpunkt dieses Ansatzes steht die Definition eines einheitlichen Formats für die Iteration über die Elemente einer Kollektion. Dieses einheitliche Format wird im folgenden als *Iterator* bezeichnet. Bei der hier gewählten Implementierung des Ansatzes wird der Iterator durch den Typoperator *Iter_T* realisiert. In Abschnitt 4.2.3 wird eine mögliche Definition für diesen Typoperator angegeben. Zu jedem Bulk-Typ existiert eine Funktion *elements*[3], die Kollektionen dieses Typs auf

[3]Dies ist für jeden Bulk-Typ eine eigene, unterschiedlich implementierte Funktion.

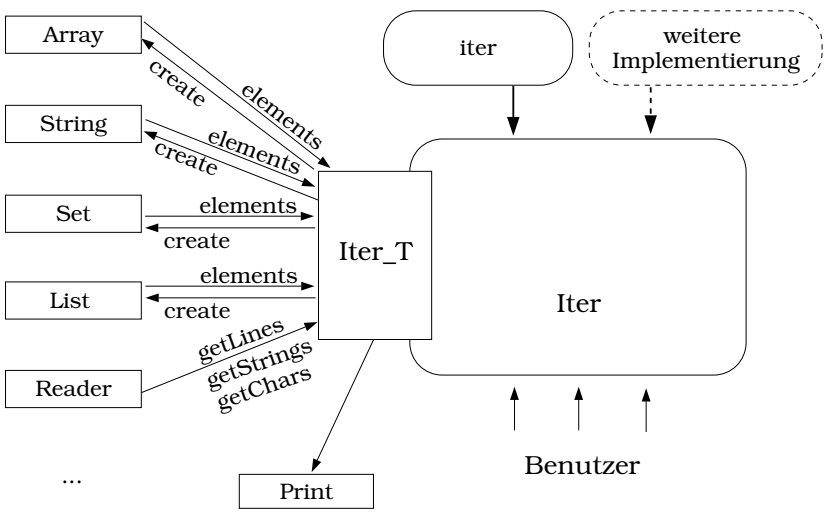

Abbildung 4.3: Iterationsabstraktion

Iteratoren abbildet. Zum Lesen geöffnete Dateien (*Reader*) können durch die Funktionen *getLines*, *getStrings* und *getChars* auf verschiedene Iteratoren abgebildet werden, auf Iteratoren über ihre Zeilen, ihre Zeichenketten oder ihre Buchstaben.

Es gibt eine Schnittstelle *Iter*, die eine Reihe von Funktionen zur höheren Iterationsabstraktion anbietet. Diese stellen die gewünschte deklarative Funktionalität zur Verfügung. Die Funktionen arbeiten nicht direkt auf den Kollektionen, sondern auf den Iteratoren. Es handelt sich dabei größtenteils um generische Funktionen höherer Ordnung. Die angebotenen Funktionen und mögliche Anwendungen im Datenbankbereich werden in Abschnitt 4.2.2 vorgestellt. Das Modul *iter* ist eine sequentielle Implementierung der Schnittstelle *Iter*. Diese Implementierung wird im Abschnitt 4.2.3 beschrieben. Man kann sich aber auch noch andere (effizientere) Implementierungen der Schnittstelle *Iter* vorstellen. Es gibt noch weitere Funktionen, die auf den Iteratoren arbeiten: Analog zu den Funktionen *elements* kann man auch zu jedem Bulk-Typ eine Funktion *create* definieren, die aus einem Iterator eine Kollektion dieses Bulk-Typs erzeugt. Außerdem existiert eine Funktion zur formatierten Ausgabe von Iteratoren.

Es ist jetzt noch zu zeigen, daß der Ansatz die oben aufgestellten Anforderungen erfüllt: Der Aufwand bei der Definition neuer Bulk-Typen ist gering. Man muß lediglich eine Funktion *elements* für die Abbildung auf den Iterator definieren. Danach können die Funktionen der Schnittstelle *Iter* benutzt werden. Diese Funktionen sind generisch und arbeiten dadurch auf beliebigen Elementtypen. Die Abstraktion vom Elementtyp ist also gegeben. Außerdem handelt es sich um Funktionen höherer Ordnung, die Funktionen als Parameter erhalten. Dies ermöglicht die Abstraktion von Details, die spezifisch für den jeweilgen Elementtyp bzw. die jeweilige Anwendung sind. Diese Details können durch die Funktionen, die man als Parameter übergibt, festgelegt werden. So gibt es z.B. keine speziellen Funktionen zur Berechnung von Aggregationsfunktionen wie *min*, *max* oder *sum*. Es existieren vielmehr nur Funktionen für die Reduktion eines Iterators auf einen Wert. Die Art, wie die Elemente dazu miteinander verknüpft werden, wird durch die als Parameter übergebene Funktion festgelegt. Da

die Funktionen auf den Iteratoren arbeiten, können sie auf alle Bulk-Typen angewandt werden. Insbesondere ist es auch möglich, Kollektionen verschiedener Struktur z.B. Relationen und Listen miteinander zu kombinieren, indem sie zunächst beide auf Iteratoren abgebildet werden. Auch die Erweiterbarkeit ist sichergestellt. Die Funktionen zur höheren Iterationsabstraktion sind nicht eingebaut, sondern werden als Dienste in Form der Schnittstelle *Iter* zur Verfügung gestellt. Man kann also die bestehenden Schnittstellen erweitern oder zusätzliche Schnittstellen mit Funktionen schreiben, die auf Iteratoren arbeiten.

4.2.2 Höhere Iterationsabstraktion

In der Literatur [OBBT89, AM87, SDDS86] existiert bereits eine Reihe von Vorschlägen für Funktionen zur Iterationsabstraktion für spezielle Bulk-Typen. Bei der Zusammenstellung der Funktionen zur höheren Iterationsabstraktion wird sich an diesen Vorschlägen orientiert. Außerdem wird versucht, die besonderen Anforderungen datenintensiver Anwendungen zu berücksichtigen. Die so ausgewählten Funktionen wurden zu einer prototypischen Schnittstelle *Iter* zusammengefaßt. In einer Testphase wurde die Schnittstelle in den verschiedensten Gebieten eingesetzt und gemäß der Ergebnisse dieser Testphase modifiziert und erweitert.

Die Funktionen der Schnittstelle *Iter*

Die Funktionen zur höheren Iterationsabstraktion werden von der Schnittstelle *Iter* angeboten. Wie bereits erwähnt, arbeiten sie nicht direkt auf den Kollektionen, sondern auf Iteratoren (= Werte der Struktur *Iter_T*) mit beliebigem Elementtyp. Außerdem sind die Funktionen möglichst allgemein gehalten, damit sie in vielen Situationen anwendbar sind. Spezielles Verhalten wird in Form von Funktionen als Parameter übergeben. Da die Anzahl der Funktionen der Schnittstelle trotzdem recht groß ist, wird hier nicht jede dieser Funktionen einzeln vorgestellt, sondern es wird jeweils auf Gruppen von Funktionen mit ähnlichen Eigenschaften eingegangen[4]. Die Einteilung in die Gruppen ist nicht eindeutig, ein Teil der Funktionen fällt in mehrere der Gruppen.

Abbildung von Iteratoren auf Iteratoren: Diese Gruppe enthält zum einen Funktionen, die einen Iterator in einen Iterator mit einem anderen Elementtyp umformen. Dazu wird eine Funktion übergeben, die die Abbildung zwischen den Elementtypen definiert. Der Iterator wird dann elementweise abgebildet. *map* ist ein Beispiel für eine solche Funktion:

let *partNames = iter.map(elements(parts)* **fun**(*p :Part*) :**String** *p.name)*

Eine weitere Art der Abbildung eines Iterators ist die Veränderung der Reihenfolge seiner Elemente. Die Funktion *reverse* z.B. bildet einen Iterator auf einen Iterator mit den selben Elementen aber in umgekehrter Reihenfolge ab.

Selektion von Elementen: Diese Funktionen können nach dem Kriterium für die Selektion der Elemente in zwei Gruppen aufgeteilt werden: Bei den Funktionen der ersten

[4]Eine komplette Auflistung der Funktionen mit ihren Signaturen findet sich in Anhang B

Gruppe wird die Selektion über ein Prädikat ausgeführt. Das Prädikat wird als Parameter übergeben. Die zweite Gruppe der Funktionen führt die Selektion abhängig von der Position bzw. Reihenfolge der Elemente durch. Beide Gruppen enthalten sowohl Funktionen, die ein einzelnes Element eines Iterators selektieren, als auch solche, die eine Reihe von Elementen selektieren und diese in Form eines Iterators zurückgeben. Ein Beispiel aus dieser Gruppe ist die Funktion *select*. Sie selektiert alle Elemente eines Iterators, die ein übergebenes Prädikat erfüllen. Sie kann z.B. zur Berechnug der Teile, die mehr als 100 Dollar kosten, verwendet werden:

let *expensiveParts = iter.select(elements(baseParts)* **fun***(p :BasePart) p.cost > 100)*

Verknüpfung von Iteratoren: Zwei oder mehr Iteratoren können auf verschiedene Weisen miteinander verknüpft werden. Man kann dabei zwischen der Vereinigung von Iteratoren und der elementweisen Verknüpfung unterscheiden. Bei der Vereinigung von Iteratoren wird ein neuer Iterator erzeugt, der die Elemente aller übergebenen Iteratoren enthält. Die zu verknüpfenden Iteratoren müssen dafür den gleichen Elementtyp besitzen. Die Funktionen dieser Gruppe unterscheiden sich durch die Reihenfolge, in der die Elemente im Ergebnis-Iterator auftreten. Ein Beispiele für diese Funktionen ist die Funktion *append*, die zwei Iteratoren aneinanderhängt:

let *allParts = iter.append(elements(parts1) elements(parts2))*

Bei der elementweisen Verknüpfung werden die einzelnen Elemente der Iteratoren miteinander verknüpft, d.h., es entsteht ein Iterator, der andere Elemente enthält als die Iteratoren, von denen ausgegangen wurde. Die zu verknüpfenden Iteratoren können verschiedene Elementtypen haben. Ein Beispiel für diese Funktionen ist *overlay*. Sie hat folgende Signatur:

overlay(E1, E2, F ::TYPE
iter1 :T(E1) iter2 :T(E2) f(:E1 :E2) :F
unit1 :E1 unit2 :E2) :T(F)

Mit Hilfe der Funktion f werden die beiden übergebenen Iteratoren elementweise miteinander verknüpft und es entsteht ein Iterator mit Elementtyp F. Für den Fall, daß die Iteratoren nicht gleich lang sind, erhält die Funktion *overlay* noch die Werte vom Typ *unit1 :E1* und *unit2 :F* als Parameter. Der kürzere Iterator wird dann durch Elemente des entsprechenden Werts verlängert.

Reduktion von Iteratoren auf ein Element: Die Reduktion des Iterators wird bei einem Teil der Funktionen dieser Gruppe durch Verknüpfung der Elemente mit Hilfe einer als Parameter übergebenen Funktion durchgeführt. Dabei wird das erste Element mit dem zweiten und dann das dritte mit dem Ergebnis der Verknüpfung der ersten beiden verknüpft und so weiter. *Iter* bietet Funktionen an, die die Verknüpfung direkt auf den Elementen des Iterators ausführen, und solche, bei denen die Elemente zunächst durch eine zusätzlich als Parameter übergebene Funktion abgebildet und die Ergebnisse dieser Abbildung miteinander verknüpft werden. Diese Funktionen eignen sich zum Beispiel für die Berechnung von Aggregationsfunktionen. Eine Funktion dieser Gruppe ist die Funktion *fold*. Sie erhält zwei Funktionen und einen Iterator als Parameter. Folgendes Beispiels zeigt eine mögliche Verwendung dieser Funktion:

```
let maxPno = iter.fold(elements(parts)
                fun(p :Part) :Int p.pno
                int.max)
```

Die erste übergebene Funktion bildet jedes Teil auf seine Nummer ab. Die zweite Funktion verknüpft jeweils zwei Nummern so miteinander, daß die höhere Nummer das Ergebnis der Verknüpfung ist. Auf diese Weise berechnet *fold* die höchste in *parts* verwendete Nummer.

Eine andere mögliche Reduktion eines Iterators ist die Reduktion auf einen booleschen Wert. Dies geschieht mit Hilfe eines Prädikats. Es wird überprüft, ob alle Elemente des Iterators das Prädikat erfüllen (*all*), bzw., ob es ein Element im Iterator gibt, das es erfüllt (*some*). Der entsprechende boolesche Wert wird zurückgegeben. Diese Funktionen entsprechen einer universellen bzw. existentiellen Quantifizierung:

```
let positive = iter.all(elements(baseParts) fun(p :BaseParts) p.cost > 0)
```

Iteration mit Seiteneffekten Diese Gruppe enthält die Schleifenkonstrukte zur Iteration über die Elemente des Iterators. Dabei werden die üblichen Schleifenkonstrukte wie z.B. *forEach*, *forDo*, *while*, *until* etc. angeboten. Als Parameter wird die Schleifenbedingung und der Schleifenrumpf in Form einer Funktion des Typs **All**(:*E*) :**Ok** übergeben. Diese Funktionen eignen sich z.B. für die Definition von Ausgabefunktionen:

```
forEach(elements(parts) fun(p :Part) print.string(p.name))
```

Inspektion und Konstruktion Es existiert eine Reihe von Funktionen zur Inspektion von Iteratoren. Darunter befinden sich Funktionen zum Test auf Gleichheit von Iteratoren, zum Ermitteln der Länge und des ersten Elements eines Iterators und für Enthaltenseins-Tests. Weiterhin gibt es auch Funktionen zur Konstruktion von Iteratoren. In dieser Gruppe sind unter anderem Funktionen zum Erzeugen von leeren Iteratoren und zum Einfügen von Elementen in Iteratoren enthalten. Als Beispiel für diese Gruppe wird die Funktion *equal* für den Test der Gleichheit zweier Iteratoren betrachtet. Sie erhält zwei Iteratoren und eine Funktion, die die Gleichheit der Elemente definiert, als Parameter und testet die Gleichheit der Iteratoren elementweise:

```
iter.equal(elements(parts1) elements(parts2)
        fun(p1, p2 :Part) p1.pno is p2.pno ∧
                string.equal(p1.name p2.name))
```

Anwendungen

Im folgenden werden einige Beispiele für die Verwendung der im vorigen Abschnitt beschriebenen Funktionen gegeben. Eine wichtige Aufgabe im Datenbankbereich ist die Formulierung von Anfragen. Für diese Aufgabe wird die Datenselektion, die Datenprojektion und die Datenkombination (*Join*) benötigt [Sch87]. Diese drei Funktionalitäten werden durch Funktionen der Schnittstelle *Iter* zur Verfügung gestellt. Die Datenprojektion kann durch die Funktion

map ausgedrückt werden. Zur Darstellung der Datenselektion kann die Funktion *select* verwendet werden. Es gibt aber auch spezielle Funktionen zur Formulierung von Anfragen. Die Funktion *get* erlaubt die Formulierung von Anfragen über einen Iterator.

*get(E, F::***TYPE** *select(:E) :F from :T(E) where(:E) :***Bool***) :T(F)*

Dabei ist *select* die Funktion zur Datenprojektion und *where* das Prädikat zur Datenselektion. Die Benennung der Parameter orientiert sich an der Syntax einer *SQL*-Anfrage [Dat89]. Bei Anfragen über mehrere Iteratoren kommt zusätzlich der Aspekt der Datenkombination hinzu. *get2* dient der Formulierung von Anfragen über zwei Iteratoren:

*get2(E1, E2, F ::***TYPE**
select(:E1 :E2):F
from1 :T(E1)
from2 :T(E2)
*where(:E1 :E2) :***Bool***) :T(F)*

Man kann z.B. die Paare von Namen des Lieferanten und Namen des Teils bestimmen, für die gilt, daß der Lieferant dieses Teil liefert:

let *partSupplierNames = get2(*
let *select(p :BasePart s :Supplier) =* **tuple** *p.name s.name* **end**
let *from1 = elements(baseParts)*
let *from2 = elements(suppliers)*
let *where(p :BasePart s :Supplier) = p.supplier* **is** *s.sno)*

Die Funktion *get2* führt einen *Join* über die Iteratoren *from1* und *from2* durch. Dabei legt das Prädikat *where* fest, welche Elemente aus dem Kreuzprodukt der beiden Iteratoren ins Ergebnis aufgenommen werden, und *select* führt wiederum die Projektion auf die gewünschten Komponenten durch. Dabei ist zu beachten, daß man mit *get* Anfragen auf beliebigen Bulk-Typen formulieren kann und mit *get2* einen *Join* über verschieden strukturierte Kollektionen durchführen kann. Wie man sieht, hängt die Anzahl der Typparameter der Funktion und das Format der übergebenen Funktionen von der Anzahl der beteiligten Iteratoren ab. Es müssen also für verschiedene Anzahlen von Iteratoren *i* auch verschiedene Funktionen *geti* geschrieben werden. Im Prototyp wird auf eine Implementierung der Funktionen für drei und mehr Iteratoren verzichtet.

Mit den gleichen Funktionen wie Anfragen können auch typisierte Sichten definiert werden. Eine weitere häufige Aufgabenstellung im Bereich der datenintensiven Anwendungen ist die Berechnung von Aggregationsfunktionen. Wie bereits im vorigen Teil des Abschnitts erwähnt, eignen sich dazu die Funktionen zur Reduktion von Iteratoren auf Elemente wie z.B. *fold*. Die Funktionen der Schnittstelle können aber auch in vielen anderen Bereichen angewandt werden, in denen mit Bulk-Typen programmiert wird (siehe z.B. die Integritätsüberwachung in Abschnitt 4.4.2). Dabei eignen sie sich besonders zur schnellen Erstellung von Prototypen.

4.2.3 Eine sequentielle Implementierung

Die allgemeinsten Iteratoren sind solche, die den sequentiellen Zugriff auf die Elemente verwenden. Es wird nur vorausgesetzt, daß es eine Reihenfolge auf den Elementen gibt. Da die Reihenfolge nicht näher spezifiziert sein muß, stellt diese Forderung keine Einschränkung dar: Die Kollektion der Elemente liegt in einer Programmiersprache vor, und somit existiert auch eine Reihenfolge auf den Elementen. Iteratoren dieses Typs haben den Vorteil, daß sie sich für sehr viele, verschieden strukturierte Kollektionen von Elementen erzeugen lassen. Allerdings nutzen sie eventuell existierende Strukturinformation wie z.B. Sortierung der Elemente oder wahlfreien Zugriff nicht aus. Daraus kann sich ein Mangel an Effizienz ergeben. Im Prototyp werden die Iteratoren und die Funktionen der Schnittstelle *Iter* sequentiell implementiert. Der erste Teil dieses Abschnitts beschreibt einen Typ *Iter_T* für sequentielle Iteratoren und die Erzeugung von Iteratoren dieses Typs. Der zweite Teil geht auf die Implementierung der Funktionen der Schnittstelle *Iter* ein.

Die Erzeugung sequentieller Iteratoren

Wie bereits erwähnt, wird für den Iterator eine sequentielle Implementierung gewählt. Es existiert ein Typoperator *Iter_T*, der einen Typ E auf den Typ eines Iterators mit Elementtyp E abbildet:

```
Def T(E ::TYPE) ::TYPE = Rec(Iter)
  Tuple
    empty() :Bool
    get() :E
    rest() :Iter
  end
```

Iter_T(E) ist ein abstrakter Datentyp. Iteratoren dieses Typs haben einen internen Zustand, auf den nur über die Funktionen *empty*, *get* und *rest* zugegriffen werden kann. Dabei testet *empty*, ob der Iterator leer ist, *get* liefert das erste Element des Iterators zurück und *rest* liefert einen neuen Iterator zurück, der aus dem alten Iterator ohne das erste Element besteht. *get* und *rest* sind nur für nicht leere Iteratoren definiert. Alle drei Funktionen belassen den ursprünglichen Iterator unverändert. Das dritte Element eines Iterators *it* z.B. erhält man durch den Aufruf:

```
it.rest().rest().get()
```

Der Iterator wird immer nur soweit berechnet, wie er gerade gebraucht wird (*Laziness*). Die Funktion *rest* ist nur eine Vorschrift zur Berechnung der weiteren Elemente des Iterators, nicht aber die tatsächliche Kollektion. Dadurch ist auch die Betrachtung unendlicher Iteratoren möglich.

Zu jedem neuen Bulk-Typ wird eine Funktion *elements* definiert, die zu einer Kollektion dieses Bulk-Typs mit dem Elementtyp E einen Iterator vom Typ *Iter_T(E)* berechnet. Die Definition einer Funktion *elements* besteht aus der Definition der drei Funktionen *empty*, *get*

und *rest* für die jeweilige Struktur. Für den Bulk-Typ *Array* kann die Definition der Funktion z.B. wie folgt aussehen:

```
let elements(E ::TYPE a :Array(E)) :Iter_T(E) =
    begin
      let rec elementsFrom(index :Int) : Iter_T(E) =
          tuple
            let empty() :Bool = index > size(a)-1
            let get() :E = a[index]
            let rest() : Iter_T(E) = elementsFrom(index+1)
          end
      elementsFrom(0)
    end
```

Die lokale Funktion *elementsFrom* definiert den Iterator über die Elemente des Felds ab dem als Parameter übergebenen Index.

Das Modul *iter*

Das Modul *iter* ist eine sequentielle Implementierung der Schnittstelle *Iter*. Die Funktionen in *iter* benutzen nur den sequentiellen Zugriff auf die Elemente. Sie arbeiten auf Iteratoren, deren Typ mit dem im vorigen Abschnitt eingeführten Typoperator *Iter_T* gebildet wurde. Es soll hier nicht auf die Implementierung jeder einzelnen Funktion eingegangen werden, sondern anhand von drei typischen Beispielen die prinzipielle Vorgehensweise erläutert werden.

Die Funktion *all* liefert einen booleschen Wert zurück:

```
let rec all(E ::TYPE iter :T(E) p(:E):Bool) :Bool =
  if iter.empty() then true
  elsif p(iter.get()) then all(iter.rest() p)
  else false
  end
```

Sie ist ein Beispiel für eine Funktion, die einen Iterator auf einen einzelnen Wert reduziert. Diese Funktionen unterscheiden meist die Fälle, ob der Iterator leer ist oder nicht. Für leere Iteratoren wird eine geeignete Aktion, hier die Rückgabe des Werts *true* durchgeführt. Ansonsten wird das übergebene Prädikat für das erste Element des Iterators getestet und abhängig vom Ergebnis fortgefahren. Dabei enthält meist mindestens eine Alternative einen rekursiver Aufruf der Funktion für den Rest des Iterators.

map ist ein Beispiel für eine Funktion, die einen Iterator als Ergebnis zurückliefert:

```
let rec map(E, F ::TYPE iter :T(E) f(:E):F) :T(F) =
    tuple
      let empty = iter.empty
      let get():F = f(iter.get())
      let rest() :T(F) = map(iter.rest() f)
    end
```

Solche Funktionen erzeugen einen neuen Wert vom Typ *Iter_T(F)*. Dazu werden die Funktionen *empty*, *get* und *rest* definiert. Dies geschieht mit Hilfe des bzw. der als Parameter übergebenen Iteratoren und Funktionen. Die Vorschriften zur Berechnung des ersten Elements und des Rests des Iterators werden auf neue Vorschriften abgebildet. Da die Komponente *rest* wiederum vom Typ *Iter_T* ist, besteht ihre Berechnung meist aus einem rekursiven Aufruf der Funktion.

Ein Beispiel für Iterationen mit Seiteneffekten ist die Funktion *whileDo*:

```
let rec whileDo(E ::TYPE iter :T(E) p(:E):Bool statement(:E):Ok) :Ok =
    if not(iter.empty()) andif p(iter.get()) then
      statement(iter.get())
      whileDo(iter.rest() p statement)
    end
```

Funktionen dieser Art bestehen in der Regel aus einem Test, ob das Schleifenprädikat für das erste Element erfüllt ist, aus einem Aufruf des als Parameter übergebenen Schleifenrumpfs für dieses Element und aus einem rekursiven Aufruf der Funktion für den Rest des Iterators.

4.3 Transaktionsorientierte Fehlererholung

Transaktionen spielen eine wichtige Rolle für die Erhaltung der Konsistenz von Datenbanken. Sie lassen sich nach Reuter [Reu87] wie folgt definieren:

Eine **Transaktion** *ist eine Folge von Operationen, welche die Datenbank in ununterbrechbarer Weise von einem konsistenten Zustand in einen (nicht notwendig verschiedenen) konsistenten Zustand überführt.*

Härder und Reuter fassen in [HR83] die wesentlichen Eigenschaften einer Transaktion zu dem inzwischen allgemein anerkannten *ACID-Prinzip* der Transaktionen zusammen:

- Ununterbrechbarkeit (**A**tomicity)
- Konsistenzerhaltung (**C**onsistency)
- Isolierter Ablauf (**I**solation)
- Dauerhaftigkeit der Ergebnisse (**D**urability)

In dieser Arbeit wird untersucht, inwieweit man ein Transaktionskonzept durch einen *Add-On*-Ansatz realisieren kann. Dabei wird die Realisierung der Eigenschaften Ununterbrechbarkeit und Dauerhaftigkeit in diesem Abschnitt untersucht. Die Frage der Konsistenzerhaltung wird im Abschnitt 4.4 behandelt. Der isolierte Ablauf spielt bei dem hier gewählten Ansatz keine Rolle, da keine parallelen Abläufe betrachtet werden.

Die Eigenschaft der Ununterbrechbarkeit besagt, daß eine Transaktion entweder ganz oder gar nicht ausgeführt werden soll. Die Eigenschaft der Dauerhaftigkeit fordert, daß nach erfolgreichem Abschluß einer Transaktion ihre Ergebnisse für alle Zeiten erhalten bleiben sollen. Ein wichtiger Aspekt in diesem Zusammenhang ist die Frage, welche Maßnahmen notwendig sind, um diese Eigenschaften sicherzustellen, falls während oder nach Beendigung der Transaktion Fehler auftreten. Reuter [Reu87] nennt vier verschiedene Fehlererholungsmaßnahmen, die in einem Datenbanksystem mindestens notwendig sind.

R1-Recovery: Im normalen Betrieb können Transaktionen scheitern z.B. aufgrund des Auftretens von Ausnahmen oder wegen der Verletzung semantischer Integritätsbedingungen. Deshalb müssen Vorkehrungen getroffen werden, die das Zurücksetzen unvollständiger Transaktionen erlauben (*partielles Zurücksetzen*).

R2-Recovery: Nach einem Systemausfall ist der Zustand, den die Fehlererholungsmaßnahmen herstellen sollen, zum einen dadurch definiert, daß er die Effekte aller bis zum Zeitpunkt des Systemausfalls beendeten Transaktionen enthalten muß. Dazu ist es beim Wiederanlauf des Systems evtl. erforderlich einen Teil der schon beendeten Transaktionen nachzuvollziehen (*partielles Wiederholen*).

R3-Recovery: Zum anderen ist der herzustellende Zustand auch dadurch definiert, daß die Datenbank keine Effekte von Transaktionen enthalten darf, die zum Zeitpunkt des Ausfalls noch nicht beendet waren. Um dies sicherzustellen, ist es evtl. erforderlich die Effekte solcher Transaktionen aus der Datenbank zu entfernen (*vollständiges Zurücksetzen*).

R4-Recovery: Wenn es zu einer physischen Zerstörung der Datenbank kommt, z.B. durch einen Plattenfehler, muß der jüngste transaktionskonsistente Zustand der Datenbank wiederhergestellt werden. Dazu wird eine Archivkopie der Datenbank oder des zerstörten Teils der Datenbank geladen. Danach müssen die Effekte aller seit dem Erstellen dieser Kopie beendeten Transaktionen nachgetragen werden (*vollständiges Wiederholen*).

Im folgenden werden nur die Maßnahmen *R1* bis *R3* behandelt, wobei der Schwerpunkt auf der *R1-Recovery* liegt. Die *R4-Recovery* kann von einem *Add-On*-Ansatz für Transaktionen nicht geleistet werden. Sie ist Aufgabe der tieferliegenden Schichten. Methoden hierfür werden z.B. in [Wei88] beschrieben.

Der Rest dieses Abschnitts ist wie folgt aufgeteilt: Zunächst werden ad-hoc Lösungen für die *Add-On*-Realisierung von Transaktionen vorgestellt. Dabei wird der Nutzen moderner Programmiersprachenkonzepte für einen solchen Ansatz untersucht. Gleichzeitig wird die Fehlererholung durch Verwaltung eines *Undo-Logs* motiviert. Die Beschreibung dieses Ansatzes (Abschnitt 4.3.3) und einer möglichen Realisierung im Rahmen einer *Add-On*-Transaktionsverwaltung ist der Inhalt der weiteren Teile des Abschnitts. Dabei wird in 4.3.2 zunächst untersucht, wie eine Transaktion aufgebaut ist. In 4.3.4 wird auf eine Einbettung des gewählten Ansatzes in eine Transaktionsverwaltung eingegangen. Hierbei wird auch die Frage der Dauerhaftigkeit der Ergebnisse betrachtet. In 4.3.5 wird beschrieben, welche Anforderungen in diesem Rahmen an die Operationen einer Transaktion gestellt werden müssen (Einführung des Konzepts der geschützten Operationen). Im letzten Teil des Abschnitts wird eine Anleitung zum Entwurf solcher geschützter Operationen gegeben.

4.3.1 Transaktionen in einem *Add-On*-Ansatz

Im folgenden soll die Frage behandelt werden, wie man ein Transaktionskonzept oder zumindest Teile davon in einem *Add-On*-Ansatz realisieren kann. Dabei liegt der Schwerpunkt auf dem Aspekt der Fehlererholung. Als Einstieg in das Thema wird folgendes Problem betrachtet: Für eine Prozedur bestehend aus einer Folge von Operationen soll die Ununterbrechbarkeit gewährleistet werden. Zur Lösung dieses Problems werden ad-hoc Ansätze vorgestellt. Dabei wird von einem naiven Ansatz ausgegangen, der anhand eines Beispiels schrittweise weiterentwickelt wird. Insbesondere wird dabei untersucht, welche Konzepte einer modernen Programmiersprache bei dieser Aufgabe von Nutzen sind und zu eleganteren Lösungen führen.

Naiver Ansatz: Am Anfang der Prozedur werden alle Bedingungen getestet, die zu einem Scheitern einer der Operationen führen können. Die Operationen werden nur dann ausgeführt, wenn all diese Bedingungen erfüllt sind:

```
let p() =
  if cond1 ∧ cond2 ∧ cond3 then
   op1
   op2
   op3
  end
```

Bei diesem Ansatz tritt folgendes Problem auf: Operation *op1* verändert die Umgebung, in der Operation *op2* ausgeführt wird, d.h., eine für *op2* notwendige Bedingung kann, obwohl sie am Anfang der Prozedur erfüllt ist, nach der Ausführung von *op1* verletzt sein.

Das Zurücksetzen von Operationen: Vor jeder Operation werden alle Bedingungen getestet, die erfüllt sein müssen, damit diese Operation nicht scheitern kann. Falls sie erfüllt sind, wird die Operation ausgeführt. Ansonsten müssen alle Operationen, die bis zu diesem Zeitpunkt in der Prozedur ausgeführt worden sind, wieder zurückgesetzt werden.

```
let p() =
   if cond1 then
    op1
    if cond2 then
      op2
      if cond3 then
        op3
      else
        undoOp2
        undoOp1
      end
    else
      undoOp1
    end
   end
end
```

Dazu muß zu jeder Operation eine sogenannte kompensierende Operation (z.B. *undoOp1*) entworfen werden, die den Effekt der zugehörigen Operation rückgängig macht. Bei einem solchen Ansatz ist es empfehlenswert, die Operationen in umgekehrter Reihenfolge ihrer Ausführung zurückzusetzen. Durch diese Vorgehensweise ist die Umgebung der kompensierenden Operation zum Zeitpunkt ihrer Ausführung wieder im selben Zustand wie direkt nach der Ausführung der zugehörigen Operation[5]. Dieser Umstand erleichtert den Entwurf der kompensierenden Operationen.

Die Verwendung einer Ausnahmebhandlung: Wenn die verwendete Sprache die Behandlung von Ausnahmen und insbesondere auch deren geschachtelte Behandlung unterstützt, kann der Kontrollfluß eleganter gesteuert werden. Im Gegensatz zu den ersten beiden Ansätzen wird darauf verzichtet, Bedingungen zu testen, die zum Scheitern von Operationen führen können. Wenn die Operation scheitert, kann dies durch die Ausnahmebehandlung abgefangen und entsprechend (mit dem Zurücksetzen von Operationen) darauf reagiert werden.

```
let p()=
   try op1
```

[5] Alle danach ausgeführten Änderungen wurden bereits rückgängig gemacht.

```
    try op2
      try op3
      else
        undoOp2
        undoOp1
    else
      undoOp1
    end
  end
```

In *P-Quest* dient das *try*-Konstrukt zur Ausnahmebehandlung.

Der Einsatz Funktionen höherer Ordnung Unterstützt die Programmiersprache neben der Ausnahmebehandlung auch das Konzept der Funktionsvariablen und der Funktionen höherer Ordnung, so ist eine noch kompaktere Lösung möglich. Man definiert eine Funktionsvariable *undo*, die mit der leeren Operation (hier **ok**) initialisiert wird.

```
let var undo() :Ok = ok
```

Außerdem wird eine Funktion höherer Ordnung *addUndo* definiert, die eine Funktion als Parameter erhält und diese mit dem aktuellen Wert der globalen Variablen *undo* verknüpft:

```
let addUndo(f1() :Ok) :Ok =
    undo := begin  f1() undo() end
```

Die neue Funktion *undo* besteht aus der Hintereinanderausführung der übergebenen Funktion und der bisherigen Funktion *undo*. Mit Hilfe dieser Definitionen kann das Problem wie folgt gelöst werden:

```
let p()=
    try
      op1
      addUndo(undoOp1)
      try
        op2
        addUndo(undoOp2)
        try
          op3
          addUndo(undoOp3)
        else
          undo()
        end
      else
        undo()
      end
    else
      undo()
    end
```

Durch die Aufrufe der Funktion *addUndo* werden die kompensierenden Operationen in die Funktion *undo* eingefügt. Man beachte, daß durch die Wahl der Reihenfolge der Funktionen bei der Definition von *addUndo* sichergestellt ist, daß die zuletzt ausgeführte Operation zuerst zurückgesetzt wird. Die Funktion *undo* arbeitet ähnlich wie ein Kellerspeicher für Funktionen.

Da in allen Fällen durch einen Aufruf der Funktion *undo* auf das Erzeugen einer Ausnahme reagiert wird, kann auf eine Schachtelung der Ausnahmebehandlung verzichtet werden:

```
let p() =
    try
     op1
     addUndo(undoOp1)
     op2
     addUndo(undoOp2)
     op3
     addUndo(undoOp3)
    else
     undo()
    end
```

Man überlegt sich leicht, daß *undo* auch hier genau die Operationen zurücksetzt, die zum Zeitpunkt der Ausnahme bereits ausgeführt worden sind. Wie man sieht, hat sich durch die Verwendung von Konzepten moderner Programmiersprachen ein erheblich vereinfachter Kontrollfluß ergeben.

Beim letzten der vorgestellten Ansätze ergibt sich die Notwendigkeit, kompensierende Operationen zu verwalten. Dies kann wie hier durch den schrittweisen Aufbau der Funktion *undo* geschehen oder auch durch das Einfügen der kompensierenden Operationen in eine Kollektion, die im Falle eines Zurücksetzens abgearbeitet wird. Um dem Benutzer die Aufgabe zu erleichtern, bietet es sich an, die Verwaltung der kompensierenden Operationen als Dienst zur Verfügung zu stellen. Die Weiterentwicklung dieser Idee führt zum Ansatz der Fehlererholung durch die Verwaltung eines *Undo-Logs*, der in den folgenden Abschnitten vorgestellt wird.

4.3.2 Aufbau der Transaktionen

Um eine transaktionsorientierte Fehlererholung als *Add-On*-Dienst anzubieten, muß zunächst überlegt werden, welche Funktionen zum Aufbau einer Transaktion benötigt werden. Abbildung 4.4 zeigt die Grobstruktur einer Transaktion.

Eine Transaktion überführt eine Datenbank von einem konsistenten Zustand *i* in einen neuen konsistenten Zustand *i+1* (vgl. Definition in der Einleitung dieses Abschnitts). Sie setzt sich aus einer Reihe von Operationen *Op 1*, *Op 2*, ... *Op n* zusammen, die durch ein *BOT* (*Begin of Transaction*) eingeleitet und durch ein *EOT* (*End of Transaction*) beendet wird. Die Operationen sollen entweder alle durchgeführt werden, oder keine von ihnen (Ununterbrechbarkeit). Bei einem Abbruch (*Abort*) der Transaktion muß die Wirkung der bis zu diesem

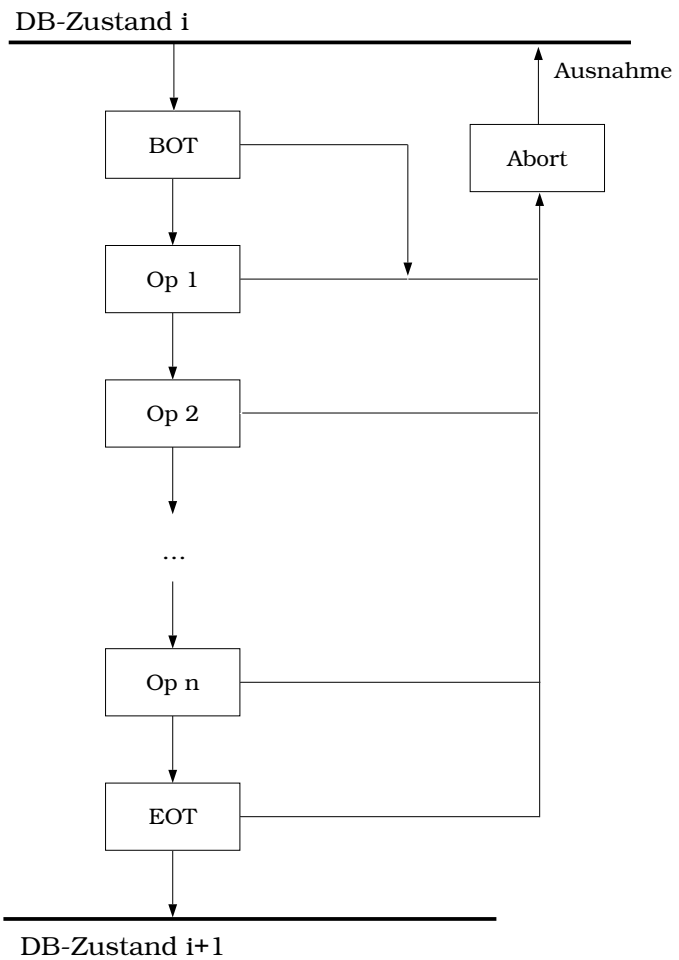

Abbildung 4.4: Transaktion

Zeitpunkt der Transaktion ausgeführten Operationen auf die Datenbank rückgängig gemacht werden (zurück zum Zustand *i*). Falls die Transaktion ohne Probleme durchläuft, wird die Transaktion verlassen (*EOT*) und die Datenbank befindet sich im Zustand *i+1*.

Der Benutzer benötigt also Funktionen *BOT* und *EOT*, um Anfang und Ende der Transaktion kennzeichnen zu können, und eine Funktion *Abort*, die einen Abbruch der Transaktion erlaubt.

4.3.3 Fehlererholung durch Verwaltung eines *Undo-Logs*

Für die Fehlererholung in Transaktionen ist es notwendig, daß Effekte von bereits ausgeführten Operationen rückgängig gemacht werden können. Dazu gibt es zwei wichtige Ansätze: das Arbeiten mit Versionen (Kopien) und die Verwaltung eines *Undo-Logs*.

Das Arbeiten mit Versionen: Hierbei wird zu Beginn der Transaktion eine Kopie der Datenbank bzw. der Objekte, die während der Transaktion verändert werden, angelegt.

Es kann dann auf zwei verschiedene Weisen fortgefahren werden:

- Die Transaktion arbeitet auf der Kopie: Im Falle eines Abbruchs wird die Kopie nicht mehr benötigt. Da das Original keine Effekte der Transaktion enthält, sind keine weiteren Aktionen erforderlich. Im Falle eines *Commits* wird das Original durch die Kopie überschrieben, die die Effekte der Transaktion enthält.
- Die Transaktion arbeitet auf dem Original: Im Falle eines Abbruchs wird das Original durch die Kopie überschrieben[6]. Im Falle eines *Commits* kann die Kopie vernichtet werden. Das Original enthält bereits die Effekte der Transaktion.

Es gibt eine Reihe von Variationen dieses Ansatzes, auf die aber im folgenden nicht weiter eingegangen wird. Beschreibungen finden sich z.B. in [Wei88]

Die Verwaltung eines *Undo-Logs*: Ein weiterer Ansatz für die Rücksetzbarkeit von Transaktionen ist die Verwaltung eines *Undo-Logs*. Hierbei wird während der Transaktion zu jeder Operation O, die ausgeführt worden ist, eine kompensierende Operation abgelegt. Wenn es zu einem Abbruch der Transaktion kommt, werden die abgespeicherten kompensierenden Operationen nacheinander ausgeführt. Dies macht die Effekte aller bis zu diesem Zeitpunkt in der Transaktion ausgeführten Operationen rückgängig. Die kompensierenden Operationen werden während der Ausführung der Transaktion in einem sogenannten *Undo-Log* verwaltet.

Abbildung 4.5 gibt einen Überblick über die möglichen Kontrollflüsse bei dieser Methode der Fehlererholung. Dabei stellen die rechteckigen Kästen die Operationen der Transaktion dar und die sechseckigen die kompensierenden Operationen. Dieser Ansatz wird z.B. in [Sch84] beschrieben.

Im folgenden wird untersucht, wie die Verwaltung eines *Undo-Logs* als *Add-On*-Dienst im Rahmen einer Transaktionsverwaltung implementiert werden kann. Dabei wird in Abschnitt 4.3.4 darauf eingegangen, welche Funktionalität die Transaktionsverwaltung (in der Abbildung grau unterlegt) zu diesem Zweck zur Verfügung stellen muß. In den Abschnitten 4.3.5 und 4.3.6 wird beschrieben, welche Anforderungen die Operationen einer Transaktion erfüllen müssen.

4.3.4 Aufgaben der Transaktionsverwaltung

Eine Transaktionsverwaltung muß gewisse Grundfunktionalitäten wie das Starten und Beenden einer Transaktion zur Verfügung stellen. Weiterhin soll sie die oben beschriebenene Fehlererholung durch Verwaltung eines *Undo-Logs* anbieten. Im folgenden wird untersucht, wie eine solche Transaktionsverwaltung als *Add-On*-Dienst realisiert werden kann, und es werden die wichtigsten Aspekte der Implementierung eines solchen Ansatzes in einer modernen Programmiersprache (hier *P-Quest*) aufgezeigt. Es wird eine Schnittstelle *TransactionBasics* beschrieben, die eine solche Transaktionsverwaltung als Dienst anbietet. Diese Schnittstelle stellt einen abstrakten Datentyp T zur Verfügung. Eine Transaktion t ist ein Wert dieses Typs.

[6] Die Kopie enthält den Zustand vor der Transaktion.

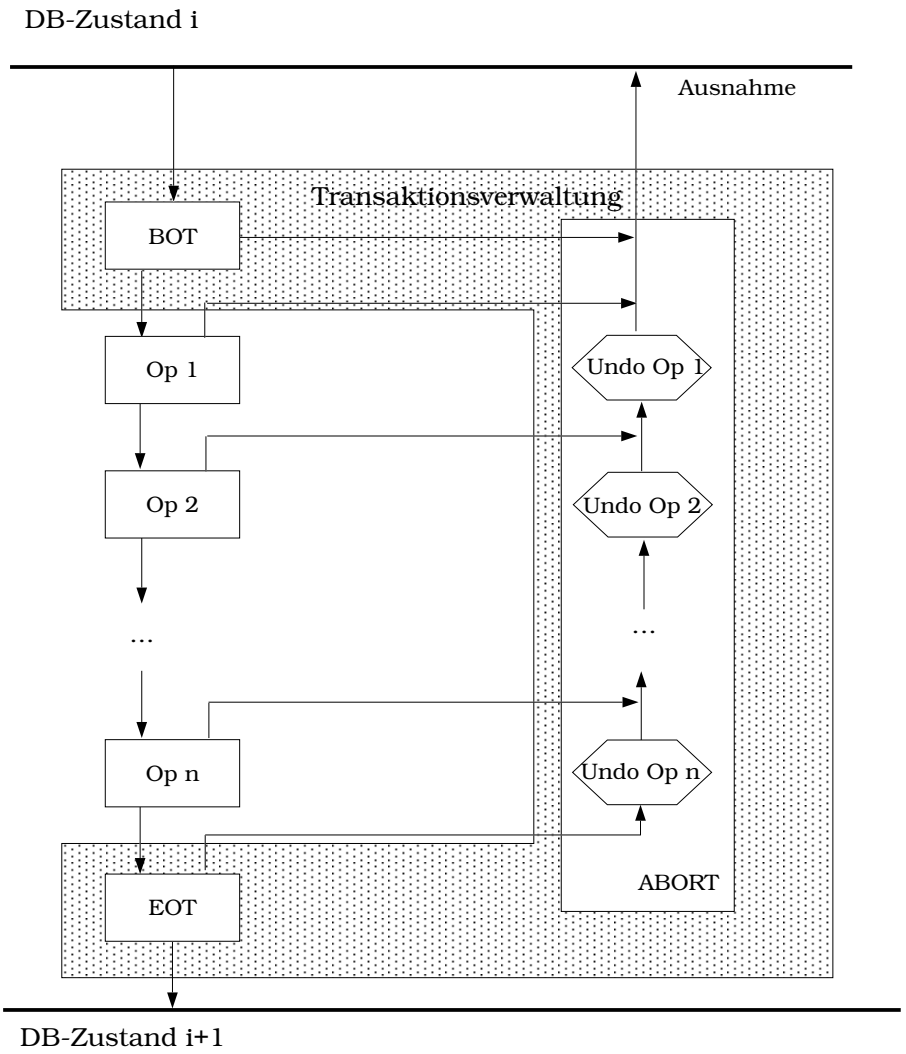

Abbildung 4.5: Fehlererholung durch Verwaltung eines Undo-Logs

Das *Undo-Log*

Nach jeder Operation wird eine kompensierende Operation im *Undo-Log* abgelegt. Diese Operationen müssen in einer geeigneten Struktur verwaltet werden. Man benötigt dazu eine Kollektion, deren Elemente Funktionen sind. Die Realisierung einer solchen Struktur ist nur möglich, da in *P-Quest* Funktionen Werte erster Klasse sind und deshalb auch in Kollektionen abgelegt werden können. Im Falle, daß ein *Undo* notwendig wird, müssen die kompensierenden Operationen in umgekehrter Reihenfolge ihrer Abspeicherung aufgerufen werden, was man sich an folgendem kleinen Beispiel veranschaulichen kann:

Die kompensierende Operation zum Einfügen eines Elements in eine Kollektion ist das Löschen dieses Elements und umgekehrt[7]. Eine Transaktion enthalte die Operationenfolge:

set.insert(persons peter) *set.delete(persons peter)*

(*persons* sei eine Menge von Personen.) Die zugehörigen kompensierenden Operationen sind dann (in Reihenfolge ihrer Abspeicherung):

set.delete(persons peter) *set.insert(persons peter)*

Nur wenn sie in umgekehrter Reihenfolge ihrer Abspeicherung abgearbeitet werden, erhält man den gewünschten Effekt der Kompensation der Operationen.

Es bietet sich also an, die kompensierenden Operationen auf einem Kellerspeicher abzulegen.

Zum Zeitpunkt, zu dem eine kompensierende Operation ausgeführt werden soll, ist deren Umgebung, insbesondere die Parameter, mit denen sie aufgerufen werden soll, nicht mehr bekannt. Die kompensierende Operation muß deshalb so gestaltet werden, daß ihr Aufruf unabhängig von der Umgebung ist, bzw. sie den benötigten Teil der Umgebung enthält. Sie muß also eine parameterlose Funktion sein, die den Wert **ok** zurückliefert, da auch Rückgabewerte zu diesem Zeitpunkt nicht mehr berücksichtigt werden können. Das Prinzip des Entwurfs von parameterlosen kompensierenden Operationen soll an obigem Beispiel veranschaulicht werden[8]:

set.insert(persons peter) $\longrightarrow$ **fun**() *set.insert(persons peter)*

Auf diese Weise sind alle kompensierenden Operationen vom gleichen Typ **All**() **Ok** und können in einer homogenen Struktur abgespeichert werden. Als *Undo-Log* kann man also einen Kellerspeicher mit Elementtyp **All**() :**Ok** verwenden.

FUNKTION DER SCHNITTSTELLE: *addUndo(t :T op() :***Ok***) :***Ok**

Start einer Transaktion

Durch den Start einer Transaktion wird ein Punkt festgesetzt, bis zu dem im Falle eines Abbruchs zurückgesetzt wird. Alle nachfolgenden Operationen werden innerhalb der Transaktion ausgeführt, bis die Transaktion explizit beendet wird oder ein Abbruch der Transaktion auftritt. Beim Start einer Transaktion muß sichergestellt werden, daß die Strukturen zur Speicherung der kompensierenden Operationen geeignet initialisiert sind.

FUNKTION DER SCHNITTSTELLE: *start() :T*

[7] Der Entwurf kompensierender Operationen wird in Abschnitt 4.3.6 ausführlich behandelt

[8] Die Funktionen *insert* und *delete* seien so definiert, daß sie den Wert **ok** zurückgeben.

Abbruch einer Transaktion (*Abort*)

Abbildung 4.6 gibt einen Überblick über die Vorgehensweise beim Abbruch einer Transaktion.

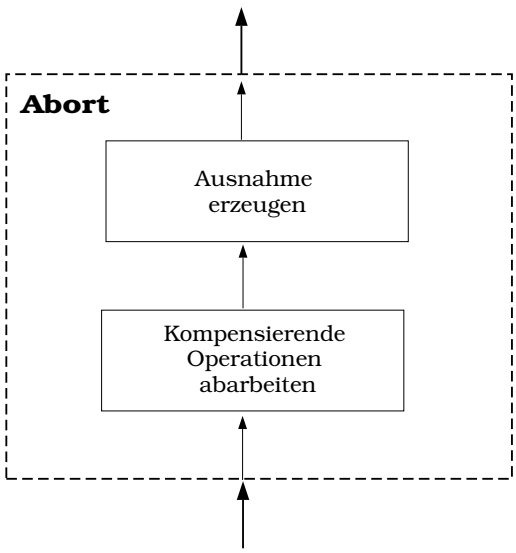

Abbildung 4.6: Abbruch einer Transaktion

Bei einem Abbruch müssen alle bisher in der Transaktion durchgeführten Operationen rückgängig gemacht werden (*Undo*). Dazu arbeitet man die kompensierenden Operationen in umgekehrter Reihenfolge ihrer Abspeicherung ab. Die kompensierenden Operationen werden nacheinander vom Kellerspeicher genommen und aufgerufen. Wenn die kompensierenden Operationen korrekt sind, befindet sich die Datenbank danach wieder im selben Zustand wie vor der Transaktion. Das *Undo* entspricht der oben aufgeführten *R1-Recovery*. Da der Abbruch ein unplanmäßiges Ende der Transaktion ist, sollte eine Ausnahme erzeugt werden, um das aufrufende Programm darüber zu informieren.

FUNKTION DER SCHNITTSTELLE: *abort(E ::***TYPE** *t :T message :***String***) :E*

Beenden einer Transaktion (*EOT*)

Wenn die Transaktion ohne Probleme bis zum Ende durchläuft, kann sie erfolgreich beendet werden. Ansonsten wird, wie oben beschrieben, ein Abbruch der Transaktion durchgeführt.

Nach erfolgreichem Abschluß einer Transaktion sollen ihre Effekte auf die Datenbank dauerhaft gemacht werden (*Durability*). Dies kann in *P-Quest* durch einen Aufruf der Funktion *store.stabilise* geschehen. Dieser Aufruf macht, wie in Abschnitt 3.8.2 beschrieben, alle bis zu diesem Zeitpunkt durchgeführten Änderungen an Objekten des persistenten Speichers dauerhaft. Ausgehend davon, daß sich die Datenbank im persistenten Speicher befindet, wird somit der Datenbankzustand, den die Transaktion erzeugt hat, persistent gemacht.

Da durch den Aufruf von *store.stabilise* auch ein Sicherungspunkt gesetzt wird, erhält man durch das Einfügen des Befehls am Ende der Transaktion auch die *R2-* und *R3-Recovery*: Bei einem Systemausfall wird der Zustand des persistenten Speichers auf den Zustand beim letzten Sicherungspunkt zurückgesetzt. Wenn man außer am Ende der Transaktionen keine Sicherungspunkte setzt, wird die Datenbank auf den Zustand, den sie am Ende der letzten vollständig ausgeführten Transaktion hatte, zurückgesetzt. Die Datenbank enthält also keine Effekte von Transaktionen, die zum Zeitpunkt des Systemausfalls noch nicht abgeschlossen waren (*R3-Recovery*). Zugleich werden durch die Sicherungspunkte alle Effekte von vollständig ausgeführten Transaktionen persistent gemacht. Bei einem Systemausfall ist also kein partielles Wiederholen von Transaktionen notwendig (*R2-Recovery*).

Es bleibt die Frage zu behandeln, was geschieht, wenn noch an anderen Stellen außer am Ende der Transaktionen Sicherungspunkte gesetzt werden. Dabei muß man zwischen Sicherungspunkten außerhalb und innerhalb von Transaktionen unterscheiden. Das Setzen solcher Punkte außerhalb von Transaktionen stellt kein Problem dar, da sie für das oben beschriebene Verhalten nicht relevant sind. Sicherungspunkte innerhalb einer Transaktion hingegen können bei einem Systemausfall zu inkonsistenten Zuständen führen, wenn dieser innerhalb der Transaktion nach dem Sicherungspunkt auftritt. Die Datenbank enthält dann Effekte dieser Transaktion, obwohl sie nicht vollständig ausgeführt worden ist. Es müßte dann eine *R3-Recovery* durchgeführt werden. Um dies zu umgehen, sollte das Setzen von Sicherungspunkten innerhalb von Transaktionen vermieden werden.

FUNKTION DER SCHNITTSTELLE: *commit(t :T)* :**Ok**

Protokollierung

Um dem Benutzer bzw. dem aufrufenden Programm eine differenzierte Reaktion auf den Abbruch einer Transaktion zu ermöglichen, sollte ihm Information über den Grund des Abbruchs zur Verfügung gestellt werden. Eine Möglichkeit dafür ist es, die während der Ausführung an die Transaktion geschickten Fehlermeldungen zu protokollieren und nach Abbruch der Transaktion zugreifbar zu machen. Dazu müssen die abgesetzten Fehlermeldungen in Strukturen abgespeichert werden, die die Transaktion überleben. Insbesondere müssen sie auch erhalten bleiben, falls die Transaktion zurückgesetzt wird.

FUNKTIONEN DER SCHNITTSTELLE:
addMessage(t :T message :**String**) :**Ok**, *messages(t :T) :Iter_T(***String***)*

Die Funktionen *addUndo*, *abort*, *addMessage* und *messages* werden auch von der Schnittstelle *Transaction* zur Verfügung gestellt, allerdings jeweils ohne den Parameter, der die Transaktion spezifiziert. Die Aufrufe beziehen sich automatisch auf die aktuelle Transaktion. Die Funktionen dieser Schnittstelle sind dadurch weniger flexibel, aber auch sicherer. Die Schnittstelle *Transaction* bietet außerdem auch Transaktionsgeneratoren an (siehe dazu Abschnitt 4.5.1).

4.3.5 Kompensierende und geschützte Operationen

In diesem und dem folgenden Abschnitt wird untersucht, wie die Operationen in der Transaktion definiert werden müssen, um den Ansatz der Fehlererholung durch die Verwaltung eines *Undo-Logs* zu unterstützen. Ein wichtiger Aspekt in diesem Zusammenhang ist die Frage, wel-

che der während einer Transaktion aufgetretenen Operationen zurückgesetzt werden müssen. Gray [Gra81] teilt die Operationen dazu in drei Gruppen ein:

Ungeschützte Operationen: Für diese Operationen ist kein Rücksetzen notwendig. Beispiele sind Operationen auf temporären Dateien und anderen temporären Daten.

Zu schützende Operationen: Diese Operationen müssen rückgängig gemacht werden, falls die Transaktion zurückgesetzt wird. In diese Gruppe fallen alle Operationen, die Datenbanken verändern.

Reale Operationen: Hierbei handelt es sich um Operationen, die nicht zurückgesetzt werden können, da sie einen nicht widerrufbaren Effekt in der realen Welt haben. Beispiele hierfür sind die Bearbeitung eines Werkstücks oder die Auszahlung von Wechselgeld. Für diese Art Operationen empfiehlt Gray ihre Ausführung möglichst bis ans Ende der Transaktion zu verzögern. Diese Gruppe von Operationen wird im folgenden nicht weiter berücksichtigt.

Es müssen also nur die zu schützenden Operationen zurückgesetzt werden. Um Effekte bereits ausgeführter Operationen auf die Datenbank rückgängig machen zu können, muß man bei dem hier gewählten Ansatz (Verwaltung eines *Undo-Logs*) kompensierende Operationen zu den zu schützenden Operationen ablegen. Eine *kompensierende* Operation O^{-1} zu einer Operation O, ist eine Operation, deren Aufruf die Wirkung von O rückgängig macht.

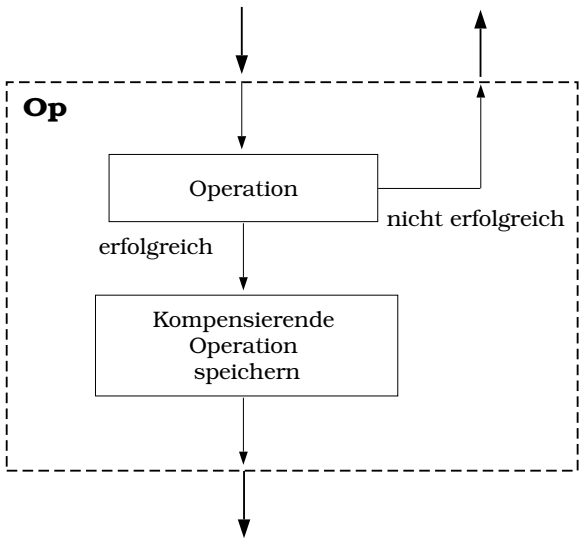

Abbildung 4.7: Aufbau einer geschützten Operation

Im folgenden sollen unter *geschützten* Operationen solche Operationen verstanden werden, die ihre zugehörige kompensierende Operation im *Undo-Log* ablegen. Abbildung 4.7 zeigt den Aufbau einer geschützten Operation. Zunächst wird die eigentliche Operation ausgeführt. Falls die Operation gelingt, wird die zugehörige kompensierende Operation abgelegt. Ansonsten muß ein Abbruch der Transaktion durchgeführt werden.

Beim Entwurf der kompensierenden Operation zu einer Operation O ist darauf zu achten, daß O bezüglich seiner Effekte auf die Datenbank entweder ganz oder gar nicht ausgeführt wird: Falls O um eine zusammengesetzte Operation ist und es nach einer Teilausführung von O zu einem Abbruch kommen kann, werden in diesem Fall bereits entstandene Effekte auf die Datenbank nicht zurückgesetzt. Um dies zu verhindern, muß beim Entwurf der geschützten Operationen die Granularität feiner gewählt werden. Die geschützten Operationen sollten also auf der Basis von einfachen Operationen entworfen werden. Geschützten Operationen für komplexere Operationen können dann aus diesen zusammengesetzt werden. Der Entwurf von geschützten Operationen und insbesondere der Entwurf von geeigneten kompensierenden Operationen werden im nächsten Abschnitt behandelt.

4.3.6 Der Enwurf von geschützten Operationen

In diesem Abschnitt wird die Problematik des Entwurfs von geschützten Operationen behandelt. Der Schwerpunkt liegt dabei nicht auf der Angabe von geschützten Operationen für spezielle Fälle, sondern auf der prinzipiellen Vorgehensweise beim Entwurf. Außerdem wird aufgezeigt, an welchen Stellen Probleme auftreten können. Es werden allgemeine Regeln für den Entwurf von geschützten Operationen aufgestellt und die Vorgehensweise für Operationen auf einzelnen Werten und Kollektionen (Bulk-Typen) genauer untersucht.

Allgemeine Vorgehensweise

Im Zusammenhang mit dem Entwurf von geschützten Operationen sind folgende Probleme zu lösen: Es sind kompensierende Operationen zu entwerfen und die kompensierenden Operationen müssen im *Undo-Log* der aktuellen Transaktion abgelegt werden. Es soll hier eine schrittweise Anleitung für den Entwurf von geschützten Operationen gegeben werden. Dabei wird davon ausgegangen, daß wie in folgendem Beispiel bereits eine Schnittstelle, die einen abstrakten Datentyp (ADT) mit den zugehörigen Operationen exportiert, vorliegt:

interface *A*
export
  T ::**TYPE**
  new() :T
  operation1(:T) :**Ok**
  operation2(:T) :**Ok**
  operation3(:T) :F
end

module *a :A*
....

Schnittstelle A exportiert einen abstrakten Datentyp T, eine Funktion *new* zum Erzeugen neuer Werte dieses Typs und drei Operationen auf Werten dieses Typs. Modul *a* sei eine Implementation dieser Schnittstelle. Durch den Übergang von einer Operation zur zugehörigen geschützten Operation ändert sich deren Signatur nicht. Man kann also die Schnittstelle

(hier *A*) beibehalten. Es genügt, eine neue Implementierung *protectedA* zu der ursprünglichen Schnittstelle *A* zu schreiben. *P-Quest* erlaubt es, die ursprüngliche Implementierung *a* in *protectedA* zu importieren und bei der Implementierung von *protectedA* die durch *a* definierten Funktionen zu verwenden. Dies macht die Aufgabe in der Regel erheblich einfacher. Die Vorgehensweise wird anhand dieses Beispiel erläutert:

1. **Wahl der zu schützenden Operationen**
   Alle Operationen auf einer Struktur, die Veränderungen durchführen, sollten geschützt werden. Dies erleichtert den Entwurf von kompensierenden Operationen: Beim Zurücksetzen einer Operation *i* kann man davon ausgehen, daß alle nach *i* ausgeführten Operationen, die die Struktur verändert haben, bereits rückgängig gemacht worden sind. Die Struktur befindet sich also wieder im selben Zustand wie direkt nach der Operation *i*. Dadurch ist beim Entwurf der geschützten Operationen Lokalität gewährleistet.

   Im Beispiel seien *operation1* und *operation2* die zu schützenden Operationen.

2. **Entwurf der kompensierenden Operationen**
   Für den Entwurf der kompensierenden Operationen kann man keine allgemeingültigen Regeln angeben. Sie müssen den Effekt der Operation, zu der sie gehören, rückgängig machen. Ihre Form hängt in starkem Maße von der ursprünglichen Operation ab. Sie haben meist die selbe Signatur wie diese Operation. In den folgenden Abschnitten werden die kompensierenden Operationen für einige einfache aber wichtige Fälle untersucht. Die kompensierenden Operationen zu den Operationen *operation1* und *operation2* seien *compOp1* und *compOp2*.

3. **Umformung in parameterlose Funktionen**
   Zum Zeitpunkt der Ausführung der kompensierenden Operation sind ihre Umgebung und insbesondere ihre Parameter nicht mehr bekannt (vgl. Abschnitt 4.3.4). Die Operationen müssen deshalb in parameterlose Funktionen mit Rückgabewert *ok* umgeformt werden. Der Übergang zu einer parameterlosen Funktion wird anhand des Beispiels verdeutlicht. Der Aufruf der Operation *operation1* für einen Wert *e :T* werde durch den Aufruf der kompensierenden Operation *compOp1* für den selben Wert kompensiert. In diesem Fall wird folgende Operation ins *Undo-Log* abgelegt:

   let *undoOp1* = **fun**() :**Ok** *compOp1(e)*

4. **Implementierung der geschützten Operationen**
   Geschützte Operationen sollen nur in Transaktionen ausgeführt werden. Deshalb muß zunächst getestet werden, ob eine Transaktion aktiv ist. Dies kann durch die Funktion *testActive* der Schnittstelle *Transaction* geschehen. Wenn eine Transaktion aktiv ist, kann die ursprüngliche Operation ausgeführt werden. Falls die Ausführung der Operation gelingt, wird ihre kompensierende Operation durch Aufruf der dafür in der Schnittstelle *Transaction* vorgesehenen Funktion *addUndo* abgespeichert. Falls eine Ausnahme auftritt, muß diese abgefangen und mit einem Abbruch der Transaktion darauf reagiert werden. Die geschützte Operation zu *operation1* könnte also z.B. wie folgt aussehen:

```
let operation1(e :T) :Ok =
    begin
     transaction.testActive()
      try
       a.operation1(e)
       transaction.addUndo(undoOp1)
      else
       transaction.abort("Operation failed")
      end
```

5. **Übernahme der übrigen Operationen**
   Für Operationen, die nicht geschützt werden müssen, z.B. lesende Zugriffe, kann auf die Implementation im ursprünglichen Modul zurückgegriffen werden. Ihre Definition im neuen Implementationsmodul ist deshalb trivial, wie an der Definition von *operation3* aus dem Beispiel zu sehen ist:

   ```
   let operation3 = a.operation3
   ```

Einzelne Werte

Es wird hier zunächst der Fall einer modifizierbaren Variablen vom Typ E betrachtet. Die zu schützende Operation ist die Veränderung des Werts der Variablen. Für diesen Fall können kompensierende und geschützte Operationen nach einem einfachen Grundprinzip entworfen werden. Der aktuelle Wert der Variablen wird zwischengespeichert, bevor der neue Wert auf die Variable zugewiesen wird. Die kompensierende Operation besteht dann einfach aus der Zuweisung des zwischengespeicherten alten Werts auf die Variable. Das folgende Programmstück soll die Vorgehensweise veranschaulichen:

```
let temp = a
a := b
transaction.addUndo(fun() a := temp)
```

Es ist möglich, eine Schnittstelle *Protected* zu schreiben, die es erlaubt, nach diesem Prinzip automatisch geschützte Operationen auf einzelnen Werten zu erzeugen. Hierbei sind die in *P-Quest* gegebenen Möglichkeiten, Typoperatoren und Generatoren für Funktionen zu definieren, eine große Hilfe. Tabelle 4.1 gibt einen Überblick über die Funktionen der Schnittstelle: Ein Element vom Typ E wird zu einem Element des Typs *protected.T(E)* (ADT). Dieser kann z.B. wie folgt implementiert werden:

```
Let T(E ::TYPE) ::TYPE =
   Tuple
     var value :E
   end
```

Die Funktion *update* ist ein Generator für geschützte Operationen. Sie ist im Prototyp wie folgt implementiert:

Funktion	**Wirkung**
*new(E ::***TYPE** *:E) :T(E)*	erzeugt aus dem übergebenen Wert einen neuen Wert vom Typ *T(E)*
*update(E ::***TYPE** *f(:E) :E) :***All***(:T(E)) :***Ok**	erzeugt eine geschützte Funktion, deren Anwendung auf ein Element vom Typ *T(E)* dessen Wert gemäß der übergebenen Funktion *f* verändert
*value(E ::***TYPE** *:T(E)) :E*	liefert den aktuellen Wert vom Typ *E* zurück

Tabelle 4.1: Die Funktionen der Schnittstelle *Protected*

```
let update(E ::TYPE f(:E) :E) :All(:T(E)) Ok =
   fun(object :T(E))
    begin
      transaction.testActive()
      let old = object.value
      object.value := f(old)
      transaction.addUndo(fun() object.value := old)
    end
```

Bei der Erzeugung der geschützten Operation wird nach dem oben vorgestellten Prinzip vorgegangen.

Die Verwendung der Schnittstelle *protected* soll am Beispiel eines Zählers illustriert werden. Ein Zähler mit Anfangswert 0 kann durch die Operation

```
let count = protected.new(:Int 0)
```

erzeugt werden. *count* ist vom Typ *protected.T(*:**Int***)*. Mit Hilfe der Funktion *update* der Schnittstelle kann man Funktionen zum kontrollierten Inkrementieren und Dekrementieren des Zählers definieren:

```
let increment = protected.update(:Int  fun(i :Int) i + 1)
let decrement = protected.update(:Int  fun(i :Int) i - 1)
```

Die Funktionen *increment* und *decrement* sind vom Typ **All***(protected.T(***Int***))* :**Ok**. Wenn diese Operationen innerhalb einer Transaktion verwendet werden, hat der Zähler im Falle eines Abbruchs am Ende wieder den gleichen Wert wie vor der Transaktion. Der Aufruf *protected.value(count)* liefert den aktuellen Wert des Zählers.

Bei dem bis jetzt vorgestellten Ansatz wird davon ausgegangen, daß eine Variable vom Typ *E* auf einen neuen Wert zugewiesen werden soll, d.h., daß die Operationen zur Änderung des

Werts der Variablen vom Typ **All***(:E) :E* sind, also einen neuen Wert vom Typ *E* erzeugen. In vielen Fällen, z.B. wenn nur einer Komponente der Variablen ein neuer Wert zugewiesen werden soll, möchte man aber Operationen mit Seiteneffekten verwenden (Funktionen vom Typ **All***(:E):* **Ok**). Dies ist z.B. notwendig, wenn das Element von einem abstrakten Datentyp ist, dessen Schnittstelle aus Operationen mit Seiteneffekten besteht.

Um vor Änderungsoperationen mit Seiteneffekten den ursprünglichen Wert des Elements für die Herleitung der kompensierenden Operation zwischenzuspeichern, muß eine Kopie des Elements erzeugt werden. Die Kopie hat aber eine andere Identität als das ursprüngliche Element, was in Umgebungen, bei denen mit Seiteneffekten gearbeitet wird, oft nicht akzeptabel ist. Ohne eine Kopie des ursprünglichen Werts kann aber die kompensierende Operation nicht mehr automatisch hergeleitet werden. Um für diesen Fall einen Generator für geschützte Funktionen zu schreiben, muß die kompensierende Operation dem Generator als Parameter übergeben werden. Daraus ergibt sich folgende Signatur für die Funktion *update2*:

*update2(E ::***TYPE** *ob: T(E) f(:E) :***Ok** *fcomp(:E) :***Ok***) :***All***(:E) :***Ok**

Bei abstrakten Datentypen können nur Operationen der zugehörigen Schnittstelle oder daraus aufgebaute Operationen als Änderungsfunktion *f* und als kompensierende Funktion *fcomp* übergeben werden. In einer objekt-orientierten Umgebung sind für diese Parameter nur Methoden des Objekts oder daraus aufgebaute Operationen möglich.

Die Anwendung dieses neuen Generators soll ebenfalls anhand eines Zählers veranschaulicht werden, der diesmal in Form eines ADTs gegeben sei:

```
interface Count
  Def T =
      Tuple
        value() :Int
        incr()  :Ok
        decr()  :Ok
      end
    new(i :Int) :T
end
```

Die Funktionen zum geschützten Inkrementieren und Dekrementieren können dann wie folgt definiert werden:

```
let increment = protected.update2(:count.T  count.incr  count.decr)
let decrement = protected.update2(:count.T  count.decr  count.incr)
```

Bulk-Typen

Die modifizierenden und damit zu schützenden Operationen auf Bulk-Typen (Definition siehe Abschnitt 1.2) sind das Einfügen und Löschen von Elementen und Änderungsoperationen auf einzelnen Elementen. Es wird sich hier auf die Betrachtung von Einfüge- und Löschoperationen beschränkt. Geschützte Änderungsoperationen auf einzelnen Elementen werden bereits im vorigen Abschnitt beschrieben.

Grundsätzlich wird das Einfügen eines Elements durch das Löschen des gleichen Elements kompensiert und umgekehrt. Probleme treten jedoch bei der Behandlung von Duplikaten auf und bei Operationen, die mehrere Elemente löschen. Es werden hier nur ungeordnete Bulk-Typen betrachtet.

Operation	#(e) vorher	#(e) nachher	kompens. Operation	Bemerkung
Bulk-Typ ohne Duplikate				
Einfügen				
insert1(e) [1]	0	1	delete(e)	
	1	1	–	Ausnahme
insert2(e) [2]	0	1	delete(e)	
	1	1	**ok**	
Löschen				
delete1(e) [3]	1	0	insert(e)	
	0	0	–	Ausnahme
delete2(e) [4]	1	0	insert(e)	
	0	0	**ok**	
Bulk-Typ mit Duplikaten				
Einfügen				
insert(e)	n	n+1	delete(e)	
Löschen				
delete1(e)	n>0	n−1	insert(e)	siehe([3])
	0	0	–	Ausnahme
delete2(e)	n>0	n−1	insert(e)	siehe([4])
	0	0	**ok**	
deleteAll1(e) [5]	n	0	n x insert(e)	siehe([3])
	0	0	–	Ausnahme
deleteAll2(e)	n	0	n x insert(e)	siehe([4])
	0	0	**ok**	

Tabelle 4.2: Kompensierende Operationen bei Bulk-Typen

Tabelle 4.2 gibt einen Überblick über die kompensierenden Operationen für ungeordnete Bulk-Typen. Bulk-Typen mit und ohne Duplikate werden dabei getrennt betrachtet. Bei jeder Operation wird zwischen den beiden Fällen unterschieden, ob das zu löschende bzw. ein-

[1] Falls e bereits existiert, wird eine Ausnahme erzeugt.
[2] Falls e bereits existiert, wird der Befehl ignoriert.
[3] Falls e nicht vorhanden ist, wird eine Ausnahme erzeugt.
[4] Falls e nicht vorhanden ist, wird der Befehl ignoriert.
[5] Der Befehl *deleteAll* löscht alle Vorkommen von e.

zufügende Element e in der Kollektion vorhanden ist oder nicht (#(e) vorher). Die Angabe "-" bei den kompensierenden Operationen bedeutet, daß diese beliebig gewählt werden kann.

Beim Entwurf der kompensierenden Operationen sind, wie in Tabelle 4.2 zu sehen ist, grundsätz lich zwei Typen[9] von Operationen zu unterscheiden. Operationen vom Typ 1 erzeugen eine Ausnahme, wenn die Operation nicht zulässig ist[10]. Im Fall, daß es zu einer solchen Ausnahme kommt, wird ein Abbruch durchgeführt, bevor die kompensierende Operation abgelegt wird, d.h., die kompensierende Operation kann beliebig gewählt werden. Beim Typ 2 werden Operationen ignoriert, die nicht zulässig sind, d.h. es wird einfach fortgefahren, als ob diese Operation nicht existiert hätte. In diesem Fall sollte keine Operation bzw. ein **ok** als kompensierende Operation abgelegt werden. Bei Operationen vom Typ 2 müssen also abhängig davon, ob die Operation zulässig ist oder nicht, verschiedene kompensierende Operationen abgelegt werden. Dies macht, wie das folgende Beispiel einer Löschoperation zeigt, einen Enthaltenseins-Test vor der Operation notwendig, um anhand von dessen Ergebnis über die kompensierende Operation zu entscheiden.

```
let protecteDelete2(bulk :Bulk(E) e :E) :Ok =
    begin
      let exists = member(bulk e)
      delete(bulk e)
      if exists then
        addUndo(fun() insert(e))
      else (* Der else-Zweig kann weggelassen werden. *)
        addUndo(fun() ok)
      end
    end
```

Bei Operationen vom Typ 1 hingegen braucht nicht zwischen diesen beiden Fällen unterschieden zu werden. Es kann für beide Fälle die gleiche kompensierende Operation abgelegt werden.

Bisher wurden nur Löschoperationen betrachtet, die ein einzelnes Element löschen. Es gibt jedoch auch Löschoperationen, deren Ausführung evtl. eine Reihe von Elementen löscht. Ein Beispiel hierfür ist das Löschen aller Vorkommen eines Elements (*deleteAll* in Tabelle 4.2) bei Kollektionen mit Duplikaten. Die kompensierende Operation muß in diesem Fall soviele Vorkommen dieses Elements wiedereinfügen, wie gelöscht worden sind. Dazu ist es notwendig vor der Löschoperation die Anzahl der Vorkommen des Elements zu ermitteln.

Ein weiteres Beispiel für solche Operationen ist das *prädikative Löschen*. Beim prädikativen Löschen werden alle Elemente gelöscht, die das bei der Löschoperation angegebene Prädikat erfüllen. In diesem Fall müssen vor der Löschoperation die Elemente, die das Prädikat erfüllen, ermittelt und zwischengespeichert werden. Die kompensierende Operation muß alle zwischengespeicherten Elemente wieder einfügen. Die zugehörige geschützte Operation wird also nach folgendem Schema aufgebaut:

[9] Sie sind durch Anhängen einer 1 bzw. 2 an ihren Namen gekennzeichnet.

[10] Das ist der Fall, wenn beim Löschen das angegebene Element nicht existiert oder beim Einfügen in eine Kollektion ohne Duplikate das einzufügende Element bereits existiert.

```
let protectedDeletePred(bulk :Bulk(E) p(:E) :Bool) :Ok =
   begin
    let temp = iter.select(elements(bulk) p)
    deletePred(bulk p)
    addUndo(fun() iter.forEach(temp fun(e :E) insert(bulk e))
   end
```

Bei Operationen, die mehrere Elemente löschen, ist außerdem das in Abschnitt 4.3.5 über die Granularität von Operationen Gesagte zu beachten: Es muß gewährleistet sein, daß nicht nach dem Löschen eines Teils der Elemente ein Abbruch der Transaktion auftreten kann, da sonst die bereits durchgeführten Löschoperationen nicht kompensiert werden. Falls dies nicht sichergestellt werden kann, muß das Löschen in Einzeloperationen zerlegt werden, wobei nach jeder Operation die zugehörige kompensierende Operation abgelegt wird.

Bei geordneten Kollektionen (wie z.B. Listen) treten wegen der festen Reihenfolge der Elemente zusätzliche Probleme auf. So muß man z.B. bei den Löschoperationen darauf achten, daß die kompensierende Operation das Element an der richtigen Stelle wieder einfügt. Auf diese Probleme soll hier aber nicht weiter eingegangen werden.

4.4 Transaktionsorientierte Integritätsüberwachung

Die Konsistenz einer Datenbank ist durch eine Menge von Integritätsbedingungen definiert: Wenn all diese Bedingungen erfüllt sind, ist die Datenbank in einem konsistenten Zustand. Jede Änderungsoperation auf der Datenbank kann zu einer Inkonsistenz führen. Ein korrekter, aber naiver Ansatz zur Integritätsüberwachung ist es, nach jeder Operation bzw. am Ende jeder Transaktion alle Integritätsbedingungen zu testen. Dieser Ansatz führt zu großen Effizienzproblemen, da nach [Qia88] (1) Integritätsbedingungen allgemeine Aussagen über Datenmengen der Datenbank sind und somit Zugriffe auf große Datenmengen notwendig machen, (2) Datenbanken häufig verändert werden und die teuren Auswertungsoperationen somit oft ausgeführt werden müssen und (3) falls eine Bedingung verletzt ist, die ganze Transaktion zurückgesetzt werden muß, was wiederum eine teure Operation ist.

Nicht jede Operation kann jede Integritätsbedingung verletzen. Um unnötige Tests zu vermeiden, kann man die Integritätsbedingungen den Operationen zuordnen, die sie verletzen können. Eine Integritätsbedingung muß dann nur getestet werden, wenn eine der Operationen, der sie zugeordnet ist, ausgeführt wird. Dazu wird hier ein dynamischer Ansatz gewählt: Die Integritätsbedingungen werden in Kollektionen verwaltet, die den Operationen zugeordnet sind. Die Kollektionen können dynamisch durch Einfügen und Löschen verändert werden, auch wenn die Datenbank bereits existiert. Bei Ausführung einer Operation werden alle aktuell zu ihr gespeicherten Integritätsbedingungen getestet. Es wird sich hier auf die Betrachtung statischer Integritätsbedingungen beschränkt.

Wenn Bedingungen bei Operationen getestet werden, kann man zwischen Vor- und Nachbedingungen unterscheiden, je nachdem, ob die Bedingung vor oder nach der Operation getestet wird. In der Regel muß die Bedingung für die beiden Fälle unterschiedlich formuliert werden. Manche Zusammenhänge lassen sich besser als Vor- und manche besser als Nachbedingungen ausdrücken. Es läßt sich jedoch für die hier betrachteten Klassen von Integritätsbedingungen jede Bedingung, die als Vorbedingung formuliert werden kann, auch als Nachbedingung ausdrücken und umgekehrt [HL74]. Vorbedingungen haben den Vorteil, daß man die Operation nicht zurücksetzen muß, falls die Bedingung verletzt ist. Es wird deshalb versucht, die Integritätsüberwachung soweit wie möglich durch den Test von Vorbedingungen zu realisieren. Wie bereits in Abschnitt 2.5.1 erwähnt, gibt es jedoch auch Bedingungen, die verzögert getestet werden müssen. Verzögerte Bedingungen sind stets Nachbedingungen. Bei einem transaktionsorientierten Ansatz werden die Tests dabei auf das Ende der Transaktion verschoben. Da Verletzungen dieser Bedingungen somit erst zu diesem Zeitpunkt festgestellt werden, müssen evtl. sehr viele Operationen zurückgesetzt werden. Es empfiehlt sich also, möglichst auf die Verwendung verzögerter Bedingungen zu verzichten.

Wenn Integritätsbedingungen Operationen zugeordnet werden, kann man häufig die bei den Operationen durchzuführenden Tests vereinfachen. Dafür existiert eine Reihe von Methoden [Bla81, Nic82, Kre88, BMM92]. Die Zuordnung der Integritätsbedingungen zu den Operationen und die Umformung der Tests zur Vereinfachung sind zum Teil keine trivialen Aufgaben. Ziel dieses Ansatzes ist jedoch auch nicht die Überwachung allgemeiner Integritätsbedingungen zu unterstützen. Vielmehr soll die Überwachung eines begrenzten Satzes häufig verwendeter Integritätsbedingungen, wie etwa Eindeutigkeitsbedingungen und referentielle Integrität, effizient unterstützt werden. Der Benutzer gibt diese Bedingungen in deklarativer Form an, und die Zuordnung und die Transformation in geeignete Tests wird automatisch sichergestellt.

Der Abschnitt ist wie folgt aufgebaut: Im ersten Teil wird untersucht, wie sich Integritätstests in der Sprache *P-Quest* darstellen lassen. Die Verwaltung der Integritätsbedingungen in Kollektionen und der Test solcher Kollektionen wird im zweiten Teil behandelt. Bei diesem Ansatz werden sowohl unmittelbare als auch verzögerte Integritätsbedingungen betrachtet. Deren Verwaltung und Test in einem transaktionsorientierten Ansatz ist Thema des nächsten Teils des Abschnitts. Dabei wird untersucht, wie die Integritätsüberwachung aussieht und welche Erweiterungen der im vorigen Abschnitt vorgestellten Transaktionsverwaltung dafür notwendig sind. Der vorletzte Teil beschäftigt sich mit der Frage, wie Operationen modifiziert werden müssen, damit sie die gewünschte Integritätsüberwachung durchführen (konsistenzerhaltende Operationen). Es wird dabei eine allgemeine Anleitung für den Entwurf solcher Operationen gegeben, und für einzelne Objekte und Bulk-Typen wird der Entwurf genauer untersucht. Im Abschnitt über die Integritätsüberwachung bei Bulk-Typen wird außerdem ein alternativer Ansatz zu dem hier vorgestellten aufgezeigt. Im letzten Teil des Abschnitts wird kurz auf die Zuordnung von Integritätsbedingungen zu Operationen und auf die Transformation von Integritätsbedingungen eingegangen. Eine ausführlichere Behandlung dieses Themas findet sich im Abschnitt 5.2.3. Dort wird für wichtige Integritätsbedingungen auf Klassen exemplarisch gezeigt, wie die Zuordnung zu den Operationen und mögliche Umformungen aussehen können.

4.4.1 Darstellung und Test von Integritätsbedingungen

Integritätsbedingungen sind Prädikate über Datenbankvariablen. Prädikate können als boolesche Funktionen dargestellt werden. So kann z.B. ein einstelliges Prädikat als Funktion betrachtet werden, die einen Wert vom Typ E auf einen booleschen Wert abbildet, also als Funktion vom Typ **All***(:E)* :**Bool**.

Bei diesem Ansatz werden die Integritätsbedingungen den Operationen zugeordnet. Man braucht also nicht allgemeine Integritätsbedingungen darzustellen, sondern Integritätstests, die bei den Operationen ausgeführt werden. Die booleschen Funktionen, die solche Tests darstellen, haben in der Regel die selben Parameter wie die Operation oder eine Teilmenge davon. Betrachtet man z.B. die allgemeine Integritätsbedingung "Der Preis aller Basisteile ist positiv". Diese Bedingung kann z.B. durch das Einfügen eines neuen Teils verletzt werden. Vor dem Einfügen eines Teils genügt es zu testen, ob der Preis dieses Teils positiv ist. Dieser Test läßt sich wie folgt darstellen:

let *positiveCost(bp :BasePart)* :**Bool** = *bp.cost > 0*

Wenn eine Integritätsbedingung in Form einer booleschen Funktion vorliegt, kann sie durch einen Aufruf dieser Funktion getestet werden. Dabei sind die Werte als Parameter zu übergeben, für die die Bedingung getestet werden soll. Der Test des obigen Prädikats für ein Teil *bp1* sieht wie folgt aus:

positiveCost(bp1)

Zur Formulierung komplexerer Bedingungen, die Bezug auf eine ganze Kollektion nehmen, können die Funktionen der Schnittstelle *Iter* verwendet werden (siehe Abschnitt 4.2.2), wie an folgendem Beispiel für eine Eindeutigkeitsbedingung zu sehen ist:

"Die Nummer des Teils *bp1* ist eindeutig in der Menge der Basisteile":

iter.all(set.elements(baseParts) **fun***(p: BasePart) not(p.pno is bp1.pno))*

4.4.2 Verwaltung von Integritätsbedingungen

Die Integritätsbedingungen werden in Kollektionen verwaltet. Dabei werden solche Integritätsbedingungen zu einer Kollektion zusammengefaßt, die bei den gleichen Operationen getestet werden und die gleiche Signatur besitzen. Auf diese Weise können alle Integritätsbedingungen einer Kollektion durch eine Iteration über deren Elemente getestet werden.

Integritätsbedingungen werden durch Funktionen dargestellt. Je nach Arität der Funktionen, d.h. je nach Stelligkeit der Prädikate, benötigt man verschiedene Schnittstellen zur Verwaltung der Integritätsbedingungen. Es wird sich hier aber auf die Betrachtung von einstelligen Prädikaten beschränkt. Dies hat sich z.B. bei der Untersuchung der Integritätsbedingungen für das *Object-Relationship*-Modell als ausreichend erwiesen (siehe Abschnitt 5). Bei der Erstellung von Schnittstellen für Prädikate mit anderer Stelligkeit kann jedoch nach der gleichen Methode vorgegangen werden.

Die Schnittstelle zur allgemeinen Verwaltung von Integritätsbedingungen *ConstrBasics* stellt Funktionen zur Verwaltung und zum Test von Kollektionen von Integritätsbedingungen zur Verfügung. Auf diese Funktionen wird im folgenden genauer eingegangen.

Die Kollektionen

Neben der booleschen Funktion, die die Integritätsbedingung darstellt, wird zu jeder Bedingung noch ein eindeutiger Name und eine Fehlermeldung abgespeichert. Im Falle der Verletzung einer Integritätsbedingung kann diese durch ihren Namen identifiziert und die abgespeicherte Fehlermeldung ausgegeben werden. Der Elementtyp der Kollektionen kann also etwa wie folgt aussehen:

Let *Constraint(E ::*__TYPE__*) ::*__TYPE__ =
  Tuple
    *name :*__String__
    *test(:E) :*__Bool__
    *message :*__All__*(:E)* __String__
  end

Die Fehlermeldung ist dabei eine Funktion des Typs *E*. Dies ermöglicht es, präzisere Fehlermeldungen auszugeben. So kann man z.B. im Fall von Kollektionen auch den Namen oder eine andere Identifikation des Objekts, das die Integritätsbedingung verletzt, in die Fehlermeldung aufnehmen. So kann die Meldung zu der oben eingeführten Bedingung *positiveCost* etwa wie folgt aussehen:

let *priceMessage(p :BasePart) :*__String__ =
  "Der Preis des Teils " <> p.name <> "ist nicht positiv."

Die Definition von Kollektionen mit einem Elementtyp wie *Constraint* ist nur möglich, da in *P-Quest* Funktionen Werte erster Klasse sind und somit auch als Komponenten von Tupeln auftreten können.

Einfügen und Löschen von Integritätsbedingungen

Die Operationen zum Einfügen und Löschen von Integritätsbedingungen haben in der Schnittstelle *ConstrBasics* folgende Signaturen:

*insert(E ::***TYPE** *constraints :constrBasics.T(E) name :***String**
*p(:E) :***Bool** *message :Error(E)) :***Ok**
*delete(E ::***TYPE** *constraints :constrBasics.T(E) name :***String***) :***Ok**

Vor dem Einfügen einer Bedingung wird die Eindeutigkeit ihres Namens innerhalb der Kollektion getestet. Da es sich bei den Bedingungen um Funktionen handelt, wird für die Einfügeoperation eine Funktion höherer Ordnung benötigt. Das Löschen von Integritätsbedingungen geschieht über ihren Namen. Falls eine Bedingung mit dem angegebenen Namen existiert, wird sie gelöscht. Ansonsten wird eine Ausnahme erzeugt

Test durch Iterationsabstraktion

Der Test einer Kollektion von Integritätsbedingungen besteht aus einer Reihe von Einzeltests, deren Ergebnisse verknüpft werden. Mit Hilfe der Funktionen zur Iterationsabstraktion (siehe Abschnitt 4.2.2) kann eine Kollektion von Integritätsbedingungen einfach gestestet werden. Die Schnittstelle *ConstrBasics* stellt zwei Funktionen zum Testen zur Verfügung: *test* und *violations*. Die Funktion *test* testet, ob alle Integritätsbedingungen erfüllt sind und ist wie folgt implementiert:

let *test(E ::***TYPE** *constraints :T(E) object :E) :***Bool** =
iter.all(elements(constraints) **fun***(c :Constraint(E)) :***Bool** *c.test(object))*

Die Funktion *violations* erzeugt eine Iteration über alle Integritätsbedingungen, die für ein bestimmtes Element verletzt sind:

let *violations(E ::***TYPE** *constraints :T(E) object :E) :Iter_T(Constraint(E))=*
iter.reject(elements(constraints)
fun*(c :Constraint(E)) c.test(object))*

Wie sich im folgenden noch zeigen wird, ist diese Art Test oft sehr nützlich.

Eine weitere Funktion (*elements*) dient zur Berechnung einer Iteration über alle Integritätsbedingungen, die sich in der Kollektion befinden. Dadurch erhält der Anwender die Möglichkeit, weitere Operationen wie z.B. andere Test als die oben beschriebenen auf den Integritätsbedingungen auszuführen oder auch eine Liste aller gespeicherten Integritätsbedingungen auszugeben.

4.4.3 Übergang zur transaktionsorientierten Integritätsüberwachung

Wie bereits im Abschnitt 2.5.1 erwähnt, können nicht alle Integritätsbedingungen unmittelbar getestet werden: Es gibt logische Operationen, die in mehrere Teiloperationen zerfallen und evtl. erst am Ende all dieser Teiloperationen wieder einen gemäß der Integritätsbedingungen konsistenten Zustand herstellen. Die entsprechenden Tests können erst zu diesem Zeitpunkt ausgeführt werden. Bei einem transaktionsorientierten Ansatz bietet sich das Ende der jeweiligen Transaktion als (spätester) Zeitpunkt für die verzögerten Tests an, da erst am Ende der Transaktion wieder ein nach außen hin sichtbar zu machender Zustand erreicht ist.

Verwaltung unmittelbarer und vezögerte Integritätstests

Es müssen in diesem Zusammenhang also zwei Arten von Integritätsbedingungen verwaltet werden, die unmittelbar und die verzögert zu testenden. Die Einteilung der Integritätsbedingungen in diese Gruppen ist anwendungsabhängig und muß vom Benutzer durchgeführt werden. Funktionen zur Verwaltung von unmittelbaren und verzögerten Integritätsbedingungen werden von der Schnittstelle *Constr* angeboten. Sie enthält unter anderem Funktionen zum Einfügen und Löschen von Integritätsbedingungen, zum Test der einzelnen Arten der Integritätsbedingungen und zur Iteration über die Kollektionen. Die Implementierung der Schnittstelle baut auf den Funktionen der Schnittstelle *ConstrBasics* auf.

Für die Verwaltung der beiden Arten von Integritätsbedingungen kann man sich zwei Ansätze vorstellen. Man kann beide Arten in einer Kollektion abspeichern und sie zur Unterscheidung mit einem zusätzlichen Attribut versehen oder man kann für jede Art eine eigene Kollektion anlegen. Auf jeden Fall muß die Schnittstelle zur Verwaltung der Integritätsbedingungen dem Benutzer die Möglichkeit anbieten, beim Einfügen von Bedingungen festzulegen, ob sie unmittelbar oder verzögert getestet werden sollen. Sie kann dazu zwei verschiedene Funktionen zur Verfügung stellen oder nur eine Funktion mit einem zusätzlichen Parameter, durch den die Art der Bedingung festgelegt werden kann. Die Schnittstelle *Constr* stellt zu diesem Zweck die beiden Funktionen *addPrecondition* und *addDelayedCondition* zur Verfügung. Das Löschen von Integritätsbedingungen geschieht wiederum über ihren Namen. Wenn dabei, wie das bei der Funktion *delete* der Schnittstelle *Constr* der Fall ist, die Art der Integritätsbedingung nicht mit angegeben wird, müssen beide Kollektionen durchsucht werden.

Integritätsüberwachung

Die Aufgabe der Integritätsüberwachung besteht darin, bei jeder Operation sowohl deren unmittelbare als auch deren verzögerte Integritätsbedingungen zu testen. Die unmittelbaren Integritätsbedingungen werden, da es sich um Vorbedingungen handelt, vor der Operation getestet. Falls sie nicht alle erfüllt sind, muß ein Abbruch der Transaktion initialisiert werden. Vorher werden die Namen und Fehlermeldungen der verletzten Bedingungen an die Transaktion übermittelt, um es dem Anwender zu erleichtern, den Grund für einen Abbruch herauszufinden und geeignet darauf zu reagieren. Der Test der verzögerten Integritätsbedingungen kann erst am Ende der Transaktion durchgeführt werden. Diese Bedingungen müssen deshalb zwischengespeichert werden. Wie für die kompensierenden Operationen (siehe Abschnitt 4.3.4), gilt auch für die verzögerten Integritätsbedingungen, daß zum Zeitpunkt ihres

Tests ihre Umgebung nicht mehr bekannt ist. Sie müssen deshalb ebenfalls als parameterlose Funktionen gespeichert werden. Da es sich um Tests handelt, ist der Rückgabewert der Funktionen vom Typ **Bool**. Eine Integritätsbedingung p vom Typ **All***(:E)* :**Bool**, die am Ende der Transaktion für ein Element e vom Typ E getestet werden soll, wird dafür in folgende parameterlose Funktion umgewandelt:

fun*()* :**Bool** *p(e)*

Der Test der einzelnen Bedingungen am Ende der Transaktion kann dann jeweils durch einen einfachen Aufruf der abgespeicherten Funktion durchgeführt werden. Falls beim Test eine Konsistenzverletzung festgestellt wird, d.h., eine oder mehrere der Bedingungen nicht erfüllt sind, muß die Transaktion zurückgesetzt werden.

Für die Integritätsüberwachung müssen also sowohl die Operationen modifiziert (Durchführung geeigneter Tests) als auch die Funktionalität der Transaktionsverwaltung erweitert werden. Der Entwurf geeigneter Operationen wird im nächsten Abschnitt (Entwurf konsistenzerhaltender Operationen) beschrieben. Auf die notwendigen Erweiterungen der Transaktionsverwaltung wird im nächsten Teil dieses Abschnitts eingegangen.

Um den Entwurf von konsistenzerhaltenden Operationen zu erleichtern, bietet die Schnittstelle *Constr* eine umfassende Funktion *test* für den Test von Integritätsbedingungen innerhalb einer Transaktion an. Diese Funktion führt den Test der unmittelbaren Bedingungen und die Ablage der verzögerten Bedingungen für ein Element wie oben beschrieben durch.

Erweiterung der Transaktionsverwaltung

Verzögerten Integritätsbedingungen müssen bis zum Ende der Transaktion zwischengespeichert werden. Die Transaktionsverwaltung muß dafür eine geeignete Struktur zur Verfügung stellen. Da die Tests Funktionen des Typs **All***()* :**Bool** sind, wird dafür eine Kollektion mit diesem Elementtyp benötigt.

Neben dieser parameterlosen Funktion sollten noch weitere Informationen, wie etwa der Name der Bedingung (zur Identifikation im Fehlerfalle) und die zu der Integritätsbedingung abgespeicherte Fehlermeldung an die aktuelle Transaktion übermittelt werden. Die Schnittstelle *Transaction* stellt dafür folgende Funktion zur Verfügung:

addDelayedCondition(name :**String** *p()* :**Bool** *message* :**String***)* :**Ok**

Weiterhin soll es möglich sein, Namen und Fehlermeldungen verletzter (unmittelbarer) Integritätsbedingungen in einer geeigneten Struktur der Transaktionsverwaltung abzuspeichern. Die Protokollierung von Fehlermeldungen wird bereits im Abschnitt 4.3.4 beschrieben. Die Abspeicherung der Namen verletzter Integritätsbedingungen kann auf die selbe Weise implementiert werden.

4.4.4 Der Entwurf konsistenzerhaltender Operationen

Konsistenzerhaltende Operationen sind solche Operationen, die gewährleisten, daß sie keine Integritätsbedingungen verletzen. Da beim Entwurf des Prototyps auf aktive Komponenten

verzichtet wird (siehe dazu Abschnitt 4.5.4), gibt es keine Maßnahmen zur Korrektur von Konsistenzverletzungen, wie sie z.B. in [CW90] und [AGO91b] verwedet werden. Die Konsistenzerhaltung sieht hier deshalb wie folgt aus: Die eigentliche Operation wird nur dann ausgeführt, wenn sie keine unmittelbare Bedingung verletzt, bzw. sie wird am Ende der Transaktion zurückgesetzt, falls ihre Ausführung eine Bedingung verletzt und diese Verletzung nicht bis zum Ende der Transaktion korrigiert wird (verzögerte Bedingungen).

Zwischen konsistenzerhaltenden und geschützten Operationen besteht folgender Zusammenhang: Bei dem hier gewählten Ansatz werden sowohl unmittelbar als auch verzögert zu testende Integritätsbedingungen verwaltet. Der Test der verzögerten findet erst am Ende der Transaktion statt, d.h., nachdem die Operation, die diesen Test nötig macht, ausgeführt worden ist. Falls die Bedingung am Ende der Transaktion nicht erfüllt ist, muß die ganze Transaktion und damit auch diese Operation zurückgesetzt werden. Es empfiehlt sich also, beim Entwurf der konsistenzerhaltenden Operationen bereits von geschützten Operationen auszugehen. Der Weg von der ursprünglichen bis zur konsistenzerhaltenden Operation geht somit über die geschützten Operationen. Man kann die beiden Stufen natürlich auch zu einem Schritt zusammenfassen. Sie werden der Übersichtlichkeit halber hier aber getrennt dargestellt.

Wie im Fall der geschützten Operationen (Abschnitt 4.3.6) werden auch hier zunächst allgemeine Regeln angegeben, nach denen beim Entwurf von konsistenzerhaltenden Operationen vorgegangen werden kann. In den darauffolgenden Abschnitten wird die Integritätsüberwachung für einzelne Objekte und für Bulk-Typen genauer behandelt.

Allgemein ist zu sagen, daß es nicht Ziel dieses Ansatzes ist, daß sich der Endbenutzer oder der Anwendungsprogrammierer mit dem Entwurf konsistenzerhaltender Operationen auseinandersetzen muß. Vielmehr sollten die konsistenzerhaltenden Operationen beim Entwurf neuer Bulk-Typen und anderer Strukturen mitberücksichtigt und als Dienst zur Verfügung gestellt werden. So wird z.B. bei den Schnittstellen für das *Object-Relationship*-Modell (siehe Abschnitt 5.2.3) eine Funktion zur Verfügung gestellt, die den automatischen Übergang zu Klassen mit konsistenzerhaltenden Operationen erlaubt.

Allgemeine Vorgehensweise

Im Zusammenhang mit dem Entwurf konsistenzerhaltender Operationen auf Datenstrukturen sind folgende Aufgaben zu lösen: Die Integritätsbedingungen müssen in Kollektionen verwaltet werden und bei modifizierenden Operationen asuf der Struktur muß eine Integritätsüberwachung durchgeführt werden. Zur Lösung dieser Aufgaben können die Funktionen der in den vorigen Abschnitten vorgestellten Schnittstellen, insbesondere der Schnittstelle *Constr* verwendet werden.

Wie bei den geschützten Operationen wird auch hier eine schrittweise Anleitung für den Entwurf gegeben. Zur Erläuterung wird die gleiche Beispielsschnittstelle A verwendet, die einem abstrakten Typ T mit einer Reihe von Operationen auf diesem Typ exportiert. Der Typ T und die Operationen seien in einem Modul a implementiert.

```
interface A
export
  T ::TYPE
  new() :T
  operation1(t :T) :Ok
  operation2(t :T) :Ok
  operation3(t :T) :F
end

module a :A
...
```

Beim Entwurf konsistenzerhaltender Operationen kann wie folgt vorgegangen werden:

1. **Wahl der konsistenzerhaltenden Operationen**
   Es ist zu überlegen, zu welchen Operationen Integritätsbedingungen verwaltet werden sollen, d.h., welche Operationen Integritätsbedingungen verletzen können. Das sind im allgemeinen alle Operationen, die den Wert des ADT verändern. Im gewählten Beispiel seien dies *operation1* und *operation2*.

2. **Entwurf einer neuen Schnittstelle.**
   Die neue Schnittstelle muß neben einem abstrakten Datentyp und einer Funktion zur Erzeugung neuer Werte dieses Typs auch alle Operationen der ursprünglichen Schnittstelle enthalten. Weiterhin braucht man Funktionen zum Einfügen und Löschen von Integritätsbedingungen. Die neue Schnittstelle könnte für das oben gewählte Beispiel etwa wie folgt aussehen:

   ```
   interface IcA
     IcT ::TYPE
     new() :IcT
     operation1(t :IcT) :Ok
     operation2(t :IcT) :Ok
     operation3(t :IcT) :F

     Operation ::TYPE
     op1, op2 :Operation
     addCondition(t :IcT op :Operation ...) :Ok
     addDelayedCondition(t :IcT  op:Operation ...) :Ok
     deleteCondition(t :IcT ConstraintName :String) :Ok
   end
   ```

3. **Definition des neuen abstrakten Datetyps**
   Für das gewählte Beispiel muß der neue ADT eine Komponente vom Typ $a.T$ für die Abspeicherung des eigentlichen Werts enthalten.

```
Let IcT=
   Tuple
    value :a.T
    op1Constraints :constr.T(a.T)
    op2Constraints :constr.T(a.T)
   end
```

Außerdem wird für jede konsistenzerhaltende Operation eine Kollektion von Integritätsbedingungen vorgesehen. Der Typ einer solchen Kollektion kann mit dem Typoperator *constr.T* (Verwaltung von unmittelbaren und verzögerten Integritätsbedingungen, siehe Abschnitt 4.4.3) erzeugt werden. Im Beispiel wird angenommen, daß die Tests für alle Operationen Prädikate der Form *p(:a.T)* :**Bool** sind. Daraus ergibt sich der Typ der Kollektion der Integritätsbedingungen zu *constr.T(a.T)*.

4. **Implementation einer Funktion zur Erzeugung neuer Werte**
   Die Funktion *new* zur Erzeugung von Werten des neuen ADT muß einen Wert des ursprünglichen ADTs erzeugen. Das kann durch einen Aufruf der Funktion *new* der ursprünglichen Schnittstelle geschehen. Weiterhin müssen leere Kollektionen zur Verwaltung der Integritätsbedingungen angelegt werden. Hierfür kann man die Funktion *new* der Schnittstelle *Constr* verwenden. Für das Beispiel ergibt sich folgende Funktion:

```
let new() :IcT =
   tuple
    let value = a.new()
    let o1Constraints = constr.new(:a.T)
    let o2Constraints = constr.new(:a.T)
   end
```

5. **Entwurf der konsistenzerhaltenden Operationen**
   Da davon ausgegangen wird, daß die unmittelbaren Integritätsbediungungen in Form von Vorbedingungen vorliegen, müssen sie vor der Operation getestet werden. Weiterhin müssen die verzögerten Integritätsbedingungen an die aktuelle Transaktion übermittelt werden, damit sie an deren Ende getestet werden können. Zur Ausführung dieser Aufgaben kann die Funktion *test* der Schnittstelle *Constr* verwendet werden. Unter Verwendung dieser Funktion kann z.B. die Definition der Operation *operation1* mit Integritätsüberwachung wie folgt aussehen[11]:

```
let operation1(t :IcT) :Ok =
   try
    constr.test(t.o1Constraints t.value)
    a.operation1(t.value)
   else
    transaction.abort(transaction aborted")
   end
```

[11] Bei unmittelbaren Integritätsbedingungen in Form von Nachbedingungen muß die Funktion *test* nach Ausführung der Operation und nach dem Abspeichern der kompensierenden Operation ausgeführt werden.

Falls nicht alle unmittelbaren Integritätsbedingungen erfüllt sind, erzeugt *test* eine Ausnahme [12]. Ansonsten wird die eigentliche Operation ausgeführt. Während des Tests oder der Operation aufgetretene Ausnahmen werden durch das *try*-Konstrukt behandelt, und es wird mit einem Abbruch der Transaktion darauf reagiert.

6. **Implementation der Funktionen zum Einfügen von Integritätsbedingungen**
Beim Einfügen muß die Möglichkeit gegeben sein, zwischen unmittelbaren und verzögerten Integritätsbedingungen zu unterscheiden und festzulegen, an welche Operation die Integritätsbedingung gebunden werden soll. Aus Konsistenzgründen mit der Schnittstelle *Constr* empfiehlt es sich, getrennte Funktionen zum Einfügen verzögerter und unmittelbarer Integritätsbedingungen anzubieten. Zur Unterscheidung zwischen den Operationen kann ein Optionstyp *Operation* der folgenden Form eingeführt werden:

```
Let Operation = Option op1 op2 end
let op1 = option op1 of Operation end
let op2 = option op2 of Operation end
```

Für die Zuordnung der Integritätsbedingungen zu den Operationen können Werte dieses Typs als Parameter angegeben werden. Die übrigen Parameter der Funktionen zum Einfügen von Integritätsbedingungen sind die gleichen wie bei den entsprechenden Funktionen der Schnittstelle *Constr*.

Wie am Beispiel der Funktion *addCondition* zum Einfügen unmittelbarer Integritätsbedingungen zu sehen ist, besteht die Implementierung im wesentlichen aus einer Fallunterscheidung für die verschiedenen Operationen und aus Aufrufen von Funktionen der Schnittstelle *Constr* mit geeigneten Parametern:

```
let addCondition(t :IcT op :Operation ...) :Ok =
  case op
    when op1 then constr.addCondition(t.o1Constraints ...)
    when op2 then constr.addCondition(t.o2Constraints ...)
  end
```

Bei dem hier gewählten Ansatz können auch noch neue Integritätsbedingungen eingefügt werden, wenn die Datenbank bereits existiert und Daten enthält. Um die Konsistenz der Datenbank bezüglich einer neu eingefügten Integritätsbedingung zu gewährleisten, muß getestet werden, ob bereits existiernde Werte diese Bedingung erfüllen. Ansonsten kann die Bedingung nicht eingefügt werden. Wie der Test der Bedingung für bereits existierende Werte aussieht hängt in hohem Maße von der Struktur von T ab. In den folgenden Abschnitten wird das Problem für einzelne Objekte und für Bulk-Typen untersucht.

7. **Entwurf einer Funktion zum Löschen von Integritätsbedingungen**
Wenn die Signatur der Löschoperation wie im Beispiel gewählt wird (nur den Namen der Bedingung als Parameter), müssen die Kollektionen aller Operationen nach der zu löschenden Bedingung durchsucht werden. Dies geschieht durch Aufrufe der Löschfunktion der Schnittstelle *Constr* für jede Kollektion.

[12] Eine genauere Beschreibung dieser Funktion findet sich in Abschnitt 4.4.3.

8. **Übernahme der übrigen Operationen der ursprünglichen Schnittstelle**
   Alle Operationen, für die keine Integritätsbedingungen verwaltet werden, sollen auch in der neuen Schnittstelle enthalten sein. Ihre Definition im neuen Implementationsmodul ist, wie am Beispiel zu sehen, trivial:

   let *operation3(t :IcT)* = a.*operation3(t.value)*

Bis jetzt wird davon ausgegangen, daß die Integritätsbedingungen für die verschiedenen Operationen alle das gleiche Format haben. Falls dies nicht der Fall ist, braucht man für jede Operation eine eigene Funktion für das Einfügen von Integritätsbedingungen. Die Schnittstelle wird dadurch umfangreicher, aber bei der Implementierung sind keine entscheidend anderen Aspekte zu beachten.

Integritätsüberwachung für einzelne Objekte

Zunächst soll der einfachste Fall betrachtet werden, nämlich die Verwaltung und Überprüfung von Integritätsbedingungen für ein einzelnes Objekt. Für eine sinnvolle Integritätsüberwachung muß sichergestellt sein, daß die Integritätsbedingungen bei jeder Veränderung des Werts des Elements getestet werden. Eine Methode hierfür ist es, das Element in einen ADT einzuschließen. Die Funktionen der Schnittstelle des ADTs sind dann die einzigen Möglichkeiten, den Wert des Elements zu verändern, und die Integritätsüberwachung kann auf diese Operationen beschränkt werden.

In Abschnitt 4.3.6 wird eine Schnittstelle *Protected* vorgestellt, die den Übergang zu geschützten Operationen für einzelne Elemente als Dienst anbietet. Für den Übergang zu konsistenzerhaltenden Operationen bietet der Prototyp eine entsprechende Schnittstelle *IntegrityChecked* an und ein zugehöriges Modul *integrityChecked*, das diese Schnittstelle implementiert. *IntegrityChecked* stellt einen abstrakten Typoperator T zur Verfügung, der aus einem Elementtyp E den benötigten abstrakten Datentyp erzeugt. Zu jedem Element wird eine Kollektion mit Integritätsbedingungen verwaltet. Der neue abstrakte Typoperator kann also etwa wie folgt implementiert werden:

```
Let T(E) ::TYPE =
   Tuple
     value :E
     objectConstraints :constr.T(E)
  end
```

Auf Werte dieses abstrakten Datentyps kann nur über die Funktionen der Schnittstelle *IntegrityChecked* zugegriffen werden. Tabelle 4.3 gibt einen Überblick über diese Funktionen.

Um die Verwendung der Schnittstelle zu erläutern, wird das Beispiel des Zählers aus Abschnitt 4.3.6 erweitert. Dort wird ein Zähler *count* mit Operationen *increment* und *decrement* zum geschützten Inkrementieren und Dekrementieren implementiert. Der Zähler soll jetzt zusätzlich die Bedingung erfüllen, daß sein Wert immer im Bereich zwischen zwei Werten a und b liegt. Ein Zähler mit Integritätsüberwachung kann aus *count* durch die Operation

let *checkedCount* = *integrityChecked.new(count)*

Funktion	**Wirkung**
*new(E ::***TYPE** *e :E) :T(E)*	erzeugt aus dem übergebenen Wert einen neuen Wert vom Typ *T(E)*
*update(E::***TYPE** *f(:E)* **:Ok***)* **:All***(:T(E))* **:Ok**	erzeugt eine Funktion, deren Anwendung auf ein Element vom Typ *T(E)* dessen Wert gemäß der übergebenen Funktion *f* verändert und die gespeicherten Integritätsbedingungen testet
*value(E ::***TYPE** *ob :T(E)) :E*	liefert den aktuellen Wert vom Typ *E* zurück
*addCondition(E ::***TYPE** *ob :T(E)* *name:* **String** *p(:E)* **:Bool** *message(:E)* **:String***)*:**Ok**	fügt eine unmittelbar zu testende Integritätsbedingung *p* ein
*addDelayedCondition(E ::***TYPE** *ob :T(E)* *name:* **String** *p(:E)* **:Bool** *message(:E)* **:String***)* **:Ok**	fügt eine verzögert zu testende Integritätsbedingung *p* ein
*deleteCondition(E ::***TYPE** *ob :T(E)* *name* **:String***)* **:Ok**	löscht die durch *name* gewählte Integritätsbedingung

Tabelle 4.3: Die Funktionen der Schnittstelle *IntegrityChecked*

erzeugt werden. *checkedCount* ist dann vom Typ *integrityChecked.T(protected.T(***:Int***))*. Zur Bereichsbeschränkung des Zählers auf Werte zwischen a und *b* werden Integritätsbedingungen *lowerlimit* und *upperlimit* definiert und in die Kollektion der unmittelbaren Tests eingefügt:

let *lowerlimit(c :protected.T(Int))* **:Bool** = *protected.value(c)* > a
let *upperlimit(c :protected.T(Int))* **:Bool** = *protected.value(c)* < *b*

addCondition(checkedCount lowerlimit ...)
addCondition(checkedCount upperlimit ...)

Mit Hilfe der Funktion *update* kann man aus den geschützten Operationen *increment* und *decrement* entsprechende Operationen mit Integritätsüberwachung erzeugen:

let *checkedIncrement = integrityChecked.update(increment)*
let *checkedDecrement = integrityChecked.update(decrement)*

Wie im Fall der Schnittstelle *Protected* ist die Funktion *update* ein Generator für Änderungsoperationen, hier allerdings nicht für geschützte, sondern für konsistenzerhaltende. Sie erhält

eine Funktion f als Parameter und liefert eine Funktion zurück, die f für den aktuellen Wert des Objekts aufruft und eine Integritätsüberwachung durchführt.

Für die Implementierung der Funktionen der Schnittstelle ist zunächst zu überlegen, zu welchen Zeitpunkten Integritätsbedingungen zu testen sind: Zum einen muß nach jeder Veränderung des Werts des Objekts überprüft werden, ob der neue Wert die Integritätsbedingungen noch erfüllt. Die Bedingungen müssen hier als Nachbedingungen formuliert werden. Zum anderen muß beim Einfügen einer neuen Integritätsbedingung getestet werden, ob der aktuelle Wert die einzufügende Bedingung erfüllt. Es müssen also von den Funktionen *update*, *addCondition* und *addDelayedCondition* Integritätstests durchgeführt werden. Um genauer zu sein, muß von der Funktion *update* eine Funktion erzeugt werden, in deren Rumpf unter anderem die notwendigen Bedingungen getestet werden. Falls ein Integritätstest ein negatives Ergebnis liefert, wird bei den Funktionen *addCondition* und *addDelayedCondition* eine Ausnahme erzeugt, da die eingefügte Bedingung nicht zulässig ist. Innerhalb der von *update* erzeugten Funktion hingegen muß ein Abbruch der Transaktion initiiert werden. Für diese Aufgabe kann man die Funktion *test* der Schnittstelle *Constr* verwenden. Die Implementierung der übrigen Funktionen der Schnittstelle orientiert sich weitgehend an den oben angegebenen Regeln und wird deshalb hier nicht weiter erläutert.

Die Funktionen der Schnittstelle *IntegrityChecked* können auch auf Werte angewandt werden, die nicht durch *protected.new* erzeugt worden sind. Wie weiter oben bereits erwähnt, empfiehlt es sich jedoch, von Elementen mit geschützten Operationen auszugehen, insbesondere, da hier auch die unmittelbaren Bedingungen erst nach der Operation getestet werden und somit bei einer Verletzung ein Rücksetzen der Operation notwendig ist.

Eine mögliche Erweiterung des vorgestellten Ansatzes besteht darin, für verschiedene Änderungsoperationen eigene Kollektionen von Integritätsbedingungen zu verwalten. Das hat den Vorteil, daß die Anzahl der Tests reduziert wird, da man die Integritätsbedingungen gezielter den Operationen zuordnet, die sie verletzen können. Es ist dann aber nicht mehr möglich, eine Standardschnittstelle wie *IntegrityChecked* für den Übergang zu konsistenzerhaltenden Operationen zu schreiben, da die Schnittstelle von der Anzahl der Operationen, die für ein Objekt definiert sind, abhängt.

Integritätsüberwachung für Bulk-Typen

In diesem Abschnitt wird die Definition von konsistenzerhaltenden Operationen für Bulk-Typen (Definition siehe Abschnitt 1.2) untersucht. Dazu ist zunächst zu überlegen, welche Operationen einen Bulk-Typ verändern, da für alle verändernden Operationen eine Integritätsüberwachung durchgeführt werden soll. Normalerweise sind dies die Operationen *insert*, *delete* und *update* zum Einfügen, Löschen und Ändern von Elementen der Kollektion. Integritätsüberwachung ist jedoch nur für die Operationen sinnvoll, bei denen auch sicher ist, daß die Schnittstelle mit der Integritätsüberwachung benutzt wird. Der Wert eines einzelnen Elements der Kollektion kann aber in der Regel auch auf andere Weise als über die Funktion *update* verändert werden, nämlich über den Namen oder andere Bezüge auf das Element. Damit wird aber die Integritätsüberwachung der Funktion *update* umgangen. Es wird sich deshalb auf Integritätstests für Einfüge- und Löschoperationen beschränkt. Zur Integritätsüberwachung bei Änderungsoperationen kann auf die Methoden des vorigen Abschnitts zurückgegriffen werden.

Wegen der großen Bedeutung der Bulk-Typen im Datenbankbereich existiert im Prototyp eine eigene Schnittstelle *BulkConstraints* für die Verwaltung von Integritätsbedigungen auf Bulk-Typen. Sie bietet Funktionen zur Verwaltung unmittelbarer und verzögerter Integritätsbedingungen für Einfüge- und Löschoperationen an. Bei den Bedingungen auf wird davon ausgegangen, daß sie als boolesche Funktionen über dem Elementtyp E der Kollektion vorliegen, d.h. vom Typ **All**(*:E*) :**Bool** sind. Der Test beim Löschen bzw. Einfügen eines Elements besteht dann im Aufruf der entsprechenden Bedingungen für dieses Element. Bei der Implementierung des abstrakten Datentyps T und der Funktionen von *BulkConstraints* wird auf die Funktionen der Schnittstelle *Constr* zurückgegriffen. So ist z.B. T wie fogt definiert:

```
Let T(E ::TYPE) ::TYPE=
    Tuple
      insertConditions :constr.T(E)
      deleteConditions :constr.T(E)
    end
```

Für eine Kollektion mit Elementtyp E ist die Kollektion der Integritätsbedingungen vom Typ *bulkConstraints.T(E)*.

Die Funktion *new* der Schnittstelle *BulkConstraints* legt entsprechend zwei Kollektionen vom Typ *Constr.T* an. Bei den übrigen Funktionen der Schnittstelle (Einfügen, Löschen und Test von Integritätsbedingungen) wird ebenfalls gemäß der oben angegebenen Regeln vorgegangen: Sie bestehen hauptsächlich aus Fallunterscheidungen für die Operationen *insert* und *delete* und aus Aufrufen der entsprechenden Funktion der Schnittstelle *Constr* mit der geeigneten Kollektion von Integritätsbedingungen (*insertConditions* oder *deleteConditions*).

Für den Entwurf der konsistenzerhaltenden Operationen wird davon ausgegangen, daß der Bulk-Typ als abstrakter Datentyp *Bulk* mit einem internen Zustand und den Operationen *insert*, *delete* und *elements* definiert ist:

```
Def Bulk(E ::TYPE) ::TYPE =
    Tuple
      insert(e :E) :Ok
      delete(e :E) :Ok
      elements() :Iter(E)
    end
```

```
Def ICheckedBulk(E ::TYPE) ::TYPE =
    Tuple
      insert(e :E) :Ok
      delete(e :E) :Ok
      elements() :Iter(E)
      constraints :BulkConstraints.T(E)
    end
```

Der entsprechende Typ mit Integritätsüberwachung kann dann z.B. von der Form wie *ICheckedBulk* sein. Die Funktionen für das Einfügen und Löschen von Integritätsbedingungen werden von der Schnittstelle *BulkConstraints* zur Verfügung gestellt.

Wie bei der Integritätsüberwachung bei einzelnen Objekten so ist auch hier bei der Implementierung zunächst zu überlegen, wann ein Test von Integritätsbedingungen notwendig wird: Zum einen müssen beim Einfügen von Elementen in die Kollektion und beim Löschen von Elementen aus der Kollektion Integritätsbedingungen getestet werden, und zwar sowohl unmittelbare als auch verzögerte. Falls zum Zeitpunkt, zu dem eine neue Bedingung eingefügt wird, bereits Elemente in der Kollektion existieren, muß außerdem sichergestellt werden, daß die Bedingung für diese Elemente erfüllt ist.

Die Implementierung von Operationen mit Integritätsüberwachung wird in den allgemeinen Regeln zum Entwurf konsistenzerhaltender Operationen beschrieben. Nach diesem Muster kann bei der Erstellung der neuen Operationen *insert* und *delete* vorgegangen werden. Bei ihrer Definition können die ursprünglichen Definitionen der Operationen verwendet werden. Insbesondere kann man z.B. für einen konkreten Bulk-Typen *Bulk* eine Funktion schreiben, die von *Bulk* nach *ICheckedBulk* abbildet, d.h., die eine Kollektion als Parameter nimmt und eine Kollektion mit den gleichen Elementen und Operationen mit Integritätsüberwachung als Funktionswert zurückliefert. Eine Implemtierung dieses Ansatzes für Klassen findet sich in Abschnitt 5.2.3.

Beim Einfügen einer neuen Bedingung ergibt sich folgendes Problem: Wenn eine Bedingung p für den Bulk-Typ gelten soll, wird daraus eine Bedingung p', die vor dem Einfügen bzw. Löschen eines Elements zu testen ist, hergeleitet. p' ist eine Vorbedingung und evtl. umgeformt unter der Voraussetzung, daß nur ein einzelnes Element eingefügt bzw. gelöscht wird und sich die Datenbank vor dem Test in einem konsistenten Zustand befindet [13]. Beim Einfügen der Bedingung p' ist zu testen, daß die Elemente, die bereits in der Kollektion existieren, die ursprüngliche Bedingung p nicht verletzen. Dies kann in der Regel nicht durch einen Test der Bedingung p' für jedes Element geschehen, da es sich bei p' um eine Vorbedingung für das Einfügen eines einzelnen Elementes handelt. Es wird deshalb kein automatischer Test der eingefügten Bedingung für alle bereits existierenden Elemente durchgeführt. Stattdessen bietet die Schnittstelle *BulkConstraints* eine Funktion *testCondition* für den Test einer Bedingung für eine Kollektion von Elementen an. Sie erhält einen Iterator und eine Bedingung als Parameter und testet die Bedingung für alle Elemente der Iteration[14]. Dieser Funktion kann die Bedingung p' oder eine andere geeignete Bedingung als Parameter übergeben werden.

Bei dem hier vorgestellten Ansatz werden die Integritätsbedingungen in Kollektionen verwaltet und bei jeder Operation werden die aktuell zu ihr abgespeicherten Bedingungen getestet. Für die Überwachung der Integritätsbedingungen kann auch ein anderer Ansatz gewählt werden. Er läßt sich besonders gut am Beispiel des oben eingeführten Typs *Bulk* erläutern. Beim Einfügen einer neuen Bedingung p für die Operation o (*insert* oder *delete*) wird die Operation o neu definiert als:

if p **then** o **else raise** *error* **end end**

Die Definition kann von der Funktion zum Einfügen von Integritätsbedingungen durchgeführt werden. Die Komponenten *insert* und *delete* von *Bulk* müssen dafür veränderbar sein, oder die Funktion muß einen neuen Wert vom Typ *Bulk* zurückliefern. Dieser Ansatz erlaubt es, auf die explizite Verwaltung von Integritätsbedingungen in Kollektionen zu verzichten. Er hat jedoch den Nachteil, daß das Löschen von einmal eingefügten Integritätsbedingungen problematisch ist. Man kann sich zwar zu jeder Operation eine Liste ihrer vorigen Versionen abspeichern. Dies erlaubt aber nur ein Löschen der Integritätsbedingungen in genau der Reihenfolge, in der sie eingefügt worden sind. Der Ansatz ist also nur empfehlenswert, wenn auf das Löschen von Integritätsbedingungen verzichtet werden kann.

[13] Beispiele für solche Umformungen finden sich bei den Integritätbedingungen auf Klassen (siehe Abschnitt 5.2.3).

[14] Diese Funktion eignet sich nur für Integritätsbedingungen, die universell über die Kollektion quantifiziert sind. Andere Tests können mit Hilfe der Funktionen der Schnittstelle *Iter* formuliert werden.

4.4.5 Zuordnung der Integritätsbedingungen

In der Regel liegen die Integritätsbedingungen als allgemeine Prädikate über den Datenbankvariablen vor. Bei diesem Ansatz sollen sie den Operationen zugeordnet werden. Dabei sind zwei Fragen zu untersuchen: (1) Welche Operation kann welche Integritätsbedingung verletzen? (2) Welcher Test muß durchgeführt werden, um eine solche Verletzung zu erkennen? Der Aufwand für den Test sollte dabei möglichst gering sein.

In der Literatur wird eine Reihe von Methoden für diese Aufgaben vorgestellt. So gibt es z.B. Regeln, die von Prädikaten in Pränex-Normalform ausgehen und bei denen abhängig von der Quantifizierung der Variablen entschieden werden kann, welche Operationen eine Integritätsbedingung verletzen können. Solche Regeln werden z.B. in [LR84] und [Kre88] untersucht [15]. Andere Ansätze untersuchen den Aufbau des eigentlichen Prädikats und leiten daraus mögliche Optimierungen her. Diese Untersuchungen konzentrieren sich meist auf bestimmte Klassen von Prädikaten. Ein Beispiel hierfür ist die Arbeit von Blaustein [Bla81]. Allgemein läßt sich sagen, daß die Umformungen zu effizienten Tests nicht trivial sind. Wie bereits in der Einleitung dieses Abschnitts erwähnt, ist es jedoch nicht Ziel dieses Ansatzes, daß der Benutzer diese Zuordnungen und Umformungen durchführt. Ihm soll vielmehr ein Satz spezieller Integritätsbedingungen zur Verfügung gestellt werden. Er kann diese Bedingungen in deklarativer Form angeben, ohne sie Operationen zuzuordnen. Diese Zuordnung wird zusammen mit der Umformung zu geeigneten Tests von den Funktionen vorgenommen, die den Dienst anbieten. In Abschnitt 5.2.3 wird dieser Ansatz anhand des Beispiels der Integritätsbedingungen auf Klassen genauer erläutert.

[15] Sie werden bei der Umformung von Integritätsbedingungen auf Klassen vorgestellt und verwendet (Abschnitt 5.2.3.

4.5 Weitergehende Ansätze

In diesem Abschnitt werden weitergehende Ansätze bzgl. Transaktionsverwaltung, Fehlererholung und Integritätsüberwachung vorgestellt. Dabei sind nicht alle angesprochenen Konzepte im Prototyp implementiert. Für einen Teil werden nur Anregungen für eine mögliche Implementierung gegeben. Es werden folgende Erweiterungen angesprochen: Im ersten Teil des Abschnitts wird der Entwurf eines Transaktionsgenerators, der aus Funktionen automatisch Transaktionen erzeugt, untersucht. Thema des zweiten und dritten Teils ist die partielle Rücksetzbarkeit von Transaktionen. Dabei werden im zweiten Teil Rücksetzpunkte als möglicher Ansatz besprochen. Im dritten Teil wird die Erweiterung der bisher vorgestellten Transaktionsverwaltung auf geschachtelte Transaktionen behandelt. In diesem Zusammenhang wird versucht, den im ersten Abschnitt vorgestellten Transaktionsgenerator auf geschachtelte Transaktionen zu erweitern. Im letzten Teil wird auf die Frage einer möglichen Implementierung von aktiven Komponenten eingegangen.

4.5.1 Ein Transaktionsgenerator

Ein Transaktionsgenerator hat die Aufgabe, aus Funktionen automatisch Transaktionen zu erzeugen. Der Generator nimmt eine Funktion als Parameter und gibt als Funktionswert eine Transaktion mit der gleichen Funktionalität und zusätzlich den Eigenschaften einer Transaktion zurück. Dem Benutzer wird dadurch eine höhere Abstraktionsebene als Dienst zur Verfügung gestellt. Er braucht sich nicht mehr mit den Details des Aufbaus einer Transaktion auseinanderzusetzen.

Die Signatur

Das erste Problem beim Entwurf eines Transaktionsgenerators ergibt sich bei seiner Signatur. Ein solcher Generator soll beliebige Funktionen als Parameter akzeptieren, also insbesondere auch Funktionen mit beliebiger Parameterzahl. Bei der Angabe einer Signatur für eine Funktion höherer Ordnung, also hier für den Generator, wird aber der Typ der als Parameter übergebenen Funktion und damit die Anzahl ihrer Parameter festgelegt. Der Typ der Parameter der übergebenen Funktion hingegen kann noch offen gehalten werden, indem man den Generator durch die Einführung von Typparametern generisch gestaltet. Das folgende ist ein Beispiel für die Signatur eines Generators, der Funktionen mit zwei Parametern beliebigen Typs akzeptiert:

*generate2(E, F, G ::***TYPE** *f(:E :F) :G) :***All***(:E :F) :G*

Der Generator hat für jede Parameterzahl der übergebenen Funktion eine andere Signatur, d.h., man erhält verschiedene Generatoren für verschiedene Parameterzahlen.

Um dies zu umgehen, kann man die zu übergebenden Funktionen zu Funktionen mit nur einem Parameter vereinheitlichen. Dazu werden die n Parameter einer Funktion zu einem Tupel zusammengefaßt. Eine Funktion f mit der Signatur *f(e :E g :G) :H* wird z.B. in eine Funktion *f2* mit der Signatur

*f2(x :***Tuple** *e :E g :G* **end***) :H*

umgeformt. Innerhalb der Funktion muß dieser Unformung Rechnung getragen werden. So muß im Beispiel jeder Bezug auf den Parameter *e* durch einen Bezug auf *x.e* ersetzt und übergebene Parameter müssen in *tuple end* eingeschlossen werden. Durch diese Vereinheitlichung kann man die Signatur des Generators für beliebige Funktionen einheitlich wie folgt wählen:

*generate(E, F ::***TYPE** *f(:E) :F) :***All***(:E) :F*

Die Vereinheitlichung hat jedoch den Nachteil, daß man für die Parameter eine zusätzliche Stufe der Indirektion erhält (durch die Zusammenfassung in ein Tupel). Im Prototyp werden deshalb für verschiedene Parameterzahlen verschiedene Generatoren angeboten. Die Schnittstelle *Transaction* stellt Generatoren für Funktionen mit einem bis fünf Parametern zur Verfügung.

Automatische Erzeugung vs. Anwendungsabhängigkeit

Im folgenden ist zu untersuchen, wie man von einer Funktion zu einer Transaktion gelangt und welche Teile dieser Umwandlung automatisch, d.h. durch einen Generator durchgeführt werden können. Zusätzlich zur normalen Funktionsausführung müssen bei einer Transaktion folgende Schritte ausgeführt werden: Die Transaktion muß gestartet und zu jeder innerhalb der Funktion ausgeführten zu schützenden Operation muß eine kompensierende Operation und evtl. auch verzögerte Integritätstests abgelegt werden. Vor dem Beenden der Transaktion müssen die verzögerten Integritätsbedingungen getestet werden. Weiterhin muß man, falls ein Fehler auftritt oder nicht alle Integritätsbedingungen erfüllt sind, ein Abbruch der Transaktion mit Zurücksetzen der bereits ausgeführten Operationen durchführen. Die Transaktion muß mit einem *Commit* beendet werden.

Für das Starten der Transaktion, die Initialisierung eines Abbruchs im Fehlerfalle und das Beenden der Transaktion einschließlich des Tests der verzögerten Integritätsbedingungen können die in den vorigen Abschnitten beschriebenen Funktionen der Schnittstelle *TransactionBasics* verwendet werden. Aufrufe dieser Funktionen können vom Generator automatisch erzeugt und eingefügt werden, wie an folgendem Rahmen für die Funktionsweise eines Transaktionsgenerators veranschaulicht wird. Zu einer Funktion *f :***All***(x :T) E* wird folgende Transaktion erzeugt:

```
fun(x :T): E
  begin
    let t = transactionBasics.start()
    try
      let result = f(x)
      transactionBasics.commit(t)
      result
    else
      transactionBasics.abort(:E t "transaction aborted")
    end
  end
```

Bei diesem Ansatz bleibt folgendes Problem offen: Während der Ausführung einer Transaktion muß zu jeder zu schützenden Operation eine kompensierende Operation abgespeichert

werden. Diese Operationen können nicht automatisch vom Generator erzeugt werden, da sie in hohem Maße anwendungsabhängig sind. Die übergebene Funktion muß bereits Aufrufe zur Abspeicherung geeigneter kompensierender Funktionen enthalten. Sie sollte also bereits aus geschützten Operationen bestehen. Dies verträgt sich jedoch auch gut mit der Idee des Ansatzes, zu wichtigen Datenstrukturen, wie z.B. Klassen bereits geschützte Operationen auf diesen Strukturen zur Verfügung zu stellen.

4.5.2 Partielle Rücksetzbarkeit von Transaktionen

In manchen Fällen, besonders bei langen Transaktionen, ist es nicht wünschenswert, bei Fehlern oder Integritätsverletzungen die Effekte der ganzen Transaktion zurückzusetzen. Es sollte eine Möglichkeit angeboten werden, die es erlaubt, Transaktionen nur teilweise zurückzusetzen und evtl. statt der mißlungenen Operation andere Operationen auszuführen, um die Transaktion doch noch erfolgreich zu beenden. Dies führt zu einer Verringerung des Aufwands, da nur soviel wie nötig und nicht die ganze Transaktion zurückgesetzt wird.

Es wird hier auf zwei mögliche Ansätze für die partielle Rücksetzbarkeit eingegangen: Rücksetzpunkte und geschachtelte Transaktionen. Dabei werden die Rücksetzpunkte in diesem Abschnitt behandelt und die geschachtelten Transaktionen im nächsten.

Rücksetzpunkte

Idee der Rücksetzpunkte ist es, ein partielles Zurücksetzen der Effekte einer Transaktion zu erlauben. Innerhalb einer Transaktion kann zu einem Zeitpunkt *T1* der Ausführung ein Punkt als Rücksetzpunkt festgelegt werden. Die Transaktion kann zu einem späteren Zeitpunkt *T2* zu diesem Punkt zurückgesetzt werden. Es wird dabei zum Zustand der persistenten Datentstrukturen zu dem Zeitpunkt *T1* zurückgekehrt, nicht aber zum Ausführungspunkt des Programms. Dazu müssen alle geschützten Operationen, die seit dem Zeitpunkt *T1* ausgeführt worden sind, rückgängig gemacht werden. Dies entspricht der Abarbeitung aller *Undo*-Operationen, die seit dem Setzen dieses Rücksetzpunkts abgespeichert worden sind.

Abbildung 4.8 zeigt den Kontrollfluß einer Transaktion mit dem Rücksetzpunkt *R*. Dabei ist der dick eingezeichnete Pfad der Kontrollfluß zwischen dem Setzen eines Rücksetzpunkts *R* und dem späteren Zurücksetzen zu diesem Punkt. Die dünn eingezeichneten Pfade sind mögliche Pfade, wenn es zu einem Abbruch der Transaktion kommt (siehe auch Abbildung 4.5).

Um es dem Benutzer zu ermöglichen, Rücksetzpunkte zu benutzen, bietet die Schnittstelle zur Transaktionsverwaltung eine Funktion zum Setzen eines solchen Punkts an und eine Funktion zum Zurücksetzen zu einem definierten Rücksetzpunkt:

checkpoint() :Checkpoint
rollbackTo(c :Checkpoint) :**Ok**

Um mehrere Rücksetzpunkte verwalten zu können, erhält der Benutzer einen Identifikator vom Typ *Checkpoint* für den gesetzten Punkt als Funktionswert zurück.

Zur Implementierung von Rücksetzpunkten im bis jetzt vorgestellten Rahmen genügt es, zum Zeitpunkt *T1* eine Markierung (einen Zeiger) auf das aktuelle oberste Element des *Undo-Logs*

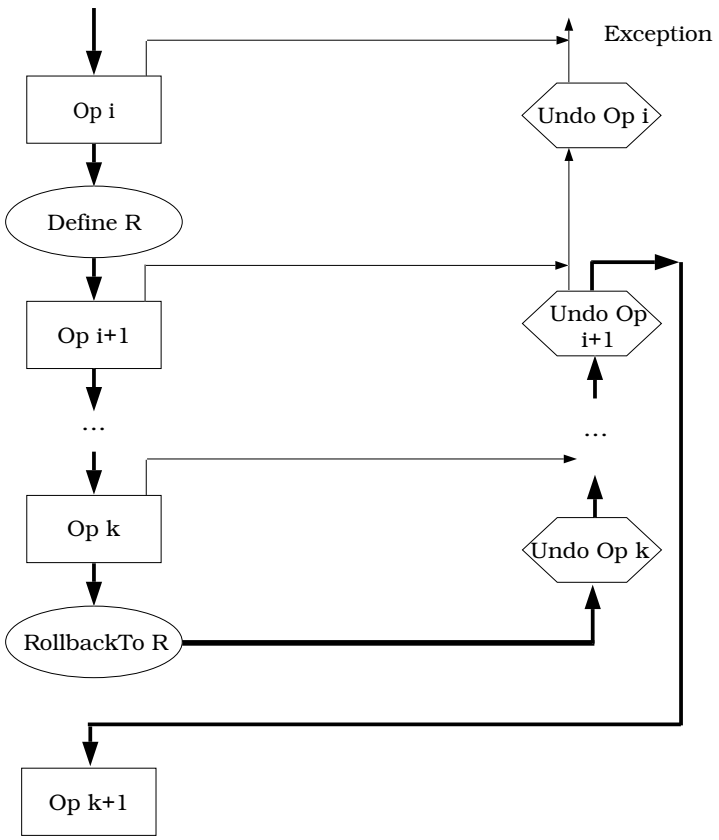

Abbildung 4.8: Partielles Zurücksetzen durch Verwendung von Rücksetzpunkten

anzulegen. Beim Zurücksetzen zum Zeitpunkt *T2* wird das *Undo-Log* bis zum Erreichen dieser Marke abgearbeitet und danach mit der Ausführung der Transaktion fortgefahren. Weiterhin müssen auch verzögert zu testende Integritätsbedingungen gelöscht werden, und zwar die, die zwischen den Zeitpunkten *T1* und *T2* abgespeichert worden sind, da der Grund für ihren Test weggefallen ist. Das Löschen dieser Bedingungen kann ebenfalls durch die Markierung der ersten nach dem Zeitpunkt *T1* abgelegten Bedingung implementiert werden.

Der Vorteil der Rücksetzpunkte ist ihre relativ unkomplizierte Implementierung. Bezüglich der Strukturierung einer Transaktion sind sie aber etwa mit einem GOTO in der traditionellen Programmierung zu vergleichen ([Dij68, Knu74]). Sie erlauben ein Zurücksetzen ohne Beachtung der hierarchischen Strukturen einer Transaktion. Ein Überspringen von Transaktionsgrenzen ist bei der hier gewählten Implementierung jedoch nicht möglich. Die im nächsten Abschnitt beschriebenen geschachtelten Transaktionen bieten einen strukturierteren Ansatz für das partielle Zurücksetzen einer Transaktion.

4.5.3 Geschachtelte Transaktionen

Bei geschachtelten Transaktionen ist es möglich, innerhalb einer Transaktion andere Transaktionen aufzurufen. Die von einer *Top-Level*-Transaktion T aufgerufenen Transaktionen bilden einen Baum, der während der Ausführung wächst. T bildet die Wurzel des Baums. Die Blätter des Baums sind die Transaktionen, die bis zum Ende durchlaufen (*Commit*) und keine weiteren Transaktionen aufrufen.

Die Einführung von geschachtelten Transaktionen läßt sich am besten an einem Beispiel motivieren. Es wird dafür das Beispiel einer Reiseagentur verwendet, das inzwischen zu einem Standardbeispiel für die Einführung von geschachtelten Transaktionen geworden ist [Wei88, Gra81].

Eine Reiseagentur führt Buchungen für Individualreisen durch. Eine Reise besteht aus einer Flugbuchung, einer Hotelreservierung und der Reservierung eines Mietwagens. Aus der Sicht des Kunden ist die Buchung der gesamten Reise eine einzige Transaktion: Es soll nur eine komplette Reise gebucht werden. Es nutzt dem Kunden nichts, wenn er z.B. zwar einen Flug zum Urlaubsort und einen Mietwagen am Urlaubsort hat, aber kein Hotel. Für die Reiseagentur zerfällt diese Transaktion in mehrere Einzeltransaktionen: Flug buchen, Hotel reservieren, Mietwagen reservieren. Diese Operationen werden getrennt und meist auch mit verschiedenen Geschäftspartnern durchgeführt. Dabei muß z.B. im allgemeinen eine Flugbuchung bereits abgeschlossen werden (für die Fluggesellschaft ist das dann bereits ein nach außen hin sichtbarer Zustand), bevor man überhaupt weiß, ob ein Hotel an diesem Urlaubsort gefunden werden kann. Um z.B. die Buchung eines Fluges rückgängig zu machen, muß als kompensierende Transaktion eine explizite Stornierung der Reservierung bei der Fluggesellschaft durchgeführt werden. Für die Fluggesellschaft ist das nicht ein einfaches Zurücksetzen der durchgeführten Operationen, da evtl. Wartelisten für diesen Flug bearbeitet werden müssen.

Geschachtelte Transaktionen stellen ein gutes Hilfsmittel zur Strukturierung von Transaktionen zur Verfügung und geben die Möglichkeit, beim Auftreten von Fehlern nur die betroffenen Teile und nicht die ganze Transaktion zurückzusetzen. Wenn z.B. bei der Reservierung des Hotels Probleme auftreten, kann diese Operation zurückgesetzt werden, um ein anderes Hotel zu probieren, ohne auch die bereits getätigte Flugreservierung rückgängig zu machen. Gegebenenfalls (z.B. kein Hotel mehr an diesem Urlaubsort) muß natürlich auch ein größerer Teil der Transaktion oder die ganze Transaktion zurückgesetzt werden.

Eine weiterer Vorteil von geschachtelten Transaktionen ist die Strukturierung bezüglich der verzögerten Integritätstests. So können, um bei obigem Beispiel zu bleiben, die verzögerten Integritätsbedingungen, die sich nur auf Flüge beziehen, am Ende der Flug-Transaktion getestet werden. Wenn sie zu diesem Zeitpunkt nicht gelten, werden sie auch durch spätere Operationen nicht mehr wahr. Somit ist, falls eine dieser Bedingungen nicht erfüllt ist, ein Abbruch der Transaktion (oder teilweises Zurücksetzen) zu einem relativ frühen Zeitpunkt möglich. Ansonsten würde die verletzte Bedingung erst am Ende der Gesamttransaktion entdeckt werden, was in der Regel zu wesentlich umfangreicheren Rücksetzaktionen führt.

Eine große Bedeutung kommt dem Konzept der geschachtelten Transaktionen auch in verteilten Systemen zu. Aspekte der Verteilung werden in dieser Arbeit jedoch nicht behandelt (siehe hierzu [Mos81, Wei88]).

Im folgenden wird zunächst auf die Semantik der geschachtelten Transaktionen eingegangen. Danach wird untersucht, wie die im Abschnitt 4.5.1 vorgestellte Transaktionsverwaltung für geschachtelte Transaktionen erweitert werden muß. Dabei wird das Problem der Kompensation von Subtransaktionen getrennt im dritten Teil des Abschnitts betrachtet. Thema des letzten Teils des Abschnitts ist die automatische Generierung von Subtransaktionen.

Semantik

Bei der Semantik der geschachtelten Transaktionen werden die von Moss in [Mos81] und [Mos87] vorgestellten Konzepte zugrundegelegt. Es wird hier die Semantik bezüglich der Fehlererholung untersucht. Dazu werden die Konsequenzen des Scheiterns einer Transaktion T (a) für ihre Subtransaktionen und (b) für die Transaktion, von der T aufgerufen wurde, betrachtet. Falls innerhalb einer Transaktion T ein Fehler auftritt oder eine Integritätsbedingung verletzt ist, wird, wie für den Fall der einfachen Transaktion beschrieben, ein Abbruch der Transaktion durchgeführt. Zusätzlich müssen aber alle Subtransaktionen von T zurückgesetzt werden.

Zur weiteren Erläuterung wird folgende Situation betrachtet: Transaktion T rufe nacheinander die Transaktionen *T1* und *T2* auf. *T1* laufe bis zum Ende ohne Fehler durch, d.h., *T1* führt ein *Commit* durch. Es handelt sich dabei um ein *Commit* bezüglich T und nicht um ein absolutes *Commit*. Absolute *Commits* werden nur von den *Top-Level*-Transaktionen durchgeführt. Man kann also sagen, T bildet eine *Commit*-Sphäre für ihre direkten Subtransaktionen. Innerhalb von *T2* komme es zu einem Abbruch, d.h., die Transaktion wird zurückgesetzt. Über das weitere Vorgehen entscheidet jetzt die Transaktion T. Falls der Abbruch von *T2* abgefangen wird (z.B. Übernahme der Aufgabe von *T2* durch eine andere Transaktion), kann T weiterlaufen. Ansonsten stellt der Abbruch von *T2* einen Fehler innerhalb von T dar, d.h., T und damit auch *T1* (*T1* ist Subtransaktion von T) müssen zurückgesetzt werden. Der Abbruch von *T2* wird also eventuell auch rekursiv nach oben weiter propagiert. Die Propagierung kann sich bis zu der *Top-Level*-Transaktion fortsetzen.

Der Übergang zu geschachtelten Transaktionen

Die im Abschnitt 4.3 vorgestellte und im Abschnitt 4.4 um die Integritätsüberwachung erweiterte Transaktionsverwaltung soll so modifiziert werden, daß sie auch geschachtelte Transaktionen unterstützt. Dazu ist zunächst zu überlegen, an welchen Stellen sich das Verhalten von Subtransaktionen von dem Verhalten von *Top-Level*-Transaktionen unterscheidet.

Beim Übergang zu geschachtelten Transaktionen tritt die Situation auf, daß es zu einem Zeitpunkt mehrere Transaktionen geben kann, die noch nicht beendet worden sind (die aktuelle *Top-Level*-Transaktion und all ihre noch nicht beendeten Subtransaktionen). Für jede dieser Transaktionen muß ein eigenes *Undo-Log* verwaltet werden, und es muß eine eigene Struktur existieren, um verzögerte Integritätstests zu speichern. Es ergibt sich also die Aufgabe gleichzeitig mehrere *Undo*-Logs zu verwalten.

Die Effekte einer Subtransaktion dürfen erst dann persistent gemacht werden, wenn die zugehörige Wurzeltransaktion erfolgreich abgeschlossen wird. Vorher kann eine Subtransaktion, auch wenn sie bereits erfolgreich beendet worden ist, noch durch das Zurücksetzten der sie

aufrufenden Transaktion rückgängig gemacht werden. Es darf also am Ende einer Subtransaktion kein Aufruf der Funktion *store.stabilise* geben.

Nach Beendigung einer Transaktion steht der Inhalt ihres *Undo-Logs* nicht mehr zur Verfügung. Am Ende einer Subtransaktion muß diese also geeignete Information auf dem *Undo-Log* der aufrufenden Transaktion ablegen, um ein späteres Rücksetzen zu ermöglichen[16].

Um dem unterschiedlichen Verhalten von Subtransaktionen und *Top-Level*-Transaktionen gerecht zu werden, braucht man entweder einen eigenen Satz Funktionen zum Starten, Beenden und Abbrechen von Subtransaktionen oder die ursprünglichen Funktionen müssen geeignet modifiziert werden. Wenn für die beiden Arten von Transaktionen die gleichen Funktionen verwendet werden sollen, muß in der Implementierung eine Fallunterscheidung gemacht werden, je nachdem, um welche Art Transaktion es sich handelt. Dazu kann z.B. ein Zähler verwaltet werden, der zu jedem Zeitpunkt die aktuelle Ebene im Transaktionsbaum angibt.

- Ebene 0: keine Transaktion aktiv
- Ebene 1: *Top-Level*-Transaktion
- Ebene 2 – n: Subtransaktion

Der Zähler muß beim Start jeder Transaktion inkrementiert werden und bei jedem Beenden einer Transaktion (auch bei einem Abbruch) dekrementiert werden, und zwar jeweils um eins. Es kann dann abhängig vom Wert des Zählers entschieden werden, von welcher Art die aktuelle Transaktion ist.

Die Kompensation von Subtransaktionen

Am Ende einer Subtransaktion muß genügend Information auf dem *Undo-Log* der aufrufenden Transaktion abgelegt werden, um ein späteres Rücksetzen der Subtransaktion zu ermöglichen. Man kann sich dafür zwei verschiedene Ansätze vorstellen, die sich darin unterscheiden, welche Information auf dem *Undo-Log* der aufrufenden Transaktion abgelegt wird:

Ablage des *Undo-Logs*: Wenn eine Subtransaktion ihr *Commit* erreicht, legt sie ihr *Undo-Log* als Element auf dem *Undo-Log* der aufrufenden Transaktion ab. Diese muß, falls die Subtransaktion zurückgesetzt werden soll, das abgelegte Log abarbeiten. Dieser Ansatz hat den Vorteil, daß sich der Benutzer nicht mit der Kompensation der Subtransaktion beschäftigen muß. Die Ablage des Undo-Logs kann bei einem *Commit* automatisch von der Transaktionsverwaltung durchgeführt werden. Ein Nachteil dieses Ansatzes ist es, daß evtl. unnötig viele Einzeloperationen zur Kompensation abgearbeitet werden müssen. Außerdem macht die Ablage eines ganzen Undo-Logs auf dem Undo-Log der aufrufenden Transaktion die Abarbeitung dieses Logs komplizierter.

Ablage einer kompensierenden Transaktion: Zu einer Subtransaktion wird eine kompensierende Transaktion definiert, die die Effekte der Transaktion rückgängig macht. Im Falle eines *Commits* der Subtransaktion wird diese kompensierende Transaktion auf dem *Undo-Log* der aufrufenden Transaktion abgelegt. Ein Vorteil dieses Ansatzes ist

[16] Auf diesen Punkt wird im nächsten Teil dieses Abschnitts genauer eingegangen.

die höhere Flexibilität bei der Wahl der kompensierenden Operationen. Wie bereits bei dem Buchungsbeispiel erwähnt, ist in manchen Situationen (z.B. Stornierung eines Flugs) zur Kompensation des Effekts eine andere Operation als das reine Zurücksetzen notwendig. Weiterhin ist der Ansatz leicht zu implementieren. Er hat jedoch den Nachteil, daß ein Mehraufwand für den Benutzer der Transaktionsverwaltung entsteht. Er muß zu jeder Subtransaktion eine kompensierende Transaktion schreiben, was oft von der gleichen Komplexität wie das Schreiben der eigentlichen Funktion ist. Außerdem ist es die Idee des vorgestellten Ansatzes zur Fehlererholung, daß die meisten Operationen ihre kompensierenden Operationen selbst im *Undo-Log* ablegen (Verwendung von geschützten Operationen, siehe Abschnitt 4.3.5). Dieser Vorgang ist für den Anwender unsichtbar, d.h., er weiß evtl. auch nicht, wie die kompensierenden Operationen aussehen. Beim Schreiben einer kompensierende Transaktion muß er sich aber mit diesen kompensierenden Operationen auseinandersetzen.

Der Ansatz ist in Umgebungen sinnvoll, in denen von Transaktionen Sperren auf Objekte oder andere Resourcen gesetzt werden. Bei Ablage des gesamten *Undo-Logs* müssen die Sperren der Subtransaktion in der Regel bis zum Ende der *Top-Level*-Transaktion gehalten werden[17]. Bei der Definition einer kompensierenden Transaktion kann festgelegt werden, welche Sperren noch benötigt werden. Die übrigen Sperren können am Ende der Subtransaktion freigegeben werden.

Ein Transaktionsgenerator für geschachtelte Transaktionen

Nach Einführung des Konzepts der geschachtelten Transaktionen stellt sich die Frage, wie man den in Abschnitt 4.5.1 vorgestellten Transaktionsgenerator erweitern muß, um einen Generator für Subtransaktionen zu erhalten. Wie oben erläutert, haben *Top-Level*-Transaktionen und Subtransaktionen stellenweise ein anderes Verhalten. Es bietet sich also an, zwei getrennte Generatoren zu schreiben. Jeder von ihnen kann dem speziellen Verhalten der verschiedenen Transaktionsarten gerecht werden. Dieser Ansatz hat jedoch neben der Verbreiterung der Schnittstelle den weiteren Nachteil, daß man bereits bei der Erzeugung einer Transaktion festlegen muß, ob sie als *Top-Level*-Transaktion oder als Subtransaktion verwendet werden soll. Falls man sie in beiden Rollen verwenden möchte, muß man zwei verschiedene Transaktionen erzeugen.

Abhängig davon, welchen Ansatz man für die Kompensation der Subtransaktionen wählt, ergeben sich noch weitere Konsequenzen für den Transaktionsgenerator. Falls kompensierende Transaktionen verwendet werden sollen, muß bei der Erzeugung einer Subtransaktion neben der eigentlichen Funktion auch eine kompensierende Funktion als Parameter übergeben werden, die im Falle eines *Commits* der Subtransaktion auf dem *Undo-Log* der aufrufenden Transaktion abgelegt wird. Bei Verwendung eines gemeinsamen Generators für *Top-Level*-Transaktionen und Subtransaktionen, muß auch bei der Erzeugung von *Top-Level*-Transaktionen eine solche Funktion angegeben werden. Wenn man eine Transaktion erzeugt, von der man weiß, daß sie nur als *Top-Level*-Transaktion verwendet werden soll, kann man als kompensierende Funktion eine Funktion angeben, deren Rumpf aus der leeren Anweisung (in *P-Quest*: *ok*) besteht. Die größte Benutzerfreundlichkeit erhält man bei Verwendung eines einzigen Transaktionsgenerators für beide Arten von Transaktionen in Kombination mit

[17] Das *Undo-Log* kann noch Operationen auf allen Objekten enthalten

der automatischen Ablage des *Undo-Logs* der Subtransaktion. Der Anwendungsprogammierer muß sich keine Gedanken über die Kompensation der Subtransaktionen machen, und mit dem Generator erzeugte Transaktionen können sowohl als *Top-Level*-Transaktionen als auch als Subtransaktionen verwendet werden. Dieser Ansatz wird im Prototyp implementiert.

4.5.4 Aktive Komponenten

Datenbanken können durch die Einführung von Regeln zu aktiven Datenbanken [MD89, Mor83] erweitert werden. Eine wichtige Anwendung solcher aktiver Komponenten ist der Bereich der Konsistenzerhaltung (vgl. Abschnitt 2.5.3 und [CW90]): Bei Integritätsverletzungen werden gemäß der definierten Regeln automatisch Operationen durchgeführt, um die Konsistenz wiederherzustellen. Eine solche Komponente zur Konsistenzerhaltung läßt sich auch in den hier vorgestellten Ansatz integrieren: Man kann z.B. zu jeder Bedingung zusätzlich eine Komponente *action* abspeichern:

Let *Constraint(E ::***TYPE***) ::***TYPE** =
  Tuple
    *name :***String**
    *test(:E) :***Bool**
    *message :***All***(:E)* **String**
    *action(:E) :***Ok**
  end

Die Komponente *action* kann eine beliebig komplexe Prozedur enthalten [18].

Bei unmittelbaren Integritätstests wird wie folgt verfahren: Zunächst wird die Bedingung getestet. Falls sie erfüllt ist, kann die eigentliche Operation ausgeführt werden. Ansonsten wird die in der Komponente *action* angegebene Operation ausgeführt. Diese korrigiert die Integritätsverletzung [19] und die eigentliche Operation kann ausgeführt werden. Bei verzögerten Integritätstests wird die Bedingung erst am Ende der Transaktion getestet und somit auch die korrigierenden Operationen erst am Ende der Transaktion, also nach der eigentlichen Operation, durchgeführt.

Im folgenden wird anhand einer Existenzbedingung ein Beispiel dafür gegeben, wie eine solche korrigierende Aktion aussehen kann:

> Bedingung: "Zu jedem Element in einer Menge A muß ein Element in einer Menge B existieren."
>
> Vor dem Einfügen eines Elements e in die Menge A ist dann zu testen, daß ein zugehöriges Element in der Menge B existiert. Falls dies nicht der Fall ist, führt das in dem bisher vorgestellten Ansatz zu einer Ausnahme. Als korrigierende Operation bietet sich in diesem Fall das Einfügen eines geeigneten Elements in die Menge B an. Danach kann das Element e in die Menge A eingefügt werden.

[18] Bei vielen in der Literatur vorgestellten Ansätzen [Mar90, AGO91b] sind die möglichen Aktionen festgelegt oder hängen von der Art der Integritätsbedingung ab.

[19] Es wird hier davon ausgegangen, daß *action* dafür geeignet definiert werden.

In einen solchen Ansatz kann der in Abschnitt 4.4 vorgestellte Ansatz, bei dem auf jede Integritätsverletzung mit einem Abbruch der Transaktion reagiert wird, als Spezialfall eingebettet werden: Man gibt als Aktion die Erzeugung einer Ausnahme an.

Bei der Verwendung von aktiven Komponenten zur Konsistenzerhaltung ist folgendes zu beachten: Eine Aktion die zur Korrektur einer Integritätsbedingung durchgeführt wird, kann wiederum Integritätsbedingungen verletzen, was neue Aktionen nach sich zieht. Aus diesem Verhalten ergeben sich folgende Probleme:

Korrektheit: Es muß sichergestellt sein, daß die ausgeführten Operationen auch wirklich die Integritätsverletzung korrigieren.

Endlichkeit und Zyklen: Durch sich immer weiter propagierende Aufrufe von Operationen können nicht terminierende Transaktionen entstehen. Das Auftreten von Zyklen muß verhindert werden.

Zusätzliche Parameter: Die angestoßenen Aktionen brauchen evtl. zusätzliche Parameter (z.B. zum Einfügen neuer Elemente). Diese müssen in irgendeiner Form zur Verfügung gestellt werden.

Widersprüche: Es ist darauf zu achten, daß die durch Angabe von Aktionen erzeugte Propagierung von Operationen widerspruchsfrei bleibt. Es können sonst abhängig von der Reihenfolge der Auswertung widersprüchliche Ergebnisse entstehen (siehe dazu auch [Mar90]). Hilfreich hierbei kann die grafische Darstellung der entstandenen Propagierungen sein, wie sie etwa in [BDRZ83] vorgestellt wird.

Unübersichtlichkeit: Eine Operation kann evtl. eine große Anzahl anderer Aktionen auslösen. Es besteht die Gefahr, daß der Benutzer nicht mehr überblicken kann, was er durch den Aufruf einer einzelnen Operation bewirkt. Dies ist besonders dann ein Problem, wenn Integritätsbedingungen mit Aktionen dynamisch eingefügt und gelöscht werden können.

Aus diesen Gründen wird im Prototyp auf eine Implementierung von aktiven Komponenten verzichtet.

Kapitel 5

Schnittstellen für ein spezielles Datenmodell

Im vorigen Kapitel wird die prototypische Implementierung einer Bibliothek generischer Dienste für datenintensive Anwendungen vorgestellt. In diesem Kapitel wird ein konkretes Datenmodell in der durch die Dienste geschaffenen Umgebung implementiert. Als Datenmodell wird das *Object-Relationship*-Modell gewählt, ein objekt-orientierter Ansatz mit expliziter Darstellung von Beziehungen (*Relationships*). Auf dieses Modell wird in Abschnitt 2.1 genauer eingegangen. In [AGO91b] wird ein Vorschlag für eine Datenbankprogrammiersprache gemacht, der dieses Modell zugrunde liegt. Die wichtigsten Aspekte dieses Vorschlags werden ebenfalls bereits in Kapitel 2 beschrieben. Da die vorgeschlagene Datenbankprogrammiersprache noch keinen Namen hat, wird in jenem Abschnitt eine Benennung (*ORM*) für sie eingeführt, um die eindeutige Bezugnahme auf sie zu erleichtern. Diese Konvention wird in diesem Kapitel weiter verwendet. Bei der hier vorgestellten Implementierung des *Object-Relationship*-Modells wird sich bei der Zusammenstellung der realisierten Schnittstellen und Konzepte an diesem Vorschlag orientiert und die erreichte Mächtigkeit mit der dieses Ansatzes verglichen.

Mit der Implementierung eines konkreten Datenmodells werden mehrere Ziele verfolgt. Zum einen dient sie dazu, die Eignung der im vorigen Kapitel vorgestellten Dienste zu testen. Es soll untersucht werden, inwiefern die Dienste eine geeignete Umgebung für die Implementierung neuer Datenmodelle schaffen. Außerdem liefert eine exemplarische Implementierung für ein spezielles Datenmodell auch viele Ideen und Lösungen, die bei der Implementierung anderer Datenmodelle in der selben Umgebung übernommen werden können.

Die Wahl des Datenmodells stellt keine Bewertung dieses Modells dar. Es wird vielmehr ausgesucht, da es eine Reihe neuerer Datenmodellierungskonzepte wie z.B. Klassen und die explizite Darstelung von Beziehungen beinhaltet. Es ist ein weiterer Zweck der Implementierung, mögliche Realisierungen dieser Konzepte im Rahmen eines *Add-On*-Ansatzes zu untersuchen.

Zur Realisierung des Datenmodells wird eine Reihe von Schnittstellen mit den zugehörigen Implementationsmodulen entworfen. Die Verwendung der vorgestellten Schnittstellen wird anhand eines einheitlichen Beispiels veranschaulicht. Zu diesem Zweck wird das Stücklistenbeispiel (vgl. [AB87]) verwendet. Eine komplette Modellierung dieses Beispiels findet sich in Anhang A.

Das Kapitel ist wie folgt strukturiert: Der erste Abschnitt gibt einen Überblick über die Architektur der Implementierung des Datenmodells. Im zweiten Abschnitt wird auf mögliche Realisierungen von Klassen als *Add-On*-Ansatz eingegangen. Dabei werden auch die Aspekte Integritätsüberwachung und Rücksetzbarkeit von Operationen auf Klassen betrachtet. Thema des dritten und des vierten Abschnitts des Kapitels sind Beziehungen zwischen Klassen bzw. zwischen deren Elementen. Im dritten Abschnitt wird die Darstellung von Inklusions- (Subklassen-) und Exklusionsionsbeziehungen untersucht. Außerdem wird die Darstellung von Beziehungen zwischen Elementen durch Objektreferenzen behandelt. Die Darstellung von Beziehungen durch ein eigenes Konstrukt in einer objekt-orientierten Umgebung ist eine Besonderheit des *Object-Relationship*-Modells. Im vierten Abschnitt wird beschrieben, wie ein solches Konstrukt in einem *Add-On*-Ansatz aussehen kann. Dabei werden wiederum die Aspekte Rücksetzbarkeit von Operationen und Integritätsüberwachung berücksichtigt. Der letzte Abschnitt des Kapitels untersucht die Darstellung von Objekten in einem *Add-On*-Ansatz.

5.1 Architektur der Implementierung

Abbildung 5.1 gibt einen Überblick über die Architektur der Implementierung des *Object-Relationship*-Modells.

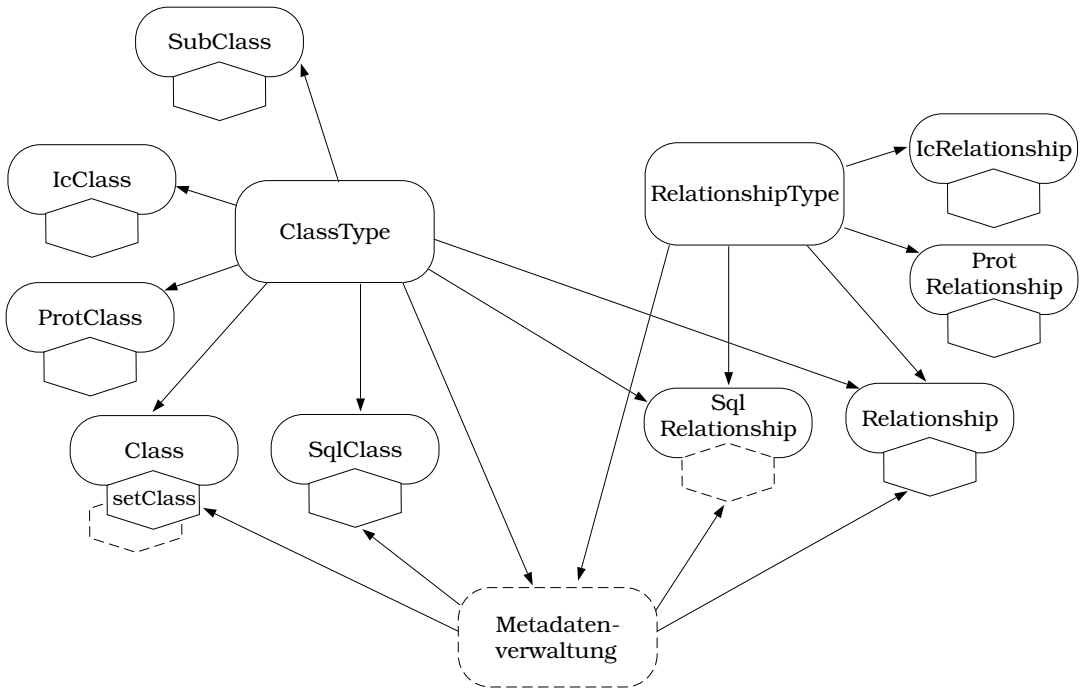

Abbildung 5.1: Architekturüberblick

Die Kasten mit den abgerundeten Ecken stellen Schnittstellen dar und die Sechsecke Implemen-

tationsmodule[1]. Die Pfeile zeigen Import-Beziehungen zwischen Komponenten an. Dabei zeigen die Pfeilspitzen stets auf die importierenden Komponenten. Nicht implementierte Teile sind durch gestrichelte Linien gekennzeichnet.

Im Mittelpunkt der Architektur stehen die Schnittstellen *ClassType* und *RelationshipType*, die die Typen für Klassen und Beziehungen (*Relationships*) ihre Verfügung stellen. Klassen sind hier so implementiert, das ihr Typ zugleich auch die Benutzerschnittstelle bildet (siehe Abschnitt 5.2.1). Dasselbe gilt für die Typen von Beziehungen (siehe Abschnitt 5.4.1). Die von *ClassType* zur Verfügung gestellten Typen werden von folgenden Schnittstellen verwendet:

Class: Diese Schnittstelle stellt Funktionen zum Erzeugen von Klassen mit und ohne Schlüssel zur Verfügung (Abschnitt 5.2.2). Das Modul *setClass*, bei dem die Elemente der Klasse in einer Menge verwaltet werden, ist eine mögliche Implementierung der Schnittstelle *Class*. Man kann sich noch weitere Implementierungen vorstellen, bei denen die Elemente der Klasse in anderen Strukturen wie z.B. Listen gespeichert werden.

SqlClass: Die Elemente einer Klasse können auch in einer *SQL*-Tabelle verwaltet werden. Wie in Abschnitt 5.2.2 beschrieben wird, sind in diesem Fall beim Erzeugen einer Klasse zusätzliche Parameter notwendig, d.h., die Funktionen zum Erzeugen von Klassen haben eine andere Signatur. Aus diesem Grund wird für das Erzeugen von Klassen auf der Basis von *SQL*-Tabellen eine eigene Schnittstelle *SqlClass* angeboten.

ProtClass: Diese Schnittstelle stellt eine Funktion zum Übergang zu Klassen mit geschützten Operationen zur Verfügung (Abschnitt 5.2.3). Geschützte Operationen werden im Fall des Abbruchs einer Transaktion zurückgesetzt (vgl. Abschnitt 4.3.5).

IcClass: Es ist möglich, für eine Klasse Integritätsbedingungen anzugeben, wenn sie vorher als Klasse mit Integritätsüberwachung definiert worden ist. Die Schnittstelle bietet sowohl eine Funktion an, die eine Umwandlung in eine solche Klasse erlaubt, als auch Funktionen zur Angabe von Integritätsbedingungen, die bei den Operationen auf Klassen automatisch getestet werden (Abschnitt 5.2.3), und zum Löschen solcher Bedingungen.

SubClass: Die Schnittstelle *SubClass* stellt Funktionen für den dynamischen Aufbau von Generalisierungshierarchien zur Verfügung. Außerdem bietet sie Funktionen an, mit denen Exklusionsbeziehungen zwischen Klassen festgelegt werden können. Sowohl Subklassenbeziehungen als auch Exklusionsbeziehungen können nicht nur dynamisch angelegt sondern, auch gelöscht werden.

Relationship/SqlRelationship: Beziehungen zwischen Klassen werden im *Object-Relationship*-Modell durch ein eigenes Konstrukt dargestellt. Diese Schnittstellen beinhalten Funktionen zum Erzeugen einer Beziehung zwischen zwei Klassen. Die Abhängigkeit der Schnittstellen von der Schnittstelle *ClassType* ergibt sich aus der Tatsache, daß bei der Erzeugung einer Beziehung die beteiligten Klassen angegeben werden (Abschnitt 5.4.2).

Die Architektur für die Beziehungen (*Relationships*) ist weitgehend symmetrisch aufgebaut. Die Schnittstellen *ProtRelationship* und *IcRelationship* (siehe Abschnitt 5.4.3) stellen die selbe Funktionalität für Beziehungen zur Verfügung wie die Schnittstellen *ProtClass* und *IcClass* für Klassen.

[1] Die Namen der meisten Implementationsmodule wurden der Übersichtlichkeit halber weggelassen.

Ein weiterer wichtiger Teil der Architektur eines Datenbanksystems ist eine Metadatenverwaltung (Datenwörterbuch). Sie verwaltet Metainformation über die definierten Datenbanken wie z.B. die Menge aller existierenden Klassen und die Beziehungen zwischen diesen Klassen.

Eine Metadatenverwaltung benötigt Zugriff auf die Typen der relevanten Strukturen wie z.B. die Typen von Klassen und Beziehungen im *Objekt-Relationship*-Modell. Zum anderen muß jede Klasse und Beziehung bei ihrer Erzeugung in das Datenwörterbuch eingetragen werden. Zu diesem Zweck importieren die Implementationsmodule zu den Schnittstellen *Class*, *SqlClass*, *Relationship* und *SqlRelationship* Funktionen der Metadatenverwaltung, die einen entsprechenden Eintrag in das Datenwörterbuch durchführen. Im Prototyp wurde auf eine Implementierung einer Metadatenverwaltung verzichtet.

5.2 Klassen

In der Sprache *ORM* werden Klassen als Mengen von Daten betrachtet, die durch Inklusionsbeziehungen miteinander verbunden sein können [2]. Basisoperationen auf Klassen sind das Erzeugen von Klassen, das Einfügen und Löschen von Elementen und die Verschiebung von Elementen zwischen Klassen und Subklassen (vgl. Abschnitt 2.2.1).

Auch in dem hier vorgestellten Ansatz liegt bei der Definition der Klassen der Schwerpunkt auf dem Aspekt der Klassenausprägung. Die Konzepte von Klassen und Objekten werden wie beim *ORM* unabhängig voneinander realisiert, was im Gegensatz zu den meisten objektorientierten Ansätzen steht, bei denen diese beiden Konzepte fest miteinander verbunden sind. Bei dem realisierten Ansatz besteht die Verbindung darin, daß Objekttypen mögliche Elementtypen für Klassen sind.

Die Realisierung von Objekten in einem *Add-On*-Ansatz wird in Abschnitt 5.5 behandelt. In diesem Abschnitt werden folgende Themen betrachtet: Zunächst wird auf die Benutzerschnittstelle und den Aufbau von Klassen eingegangen. Dabei zeigt sich, daß die Wahl der Schnittstelle einen starken Einfluß auf den Typ von Klassen hat. Es wird untersucht, wie sich Klassen mit und ohne Schlüssel voneinander unterscheiden und wie man zu einer einheitlichen Darstellung kommen kann. Der zweite Teil des Abschnitts beschäftigt sich mit speziellen Aspekten der Implementierung von Klassen. Der hier gewählte Ansatz erweist sich dabei als sehr flexibel, da die Implementierung auf verschiedene Arten von Kollektionen (z.B. Mengen, Listen, *SQL*-Tabellen) aufsetzen kann und dem Benutzer trotzdem eine einheitliche Schnittstelle zur Verfügung stellt. In Teil 5.2.3 des Abschnitts wird der Übergang zu Klassen mit geschützten Operationen und Integritätsüberwachung behandelt. Insbesondere werden Funktionen vorgestellt, die eine existierende Klasse auf eine Klasse, deren Operationen geschützt sind, bzw. auf eine Klasse mit Integritätsüberwachung abbilden. Im letzten Teil des Abschnitts wird die Realisierung einer Reihe spezieller Integritätsbedingungen auf Klassen beschrieben. Hierbei wird auf die Zuordnung der Integritätsbedingungen zu den Operationen und die Herleitung der notwendigen Tests eingegangen.

[2] Auf Inklusions- und Exklusionsbeziehungen wird im Abschnitt 5.3.1 genauer eingegangen

5.2.1 Ein Typ für Klassen

Die Form der Schnittstelle, die man dem Benutzer zur Verfügung stellen möchte, hat einen starken Einfluß auf die Struktur des Typs für Klassen. Deshalb ist zunächst zu überlegen, wie diese Schnittstelle aussehen soll. Es wird hier, wie in der Sprache *ORM* vorgeschlagen, ein Format für die Operationen auf Klassen gewählt, das an den Aufruf von Methoden bei Objekten erinnert:

> Sei *baseParts* eine Klasse von Basisteilen mit Elementtyp *BasePart* und *part1* und *part2* Elemente dieses Typs. Die Einfüge- und Löschoperationen sollen wie folgt aussehen:
>
> *baseParts.insert(part1)*
> *baseParts.delete(part2)*

Um ein solches Format zu erreichen, muß der Typ von Klassen wie folgt strukturiert sein:

Def *T(E ::*__TYPE__*) ::*__TYPE__ =*
  Tuple
    insert(e :E) :__Ok__
    delete(e :E) :__Ok__
    ...
  end

Der Typ der Klasse hängt vom Typ ihrer Elemente ab. Man braucht zu seiner Beschreibung also einen Typoperator, mit dem der jeweilige Elementtyp auf den zugehörigen Typ der Klasse abgebildet werden kann. In eine Klasse vom Typ *T(:BasePart)* kann dann durch obige Operation eingefügt werden.

Neben den Operationen *insert* und *delete* zum Einfügen und Löschen von Elementen wird dem Benutzer eine Funktion *deleteAll* für das prädikative Löschen und durch die Funktion *elements* eine Möglichkeit, über die Elemente der Klasse zu iterieren (vgl. Abschnitt 4.2.1), zur Verfügung gestellt. Die Funktion *deleteAll* erhält ein Prädikat als Parameter und löscht alle Elemente, die dieses Prädikat erfüllen. Im Gegensatz zur Funktion *delete* kann mit dieser Funktion ein Element gelöscht werden, ohne daß man seine Identität kennt. Das zu löschende Element kann durch die Verwendung eines geeigneten Prädikats über seinen Wert identifiziert werden.

Eine Klasse hat einen internen Zustand, auf den nur diese Operationen zugreifen können. Dieser beinhaltet unter anderem die eigentliche Kollektion der Elemente der Klasse. Dieser Ansatz hat den Vorteil, daß man verschiedene Implementierungen wählen kann und trotzdem die gleiche Schnittstelle für den Benutzer erhält. Neben diesem internen Zustand sollen noch weitere Informationen zu einer Klasse verwaltet werden, die auch für andere als die oben genannten Operationen zugreifbar sind. Dazu gehören z.B. der Name der Klasse und eine Ausgabefunktion für die Elemente. Diese Ausgabefunktion bildet ein Element auf eine (möglichst identifizierende) Zeichenkette ab. Sie stellt z.B. für die Erzeugung von präzisen Fehlermeldungen eine große Hilfe dar. Für den Elementtyp *Part* kann diese Funktion z.B. wie folgt aussehen:

let *partId(p :Part) = p.name*

Der Typ für die Klassen wird um eine Komponente *info* erweitert, in der diese Informationen zusammengefaßt werden [3]. Er ergibt sich damit zu:

Let *T(E ::***TYPE***) ::***TYPE** =
  Tuple
    info :InfoType(E)
    insert(e :E) **:Ok**
    delete(e :E) **:Ok**
    deleteAll(p(:E) **:Bool***)* **:Ok**
    elements() :Iter_T(E)
  end

Alternativ kann der Typoperator T zur Erzeugung der Typen von Klassen auch als abstrakter Datentyp definiert werden:

*T ::***ALL***(E ::***TYPE***) ::***TYPE**

Der Benutzer erhält als einzige Information die Signatur des Typoperators. Mit dieser Definition von T haben die Operationen *insert* und *delete* eine andere Signatur. Die Einfügeoperation kann z.B. wie folgt definiert werden:

*insert(E ::***TYPE** *class :T(E) e :E)* **:Ok**

Die Operationen müssen neben dem Element auch die betreffende Klasse als Parameter erhalten. Außerdem handelt es sich um polymorphe Funktionen, da sie auf Klassen mit beliebigem Elementtyp E arbeiten können müssen (Elementtyp E als Typparameter).

Klassen mit Schlüssel

Schlüssel werden in Klassen eingeführt, um die Eindeutigkeit der Elemente bezüglich ihrer Schlüsselwerte zu sichern und eine Identifizierung von Elementen über Werte (hier den Schlüssel) zu ermöglichen. Außerdem wird vom System meist ein schneller Zugriff auf Elemente über ihren Schlüssel unterstützt. Im folgenden wird untersucht, welche Aspekte bei Einführung des Konzepts eines Schlüssels in die Klassen zu berücksichtigen sind.

Die Schnittstelle muß um eine Funktion (*lookup*) für den Schlüsselzugriff erweitert werden. Eine solche Funktion erhält einen Schlüsselwert als Parameter und liefert das zugehörige Element der Klasse zurück. Bei Angabe eines nicht existierenden Schlüssels wird eine Ausnahme erzeugt. Wenn der Schlüssel vom Typ K ist und der Elementtyp der Klasse E, dann ist die Funktion *lookup* vom Typ **All***(:K) :E*. Die Anwendung der Funktion sieht für die Klasse *parts* wie folgt aus:

[3] Weitere Komponenten von *info* werden an gegebener Stelle motiviert und eingeführt.

Die Anfrage

parts.lookup(3)

liefert das Teil mit der Nummer drei zurück.

Der Typ der Klassen wird um die Funktion für den Schlüsselzugriff erweitert:

Let *T(E, K ::***TYPE***) ::***TYPE** =
  Tuple
    info : InfoType(E K)
    insert(:E) :**Ok**
    delete(:E) :**Ok**
    deleteAll(p(:E) :**Bool***)* :**Ok**
    lookup(:K) :E
    elements() :Iter_T(E)
  end

Wie man sieht, hängt der Typ einer Klasse nicht mehr nur vom Typ ihrer Elemente, sondern auch vom Typ ihres Schlüssels ab.

Bei dem hier gewählten Ansatz besteht die Definition eines Schlüssels in der Angabe von zwei Funktionen: Einer Funktion *key* zur Schlüsselgewinnung und einer Funktion *equal* zum Schlüsselvergleich. Die Funktion *key* legt fest, wie ein Element vom Typ E auf seinen Schlüssel vom Typ K abgebildet wird. Sie ist vom Typ **All***(:E) :K* und besteht oft nur aus einer Komponenten-Selektion (ein Attribut als Schlüssel) oder der Kombination mehrerer selektierter Komponenten zu einem Schlüssel. Für die Klasse der Teile sieht die Funktion zur Schlüsselgewinnung z.B. wie folgt aus:

let *partKey(p :Part)* :**Int** = *p.pno*

Es sind aber auch beliebig komplexe Operationen zur Schlüsselgewinnung denkbar, wie sie etwa bei mengenwertigen Attributen auftreten können.

Die Funktion *equal* legt fest, wann zwei Schlüssel gleich sind. Sie nimmt zwei Schlüssel vom Typ K als Parameter und testet sie auf Gleichheit, ist also vom Typ **All***(:K :K)* :**Bool**. Eine Funktion zum Schlüsselvergleich wird benötigt, da die Verwendung des Operators *is* zwar auf allen Typen zulässig ist, jedoch auf Identität testet. Dies liefert besonders bei zusammengesetzten Schlüsseln oft nicht den gewünschten Effekt, da man nicht auf Identität, sondern auf Wertegleichheit der Komponenten testen möchte. Für das gewählte Beispiel genügt allerdings ein einfacher Vergleich durch *is*:

let *pnoEqual(i1, i2* :**Int***)* :**Bool** = *i1* **is** *i2*

Die Funktionen für Schlüsselgewinnung und -vergleich müssen vom Benutzer bei der Definition einer Klasse mit Schlüssel angegeben werden, um den Schlüssel festzulegen. Man benötigt sie für den Schlüsselzugriff und die Wahrung der Eindeutigkeit bezüglich der Schlüsselattribute.

Die Funktionen werden in die Komponente *info* des Typs einer Klasse aufgenommen[4], um anderen Operationen einen Zugriff auf sie zu ermöglichen. Dies kann sich z.B. bei der Formulierung von Integritätsbedingungen als hilfreich erweisen. Durch diese Erweiterung hängt auch der Typ der Komponente *info* vom Schlüsseltyp der Klasse ab:

Tuple
...
key(:E) :K
equal(:K :K) **:Bool**
...
end

Vereinheitlichung von Klassen mit und ohne Schlüssel

Durch die zusätzliche Abhängigkeit der Klassen mit Schlüssel vom Typ ihres Schlüssels haben sich verschiedene Typoperatoren für die Klassen mit und ohne Schlüssel ergeben. Das hat den Nachteil, daß man alle Operationen auf Klassen in zwei Versionen schreiben muß. Um dies zu vermeiden, kann man dazu übergehen, Klassen ohne Schlüssel als spezielle Klassen mit Schlüssel mit folgenden Eigenschaften zu betrachten:

1. $K = E$ (Elementtyp als Schlüsseltyp)
2. *key(e :E) :E* $= e$ (Identitätsfunktion zur Schlüsselgewinnung)
3. *equal(e1, e2 :E) = e1 is e2* (Identität als Schlüsselgleichheit)

Damit ist der Typoperator für die Klassen mit Schlüssel für alle Klassen zu verwenden. Klassen ohne Schlüssel mit Elementtyp E sind vom Typ *T(E E)*. Die Funktion *lookup* für den Schlüsselzugriff ist bei Klassen ohne Schlüssel ein Enthaltenseins-Test. Sie erhält wegen $K = E$ ein Element vom Typ E als Parameter und testet, ob es in der Klasse enthalten ist. Ist dies nicht der Fall, wird wie bei Angabe eines nicht existierenden Schlüssels eine Ausnahme erzeugt.

5.2.2 Implementierungen

Das Erzeugen einer Klasse besteht im wesentlichen aus dem Anlegen einer Struktur, in der die Elemente der Klasse abgelegt werden, und aus der Definition der Operationen *insert*, *delete*, *deleteAll*, *lookup* und *elements* für diese Struktur. Wie Abbildung 5.2 zeigt, kann es mehrere Implementierungen für Klassen geben, basierend auf unterschiedlichen Kollektionstypen wie z.B. Listen, Mengen und SQL-Tabellen.

Jede Implementierung stellt eigene Funktionen *create* und *createKeyed* zur Erzeugung neuer Klassen zur Verfügung. Diese können verschiedene Parameterlisten besitzen, erzeugen aber für den gleichen Elementtyp E und den gleichen Schlüsseltyp K alle Werte vom gleichen Typ, nämlich *ClassType_T(E K)*. Somit haben alle erzeugten Klassen den selben Typ und die selbe

[4] Dies ist möglich, da Funktionen Werte erster Klasse sind und somit auch als Tupelkomponenten abgespeichert werden können.

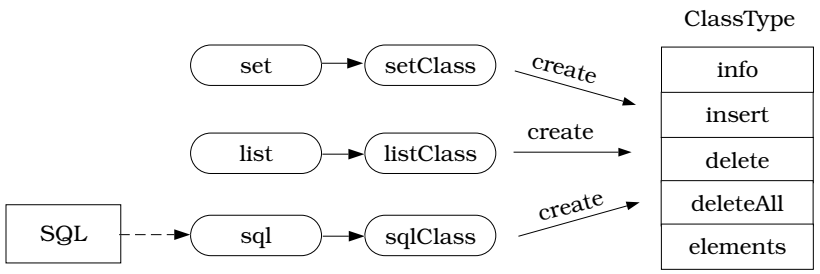

Abbildung 5.2: Implementierungen für Klassen

Benutzerschnittstelle. Dies hat neben der Benutzerfreundlichkeit den Vorteil, daß man höhere Konzepte über Klassen wie z.B. Beziehungen unabhängig von der Struktur definieren kann, die der Klasse zugrundeliegt.

Im Rahmen der vorliegenden Arbeit wurden exemplarisch ein Modul *setClass* und ein Modul *sqlClass* implementiert. Im Modul *setClass* werden die Elemente der Klasse in einer Menge vom Typ *set.T(E)* verwaltet. Beim Erzeugen einer Klasse wird eine leere Menge mit dem Elementtyp E der Klasse angelegt. Die Implementierung der Einfüge- und Löschoperationen besteht im wesentlichen aus Aufrufen der entsprechenden Funktionen der Schnittstelle *Set*. Auch für die Berechnung einer Iteration über die Elemente der Klasse existiert bereits eine Funktion *set.elements*. Da die Schnittstelle *Set* weder prädikatives Löschen noch einen Schlüsselzugriff unterstützt, müssen die Funktionen *lookup* und *deleteAll* mit Hilfe der Iteration über die Elemente der Menge implementiert werden. Bei *deleteAll* werden zunächst alle Elemente der Menge bestimmt, die das angegebene Prädikat erfüllen[5]. Diese Elemente werden dann einzeln aus der Menge gelöscht. Für die Funktion *lookup* kann das Element e mit dem angegebenen Schlüssel k z.B. auf folgende Weise bestimmt werden:

let e = *iter.any(set.elements(classExt)* **fun**$(e : E)$ *equal(key(e)) = k),*

wobei *key* und *equal* die oben eingeführten Funktionen für Schlüsselzugriff und -vergleich sind. Bei Klassen mit Schlüsseln muß die Eindeutigkeit der Elemente bezüglich des Schlüssels sichergestellt werden. Mengen sind im Modul *set* so implementiert, daß das Einfügen von Duplikaten verhindert wird (Erzeugen einer Ausnahme). Dabei besteht die Möglichkeit, beim Erzeugen einer Menge eine Funktion anzugeben, die festlegt, wann zwei Elemente gleich sind. Im Modul *setClass* kann man die Eindeutigkeit der Elemente bezüglich des Schlüssels also sicherstellen, indem man zur Definition der Gleicheit der Elemente der Menge die Schlüsselgleichheit wählt.

Die Funktion *setClass.create* zum Erzeugen von Klassen benötigt folgende Parameter: den Typ der Elemente, den Namen der Klasse und eine Ausgabefunktion für die Elemente, die bereits in Abschnitt 5.2.1 beschrieben wird. Bei Klassen mit Schlüssel (*createKeyed*) kommen der Typ des Schlüssels und die Funktionen zur Schlüsselgewinnung und zum Schlüsselvergleich hinzu. Der Aufruf zum Erzeugen der Klasse der Teile *parts*, z.B., sieht mit den weiter oben bereits definierten Funktionen *partId*, *partKey*, *pnoEqual* wie folgt aus:

[5]Dies kann z.B. durch einen Aufruf der Funktion *select* der Schnittstelle *Iter* geschehen.

```
let parts = class.createKeyed(:Part :Int "Parts" partId partKey pnoEqual)
```

Beim Modul *sqlClass* werden die Elemente der Klasse in einer *SQL*-Tabelle verwaltet. Für die Implementierung wurde die am Arbeitsbereich DBIS der Universität Hamburg erstellte Schnittstelle *Sql* verwendet, die es erlaubt, aus der Sprache *P-Quest* heraus auf *SQL*-Datenbanken zuzugreifen. Da der Sprache *SQL* [Dat89, SQL87] das relationale Modell zugrundeliegt, ergeben sich die Einschränkungen, daß sowohl für den Elementtyp als auch für den Schlüsseltyp nur flache Tupel zugelassen sind und als Attributtypen nur in *SQL* zulässige Typen.

Beim Erzeugen einer Klasse wird eine *SQL*-Tabelle angelegt. Die Erzeugung einer *SQL*-Tabelle kann z.B. für die Klasse *parts* wie folgt aussehen:

```
CREATE TABLE parts (pno INT2 NOT NULL WITH DEFAULT 0,
                    name CHAR WITH NULL NOT DEFAULT)
```

Das *SQL*-System erwartet diesen Befehl in Form von einer Zeichenkette. Diese Zeichenkette wird mit Typüberprüfung von der Funktion *createTable* der Schnittstelle *Sql* erzeugt. Sie benötigt dafür jedoch Metainformation über den Aufbau des Elementtyps und Information über Defaultwerte und die Zulässigkeit von Nullwerten. Die Schnittstelle *Sql* stellt Funktionen zur Verfügung[6], die es erlauben, die benötigten Angaben für einzelne Attribute zu machen. Beim Erzeugen einer Klasse muß der Elementtyp durch geeignete Aufrufe dieser Funktionen beschrieben werden. Außerdem muß bei Klassen mit Schlüssel angegeben werden, welche Attribute Teil des Schlüssels sind. Daraus ergibt sich folgende Signatur für die Funktion *createKeyed* der Schnittstelle *SqlClass*:

```
createKeyed(E, K ::TYPE name :String objId(:E) :String
        key(:E) :K  eq(:K :K) :Bool
        atributes :Array(sql.Attribute) keyAttributes: Array(Int)) :T(E K)
```

Die Klasse *parts* kann durch folgenden Aufruf der Funktion erzeugt werden:

```
let parts = createKeyed(:Parts :Int "Parts" partId partKey pnoEqual
        array of
        sql.newInt2Attribute(let name = "pno"
                             let nullable = false
                             let default = 0)
        sql.newCharAttribute(let name = "name"
                             let nullable = true
                             let maxsize = 45
                             let default = optional.nil(:String))end
        array of 0 end)
```

Für die Definition von *insert* und *delete* stellt die Schnittstelle *Sql* geeignete Funktionen zur Verfügung, die das Einfügen bzw. Löschen einzelner Tupel in eine bzw. aus einer Tabelle

[6] Es gibt für jeden *SQL*-Typ eine eigene Funktion.

erlauben. Auch für die Berechnung einer Iteration über die Elemente einer Tabelle bietet die Schnittstelle eine entsprechende Funktion an. Es sind also nur noch die Funktionen *deleteAll* und *lookup* zu untersuchen.

In *SQL* ist es möglich, einen Schlüsselindex festzulegen. Die Schnittstelle *Sql* stellt eine Funktion *modifyToUnique* zur Verfügung, die zu einer Tabelle einen Schlüsselindex erzeugt. Nach der Festlegung eines solchen Indexes wird ein schneller Zugriff über den entsprechenden Schlüssel unterstützt. Ein Schlüsselzugriff für eine Tabelle *A* ist in *SQL* von der Form:

*SELECT **
FROM A
WHERE
(keyAttrName1 = keyAttrValue1) $\wedge$
(keyAttrName2 = keyAttrValue2) $\wedge$
...

Die Schnittstelle *Sql* stellt eine Funktion *lookup* zur Verfügung, die den Schlüsselzugriff unterstützt. Sie kann zur Implementierung der Funktion *lookup* für die Klassen verwendet werden. Nachdem für eine Klasse ein Schlüsselindex angelegt worden ist, stellt das *SQL*-System auch Verletzungen der Eindeutigkeit des Schlüssels fest. Diese führen aber zu *SQL*-Fehlern und nicht, wie gewünscht, zu einer Ausnahme in der Sprache *P-Quest*. Um dies zu umgehen, kann vor dem Einfügen eines Elements durch eine *if*-Abfrage getestet werden, ob bereits ein Element mit diesem Schlüssel existiert. Die genaue Form des notwendigen Tests wird bei der Behandlung der Eindeutigkeitsbedingungen in Abschnitt 5.2.4 hergeleitet.

Beim prädikativen Löschen tritt folgendes Problem auf: Die Schnittstelle *Sql* stellt zwar eine Funktion für diesen Zweck zur Verfügung, die zu löschenden Elemente können aber nur über ein SQL-Prädikat selektiert werden[7]. Die Funktion *deleteAll* hingegen erhält ein *P-Quest*-Prädikat, d.h. eine boolesche Funktion als Parameter. Diese kann nicht in ein SQL-Prädikat umgewandelt werden. Deshalb wird die Funktion *deleteAll* wie im Modul *setClass* implementiert: Man berechnet aus der Iteration über alle Elemente die Elemente, die das Prädikat erfüllen, und löscht sie einzeln aus der Tabelle[8].

Neben den Funktionen *create* und *createKeyed* zum Erzeugen neuer Klassen auf der Basis von *SQL*-Tabellen implementiert das Modul *sqlClass* auch noch Funktionen *convertToClass* und *convertToKeyedClass*, die es erlauben, *SQL*-Tabellen in Klassen umzuwandeln. Der Datenbestand der Tabelle wird dabei übernommen. Auf diese Weise können bereits existierende *SQL*-Tabellen mit evtl. erheblichen Datenbeständen ohne Verlust von Daten in die Umgebung des *Object-Relationship*-Modells integriert werden. Die Funktionen sind ähnlich wie die Funktionen zum Erzeugen von Klassen implementiert, nur daß das Anlegen einer Tabelle wegfällt. Als zusätzlichen Parameter erwarten die Funktonen den Namen der *SQL*-Tabelle, die integriert werden soll.

Der Benutzer der Klasse erhält unabhängig von der Implementierung die gleiche Schnittstelle. So kann mit der folgenden Operation in beide erzeugten Klassen *parts* eingefügt werden:

*parts.insert(***tuple let** *pno =3* **let** *name = "Table"* **end***)*

[7] Diese müssen zur Konstruktion der Anfrage außerdem als Zeichenketten vorliegen.

[8] Diese Methode führt zu einem Verlust an Effizienz, der zugunsten der Allgemeinheit des Prädikats in Kauf genommen wird.

5.2.3 Geschützte Operationen und Integritätsüberwachung

In diesem Abschnitt geht es darum, die in den Abschnitten 4.3.5 und 4.4.4 vorgestellten Konzepte der geschützten Operationen und der Operationen mit Integritätsüberwachung auf Klassen anzuwenden. Wie im vorigen Abschnitt beschrieben wird, gibt es verschiedene Implementierungen für Klassen. Der Übergang von einer Klasse zu einer solchen mit geschützten Operationen bzw. mit Integritätsüberwachung sollte möglichst unabhängig von der Implementierung sein, damit man ihn nicht für jede Implementierung neu entwerfen muß. Hierbei ist es von Vorteil, daß die Klassen unabhängig von der unterliegenden Implementierung den gleichen Typ (die gleiche Schnittstelle) besitzen. Dadurch ist es möglich, eine Funktion *protect* zu schreiben, die eine beliebige Klasse C auf eine neue Klasse mit geschützten Operationen abbildet und eine Funktion *iChecked*, die ebenfalls eine Klasse C als Parameter erhält und eine Klasse, bei deren Operationen Integritätsüberwachung durchgeführt wird, zurückliefert. Abbildung 5.3 zeigt die Wirkung der Funktionen.

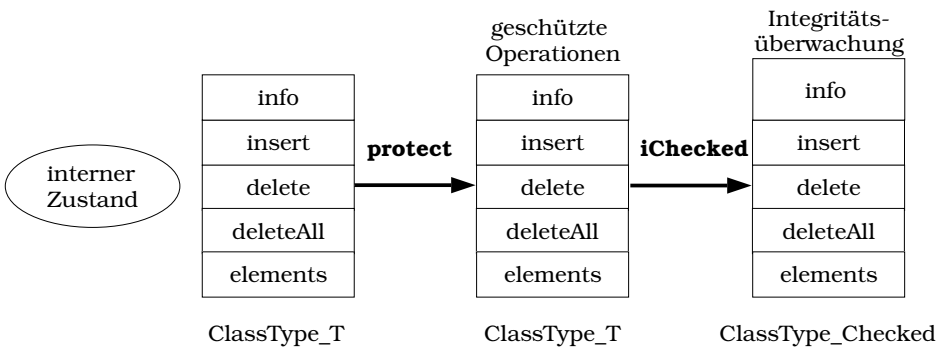

Abbildung 5.3: Die Funktionen *protect* und *iChecked*

protect und *iChecked* definieren die Operationen der neuen Klasse unter Verwendung der Operationen der als Parameter übergebenen Klasse C. Dadurch werden sowohl implementationsabhängige Details als auch indirekt der Zustand der Klasse C und damit die Kollektion ihrer Elemente übernommen. Wie die Abbildung bereits andeutet, empfiehlt es sich, *iChecked* nur auf Klassen mit geschützten Operationen anzuwenden: Wenn eine Operation eine verzögerte Integritätsbedingung nicht erfüllt, wird dies erst am Ende der Transaktion festgestellt. Diese Operation sollte dann zurückgesetzt werden, um wieder einen konsistenten Zustand herzustellen. Die Funktion *protect* wird von der Schnittstelle *ProtClass* zur Verfügung gestellt und die Funktion *iChecked* von der Schnittstelle *IcClass*. Im folgenden werden die beiden Funktionen genauer beschrieben.

Klassen mit geschützten Operationen

Die Funktion *protect* nimmt eine Klasse als Parameter und gibt eine Klasse mit geschützten Operationen zurück. Wenn man z.B. die Klasse der Teile definiert als

let *baseParts* = *protect(setClass.createKeyed(:BaseParts ...))*

wird die Funktion *baseParts.insert* innerhalb von Transaktionen als geschützte Operation ausgeführt und im Falle eines Abbruchs der Transaktion zurückgesetzt. *protect* kann jedoch auch für Klassen aufgerufen werden, die bereits Elemente enthalten. Die neue Klasse übernimmt dann die Kollektion der Elemente der ursprünglichen Klasse.

Bei der Implementierung der Funktion *protect* wird nach den in Abschnitt 4.3.6 für den Entwurf geschützter Operationen aufgestellten Regeln vorgegangen: Es ist zunächst zu überlegen, welche Operationen zu schützen sind. Bei den Klassen sind das die Operationen *insert*, *delete* und *deleteAll*. Bei der Definition der Operationen der neuen Klasse werden die der ursprünglichen Klasse verwendet. Die notwendigen kompensierenden Operationen werden bereits in Abschnitt 4.3.6 hergeleitet: *insert* wird durch *delete* kompensiert und umgekehrt. Beim prädikativen Löschen müssen die Elemente, die das Prädikat erfüllen, zwischengespeichert werden. Die kompensierende Operation fügt diese Elemente einzeln wieder ein. Dazu wird die Einfügeoperation der ursprünglichen Klasse verwendet.

Klassen mit Integritätsüberwachung

Die Funktion *iChecked* nimmt eine Klasse als Parameter und bildet sie auf eine Klasse mit Integritätsüberwachung ab. Durch die Definition

let *baseParts = iChecked(protect(setClass.createKeyed(:BaseParts ...)))*

werden für die Klasse *baseParts* Kollektionen für Integritätsbedingungen angelegt. Außerdem werden die Operationen auf der Klasse mit Integritätsüberwachung definiert: Beim Aufruf der Operation *baseParts.insert*, z.B., werden die Integritätsbedingungen gestestet, die aktuell zu dieser Operation gespeichert sind.

Die Implementierung der Funktion *iChecked* orientiert sich an den Regeln aus Abschnitt 4.4.4. Dabei sind zwei Aufgaben zu erfüllen: Es müssen Kollektionen für die Integritätsbedingungen angelegt werden, und die Integritätsüberwachung bei den Operationen muß realisiert werden.

Zum Anlegen und für die Verwaltung der Kollektionen können die Funktionen der in 4.4.4 eingeführten Schnittstelle *BulkConstraints* verwendet werden. Dadurch werden Kollektionen für Einfüge- und Löschoperationen angelegt. Die Komponente *info* wird um diese Kollektionen erweitert. So ergibt sich ein neuer Typ *ICInfoTyp* für diese Komponente und damit auch ein neuer Typ *Checked* für die ganze Klasse:

Def *ICInfoType(E, K::***TYPE***) ::***TYPE** =
  Tuple
    ...
    constraints :bulkConstraints.T(E)
  end

Def *Checked(E, K ::* **TYPE***) ::***TYPE** =
  Tuple
    info :ICInfoType(E K)
    *insert(e :E) :***Ok**
    ...
  end

Diese beiden Typen sind in der Schnittstelle *ClassType* enthalten. Die Funktion *iChecked* bildet also eine Klasse vom Typ *ClassType_T(E K)* auf eine Klasse vom Typ *ClassType_Checked(E K)* ab.

Eine Integritätsüberwachung muß bei den Operationen *insert*, *delete* und *deleteAll* durchgeführt werden. Wie bereits bei Angabe der Regeln erläutert, können die notwendigen Tests der Bedingungen durch einen Aufruf der Funktion *test* der Schnittstelle *BulkConstraints* implementiert werden. Daraus ergibt sich direkt die Implementierung der Operationen *insert* und *delete* mit Integritätsüberwachung. Beim prädikativen Löschen müssen die für das Löschen von Elementen gespeicherten Bedingungen für jedes einzelne zu löschende Element getestet werden. Diese Operation setzt sich also aus Löschoperationen mit Integritätsüberwachung für die einzelnen zu löschenden Elemente zusammen und kann durch einen wiederholten Aufruf der Funktion *delete* mit Integritätsüberwachung implementiert werden.

5.2.4 Integritätsbedingungen auf Klassen

Durch den Übergang zu Klassen mit Integritätsüberwachung erhält man Klassen mit Kollektionen für Integritätsbedingungen, die beim Einfügen und Löschen von Elementen getestet werden. Die Frage, die sich jetzt ergibt, ist, wie diese Kollektionen gefüllt werden. Die Integritätsbedingungen müssen dazu den Operationen zugeordnet werden. Da man aus Effizienzgründen nicht jede Integritätsbedingung vor jeder Operation oder am Ende jeder Transaktion testen möchte, ist für jede Integritätsbedingung zu überlegen, durch welche Operationen sie verletzt werden kann und welche Tests notwendig sind, um eine Verletzung zu vermeiden bzw. zu entdecken (siehe dazu auch Abschnitt 4.4).

Im ersten Teil dieses Abschnitts werden Regeln angegeben, die bei diesem Problem helfen können. In den weiteren Teilen werden spezielle Klassen von Integritätsbedingungen untersucht und mit Hilfe der Regeln umgeformt. Ziel des Ansatzes ist es dabei nicht, allgemeine Integritätsbedingungen auf Klassen zu unterstützen, sondern sich auf einige spezielle zu konzentrieren, die häufig auftreten. Für diese wird die Zuordnung zu den Operationen als Dienst angeboten.

Transformationsregeln

Die hier vorgestellten Regeln setzen voraus, daß die Integritätsbedingungen in Pränex-Normalform vorliegen. Das bedeutet, daß alle Quantifizierungen am Anfang der Bedingung stehen. Dies stellt jedoch keine entscheidende Einschränkung dar, da Standardmethoden existieren, um beliebige Bedingungen systematisch in Bedingungen dieser Form zu transformieren [Ric83]. Die in der Literatur angegebenen Regeln beziehen sich auf Relationen. Sie lassen

sich aber allgemein auf Kollektionen von Elementen und somit auch auf Klassen und auf die später eingeführten *Beziehungen* übertragen.

Folgende Regeln werden von Ling und Rajagopalan in [LR84] vorgestellt und deren Gültigkeit bewiesen. Sie geben an, welche Operationen welche Integritätsbedingungen (nicht) verletzen können.

Indifferenzregel:
Eine Änderungsoperation (Einfügen oder Löschen) auf einer Relation R verletzt eine Integritätsbedingung nicht, wenn keine der Variablen der Bedingung an die Relation R gebunden ist.

Regel für das Einfügen:
Das Einfügen eines neuen Tupels in eine Relation R der Datenbank verletzt eine Integritätsbedingung nicht, wenn in dieser Integritätsbedingung alle der Relation R zugeordneten Variablen existentiell quantifiziert sind.

Regel für das Löschen:
Das Löschen eines Tupels aus einer Relation R der Datenbank verletzt eine Integritätsbedingung nicht, wenn in dieser Integritätsbedingung alle der Relation R zugeordneten Variablen universell quantifiziert sind.

	Einfügen in R	Löschen aus R
NONE	gültig	gültig
SOME	gültig	
ALL		gültig
BOTH		

Tabelle 5.1: Die Gültigkeit von Integritätsbedingungen

Tabelle 5.1 faßt die Regeln zusammen. Dabei geben die Zeilen an, wie die einer Relation R zugeordneten Variablen quantifiziert sind (keine Quantifizierung, nur existentiell, nur universell oder beide Arten). Die Spalten unterscheiden zwischen den Operationen, die auf der Relation ausgeführt werden. Ein Eintrag *gültig* bedeutet, daß die Integritätsbedingung durch die Operation nicht verletzt werden kann und folglich auch nicht gestestet werden muß.

Thomas Krebs [Kre88] stellt in seiner Diplomarbeit Regeln für die Optimierung von Integritätstests auf. Es handelt sich dabei um Transformationsregeln für Integritätsbedingungen, die in Pränex-Normalform vorliegen. Diese Regeln können verwendet werden, um die Tests, die bei den einzelnen Operationen durchgeführt werden müssen, zu vereinfachen.

Vor der Angabe der Regeln ist die Einführung einiger Begriffe nötig, die in den Regeln verwendet werden. Es wird sich dabei auf eine intuitive Erklärung der Begriffe beschränkt, eine formale Definition findet sich in der Diplomarbeit von Krebs. Der *effektive Bereich* einer Relation R enthält alle Tupel aus R, die zur Erfüllung der Integritätsbedingung beitragen. Er wird mit *eff*R bezeichnet und dient zur Reduktion existentiell quantifizierter Bereichsrelationen. Der *relevante Bereich* einer universell quantifizierten Bereichsrelation R ist die Teilmenge von R, die durch ein Löschen aus einer existentiell quantifizierten Bereichsrela-

tion "betroffen" ist. Er wird mit *relR* bezeichnet und dient zur Reduktion von universell quantifizierten Bereichsrelationen. Unter der *Differenzrelation* versteht man die Menge der eingefügten Elemente.

Die ersten beiden Transformationsregeln können zur Umformung von Bedingungen beim Einfügen von Elementen und die übrigen zur Umformung bei Löschoperationen verwendet werden:

1. **Transformationsregel:**
   Eine *SOME*-quantifizierte Bereichsrelation R kann auf den effektiven Bereich reduziert werden, wenn der Bereichsterm den Geltungsbereich des aktualisierten *ALL*-quantifizierten Bereichsterms umfaßt und nicht im Geltungsbereich der Quantorem *SOME ALL* steht.

2. **Transformationsregel:**
   Eine aktualisierte *ALL*-quantifizierte Bereichsrelation R einer Integritätsregel kann auf die Differenzrelation eingeschränkt werden, es sei denn

   - R liegt im Geltungsbereich einer *SOME*-quantifizierten Relation, die nicht auf den effektiven Bereich reduziert ist.
   - R liegt im Geltungsbereich der Quantorenfolge *SOME SOME* oder *ALL SOME*.

3. **Transformationsregel:**
   Eine *SOME*-quantifizierte Bereichsrelation, die nicht im Geltungsbereich der Quantoren *SOME ALL* steht, kann auf den effektiven Bereich reduziert werden.

4. **Transformationsregel:**

   (a) Eine *ALL*-quantifizierte Bereichsrelation R, die den durch die Löschoperation aktualisierten Bereichsterm überdeckt, kann auf den relevanten Bereich reduziert werden, sofern

   - R nicht im Geltungsbereich der Quantorenfolge *SOME SOME* oder *ALL SOME* steht.
   - R nicht im Geltungsbereich eines *SOME*-quantifizierten Bereichsterms steht, dessen Bereichsrelation nicht auf den effektiven Bereich reduziert ist.
   - R nicht die Quantorenfolge *SOME ALL SOME* überdeckt.

   (b) Wenn die Löschoperation einen *SOME*-quantifizierten Bereichsterm aktualisiert, der nicht im Geltungsbereich eines *ALL*-Quantors steht, genügt es zu testen, ob der effektive Bereich durch die Löschoperation leer wird.

Es gibt noch eine Reihe weiterer Methoden zur Umformung und Vereinfachung von Integritätsbedingungen (siehe dazu z.B. [Nic82, Bla81, HS78, Kob84]). Es wird sich hier aber auf die oben vorgestellten Regeln beschränkt, da sie einfach und allgemein sind und sich gut auf die hier behandelten Bedingungen anwenden lassen. Im folgenden werden diese Transformationsregeln zur Umformung spezieller Klassen von Integritätsbedingungen benutzt.

Monoclass-Bedingungen

Monoclass-Bedingungen sind Integritätsbedingungen, die sich nur auf die Elemente einer Klasse beziehen. Als Vertreter dieser Gruppe werden hier Bedingungen mit nur einem Quantor (*ALL* oder *SOME*), Beschränkungen von Domänen und Eindeutigkeitsbedingungen betrachtet. Funktionen, die eine Angabe dieser Bedingungen für Klassen erlauben, werden von der Schnittstelle *IcClass* zur Verfügung gestellt.

***ALL*-Bedingungen:** Diese Bedingungen sind von der Form:

$$ALL\ \mathrm{a}\ IN\ A : p(\mathrm{a})$$

Da die (einzige) Variable a universell quantifiziert ist, kann die Bedingung nur durch das Einfügen von Elementen in die Klasse A verletzt werden. Vor dem Einfügen eines Elements e in die Klasse A ist folgende Bedingung zu testen:

$$ALL\ \mathrm{a}\ IN\ A \cup \{\mathrm{e}\} : p(\mathrm{a}) \xrightarrow{(2)} ALL\ \mathrm{a}\ IN\ \{\mathrm{e}\} : p(\mathrm{a}) \longrightarrow p(e)$$

Diese Bedingung kann mit Hilfe der zweiten Transformationsregel umgeformt werden. Dabei wird die Klasse $A \cup \{\mathrm{e}\}$ auf die Differenzrelation, d.h. auf $\{e\}$ reduziert. Es muß also die Bedingung **fun***(e :E) p(e)* in die Kollektion der Integritätsbedingungen aufgenommen werden, die vor dem Einfügen von Elementen in die Klasse A getestet werden. Ein Beispiel für eine solche Bedingung ist die Einschränkung von Domänen, die weiter unten in diesem Abschnitt behandelt wird. Zur Angabe von *ALL*-Bedingungen existiert die Funktion *addAll*. Sie erhält eine Klasse und ein Prädiket über deren Elementtyp als Parameter.

***SOME*-Bedingungen:** Bei diesen Bedingungen existiert nur ein existentieller Quantor:

$$SOME\ \mathrm{a}\ IN\ A : p(\mathrm{a})$$

Sie können also nur durch das Löschen aus der Klasse A verletzt werden. Vor dem Löschen eines Elements e ist der Test

$$SOME\ \mathrm{a}\ IN\ A \setminus \{\mathrm{e}\} : p(\mathrm{a}) \longrightarrow SOME\ \mathrm{a}\ IN\ A : p(\mathrm{a}) \wedge not(\mathrm{a}\ is\ e)$$

durchzuführen. Ein einfaches Beispiel für eine solche Bedingung ist die Forderung, daß eine Klasse A nicht leer werden darf:

$$SOME\ \mathrm{a}\ IN\ A : TRUE$$

SOME-Bedingungen können durch einen Aufruf der Funktion *addSome* angegeben werden. Sie hat die gleiche Signatur wie die Funktion *addAll*.

Einschränkungen von Domänen: Solche Einschränkungen beziehen sich in der Regel auf einzelne Komponenten des Elementtyps einer Klasse und schränken die zulässigen Werte für diese Komponente ein. Man braucht zur Angabe einer solchen Bedingung zwei Funktionen: Eine Funktion zur Selektion der gewünschten Komponente *comp(:E) :F* und die Bedingung, die für diese Komponente erfüllt sein soll, also ein Prädikat *p* des Typs **All***(:F)* :**Bool**.

Zur Angabe der Bedingung "Der Preis eines Basisteils ist positiv." können die beiden Funktionen z.B. wie folgt definiert werden:

let *price(bp :BasePart)* :**Int** = *bp.cost*
let *positive(p* :**Int***)* :**Bool** = *p* > *0*

Die Einschränkung einer Domäne ist von der Form:

ALL a *IN A : p(comp(*a*))*

Es handelt sich dabei um eine *ALL*-Bedingung. Die Bedingung kann also nur durch das Einfügen von Elementen in die Klasse *A* verletzt werden, und vor dem Einfügen eines Elements *e* ist der Test

p(comp(e))

notwendig. Die Funktion *addDomainRestriction* dient zur Angabe von Domäneneinschränkungen. Sie hat folgende Signatur:

*addDomainRestriction(E, K, C ::***TYPE** *class :ClassType_Checked(E K)*
constrName :**String** *comp(:E) :C p(:C)* :**Bool***)* :**Ok**

Die oben angegebene Bedingung kann z.B. durch folgenden Aufruf sichergestellt werden:

addDomainRestriction(baseParts "positivePrice" price positive)

Eindeutigkeitsbedingungen: Wie bei Schlüsselbedingungen braucht man für die Definition von Eindeutigkeitsbedingungen eine Funktion zur Selektion der Komponente bzw. Komponenten, die eindeutig sein sollen, und eine Funktion, die die Gleichheit auf dieser Komponente bzw. diesen Komponenten definiert. Diese Funktionen können zu einer Funktion *compEqual* zusammengefaßt werden, die die Gleichheit zweier Elemente als Gleichheit der gewählten Komponenten definiert.

Für das Stücklistenbeispiel könnte die Funktion zur Realisierung der Bedingung "Ein Lieferant ist durch seinen Namen und seine Adresse eindeutig bestimmt." z.B. wie folgt aussehen:

let *nameAdressEqual(s1, s2 :Supplier)* :**Bool** =
string.equal(s1.name s2.name) ∧ *string.equal(s1.address s2.address)*

Mit einer Funktion *compEqual* ist eine Eindeutigkeitsbedingung von der Form:

ALL a1 *IN A : (ALL* a2 *IN A : not(compEqual(*a1 a2*))* ∨ *(*a1 *is* a2*))*

Gemäß der obigen Regeln kann diese Bedingung nur durch das Einfügen von Elementen in die Klasse A verletzt werden. Vor dem Einfügen eines Elements e ist folgende Bedingung zu testen:

ALL a1 IN A ∪ {e} : (ALL a2 IN A : not(compEqual(a1 a2)) ∨ (a1 is a2)) $\xrightarrow{(2)}$
ALL a1 IN {e} : (ALL a2 IN A : not(compEqual(a1 a2)) ∨ (a1 is a2)) $\longrightarrow$
ALL a2 IN A : not(compEqual(e a2)) ∨ (e is a2) $\longrightarrow$
ALL a2 IN A : not(compEqual(e a2))

Diese Bedingung kann durch Anwendung der zweiten Transformationsregel vereinfacht werden. Dabei wird $A \cup \{e\}$ auf $\{e\}$ reduziert. Der Teil "∨ *(e is a2)*" in der dritten Zeile der Unformung drückt die Alternative aus, daß e bereits in A enthalten ist. Da man keine Duplikate einfügen möchte, wird diese Möglichkeit ausgeschlossen. Es ist also folgende Bedingung in die Kollektion der Integritätsbedingungen der Klasse A einzufügen:

fun*(e :E) not(iter.all(class.elements()* **fun***(e1 :E) not(compEqual(e e1)))*

Zur Angabe von Eindeutigkeitsbedingungen bietet die Schnittstelle *IcClass* die Funktion *addUniqueness* an. Sie hat folgende Signatur:

*addUniqueness(E, K ::***TYPE** *class :ClassType_Checked(E K)*
*constrName :***String** *compEqual(:E :E) :***Bool***) :***Ok**

Die oben erwähnte Bedingung im Stücklistenbeispiel kann durch folgenden Aufruf der Funktion *addUniqueness* sichergestellt werden:

addUniqueness(suppliers "uniqueNameAddress" nameAdressEqual)

DuoClass-Bedingungen

Im folgenden werden Integritätsbedingungen betrachtet, die sich auf zwei Klassen beziehen, oder genauer gesagt, die zwei Quantoren haben und sich somit auf höchstens zwei Klassen beziehen. Dabei wird auf die vier möglichen Kombinationen der beiden Quantoren eingegangen. Für jede Kombination wird hergeleitet, welche Tests für eine Integritätsbedingung dieser Form notwendig sind und wie sich diese Tests vereinfachen lassen. Bei den Untersuchungen wird von einer Klasse A mit Elementtyp E und einer Klasse B mit Elementtyp F ausgegangen. Die Schnittstelle *IcClass* stellt Funktionen *addAllSome*, *addSomeAll*, *addAllAll* und *addSomeSome* zur Verfügung, mit denen man Integritätsbedingungen dieser Form angeben kann. Sie erwarten alle zwei Klassen und ein Prädikat über den Elementtypen der beiden Klassen als Parameter. Referentielle Integritätsbedingungen sind eine spezielle Klasse der *DuoClass*-Bedingungen. Sie werden im Abschnitt 5.3.2 behandelt. Weiterhin bietet die Schnittstelle auch entsprechende Funktionen *deleteAllSome* etc. an, um die eingefügten Bedingungen wieder zu löschen.

Es gibt auch Integritätsbedingungen mit mehr als zwei Quantoren. Diese werden jedoch hier aus Platzgründen nicht behandelt. Die am Anfang des Abschnitts eingeführten Regeln gelten jedoch auch für solche Bedingungen und können somit bei ihrer Implementierung verwendet werden.

ALL/SOME-Bedingungen: Ein Beispiel für Integritätsbedingungen der Form

ALL a *IN A : (SOME b IN B : p(*a, *b))*

sind die referentiellen Integritätsbedingungen. *ALL/SOME*-Bedingungen können nach obigen Regeln nur beim Einfügen in die Klasse A und beim Löschen aus der Klasse B verletzt werden. Vor dem Einfügen eines Elements e in die Klasse A ist der Test

ALL a *IN A* $\cup$ {e} *: (SOME b IN B : p(*a, *b))* $\xrightarrow{(2)}$
ALL a *IN* {e} *: (SOME b IN B : p(*a, *b))* $\longrightarrow$
*SOME b IN B : p(*e, *b)*

notwendig. Auf diese Bedingung kann wieder die zweite Transformationsregel angewandt und somit $A \cup \{e\}$ auf $\{e\}$ reduziert werden. Vor dem Löschen eines Elements aus der Klasse B muß folgende Bedingung getestet werden:

ALL a *IN A : (SOME b IN B* $\setminus$ {e} *: p(*a, *b))* $\longrightarrow$
ALL a *IN A : (SOME b IN B : p(*a, *b)* $\wedge$ *not(b is e))* $\xrightarrow{(4a)}$
ALL a *IN relA : (SOME b IN B : p(*a, *b)* $\wedge$ *not(b is e))* $\xrightarrow{(3)}$
ALL a *IN relA : (SOME b IN effB : p(*a, *b)* $\wedge$ *not(b is e))*

In diesem Fall können die dritte Transformationsregel und Teil a) der vierten angewandt werden, um die Bedingung zu vereinfachen. Für die Umformungen ist eine Berechnung des relevanten Bereichs der Klasse A und des effektiven Bereichs der Klasse B notwendig. Der relevante Bereich der Klasse A sind alle Elemente in A, für die *p(a, e)* gilt. Er kann also wie folgt berechnet werden:

relA = iter.select(classA.elements() **fun***(*a *:E) p(*a e*))*

Der effektive Bereich der Klasse B enthält nach Definition alle Elemente aus B, die zur Erfüllung der Integritätsbedingung beitragen. Da die dritte nach der vierten Transformationsregel angewandt wird, muß dabei nicht mehr die ganze Klasse A, sondern nur noch die Elemente in *relA* berücksichtigt werden.

effB = iter.select(classB.elements()
fun*(b :F) iter.some(relA* **fun***(*a *:E) p(*a b*)))*

Effektiver und relevanter Bereich hängen von der aktuellen Ausprägung der zugehörigen Klassen ab und müssen deshalb zum Zeitpunkt des Tests berechnet werden. Es ist also zu überlegen, ob die Umformungen wirklich eine Reduktion des benötigten Aufwands darstellen. Dies hängt hier in starkem Maße von der Größe des Bereichs *relA* ab, da über diesen Bereich sowohl bei der Berechnung des effektiven Bereichs von B als auch im resultierenden Test quantifiziert wird. Wenn man davon ausgeht, daß *relA* erheblich kleiner ist als A (höchstens halb so groß) lohnt sich der Aufwand für die Berechnung von *relA* und *effB*. Der umgeformte Test eignet sich auf jeden Fall gut als Ausgangspunkt für weitere Umformungen. So genügt es z.B. bei der referentiellen Integrität zu testen, ob der relevante Bereich leer ist (siehe Abschnitt 5.3.2).

SOME/ALL-Bedingungen: Integritätsbedingungen der Form

SOME a IN A : (ALL b IN B : p(a, b))

können nach obigen Regeln nur durch das Einfügen in die Klasse B und durch das Löschen aus der Klasse A verletzt werden. Bedingungen dieser Klasse sind z.B. solche, die für alle Elemente einer Klasse B eine gemeinsame Komponente aus einer Klasse A fordern.

Ein Beispiel dafür ist die Bedingung: "Einer der Manager ist der Chef von allen Angestellten":

SOME m IN Managers : (ALL e IN Employees : boss(e) is m)

Vor dem Einfügen eines Elements e in die Klasse B ist ein Test der Bedingung

SOME a IN A : (ALL b IN B ∪ {e} : p(a, b)) $\longrightarrow$
SOME a IN A : (ALL b IN B : p(a, b)) $\wedge$ *p(a, e)*

notwendig. Um $B \cup \{e\}$ gemäß der zweiten Transformationsregel auf die Differenzrelation reduzieren zu können, muß zunächst die Klasse A auf ihren effektiven Bereich reduziert werden. Dazu muß der effektive Bereich *effA* der Klasse A berechnet werden:

let *effA = iter.select(classA.elements()*
fun(a :E) *iter.all(classB.elements()* **fun**(b :F) *p(a b))* $\wedge$ *p(a e))*

Für diese Berechnung wird für jedes Element aus A einmal über B iteriert. Die Kosten liegen also bei $\mid A \mid * \mid B \mid$. Dies ist aber bereits die oberer Schranke für die Kosten des Tests in der ursprünglichen Form[9]. Es wird deshalb auf diese Umformung des Tests verzichtet. Man kann den Test jedoch effizienter machen, indem man nur für die Elemente aus A über B iteriert, für die *p(a e)* erfüllt ist. Vor dem Löschen eines Elements e aus der Klasse A ist zu testen:

SOME a IN A \ {e} : (ALL b IN B : p(a, b))
SOME a IN A : (ALL b IN B : p(a, b)) $\wedge$ *not(a is e)*

Laut Teil b) der vierten Transformationsregel genügt es hier zu testen, ob der effektive Bereich der Klasse A durch die Löschoperation leer wird. Dazu ist aber wiederum eine Berechnung des effektiven Bereichs notwendig, woraus sich die gleichen Effizienzprobleme wie oben ergeben. Deshalb wird auch auf diese Unformung verzichtet.

ALL/ALL-Bedingungen: Diese Bedingungen sind von der Form

ALL a IN A : (ALL b IN B : p(a, b))

[9]Die äußere Quantifizierung ist existentiell, d.h. man kann abbrechen, wenn die Bedingung für das erste Element erfüllt ist.

Eindeutigkeits- und Schlüsselbedingungen sind Vertreter dieser Gruppe. Bei ihnen handelt es sich allerdings um *Monoclass*-Bedingungen, d.h., es gilt $A = B$. *ALL/ALL*-Bedingungen können durch das Einfügen von Elementen in Klasse A und B verletzt werden. Vor dem Einfügen eines Elements e in die Klasse A muß folgender Test durchgeführt werden:

$$\begin{array}{l} ALL\ a\ IN\ A \cup \{e\} : (ALL\ b\ IN\ B : p(a, b)) \stackrel{(2)}{\longrightarrow} \\ ALL\ a\ IN\ \{e\} : (ALL\ b\ IN\ B : p(a, b)) \longrightarrow \\ ALL\ b\ IN\ B : p(e, b) \end{array}$$

Dieser kann durch Anwendung der zweiten Transformationsregel umgeformt werden. Analog ergibt sich folgender Test für das Einfügen eines Elements e in die Klasse B:

$$ALL\ a\ IN\ A : p(a, e)$$

SOME/SOME-Bedingungen: Integritätsbedingungen der Form

$$SOME\ a\ IN\ A : (SOME\ b\ IN\ B : p(a, b))$$

können, da sie nur existentiell quantifizierte Variablen enthalten, nur durch das Löschen von Elementen verletzt werden. Vor dem Löschen eines Elements e aus der Klasse A ist der Test

$$\begin{array}{l} SOME\ a\ IN\ A \setminus \{e\} : (SOME\ b\ IN\ B : p(a, b)) \longrightarrow \\ SOME\ a\ IN\ A : (SOME\ b\ IN\ B : p(a, b)) \wedge not(a\ is\ e) \end{array}$$

notwendig. Laut Teil b) der vierten Transformationsregel genügt es in diesem Fall zu testen, ob der effektive Bereich $effA$ der Klasse A durch die Löschoperation leer wird. Dazu ist wiederum eine Berechnung des effektiven Bereichs notwendig und aus den gleichen Gründen wie bei den *SOME/ALL*-Bedingungen wird auf diese Umformung verzichtet. Vor dem Löschen eines Elements e aus der Klasse B muß analog die Bedingung

$$SOME\ a\ IN\ A : (SOME\ b\ IN\ B : p(a, b) \wedge not(b\ is\ e))$$

getestet werden.

Bei den *ALL*-Bedingungen und den Eindeutigkeitsbedingungen wird bereits eine Umformung der hergeleiteten Tests in eine *P-Quest* Funktion des Typs **All***(:E)* :**Bool** angegeben. Eine entsprechende Umformung ist natürlich auch für die übrigen hergeleiteten Tests notwendig. Dabei werden die Funktionen *all* und *some* der Schnittstelle *Iter* (siehe Abschnitt 4.2.2) zur Darstellung der universellen bzw. existentiellen Quantifizierung benutzt. Die resultierenden Funktionen werden in die entsprechenden Kollektionen von Integritätsbedingungen eingefügt[10].

[10] Das sind die Kollektionen, die zu den Operationen gehören, die die Integritätsbedingung verletzen können.

5.3 Beziehungen zwischen Klassen

Es gibt verschiedene Arten von Beziehungen zwischen Klassen. Man kann sie danach unterscheiden, ob es sich um Beziehungen zwischen ganzen Klassen oder zwischen einzelnen Elementen der Klassen handelt. In die erste Kategorie fallen die Inklusionsbeziehungen (Subklassenbeziehungen) und die Exklusionsbeziehungen, die Thema des ersten Teils dieses Abschnitts sind. Vertreter der zweiten Kategorie werden im zweiten Teil dieses Abschnitts und im folgenden Abschnitt behandelt.

5.3.1 Inklusions- und Exklusionsbeziehungen

Beim *ORM* wird bereits bei der Erzeugung einer Klasse festgelegt, welche direkten Superklassen sie hat und zu welchen Klassen sie disjunkt ist, d.h. in Exklusionsbeziehung steht (vgl. Abschnitt 2.4.1). Hier wird ein etwas anderer Ansatz gewählt. Bei der Erzeugung einer Klasse werden keine Angaben über Inklusions- oder Exklusionsbeziehungen zu anderen Klassen gemacht. Es gibt vielmehr Funktionen, durch deren Aufruf man Subklassenbeziehungen bzw. die Disjunktheit von zwei Klassen festlegen kann. Außerdem gibt es auch Funktionen, um diese Festlegungen rückgängig zu machen. Dieser Ansatz ist dynamischer und trennt außerdem die Erzeugung von Klassen und den Aufbau von Generalisierungshierarchien voneinander. Im folgenden wird untersucht, inwieweit man Inklusions- und Exklusionsbeziehungen zwischen Klassen durch Integritätsbedingungen realisieren kann und welche zusätzlichen Maßnahmen notwendig sind. Weiterhin wird überlegt, welche Konsequenzen Inklusionsbeziehungen zwischen Klassen auf deren Integritätsbedingungen haben. Bei den Untersuchungen wird sich auf Klassen mit Integritätsüberwachung beschränkt. Subklassenbeziehungen können aber auch für Klassen ohne Integritätsüberwachung definiert werden.

Inklusionsbeziehungen

Eine Subklassenbeziehung zwischen zwei Klassen A mit Elementtyp E und B mit Elementtyp F (B ist Subklasse von A) besagt, daß zwischen den Extensionen der Klassen eine Inklusionsbeziehung besteht, d.h., A enthält alle Elemente von B. In der Regel wird dabei eine Subtypbeziehung zwischen den Elementtypen der Klassen gefordert ($F <: E$).

In objekt-orientierten Systemen wie z.B. O_2 [Deu90] und *Galileo* [AGOO88] wird bei der Definition von Subklassenbeziehungen der Mechanismus der Vererbung verwendet (siehe Abschnitt 2.4.2). Dies ist jedoch nur in Systemen möglich, die das Konzept der Vererbung unterstützen, was in *P-Quest* nicht der Fall ist. Man muß also bei der Definition einer Subklasse deren gesamten Elementtyp angeben. Das System kann jedoch überprüfen, ob die Elementtypen in einer Subtypbeziehung stehen.

Um den Ansatz möglichst dynamisch zu halten, können Subklassenbeziehungen zu jedem Zeitpunkt für zwei bereits existierende Klassen angegeben werden. Dazu stellt die Schnittstelle *SubClass* folgende Funktion zur Verfügung:

*addSubclass(E1, K ::***TYPE** *E2 ::***POWER***(E1)*
*super: ClassType_Checked(E1 K) sub:ClassType_Checked(E2 K)) :***Ok**

Diese Funktion kann z.B. verwendet werden, um die Subklassenbeziehung zwischen der Klasse *parts* und der Klasse *baseParts* anzugeben:

addSubclass(parts baseParts)

Eine Möglichkeit, Inklusionsbeziehungen sicherzustellen, ist der für die Sprache *ORM* vorgeschlagene Ansatz: Wenn ein Element in eine Klasse eingefügt wird, so wird es auch in all seine Superklassen eingefügt, und wenn ein Element aus einer Klasse gelöscht wird, so wird es auch aus all seinen Subklassen, in denen es enthalten ist, gelöscht. Auf diese Weise werden zu einem Element in Klasse A Verweise auf dieses Element in allen Superklassen von A verwaltet. Diese Methode paßt aus zwei Gründen nicht zu dem hier gewählten Ansatz. Zum einen wurde entschieden, auf aktive Komponenten, also hier das automatische Einfügen von Elementen in die Superklassen, zu verzichten (siehe Abschnitt 4.5.4). Zum anderen führt der Ansatz zu komplizierteren kompensierenden Operationen: Das Einfügen eines Elements in eine Klasse kann nicht mehr durch ein einfaches Löschen des selben Elements aus der Klasse kompensiert werden. Das Element muß vielmehr auch aus der obersten Superklasse und allen dazwischenliegenden Klassen gelöscht werden, um die Operation zu kompensieren.

Der Versuch, die Inklusionsbeziehung alleine durch Integritätsbedingungen sicherzustellen, führt zu folgendem Problem: Die Bedingung, daß ein Element einer Klasse auch in all seinen Superklassen enthalten ist, kann als Integritätsbedingung formuliert werden. Die Wahrung dieser Bedingung erfordert jedoch ein explizites Einfügen des Elements in alle Superklassen durch den Benutzer.

Es wird deshalb ein anderer Weg gewählt: Die Elemente der Subklassen werden nicht in der Superklasse verwaltet, sondern die Extension einer Klasse wird bei Bedarf als Vereinigung der Extensionen ihrer direkten Subklassen berechnet[11]. Für die Implementierung ist zu überlegen, welche Operationen von dieser Tatsache betroffen sind. Das sind die Funktionen, die einen lesenden Zugriff auf die Elemente der Klasse haben, also insbesondere die Funktionen *elements* und *lookup*. Die Iteration über die Elemente der Klasse A berechnet sich als Vereinigung aus der Iteration über die Elemente in der Kollektion der Klasse A und den Iterationen über ihre direkten Subklassen, die wiederum auf die gleiche Weise berechnet werden. Es ist möglich, daß ein Element in mehreren Klassen enthalten ist. Deshalb ist bei der Berechnung der Iteration eine Duplikateliminierung notwendig. Beim Schlüsselzugriff für die Klasse A genügt es nicht mehr, nur in der Kollektion der Klasse A nach dem Element mit dem angegebenen Schlüssel zu suchen. Wenn das Element dort nicht enthalten ist, müssen auch die Subklassen durchsucht werden. Dazu kann die Funktion *lookup* der jeweiligen Subklasse verwendet werden. Dies setzt jedoch voraus, daß die Subklassen den gleichen Schlüssel haben[12]. Es ergibt sich also die Einschränkung, daß eine Klasse den selben Schlüssel wie all ihre Subklassen und Superklassen besitzt. Die Einschränkung läßt sich durch eine kompliziertere Definition der Funktion zum Durchsuchen der Subklassen umgehen. Darauf wird im Prototyp jedoch verzichtet.

Die Operationen *elements* und *lookup* müssen also so verändert werden, daß sie eventuell existierende Subklassen mitberücksichtigen. Dies setzt voraus, daß zu einer Klasse die Menge ihrer direkten Subklassen oder doch zumindest deren *lookup*- und *elements*-Funktionen verwaltet werden. Da Subklassenbeziehungen dynamisch festgelegt werden und nicht schon zum

[11] Die Berechnung setzt sich gegebenenfalls rekursiv fort.

[12] Eine Subtypbeziehung zwischen den Schlüsseltypen genügt nicht.

Zeitpunkt der Erzeugung einer Klasse feststehen, kann diese Menge nicht Teil des internen Zustands der Klasse sein. Sie sollte deshalb als zusätzliche Information in die Komponente *info* aufgenommen werden:

Let *ICInfoType(E, K ::***TYPE***) ::***TYPE** =
  Tuple
    ...
    *subclasses :list.T(***Tuple** *E1 <: E subclass :Checked(E1 K)* **end***)*
    ...
  end

Da *InfoType* als Typ der Komponente *Info* des Typs einer Klasse auftritt, muß der Typoperator *ClassType_Checked* für den Typ der Klassen rekursiv definiert werden. Es genügt dabei nicht, *Checked* als Typoperator zu definieren, der auf einen rekursiven Typ abbildet[13], da *Checked* bei der Komponente *subclasses* in unterschiedlich parametrisierter Form (Elementtyp der Subklassen) auftritt. Rekursive Typoperatoren sind jedoch in *P-Quest* nicht zugelassen (vgl. Abschnitt 3.3.3). Es wird sich deshalb darauf beschränkt, die Funktionen *lookup* und *elements* der Subklassen in die Komponente *info* aufzunehmen.

Um die Implementierung des Prototyps einfach zu halten, wird die Einschränkung gemacht, daß ein Element nur aus der Klasse gelöscht werden kann, in die es auch eingefügt worden ist. Dies stellt aber besonders für das prädikative Löschen eine Beschränkung dar. Alternativ dazu kann beim Löschen wie folgt vorgegangen werden: Es wird versucht, das Element bzw. die Elemente (beim prädikativen Löschen) aus der Klasse zu löschen, für die die Operation aufgerufen worden ist. Außerdem wird für alle Subklassen, die das Element (die Elemente) enthalten, eine Löschoperation ausgeführt. Beim prädikativen Löschen muß diese Prozedur für alle zu löschenden Elemente durchgeführt werden. Dieser Ansatz erfordert, daß zu einer Klasse zusätzlich auch die Löschfunktionen der direkten Subklassen abgespeichert werden. Für das Einfügen eines Elements ergeben sich keine Konsequenzen. Es wird nur in die Klasse eingefügt, für die die Operation aufgerufen worden ist.

Es soll nicht nur möglich sein, dynamisch neue Subklassenbeziehungen anzugeben, sondern auch solche Beziehungen zu löschen. Dies geschieht durch einen Aufruf der Funktion *deleteSubclass*. Sie hat die selbe Signatur wie *addSubClass*. Ein Aufruf der Funktion für zwei Klassen A und B bewirkt, daß B aus der Menge der direkten Subklassen von A gelöscht wird[14].

Die Vererbung von Integritätsbedingungen

Wenn man ein Element in eine Klasse einfügt, wird es implizit auch Element all seiner Superklassen. Es muß deshalb auch die Integritätsbedingungen all seiner Superklassen erfüllen. Die gewählte Implementierung fügt die Elemente jedoch nicht in die Superklassen ein und löscht sie auch nicht daraus, sodaß kein Test der Bedingungen der Superklassen stattfindet. Um einen solchen Test zu erzwingen, wird hier der Ansatz gemacht, daß eine Klasse die Integritätsbedingungen ihrer Superklassen erbt. Hierbei ist zu beachten, daß bei einem Aufruf

[13] Dies ist in *P-Quest* zulässig.

[14] Weitere Konsequenzen der Operation werden im Abschnitt über die Vererbung von Integritätsbedingungen behandelt.

von *addSubclass(A B)* die Integritätsbedingungen der Klasse A nicht nur an die Klasse B vererbt werden müssen, sondern auch an all ihre Subklassen. Auf diese Weise werden vor dem Einfügen und Löschen nicht nur die Integritätsbedingungen der Klasse, sondern auch all ihrer Superklassen getestet. Es ist dabei zu berücksichtigen, daß es zu Namenskonflikten kommen kann, da die Namen zwar innerhalb einer Kollektion von Integritätsbedingungen auf Eindeutigkeit gestestet werden, nicht jedoch global für verschiedene Klassen. Dieses Problem wird in der prototypischen Implementierung jedoch vernachläßigt.

Die Vererbung der Integritätsbedingungen an die Subklassen hat auch Konsequenzen für das Einfügen und Löschen von Integritätsbedingungen. Beim Einfügen muß die Integritätsbedingung nicht nur in die Kollektion der Klasse A, sondern auch in die Kollektionen aller Subklassen von A eingefügt werden. Für das Löschen von Integritätsbedingungen muß genauso verfahren werden. Dabei ergibt sich die Einschränkung, daß eine Integritätsbedingung vom Benutzer nur aus der Klasse gelöscht werden kann, in die sie auch eingefügt wurde, d.h., ererbte Integritätsbedingungen dürfen nicht gelöscht werden. Dazu muß es aber eine Möglichkeit geben, zwischen ererbten und eigenen Bedingungen zu unterscheiden. Es müssen neue Funktionen für das Einfügen und Löschen von Integritätsbedingungen geschrieben werden.

Probleme treten auf, wenn eine Subklassenbeziehung gelöscht wird. Alle aufgrund dieser Beziehung ererbten Integritätsbedingungen brauchen nicht mehr getestet zu werden und sollten deshalb aus den entsprechenden Kollektionen gelöscht werden. Es stellt sich die Frage, wie man genau diese Bedingungen wiederfindet. Man betrachtet dazu den Aufruf *deleteSubclass(A B)*. Da die Klasse B alle Integritätsbedingungen von A geerbt hat und aufgrund dieser Beziehung keine anderen, genügt es aus B alle Integritätsbedingungen zu löschen, die in A enthalten sind. Durch die Definition der Löschoperation (s.o.) werden sie damit auch aus allen Subklassen von B gelöscht.

Exklusionsbeziehungen

Beim *ORM* kann zu einer Klasse nicht nur angegeben werden, welche direkten Superklassen sie hat, sondern auch, mit welchen Klassen sie in Exklusionsbeziehung steht. Diese Angabe ist auch bei dem hier gewählten Ansatz möglich. Allerdings ist sie wiederum nicht auf den Zeitpunkt der Definition der Klasse festgelegt, sondern kann zu jedem Zeitpunkt durchgeführt werden.

Eine Exklusionsbeziehung zwischen zwei Klassen A und B besagt, daß diese beiden Klassen keine gemeinsamen Elemente besitzen. Dies läßt sich wie folgt als Integritätsbedingung formulieren:

ALL a IN A : (ALL b IN B : not(a is b))

Es handelt sich also um eine Bedingung der Form *ALL/ALL*. Gemäß der Regeln aus Abschnitt 5.2.4 kann diese Bedingung nur durch das Einfügen von Elementen in die Klassen A und B verletzt werden. Vor dem Einfügen eines Elements *e1* in die Klasse A bzw. *e2* in die Klasse B muß der Test

ALL b IN B : not(e1 is b) bzw. ALL a IN A : not(e2 is a)

durchgeführt werden. Intuitiv gesagt wird getestet, ob das einzufügende Element bereits in der anderen Klasse existiert. Die Exklusionsbeziehung kann durch das Einfügen der hergeleiteten Bedingungen in die Kollektionen der Integritätsbedingungen der Klassen A und B sichergestellt werden.

Die Festlegung einer Exklusionsbeziehung zwischen zwei Klassen durch den Benutzer geschieht über die Funktion *disjoint*. Zwei Klassen können nur dann gemeinsame Elemente haben, wenn ihre Elementtypen einen gemeinsamen Subtyp E besitzen. Daraus ergibt sich die Signatur der Funktion *disjoint* zu:

*disjoint(E, K1, K2 ::***TYPE** *E <: E1 E <: E2*
classA :ClassType_Checked(E1, K1)
*classB :ClassType_Checked(E2, K2)) :***Ok**

Im Stücklistenbeispiel sollen die Klasse der Basisteile und die Klasse der zusammengesetzten Teile in Exklusionsbeziehung stehen. Durch die Wahl der Elementtypen ist aber bereits sichergestellt, daß sie keine gemeinsamen Elemente enthalten: *BaseParts* und *CompositeParts* besitzen keinen gemeinsamen Subtyp.

Exklusionsbeziehungen zwischen mehr als zwei Klassen können durch mehrere Aufrufe der Funktion sichergestellt werden. Zum Löschen von Exklusionsbeziehungen existiert eine Funktion *deleteDisjoint*, die wiederum die beiden Klassen als Parameter nimmt. Sie entfernt die zur Wahrung der Exklusionsbeziehung in die Klassen eingefügten Integritätsbedingungen.

5.3.2 Objektreferenzen und referentielle Integrität

Bei einigen objekt-orientierten Ansätzen wie z.B. bei *Galileo* und O_2 ist es möglich, Klassen als Domänen von Attributen anzugeben. Dadurch lassen sich Beziehungen zwischen Objekten durch den Mechanismus der Aggregation darstellen (vgl. Abschnitt 2.3.2).

Da Klassen, wie sie oben definiert sind, nicht durch Typen, sondern durch Werte dargestellt werden, kann man sie nicht als Domänen von Attributen angeben. Man kann jedoch den Elementtyp E der Klasse als Typ des Attributs wählen. Auf diese Weise kann das Attribut aber beliebige Werte vom Typ E annehmen und nicht nur solche, die auch tatsächlich in der Klasse enthalten sind. Um dies zu gewährleisten, müssen geeignete Integritätsbedingungen in die beteiligten Klassen eingefügt werden. Es handelt sich hierbei um eine spezielle Art der referentiellen Integrität. Dabei kann man zwei Formen unterscheiden, je nachdem, ob das betrachtete Attribut einwertig oder mehrwertig ist. Diese beiden Formen und die relationale Form der referentiellen Integrität werden im folgenden anhand von Beispielen vorgestellt, und es wird untersucht, welche Tests durchzuführen sind, um diese Bedingungen sicherzustellen. Die Schnittstelle *IcClass* stellt Funktionen zur Angabe der verschiedenen Formen der referentiellen Integrität zur Verfügung und auch Funktionen, um diese Bedingungen wieder zu löschen.

Referentielle Integrität für einwertige Attribute

Diese Bedingung muß zum Beispiel berücksichtigt werden, wenn der Elementtyp der Klasse *baseParts* abweichend von der Definition im Anhang A wie folgt definiert wird:

```
Let BasePart =
   Tuple
     pno :Int
     ...
     supplier :Supplier
   end
```

Es soll sichergestellt werden, daß für alle Basisteile *bp* in der Klasse *baseParts* die Komponente *bp.supplier* in der Klasse *suppliers* enthalten ist.

Allgemein sieht dieses Problem wie folgt aus: Sei A eine Klasse mit Elementtyp E und c :F eine Komponente des Typs E. c soll als Werte nur Elemente der Klasse B annehmen können, die Elementtyp F hat. Mit einer Projektionsfunktion *project*[15], die vom Elementtyp E auf den Elementtyp F abbildet[16], läßt sich diese Integritätsbedingung wie folgt formulieren:

ALL a in A : (SOME b IN B : project(a) is b)

Es handelt sich also um eine spezielle *ALL/SOME*-Bedingung. Wie in Abschnitt 5.2.4 hergeleitet wird, kann die referentielle Integrität also nur beim Einfügen in die Klasse A und beim Löschen aus der Klasse B verletzt werden. Gemäß der Betrachtungen zu den *ALL/SOME*-Bedingungen ist vor dem Einfügen eines Elements e in die Klasse A die Bedingung

SOME b IN B : project(e) is b

zu testen und vor dem Löschen eines Elements e aus der Klasse B die Bedingung

ALL a IN relA : (SOME b IN effB : (project(a) is b) ∧ not(b is e))).

relA enthält nach Definition alle Elemente aus A, die vom Löschen des Elements e aus B betroffen sind, also die Bedingung *project(a) is e* erfüllen. Da *project(a)* eindeutig auf ein Element aus F abbildet, gibt es in *relA* kein Element a, das die Bedingung

SOME b IN effB : (project(a) is b) ∧ not(b is e)

erfüllt. Die oben genannte Bedingung kann also nur wahr werden, wenn der relevante Bereich leer ist. Es genügt also zu testen:

ALL a IN A : not(project(a) is e)

Die referentielle Integrität für einwertige Attribute kann für zwei Klassen durch einen Aufruf der Funktion *addRefIntegrity* sichergestellt werden. Diese Funktion fügt die obigen Tests in die entsprechenden Kollektionen von Integritätsbedingungen ein. Für das Beispiel sieht der Aufruf der Funktion wie folgt aus:

addRefIntegrity(baseParts suppliers **fun***(bp :BasePart) bp.supplier "SupplierReference")*

[15] *project* ist in der Regel eine Funktion zur Komponentenselektion.
[16] Im Beispiel ist *project* = **fun***(bp :BasePart) bp.supplier*.

Referentielle Integrität für mehrwertige Attribute

Um das Problem der referentiellen Integrität bei mehrwertigen Attributen am Stücklistenbeispiel zu veranschaulichen, muß die Definition der zusammengesetzten Teile ein wenig modifiziert werden:

> Sei der Elementtyp *CompositePart* der Klasse *compositeParts* wie folgt definiert:
>
> **Let** *CompositePart* =
> **Tuple**
> ...
> *subParts :set.T(Part)*
> **end**
>
> Bedingung: Für jedes zusammengesetzte Teil *cp* in der Klasse *compositeParts* sind alle Teile der Menge *cp.subParts* in der Klasse *parts* enthalten.

Bei einem mehrwertigen Attribute c einer Klasse A muß die referentielle Integrität für jeden der Werte sichergestellt werden. Man definiert dafür eine Funktion *project*, die ein Element der Klasse A vom Typ E auf einen Iterator über die Elemente des mehrwertigen Attributs abbildet. Die Verwendung eines Iterators hat den Vorteil, daß man von der Struktur (z.B. Menge, Liste), in der die Elemente gespeichert sind, abstrahieren kann. Mit dieser Definition von *project* ergibt sich die Bedingung:

ALL a IN A : (ALL p IN project(a) : (SOME b IN B : p is b))

Sie kann durch das Einfügen in die Klasse A und durch das Löschen aus der Klasse B verletzt werden[17]. Die zu testenden Bedingungen ergeben sich analog zu denen für den Fall von einwertigen Attributen. Vor dem Einfügen eines Elements e in die Klasse A ist zu testen, ob folgende Bedingung erfüllt ist:

ALL p IN project(e) : (SOME b IN B : p is b)

Vor dem Löschen eines Elements e aus der Klasse B ist wiederum zu testen, daß der zugehörige relevante Bereich leer ist:

ALL a IN A : not(SOME p IN project(a) : p is b)

Die Funktion *addRefIntegrityMulti* dient zur Angabe der referentiellen Integrität für mehrwertige Attribute. Sie fügt die hergeleiteten Tests in die entsprechenden Kollektionen von Integritätsbedingungen ein. Für das oben gewählte Beispiel ergibt sich der folgende Aufruf der Funktion:

let *projectToSubParts(cp :CompositeParts) : Iter_T(Part) =*
set.elements(cp.subParts)
addRefIntegrityMulti(compositeparts parts projectToSubParts "ComponentReference")

[17] Die Bedingung kann auch durch das Einfügen von Elementen in das mehrwertige Attribut verletzt werden (sofern dies erlaubt ist). Dies fällt aber in die Kategorie der Änderungsoperationen, die hier nicht betrachtet wird.

Referentielle Integrität für den relationalen Ansatz:

Wie in Abschnitt 5.2.2 erwähnt, kann einer Klasse auch eine SQL-Tabelle zugrundeliegen. In diesem Fall sind nur flache Tupel und somit keine direkten Verweise, sondern nur Referenzen (wie z.B. Schlüssel) auf Elemente anderer Klassen möglich. Aus diesem Grund besteht die Notwendigkeit, auch die Angabe von referentiellen Integritätsbedingungen im relationalen Sinne ([Dat81]) zu ermöglichen.

> Als Beispiel kann wiederum die Beziehung zwischen den Klassen der Basisteile und der Klasse der Lieferanten verwendet werden. Die oben eingeführte Definition des Typs *BaseParts* muß hierfür leicht modifiziert werden. Der Verweis auf den Lieferanten wird durch dessen Schlüssel ersetzt, der vom Typ *Int* ist:
>
> **Let** *BaseParts* =
> **Tuple**
> ...
> *supplier* :**Int**
> **end**
>
> Bedingung: Für jedes Basisteil der Klasse *baseParts* ist *bp.supplier* eine Lieferantennummer, zu der in der Klasse *suppliers* ein Lieferant existiert.

Um die referentielle Integrität zwischen zwei Klassen A und B anzugeben, werden hier zwei Funktionen *reference* und *referenced* verwendet. Die Funktion *reference* bildet ein Element der Klasse A auf die in ihm enthaltene Referenz r ab, und die Funktion *referenced* erhält eine Referenz r und ein Element b der Klasse B als Parameter und entscheidet, ob r auf b verweist. Für das Beispiel sehen diese beiden Funktionen wie folgt aus:

let *reference(bp :BasePart)* :**Int** = *bp.supplier*
let *referenced(s :Supplier sno* :**Int**) :**Bool** = *s.sno is sno*

Es wird vorausgesetzt, daß eine Referenz eindeutig ein Element in der referenzierten Klasse bestimmt[18]. Mit den beiden Funktionen kann die Bedingung wie folgt dargestellt werden:

ALL a IN A : (SOME b IN B : referenced(b reference(a)))

Diese Bedingung kann wiederum nur durch das Einfügen in die Klasse A und durch das Löschen aus der Klasse B verletzt werden. Vor dem Einfügen eines Elements e in die Klasse A ist zu testen:

SOME b IN B : referenced(b reference(e))

und vor dem Löschen eines Elements e aus der Klasse B ist zu überprüfen, ob der relevante Bereich leer ist:

ALL a IN A : not(referenced(e reference(a)))

[18] In der Regel wird der Schlüssel als Referenz verwendet.

Durch einen Aufruf der Funktion *addRefIntegrityRel* kann diese Art der referentiellen Integrität zwischen zwei Klassen sichergestellt werden. Mit den oben definierten Funktionen *reference* und *referenced* sieht der Aufruf der Funktion für das Beispiel wie folgt aus:

addRefIntegrityRel(baseParts suppliers reference referenced "SupplierReference")

5.4 Die explizite Darstellung von Beziehungen

Für die Darstellung von Beziehungen zwischen Elementen von Klassen gibt es verschiedene Möglichkeiten. Zum einen lassen sich solche Beziehungen wie im vorigen Abschnitt beschrieben durch Objektreferenzen realisieren. Man kann aber auch ein eigenes Konstrukt zur expliziten Darstellung von Beziehungen einführen. Folgendes Beispiel zeigt diese beiden Varianten bei der Darstellung für die Beziehung zwischen zusammengesetzten Teilen und ihren Komponenten im Stücklistenbeispiel:

Darstellung durch eine Objektreferenz:

```
Let CompositePart =
    Tuple
      pno :Int
      ...
      subParts :set.T(Part)
    end
```

Dartstellung durch ein eigenes Konstrukt:

```
Let CompositePart =
    Tuple
      pno :Int
      ...
    end

Let Quantity = Int

let madeFrom = setRelationship.create(compositeParts parts :Quantity)
```

Die explizite Darstellung von Beziehungen (*Relationships*) in einer objekt-orientierten Umgebung ist eine Besonderheit des *Object-Relationship*-Modells. Die Vorteile eines solchen Ansatzes werden bereits in Abschnitt 2.3 erläutert. Im folgenden wird der Begriff *Beziehung* für das spezielle Konstrukt zur Darstellung der Beziehungen verwendet.

Beim *ORM* sind Beziehungen zwischen beliebig vielen Klassen zugelassen und werden alle durch ein einheitliches Konstrukt behandelt. Wie sich in den weiteren Betrachtungen zeigen wird, hängt bei dem hier gewählten *Add-On*-Ansatz der Typ einer Beziehung unter anderem von den Elementtypen der beteiligten Klassen ab. Daraus ergibt sich zum einen, daß der

Typ einer Beziehung durch einen Typoperator dargestellt werden muß. Zum anderen erhält man für jede Anzahl von beteiligten Klassen einen anderen Typoperator, da sich die Anzahl der Typparameter ändert. Beziehungen zwischen verschieden vielen Klassen müssen also getrennt behandelt werden. Im Prototyp wird sich auf den Fall von Beziehungen zwischen zwei Klassen beschränkt. Für Beziehungen zwischen mehr als zwei Klassen müssen jeweils eigene Schnittstellen geschrieben werden. Diese unterscheiden sich jedoch nicht prinzipiell von denen für Beziehungen zwischen zwei Klassen und werden deshalb hier nicht näher erläutert.

Der Rest dieses Abschnitts gliedert sich wie folgt: Im ersten Teil wird ein Überblick über den Aufbau von Beziehungen gegeben. Eine Beziehung besteht wie eine Klasse im wesentlichen aus einer Kollektion von Elementen. Es wird darauf eingegangen, wie der Elementtyp einer solchen Kollektion aussehen kann. Dabei zeigt sich, daß auch hier eine Notwendigkeit zur Überprüfung referentieller Integrität besteht. Dieses Problem wird im zweiten Teil des Abschnitts betrachtet. Im dritten Teil wird die Benutzerschnittstelle von Beziehungen untersucht. Es wird überlegt, welche Operationen auf Beziehungen benötigt werden. Dabei wird die für Klassen gewählte Schnittstelle zugrunde gelegt. Nachfolgend werden spezielle Aspekte einer möglichen Implementierung von Beziehungen vorgestellt. Der Übergang zu Beziehungen mit geschützten Operationen und Operationen mit Integritätsüberwachung ist Inhalt des vorletzten Teils des Abschnitts. Im letzten Teil wird die Realisierung von einigen speziellen Integritätsbedingungen auf Beziehungen behandelt.

5.4.1 Beziehungen und ihre Elemente

Eine Beziehung ist ähnlich wie eine Klasse aufgebaut. Sie hat einen Zustand und bietet dem Benutzer eine Reihe von Operationen an, die auf diesem Zustand arbeiten. Außer diesen Operationen gibt es keinen Zugriff auf den internen Zustand der Beziehung. Die Kollektion der Elemente der Beziehung ist Teil des Zustands.

Um den Elementtyp dieser Kollektion festzulegen, muß man sich überlegen, welche Information in den Elementen der Beziehung gespeichert werden muß. Sei R eine Beziehung zwischen den Klassen B und C. Ein Element r der Beziehung R enthält einen Verweis auf ein Element der Klasse B und einen Verweis auf ein Element der Klasse C. Weiterhin sollen zu einer Beziehung häufig auch noch zusätzliche Attribute abgespeichert werden. Der Elementtyp hängt also von den Elementtypen der beiden Klassen und von der Anzahl und den Typen der Attribute ab. Die Abhängigkeit von den Typen kann durch die Verwendung eines Typoperators für den Elementtyp der Beziehung dargestellt werden. Problematisch hingegen sind die verschiedenen Anzahlen von Attributen. Es ergibt sich für jede Anzahl ein anderer Typoperator für den Elementtyp. Um dies zu umgehen, kann eine Vereinheitlichung von beliebigen Beziehungen zu solchen mit nur einem Attribut vom Typ A durchgeführt werden. Sei n die Anzahl der Attribute. Für

- $n = 1$: Wähle A = Typ des Attributs.
- $n = 0$: Wähle A = **Ok**.
- $n > 1$: Fasse die n Attribute zu einem Tupel zusammen. Wähle A = resultierender Tupeltyp.

Mit dieser Vereinheitlichung kann der Elementtyp für beliebige Beziehungen zwischen zwei Klassen durch folgenden Typoperator dargestellt werden:

Let *E(E1, E2, A ::***TYPE***) ::***TYPE** =
  Tuple
    first :E1
    second :E2
    attributes :A
  end

Die ersten beiden Typparameter *E1* und *E2* geben dabei die Elementtypen der beteiligten Klassen an und *A* den Typ der zusätzlichen Attribute (gemäß der Vereinheitlichung).

Für Beziehungen ohne Attribute ($n = 0$) ergibt sich aus dieser Darstellung der Nachteil, daß in jedem Element der Beziehung ein überflüssiges **ok** gespeichert wird und daß man bei einer Reihe von Operationen, z.B. beim Einfügen, zusätzlich ein **ok** (= Attributwert) als Parameter angeben muß. Es ist deshalb zu überlegen, ob man zwischen Beziehungen mit und ohne zusätzliche Attribute unterscheiden sollte. Auf diese Unterscheidung wurde im Prototyp jedoch verzichtet.

Referentielle Integrität und Eindeutigkeit

Für jedes in der Beziehung abgespeicherte Element, muß gewährleistet sein, daß die Werte seiner Komponenten *first* und *second* auch in den entsprechenden Klassen enthalten sind. Es ist also auch hier die referentielle Integrität zu überprüfen. Analog zu der oben behandelten referentiellen Integrität bei einwertigen Attributen ergeben sich für eine Beziehung R zwischen zwei Klassen B und C folgende Tests: Beim Einfügen eines Elements e in R ist zu testen, ob

SOME b IN B : e.first is b $\wedge$ *SOME c IN C : e.second is c*

gilt und beim Löschen eines Elements b aus B bzw. eines Elements c aus C ist zu überprüfen, ob gilt:

ALL r IN R : not(r.first is b) bzw. ALL r IN R : not(r.second is c)

Um die referentielle Integrität sicherzustellen, müssen also Integritätsbedingungen in die entsprechenden Kollektionen der an der Beziehung beteiligten Klassen eingefügt werden. Dies setzt voraus, daß diese Kollektionen bereits angelegt worden sind, d.h., daß es sich um Klassen mit Integritätsüberwachung handelt (siehe Abschnitt 5.2.3). Eine Beziehung kann also nur zwischen zwei Klassen mit Integritätsüberwachung erzeugt werden.

Da die referentielle Integrität in diesem Fall eine inhärente Bedingung ist, soll sie auch für Beziehungen ohne Integritätsüberwachung gewährleistet sein. Die Bedingung, die beim Einfügen in R zu testen ist, wird deshalb nicht in eine Kollektion von Integritätsbedingungen eingefügt, sondern direkt durch eine *If*-Abfrage vor dem Einfügen eines Elements getestet.

Weiterhin wird davon ausgegangen, daß ein Element der Beziehung eindeutig durch die Elemente bestimmt ist, die an der Beziehung beteiligt sind, d.h., für eine Beziehung zwischen den

Klassen *A* und *B* soll jedes Paar *(a, b)* nur einmal in der Beziehung vertreten sein. Diese Bedingung soll wie die referentielle Integrität auch für Beziehungen ohne Integritätsüberwachung sichergestellt werden und wird deshalb ebenfalls direkt getestet (siehe Abschnitt 5.4.2).

Operationen auf Beziehungen

Wie bei den Klassen soll es auch bei den Beziehungen eine benutzerfreundliche Schnittstelle geben, die die notwendigen Operationen zur Verfügung stellt. Um die Ähnlichkeit von Klassen und Beziehungen zu betonen und um eine einheitliche Programmierumgebung zu schaffen, baut die Schnittstelle der Beziehungen auf die der Klassen auf. Wie das folgende Beispiel zeigt, sind die Einfüge- und Löschoperationen bei Beziehungen denen bei Klassen sehr ähnlich.

> Im Stücklistenbeispiel tritt z.B. die Beziehung *madeFrom* zur Dartsellung der Beziehung zwischen zusammengesetzten Teilen und ihren Komponenten auf. Das Einfügen eines Elements in die Beziehung und das Löschen sehen wie folgt aus:
>
> *madeFrom.insert(chair leg 4)*
> *madeFrom.delete(chair leg)*

Ein zu löschendes Element wird durch die Angabe der beiden Elemente, auf die es sich bezieht, identifiziert.

Für die Iteration über alle Elemente der Beziehung existiert wie bei den Klassen eine Funktion *elements*. Alle Anfragen an die Beziehung können durch Verwendung dieser Funktion in Kombination mit den Funktionen der Schnittstelle *Iter* ausgedrückt werden (siehe Abschnitt 4.2.2). Auf diese Weise wird jedoch die unterschiedliche Semantik der Beziehungen gegenüber der der Klassen vernachläßigt. Deshalb werden noch weitere, speziellere Zugriffsoperationen zur Verfügung gestellt, die häufig auftretende Anfragen unterstützen. Unter diesem Aspekt wurde der Satz der Operationen über Beziehungen zusammengestellt:

```
Def T(E1, E2, A ::TYPE) ::TYPE =
  Tuple
    info: InfoType
    insert(e1 :E1 e2 :E2 a :A) :Ok
    delete(e1 :E1 e2 :E2) :Ok
    deleteAll(p(:E1 :E2 :A) :Bool) :Ok
    elements() :Iter_T(Tuple e1 :E1 e2 :E2 a :A end)
    pairs() :Iter_T(Pair_T(E1 E2))
    fst(e2 :E2) :Iter_T(E1)
    allFst() :Iter_T(E1)
    snd(e1 :E1) :Iter_T(E2)
    allSnd() :Iter_T(E2)
  end
```

RelationshipType_T(E1 E2 A) ist der Typ einer Beziehung zwischen zwei Klassen mit den Elementtypen *E1* und *E2* und dem Typ *A* für die Attribute. Durch den Aufruf der Funktion

pairs erhält man eine Iteration über die Paare von Elementen, die in Beziehung zueinander stehen. Die Funktion *fst* bzw. *snd* berechnet alle Elemente der ersten bzw. zweiten Klasse, die zu dem angegebenen Element der zweiten bzw. ersten Klasse in Beziehung stehen. Mit der Funktion *allFst* bzw. *AllSnd* kann man alle Elemente der ersten bzw. zweiten Klasse bestimmen, die an der Beziehung beteiligt sind. Folgende Beispiele sollen die Anwendung der beschriebenen Funktionen veranschaulichen:

Alle Paare von Teilen, die in der Beziehung *madeFrom* stehen:

madeFrom.pairs()

Alle Teile, aus denen ein Tisch hergestellt ist:

madeFrom.Snd(table)

Alle Teile, die von einem bestimmten Lieferanten geliefert werden:

suppliedBy.Fst(smith)

Alle Lieferanten, die an der Beziehung *suppliedBy* beteiligt sind:

suppliedBy.allSnd()

5.4.2 Implementierungen

Das Erzeugen einer Beziehung besteht im wesentlichen aus dem Anlegen einer Struktur für die Speicherung der Elemente der Beziehung und der Definition der im vorigen Abschnitt vorgestellten Operationen der Benutzerschnittstelle für die gewählte Struktur. Die Form des Elementtyps wird bereits im ersten Teil dieses Abschnitts vorgestellt.

Wie die Klassen kann man auch Beziehungen basierend auf verschiedenen Strukturen, in denen die Elemente der Beziehung gespeichert werden, implementieren. Die Schnittstelle für den Benutzer ist in allen Fällen gleich. Im Prototyp ist eine exemplarische Implementierung *setRelationship* basierend auf der Schnittstelle *Set* enthalten. Im folgenden wird kurz auf diese Implementierung eingegangen. Außerdem wird angedeutet, welche Aspekte bei einer Implementierung basierend auf *SQL-Tabellen* zu beachten sind.

setRelationship ist eine Implementierung der Schnittstelle *Relationship*. Die Funktion zum Erzeugen von Beziehungen hat folgende Signatur:

*create(E1, E2, K1, K2 ::***TYPE** *name :***String**
class1 :ClassType_Checked(E1 K1) class2 :ClassType_Checked(E2 K2)
*A ::***TYPE** *objId(:E1 :E2 :A) :***String***) :***Ok**

Die Beziehung *madeFrom* kann also durch folgenden Aufruf erzeugt werden:

let *madeFrom =*
setRelationship.create("MadeFrom" compositeParts parts :Quantity madeFromId)

wobei *madeFromId* eine Funktion ist, die Elemente der Beziehung auf eine (eindeutige) Zeichenkette abbildet. Sie kann z.B. wie folgt definiert sein:

let *madeFromId(cp :CompositePart p :Part q :Quantity)* :**String** =
"("<> *cp.name* <> "," <> *p.name* <> ")"

Bei Beziehungen, die durch einen Aufruf der Funktiuon *setRelationship.create* erzeugt worden sind, werden die Elemente in Mengen vom Typ *set.T(E(E1 E2 A))* gespeichert. Dabei ist *E* der im ersten Teil dieses Abschnitts eingeführte Typoperator für den Elementtyp von Beziehungen, und *E1* und *E2* sind die Elementtypen der beteiligten Klassen. In die Menge können nur Werte vom Typ *E(E1 E2 A)* eingefügt werden. Da die Einfügeoperation der Beziehung eine andere Signatur hat (Elemente der beiden Klassen und Attributwert als einzelne Parameter), müssen die Werte vor dem Einfügen in die Menge zu einem Tupel dieses Typs zusammengefaßt werden. Beim Löschen ergibt sich zusätzlich das folgende Problem: Bei der Löschoperation für die Beziehung wird das zu löschende Element über die beiden Elemente identifiziert, zwischen denen die Beziehung besteht. Die Löschoperation auf Mengen hingegen erwartet das zu löschende Element als Parameter. Dieses Element muß also zuerst ermittelt werden, bevor es gelöscht werden kann.

Das prädikative Löschen wird analog zu dem bei den Klassen realisiert. Bei der Implementierung der Operation *elements* kann auf die gleichnamige Funktion der Schnittstelle *set* zurückgegriffen werden. Zur Realisierung der übrigen Operationen für den lesenden Zugriff, werden die Operation *elements* und die Funktionen der Schnittstelle *Iter* verwendet, insbesondere die Funktionen *map* und *select*. Dabei ist zu beachten, daß bei der Berechnung von *allFst* und *allSnd* evtl. eine Duplikateliminierung notwendig wird, da sich Iteratoren mit Duplikaten ergeben können.

Bei einer Beziehung sind die referentielle Integrität und die Eindeutigkeit der Elemente bezüglich der an der Beziehung beteiligten Elemente zu berücksichtigen (siehe Abschnitt 5.4.1). Die Maßnahmen, die zur Wahrung der referntiellen Integrität notwendig sind, werden bereits in Abschnitt 5.4.1 beschrieben. Die Eindeutigkeit wird gewahrt, indem man verhindert, daß mehr als ein Element in die Beziehung eingefügt wird, das sich auf das gleiche Paar von Elementen bezieht. Dies kann sichergestellt werden, indem man beim Anlegen der Menge eine Funktion als Parameter übergibt, die zwei Elemente der Menge als gleich definiert, wenn sie sich auf das gleiche Paar von Elementen beziehen[19].

Es ist auch eine Implementierung denkbar, die auf *SQL*-Tabellen basiert, d.h., die Elemente der Beziehung werden in *SQL*-Tabellen gespeichert. Da in *SQL*-Tabellen jedoch nur flache Tupel erlaubt sind, können keine Verweise auf die an der Beziehung beteiligten Elemente gespeichert werden. Man muß die Verweise durch Referenzen ersetzen. Dazu müssen bei der Erzeugung einer Beziehung zwischen zwei Klassen *A* und *B* Funktionen zur Gewinnung der Referenzen aus den Elementen der Klassen *A* und *B* angegeben werden und solche, die die Gleichheit von zwei Referenzen definieren. Da es für den Benutzer unsichtbar sein soll, daß Referenzen statt Verweisen verwendet werden, muß man die Operationen auf den Beziehungen so definieren, daß sie die Verwaltung der Referenzen durchführen.

[19] Siehe dazu auch: Wahrung der Eindeutigkeit von Schlüsseln bei Klassen, Abschnitt 5.2.2.

5.4.3 Geschützte Operationen und Integritätsüberwachung

Wie bei Klassen gibt es auch bei den Beziehungen Funktionen *protect* und *iChecked*, die den Übergang zu geschützten Operationen bzw. zu Operationen mit Integritätsüberwachung ermöglichen. Sie werden von den Schnittstellen *ProtRelatonship* und *IcRelationship* zur Verfügung gestellt. Die Benutzerschnittstellen der Klassen und der Beziehungen unterscheiden sich nur in den lesenden, nicht aber in den modifizierenden Operationen. Es ergeben sich also die selben zu schützenden Operationen bzw. die selben Operationen für die Integritätsüberwachung. Die Implementierung und die Wirkungsweise der Funktionen *protect* und *iChecked* entsprechen deshalb weitgehend denen dieser Funktionen bei den Klassen (siehe Abschnitt 5.2.3), und es wird nicht weiter auf sie eingegangen. Auch bei den Beziehungen muß beim Übergang zu Operationen mit Integritätsüberwachung die Komponente *info* um die Kollektionen der Integritätsbedingungen erweitert werden. Es ergibt sich also ein neuer Typ für die Komponente *info* und damit auch ein neuer Typ *Checked* für die ganze Beziehung, der in der Schnittstelle *RelationshipType* definiert ist.

5.4.4 Integritätsbedingungen auf Beziehungen

In der Sprache *ORM* wird vorgeschlagen, vier Arten von Integritätsbedingungen auf Beziehungen zu unterstützen: referentielle Integrität, Eindeutigkeitsbedingungen, Surjektivität und die Unveränderbarkeit von Beziehungen. All diese Bedingungen werden bereits zum Zeitpunkt der Erzeugung der Beziehung festgelegt. Im Gegensatz dazu sollen sie hier dynamisch eingefügt und gelöscht werden können. Im folgenden wird untersucht, wie sich diese Bedingungen in dem hier gewählten Ansatz realisieren lassen. Bei der Betrachtung der Integritätsbedingungen wird von einer Beziehung R zwischen zwei Klassen B und C mit den Elementypen *E1* und *E2* ausgegangen. Der Typ der zusätzlichen Attribute sei A. R ist also vom Typ *RelationshipType_Checked(E1 E2 A)*. Die Funktionen zur Angabe dieser Bedingungen für eine Beziehung werden von der Schnittstelle *IcRelationship* zur Verfügung gestellt.

Referentielle Integrität: Bei der referentiellen Integrität handelt es sich um eine inhärente Bedingung von Beziehungen. Sie wird bereits weiter oben abgehandelt. In der Sprache *ORM* werden mehrere Arten von referentieller Integrität unterschieden, je nachdem wie auf eine Verletzung der Bedingung reagiert wird (automatisches Einfügen oder Abbruch der Operation). Wegen des Verzichts auf aktive Komponenten fällt diese Unterscheidung hier weg.

Eindeutigkeit: Durch eine Eindeutigkeitsbedingung wird festgelegt, daß eine Beziehung bezüglich einer oder mehrerer ihrer Komponenten eindeutig ist. Es werden wie bei den entsprechenden Bedingungen auf Klassen zwei Funktionen benötigt, eine Funktion *comp* zur Selektion der Komponente bzw. Komponenten, die eindeutig sein soll(en) und eine Funktion *equal*, um die Gleichheit auf diesen Komponenten zu definieren. Die Funktion *comp* ist vom Typ **All***(:E1 :E2 :A) :F* und die Funktion *equal* vom Typ **All***(:F :F)* :**Bool**. Auf eine Zusammenfassung zu einer Funktion wie bei den Eindeutigkeitsbedingungen auf Klassen wird hier verzichtet, da man sonst eine Funktion mit sechs Parametern erhält.

> Für die Eindeutigkeitsbedingung "Jedes Teil wird von höchstens einem Lieferanten geliefert." auf der Beziehung *suppliedBy* werden die beiden Funktionen wie folgt definiert.
>
> **let** *partComp(p :Part s :Supplier attributes :***Ok***) :Part = p*
> **let** *partsEqual(p1, p2 :Part) :***Bool** *= p1 is p2*

Analog zu den Klassen ergibt sich folgender Test: Vor dem Einfügen eines Elements e in die Beziehung R ist zu überprüfen, ob

ALL r IN R : not(equal(comp(r) comp(e)))

Eindeutigkeitsbedingungen können durch die Funktion *addUniqueness* angegeben werden. Für das oben gewählte Beispiel sieht der Aufruf der Funktion wie fogt aus:

addUniqueness(suppliedBy "uniqueSupplier" partsComp partsEqual)

Zum Löschen der Bedingung existiert die Funktion *deleteUniqueness*.

Surjektivität: R ist surjektiv (total) bezüglich der Klasse B, wenn jedes Element aus B an der Beziehung teilnimmt. Ein Beispiel hierfür ist folgende Bedingung im Stücklistenbeispiel:

> Jedes zusammengesetzte Teil besteht aus mindestens einer Komponente, d.h. nimmt an der Beziehung *madeFrom* teil.

Wenn man, wie beim *ORM* vorgeschlagen, davon ausgeht, daß das Einfügen in die Klassen vor dem Einfügen in die Beziehungen stattfindet und das Löschen aus den Beziehungen vor dem Löschen aus den Klassen, können referentielle Integritätsbedingungen sofort getestet werden, der Test von Surjektivitätsbedingungen hingegen sollte verzögert stattfinden, also ans Ende der Transaktion verschoben werden.

Die Surjektivitätsbedingung für die Beziehung R bzgl. der Klasse B läßt sich wie folgt formulieren:

ALL b IN B : (SOME r IN R : r.first is b)

Es handelt sich also um eine Bedingung der Form *ALL/SOME*, d.h., sie ist beim Einfügen in B und beim Löschen aus R zu testen. Gemäß der Umformungen in Abschnitt 5.2.4 ergibt sich für das Einfügen eines Elements e in die Klasse B der Test:

SOME r IN R : r.first is e

Das Löschen eines Elements e aus R macht den Test der Bedingung

ALL b IN relB: (SOME r IN effR : (r.first is a*)* $\wedge$ *not(r is e))*

notwendig. Vom Löschen eines Elements e aus R ist nur ein Element der Klasse B, nämlich e.first betroffen, d.h., der relevante Bereich *relB* besteht nur aus diesem Element. Der Test läßt sich also umformen zu:

SOME r IN effR : (r.first is e.first) $\wedge$ *not(r is e)*

Wie oben bereits erwähnt, werden beide Bedingungen verzögert getestet. Für die Surjektivität von R bezüglich der Klasse C sieht die Bedingung wie folgt aus:

ALL c IN C : (SOME r IN R : r.second is c)

Sie läßt sich analog umformen.

Der Typ einer Beziehung hängt von den Elementtypen der beteiligten Klassen ab. Dabei kommt es auch auf die Reihenfolge der beiden Klassen an, da sich unterschiedliche Reihenfolgen der Typparameter ergeben. Man benötigt deshalb die Funktion zur Angabe der Surjektivität in zwei Varianten, eine für die Surjektivität bezüglich der ersten Klasse und eine für die bezüglich der zweiten Klasse der Beziehung:

*addSurjectivity1(E1, E2, K1, A ::***TYPE** *rel :RelationshipType_Checked(E1 E2 A)*
class1: ClassType_Checked(E1 K1) constrName :String) **:Ok**

*addSurjectivity2(E1, E2, K2, A ::***TYPE** *rel :RelationshipType_Checked(E1 E2 A)*
class2: ClassType_Checked(E2 K2) constrName :String) **:Ok**

Zum Löschen dieser Bedingungen existieren die Funktionen *deleteSurjectivity1* und *deleteSurjectivity2*.

Unveränderbarkeit: Beim *ORM* ist diese Bedingung wie folgt definiert: Ein Element b der Klasse B ist "für alle Zeiten" (= solange das Element in der Klasse ist) mit einer festen Menge von Elementen in der Beziehung verbunden. Die Elemente dieser Menge müssen innerhalb der selben Transaktion wie b eingefügt werden und können erst in der Transaktion wieder gelöscht werden, in der auch b aus seiner Klasse B gelöscht wird. Hier tritt folgendes Problem auf: Die Bedingung für das Löschen läßt sich durch eine geeignete Integritätsbedingung, die verzögert getestet wird, sicherstellen. Zur Formulierung der Bedingung für das Einfügen von Elementen in die Beziehung, muß man jedoch wissen, ob das zugehörige Element der Klasse B in der aktuellen Transaktion eingefügt worden ist. Um dies zu entscheiden, braucht man entweder Zugriff auf die Menge der Elemente der Klasse, die während der aktuellen Transaktion eingefügt worden sind oder aber Zugriff auf den Zustand der Klasse vor der aktuellen Transaktion. Bei dem hier gewählten Ansatz verfügt man jedoch über keine dieser beiden Informationen. Es wird deshalb im Prototyp darauf verzichtet, diese Art Integritätsbedingung zu implementieren.

5.5 Objekte

Bei den meisten objekt-orientierten Ansätzen werden die Struktur und die Methoden der Objekte bei der Definition der Klasse festgelegt (vgl. Abschnitt 2.2.2). Der hier vorgestellte

Ansatz trennt, wie beim *ORM* vorgeschlagen, das Konzept der Klassen von dem der Objekte. Struktur und Methoden von Objekten werden durch Objekttypen festgelegt.

Das Konzept der Klassen wird in Abschnitt 5.2 behandelt. In diesem Abschnitt wird überlegt, inwiefern das Konzept der Objekt-Orientierung als *Add-On*-Ansatz realisiert werden kann und welche Unterstützung eine moderne Programmiersprache wie *P-Quest* dafür anbietet. Dazu sind gemäß [Weg87] drei Konzepte zu untersuchen:

- Eindeutige Identität
- Kapselung des Zustands des Objekts
- Vererbung von Methoden und Attributen

Diese Konzepte werden im folgenden getrennt betrachtet.

Identität

Objekte haben eine systemweit eindeutige, vom System verwaltete Identität. Bei allen Typen, die keine Basistypen sind, wird der Begriff der Identität von der Sprache *P-Quest* unterstützt. Zwei Werte eines solchen Typs haben nur dann die gleiche Identität (Test durch *is*), wenn sie aus der selben Erzeugung stammen, also zwei Namen des selben Elements sind. Außerdem besteht auch die Möglichkeit, Identitäten explizit zu verwalten. Dazu kann man einen abstrakten Datentyp *Id* anbieten, auf dem die Operationen *newId* zur Erzeugung neuer Identitäten und *idEqual* zum Vergleich von Identitäten definiert sind. Bei der Erzeugung jedes Objekts muß dann eine neue Identität vom Typ *Id* erzeugt und diesem Objekt fest zugeordnet werden.

Kapselung

Ein Objekt besitzt einen internen Zustand, auf den nur durch einen festen Satz von Operationen, den Methoden zugegriffen werden kann. Eine solche Kapselung kann durch Verwendung eines abstrakten Datentyps (ADT) für den Objekttyp erreicht werden. Dabei enthält die Schnittstelle des ADT die Methoden des Objekts, die die einzig zulässigen Operationen zur Manipulation des Zustands des Objekts darstellen. Der Objekttyp für die Basisteile im Stücklistenbeispiel kann z.B. wie folgt aussehen:

```
Let BasePart =
   Tuple
     getPno() :Int
     getName() :String
     getCost() :Int
     getMass() :Int
     setCost(:Int) :Ok
     setMass(:Int) :Ok
  end
```

Der Objekttyp legt die Signaturen der Methoden der Objekte fest. Weiterhin benötigt man einen Generator, mit dem neue Objekte des Objekttyps erzugt werden können. Der Generator legt die Implementation der Methoden und den Aufbau des internen Zustands des Objekts fest. Für den Objekttyp *BaseParts* kann der Generator z.B. wie folgt aussehen:

```
let newBasePart(pno :Int  name :String  cost :Int  mass :Int) :BasePart =
  begin
    let state =
       tuple
         let pno  = pno
         let name = name
         let var cost = cost
         let var mass = mass
       end
    tuple
       let getPno() :Int = state.pno
       let getName() :String = state.name
       let getCost() :Int = state.cost
       let getMass() :Int = state.mass
       let setCost(newCost :Int) :Ok = state.cost := newCost
       let setMass(newMass :Int) :Ok = state.mass := newMass
     end
  end
```

Es ist auch möglich mehrere Generatoren zu einem Objekttyp zu schreiben, die sich z.B. in der Implementierung der Methoden unterscheiden.

Vererbung

Das Konzept der Vererbung wird in *P-Quest* nicht unterstützt. Objekttypen können deshalb nicht inkrementell definiert werden.

Kapitel 6

Auswertung

Inhalt dieses Kapitels ist die Auswertung der vorgestellten Arbeit und die Zusammenfassung der Ergebnisse. Es ist wie folgt aufgebaut: Im Abschnitt 6.1 wird auf die Erfahrungen eingegangen, die bei der Implementierung der Bibliothek generischer Dienste mit der Programmiersprache *P-Quest* gemacht worden sind. Abschnitt 6.2 faßt die Ergebnisse der Arbeit zusammen und Abschnitt 6.3 gibt einen Ausblick auf mögliche Erweiterungen und Ergänzungen des vorgestellten Ansatzes.

6.1 Erfahrungen mit *P-Quest*

Die Bibliothek generischer Dienste wurde in der Sprache *P-Quest* erstellt. In diesem Rahmen konnte die Eignung der Sprache für den Entwurf von Schnittstellen und für die Erstellung einer prototypischen Implementierung untersucht werden. Dieser Abschnitt beschreibt die Erfahrungen mit der Sprache *P-Quest*. Dabei wird sowohl darauf eingegangen, welche Konzepte der Sprache sich als besonders nützlich erwiesen haben (Abschnitt 6.1.1), als auch, an welchen Stellen die Mächtigkeit der Sprache nicht ausreicht (Abschnitt 6.1.2). Zusammenfassend kann man sagen, daß die Sprache *P-Quest* zur Bewältigung der vorliegenden Aufgabe gut geeignet ist und die notwendige Unterstützung anbietet, auch wenn an einigen Stellen gewisse störende Einschränkungen auftreten. Zum anderen ist zu bemerken, daß die zur Verfügung gestellte Mächtigkeit auch tatsächlich notwendig ist, um eine Bibliothek generischer Dienste für datenintensive Anwendungen realisieren zu können.

Anhand der Sprache *TL* (=TYCOON[1] Language), die eine Weiterentwicklung der Sprache *P-Quest* ist, wird aufgezeigt, daß durch einige wenige Erweiterungen und durch den Wegfall einiger Einschränkungen der Sprache *P-Quest* viele der festgestellten Probleme überkommen werden können. Die Sprache *TL* ist eine polymorphe persistente Programmiersprache mit Funktionen und Typen als Sprachobjekten erster Klasse. Sie wurde an der Universität Hamburg im Rahmen einer Dissertation [Mat92] entwickelt. Ein Prototyp der Sprache ist implementiert.

[1] TYCOON ist das Akronym für *TYped Communicating Objects in Open eNvironments.*

6.1.1 Positive Erfahrungen

Als besonders nützlich haben sich der parametrische Polymorphismus und die Funktionen höherer Ordnung erwiesen. Auf der Typebene ist der parametrische Polymorphismus durch die Typoperatoren realisiert. Diese sind von zentraler Bedeutung für die Definition von Bulk-Typen, da sie es erlauben, den Typ von Kollektionen mit beliebigem Elementtyp zu definieren (vgl. dazu Abschnitt 5.2.1 und 4.2.1). Ohne Typoperatoren ist die *Add-On*-Definition neuer Bulk-Typen ohne Festlegung des Elementtyps nicht möglich.

Eine ideale Ergänzung zu den Typoperatoren stellen die generischen Funktionen (= parametrischer Polymorphismus bei Funktionen) dar. Sie können auf Strukturen arbeiten, die durch Typoperatoren erzeugt worden sind. Da in der Bilbliothek fast nur solche generischen Strukturen eingesetzt werden, handelt es sich bei den meisten der in den Schnittstellen der Bibliothek angebotenen Funktionen um generische Funktionen (siehe z.B. Abschnitt 4.2.2).

Weiterhin hat sich auch das Konzept der Funktionen höherer Ordnung bewährt, besonders in Kombination mit der Tatsache, daß Funktionen in *P-Quest* Werte erster Klasse sind. Dadurch ist es z.B. möglich, Funktionen als Komponenten von Tupel-Typen anzugeben und in Kollektionen abzuspeichern. Die Abspeicherung in Kollektionen wird bei den generischen Diensten z.B. bei der Verwaltung der Integritätsbedingungen (siehe Abschnitt 4.4.2) und der Verwaltung des *Undo-Logs* (siehe Abschnitt 4.3.3) verwendet. Höhere Funktionen werden in diesem Zusammenhang z.B. zum Einfügen kompensierender Operationen in das *Undo-Log* benötigt.

Ein weiteres wichtiges Einsatzgebiet für die Funktionen höherer Ordnung ist die Iterationsabstraktion. Hierbei ermöglicht es die Verwendung dieser Funktionen, abstrakte und allgemeine Funktionen zur Iterationsabstraktion zu schreiben, die zu verschiedenen Zwecken wie z.B. zur Definition von Anfragesprachen und für Ausgabefunktionen eingesetzt werden können (Abschnitt 4.2.2). Die Details für die jeweilige Anwendung der Funktion werden durch die als Parameter übergebene(n) Funktion(en) festgelegt. Ebenso wird beim Erzeugen von Klassen der Schlüssel der Klasse durch zwei als Parameter übergebene Funktionen definiert (siehe Abschnitt 5.2.1).

Funktionen höherer Ordnung ermöglichen die Definition von Generatoren für Funktionen. Ein Beispiel für solche Generatoren sind die Transaktionsgeneratoren, die eine Funktion als Parameter erhalten und eine Transaktion zurückliefern (siehe Abschnitt 4.5.1).

Als wichtig haben sich auch die rekursiven Funktionen erwiesen. Sie ermöglichen z.B. bei der Implementierung der Funktionen zur Iterationsabstraktion einen funktionalen Programmierstil (vgl. dazu Abschnitt 4.2.3).

Bei der Definition von Transaktionen hat sich die von *P-Quest* angebotene Ausnahmebehandlung als nützlich erwiesen. Sie ermöglicht es, das Zurücksetzen einer Transaktion als Ausnahmebehandlung für während der Transaktion aufgetretene Fehler (Ausnahmen) anzugeben (siehe Abschnitt 4.3.1).

Positiv zu erwähnen ist auch die Unterstützung des Programmierers durch die Typinferenz. In vielen Fällen[2] können beim Aufruf generischer Funktionen die Typparameter weggelassen werden. Das *P-Quest*-System kann die Typparameter durch Typinferenz aus den Typen der

[2]Der Typparameter muß aus den Typen der übrigen Aktualparameter ableitbar sein.

übrigen Parameter herleiten. Dieser Aspekt spielt sowohl für die Implementierung als auch für die Benutzung der generischen Dienste eine Rolle, da, wie bereits erwähnt, die Schnittstellen viele generische Funktionen enthalten.

Zwei inzwischen allgemein als nützlich anerkannte Konzepte nämlich die Datenabstraktion und die Modularisierung sind ebenfalls in der Sprache *P-Quest* enthalten und waren bei der Erstellung der Bilbliothek sehr hilfreich: Bei abstrakten Datentypen ist der Zugriff auf dessen Wert auf die Operationen beschränkt, die zu diesem Zweck definiert werden. Dies erleichtert die Integritätsüberwachung, da auf dem Datentyp definierte Integritätsbedingungen nur von diesen Operationen verletzt werden können. Ein Test ist also nur bei Ausführung dieser Operationen notwendig (vgl. Abschnitt 4.4.4).

Das Modulkonzept unterstützt den strukturierten Entwurf und Aufbau der Bilbliothek und ermöglicht deren Erweiterbarkeit. In *P-Quest* ist es möglich, zu einer Schnittstelle mehrere Implementationsmodule zu schreiben. Dabei kann ein bereits definiertes Modul bei der Definition eines weiteren Moduls zur gleichen Schnittstelle verwendet werden. Dies ermöglicht eine inkrementelle Weiterentwicklung von Implementationen.

6.1.2 Einschränkungen

Das Konzept der Datenaggregation ist sehr wichtig für datenintensive Anwendungen. Zur Darstellung dieses Konzepts bietet *P-Quest* den Tupeltyp an, der jedoch in Bezug auf die Subtypbildung sehr restriktiv ist: Die Komponenten eines Tupels sind geordnet, und man kann deshalb nur durch Anfügen von Komponenten am Ende des Tupels Subtypen bilden. Dies verhindert den Aufbau von Mehrfachvererbungshierarchien [Car84]. Für einige Anwendungen ist dieser Ansatz zu restriktiv. Abhilfe können hier die *Records* schaffen, bei denen die Komponenten ungeordnet sind. Sie sind im ursprünglichen Vorschlag für die Sprache *Quest* [Car89] enthalten, wurden aber nicht realisiert. In der Sprache *TL* ist neben dem Konzept der *Tupel* auch das Konzept der *Records* realisiert und somit der Aufbau von Mehrfachvererbungshierarchien möglich.

Deutliche Einschränkungen der Sprache *P-Quest* zeigten sich bei dem Versuch, einen objektorientierten Ansatz zu implementieren (siehe Abschnitt 5.5). Insbesondere fehlt das Konzept der Vererbung, das es erlaubt, Subklassen (bzw. Objekttypen) inkrementell zu definieren. Dies ist in den meisten objektorientierten Sprachen möglich. Auch *TL* stellt kein spezielles Konstrukt für die Vererbung zur Verfügung, sondern ein allgemeines Konstrukt *Repeat*, mit dessen Hilfe unter anderem die Vererbung zwischen Klassen realisiert werden kann. Eine bereits definierte und benannte Tupel-, Record- oder Funktionssignatur A kann innerhalb einer anderen Signatur durch Angabe von *Repeat A* wiederholt werden. Mit *open* wird in *TL* außerdem ein entsprechendes Konstrukt auf der Werteebene zur Verfügung gestellt. Es ermöglicht die Wiederholung benannter Bindungen in Bindungen.

Zur Definition des Typs einer Klasse wird ein Typoperator verwendet, der einen Typ E auf den Typ einer Klasse mit diesem Elementtyp abbildet. Bei dem hier gewählten Ansatz soll der Typ einer Klasse eine Liste ihrer direkten Subklassen als Komponente enthalten (vgl. Abschnitt 5.3.1). Da die Subklassen in der Regel nicht den selben Elementtyp wie ihre Superklasse haben, wird für die Definition des Typs einer Klasse ein rekursiver Typoperator benötigt:

```
Let Rec T(E, K ::TYPE) ::TYPE =
  Tuple
   info : Tuple
            ...
            subclasses :list.T(Tuple E1 <: E subclass: T(E1 K) end)
            ...
          end
   insert(:E) :Ok
   delete(:E) :Ok
   deleteAll(p(:E) :Bool) :Ok
   lookup(:K) :E
   elements() :Iter_T(E)
  end
```

Aufgrund einer Einschränkung der Sprache *P-Quest* sind jedoch rekursive Typoperatoren nicht zulässig. Ersatzweise können Typoperatoren definiert werden, die auf rekursive Typen abbilden. Ein solcher Operator genügt jedoch in diesem Fall nicht, da der Typoperator bei der Defintion der Klasse in unterschiedlich parametrisierter Form (Elementtypen der Subklassen) auftritt. In der Sprache *TL* sind rekursive Typoperatoren (mit Namensäquivalenzregel) erlaubt.

Da Superklassen die Elemente all ihrer Subklassen enthalten, haben die Elemente der Ausprägung einer Klasse evtl. nicht alle den gleichen Typ. Es kann sich bei den Ausprägungen also um inhomogene Kollektionen handeln. In manchen Situationen ist es bei der Bearbeitung einer solchen Klassenausprägung notwendig, eine Fallunterscheidung abhängig vom tatsächlichen Typ jedes Elements durchzuführen. Ein Beispiel hierfür ist die Berechnung der Kosten eines Teils beim Stücklistenbeispiel. Abhängig vom Elementtyp (*BasePart* oder *CompositePart*) muß bei der Berechnung der Kosten anders verfahren werden. In *P-Quest* sind jedoch nur wertbasierte Fallunterscheidungen möglich. *TL* bietet mit dem Konstrukt *typecase* auch eine Möglichkeit zur typgesteuerten Fallunterscheidung an.

Die bisher genannten Einschränkungen der Sprache *P-Quest* können, wie am Beispiel der Sprache *TL* zu sehen ist, durch einige wenige Erweiterungen der Sprache bzw. durch den Wegfall von Einschränkungen aufgehoben werden. Es sind jedoch bei der Entwicklung der generischen Bibliothek auch Probleme mit der Sprache *P-Quest* aufgetreten, die (zumindest für typisierte Sprachen) mehr prinzipieller Natur sind.

So kann man z.B. keinen Transaktionsgenerator schreiben, dem beliebige Funktionen als Parameter übergeben werden können (siehe Abschnitt 4.5.1). Das Problem ist dabei die Anzahl der Parameter der Funktion. Sie wird mit der Angabe der Signatur des Generators festgelegt, wie am Beispiel der Signaturen der Transaktionsgeneratoren für Funktionen mit einem und mit zwei Parametern zu sehen ist[3]:

```
generate1(E,F ::TYPE   f(:E) :F) :All(:E) :F
generate2(E,F,G ::TYPE   f(:E :F) :G) :All(:E :F) :G
```

Dieses Phänomen tritt in ähnlicher Form noch an einigen anderen Stellen auf, so z.B. bei dem Versuch, n-äre Beziehungen für beliebige n durch ein einheitliches Konstrukt darzustellen.

[3]Die Typen der Parameter der Funktion können durch Angabe von Typparametern offen gehalten werden.

Es liegt in der Natur von Signaturen die Anzahl der Parameter festzulegen, und es gibt keine typsichere Methode diese Anzahl variabel zu gestalten. Eine mögliche Lösung dieses Problems stellt die Verwendung von Reflektion bei der Generierung von Operationen dar (vgl. dazu Abschnitt 6.3).

Wegen der Kontravarianzregel (Siehe [Car89], Seite 34) für die Subtypbeziehung auf Funktionen ist es bei dem hier gewählten Ansatz nicht möglich, daß der Typ einer Klasse Subtyp des Typs ihrer Superklasse ist. Die Funktion für den Schlüsselzugriff *lookup* [4] ist wie alle anderen Operationen auf Klassen Komponente des Typs von Klassen (s.o.). Wegen der Kontravarianzregel muß man also fordern, daß eine Klasse den selben Schlüsseltyp wie ihre Superklasse hat. Klassen ohne Schlüssel sind als Spezialfall der Klassen mit Schlüssel definiert, wobei diese als Schlüsseltyp ihren Elementtyp haben. Bei diesen Klassen besitzt *lookup* die Signatur:

lookup(:E) :E

Da der Elementtyp in Ko- und Kontravarianter Position auftritt, stehen die Typen von Sub- und Superklassen ohne Schlüssel nur bei gleichem Elementtyp in Subtypbeziehung. Dies stellt eine inakzeptable Einschränkung dar. Die Kontravarianzregel kann also bei realen Programmieraufgaben zu Einschränkungen führen.

6.2 Zusammenfassung der Ergebnisse

Ziel dieser Arbeit war es zu untersuchen, ob man eine Umgebung für datenintensive Anwendungen als reinen *Add-On*-Ansatz implementieren kann. Zu diesem Zweck wurde ein Prototyp für eine Bibliothek generischer Dienste implementiert, die eine solche Umgebung zur Verfügung stellt. Die Bibliothek bietet eine Reihe von Diensten, wie z.B. Iterationsabstraktion, Transaktionsverwaltung und Integritätsüberwachung. Die geschaffene Umgebung ist dabei nicht an ein bestimmtes Datenmodell gebunden. Sie stellt die Dienste in modellunabhängiger Form zur Verfügung.

Die Bibliothek enthält eine Reihe von Schnittstellen für grundlegende Bulk-Typen wie Listen und Mengen. Diese können als Basis für die Implementation spezieller Bulk-Typen wie Klassen oder Relationen verwendet werden. Ein wichtiger Bestandteil der Bibliothek ist die Schnittstelle zur Iterationsabstraktion. Sie bietet eine Reihe allgemeiner höherer Funktionen an, mit denen unter anderem eine Anfragesprache für beliebige Bulk-Typen definiert werden kann. Die Funktionen können aber auch bei der Definition von Integritätsbedingungen und Sichten eingesetzt werden.

Die implementierte Transaktionsverwaltung unterstützt das Zurücksetzen von Operationen im Falle des Abbruchs einer Transaktion, den Test verzögerter Integritätsbedingungen und die Protokollierung von Fehlermeldungen. Sie kann auch geschachtelte Transaktionen verwalten[5]. Weiterhin wurde ein Transaktionsgenerator implementiert, der aus einer Funktion automatisch eine Transaktion erzeugen kann.

[4] Sie bildet vom Schlüsseltyp der Klasse auf ihren Elementtyp ab.

[5] Aspekte der Nebenläufigkeit konnten nicht behandelt werden, da die verwendete Sprache *P-Quest* hierfür keine Unterstützung anbietet.

Für die Integritätsüberwachung wurde ein dynamischer Ansatz gewählt. Bei dem zur Verfügung gestellten Dienst werden die Integritätsbedingungen in Kollektionen verwaltet, die Operationen zugeordnet werden können. Die Kollektionen können nach Bedarf durch Einfügen und Löschen von Bedingungen aktualisiert werden. In der Arbeit werden Regeln angegeben, nach denen man beim Übergang zu Datenstrukturen mit Integritätsüberwachung vorgehen kann.

Um die Eignung der Dienstbibliothek zu testen, wurde in der von den Diensten zur Verfügung gestellten Umgebung ein Datenmodell implementiert. Zu diesem Zweck wurde das *Object-Relationship*-Modell [AGO91b] gewählt, ein objektorientiertes Modell mit einem eigenen Konstrukt für die Darstellung von Beziehungen.

Neben dem Test der Dienstbibliothek hatte die Implementation auch den Zweck zu untersuchen, inwieweit sich Konzepte der modernen Datenmodellierung wie Klassen, Objekte und Generalisierungshierarchien als *Add-On*-Ansatz realisieren lassen. Der Schwerpunkt lag dabei auf dem Entwurf benutzerfreundlicher Schnittstellen. Es wurden aber auch eine prototypische Implementierung dieser Schnittstellen erstellt, um die Durchführbarkeit einer solchen Implementierung zu zeigen[6].

Klassen abstrahieren von Implementationsdetails der zugrundeliegenden Massendatenverwaltung. So ist es möglich, in einer Datenbank verschiedene unterliegende Strukturen und trotzdem einheitliche Benutzerschnittstellen zu haben. Insbesondere können die Elemente einer Klasse auch in Form einer *SQL*-Tabelle vorliegen. Dies ermöglicht die harmonische Integration bereits existierender *SQL*-Datenbanken in die Umgebung des *Object-Relationship*-Modells.

Zur Realisierung der Integritätsüberwachung auf Klassen konnte eine Funktion geschrieben werden, die eine Klasse auf eine Klasse mit Integritätsüberwachung abbildet. Bereits in der abgebildeten Klasse existierende Elemente werden dabei übernommen. Die Integritätsüberwachung konzentiert sich auf einige spezielle Klassen von Integritätsbedingungen. Diese können in deklarativer Form angegeben werden. Die bei dem gewählten Ansatz notwendige Bindung an Operationen wird automatisch durchgeführt und es werden geeignete Tests generiert, die die Wahrung der Bedingungen sicherstellen.

Für die Darstellung von (binären) Beziehungen zwischen Klassen wurde gemäß des *Object-Relationship*-Modells ein spezielles Konstrukt implementiert. Es stellt im wesentlichen eine Erweiterung des Konzepts der Klassen dar. Im Unterschied zu den Klassen werden Operationen angeboten, die spezifisch für Beziehungen sind, und die referentielle Integrität für die beteiligten Elemente sicherstellen. Auch für die Beziehungen gibt es eine Funktion, die den Übergang zu Beziehungen mit Integritätsüberwachung unter Beibehaltung bereits existierender Elemente ermöglicht.

Beim Aufbau von Generalisierungshierarchien auf Klassen wurde der Aspekt der Inklusionsbeziehungen zwischen den Ausprägungen von Sub- und Superklassen betont. Es wurde hierbei ein Ansatz gewählt, der die Ausprägung einer Superklasse bei Bedarf als Vereinigung der Ausprägungen ihrer Subklassen berechnet. Außerdem wurde eine Vererbung von Integritätsbedingungen einer Superklasse an ihre Subklassen implementiert.

Als besondere Vorteile der Realisierung einer Umgebung für datenintensive Anwendungen als *Add-On*-Ansatz gegenüber einem *Built-In*-Ansatz sind folgende Punkte zu nennen:

[6] Effizienzaspekte wurden nicht berücksichtigt.

- Es besteht die Möglichkeit, verschiedene Datenmodelle in der selben Umgebung generischer Dienste zu implementieren.
- Die Dienste können einfach erweitert und individuellen Bedürfnissen angepaßt werden, ohne in die "Tiefen" des Systems vordringen zu müssen.
- Der Benutzer erhält eine gute Unterstützung bei der Definition neuer Bulk-Typen.
- Dies alles kann im selben einheitlichen sprachlichen Rahmen geschehen, in dem auch die Anwendungen geschrieben werden. Dies führt dazu, daß die festen Grenzen zwischen Anwendungsprogrammierung und Systemprogrammierung wegfallen.

6.3 Ausblick

Die hier vorgestellte Bibliothek generischer Dienste stellt einen ersten Prototyp dar. Es existiert deshalb eine Reihe von Aspekten, die noch weiter untersucht werden müssen, und es sind noch viele Erweiterungen und Ergänzungen des Ansatzes möglich:

Optimierung: Bei der Implementierung der Schnittstellen wurde der Aspekt der Effizienz weitgehend vernachlässigt. Es besteht deshalb noch die Notwendigkeit der Optimierung der vorgestellten Lösungen. Von besonderem Interesse für den Bereich der datenintensiven Anwendungen ist dabei die Anfrageoptimierung und die Optimierung von Integritätstests.

Andere Datenmodelle: In der Umgebung, die von der Bibliothek generischer Dienste zur Verfügung gestellt wird, können auch andere Datenmodelle wie z.B. das relationale Modell implementiert werden.

Aktive Systeme: Auf die Implementierung von aktiven Komponenten wurde bei dem hier vorgestellten Ansatz bewußt verzichtet. Wie in der Arbeit bereits angedeutet wurde, ist es jedoch ohne Probleme möglich, auch aktive Komponenten als generischen Dienst anzubieten. Ein mögliches Einsatzgebiet für solche Komponenten ist die Wahrung bzw. Wiederherstellung von Integritätsbedingungen durch korrigierende Operationen.

Metadatenverwaltung und Reflektion: In einer Datenbank sind nicht nur Daten sondern auch Metadaten (Informationen über das Schema) zu verwalten. Der Aspekt der Metadatenverwaltung (Datenwörterbuch) wurde in der Arbeit bereits erwähnt, aber nicht weiter behandelt. Schnittstellen zur Metadatenverwaltung können in die implementierte Umgebung integriert werden. Dabei ist es besonders hilfreich, daß Daten und Metadaten im selben sprachlichen Rahmen behandelt werden können.

Eine Schemabeschreibung enthält strukturelle Information über die Anwendung. Aus dieser kann die Struktur grundlegender Operationen (Transaktionen) abgeleitet werden, die die strukturelle Integrität erhalten (vgl. [BR84a]). In [SSS+92a] wird vorgeschlagen, mi Hilfe von Reflektion ([SSS92b]) aus der Schemabeschreibung automatisch Operationen zu generieren, die die modellinhärenten Integritätsbedingungen erhalten. Dies soll im selben sprachlichen Rahmen geschehen wie die Anwendungsprogrammierung. Dazu ist es nötig, daß man in der Sprache über die Struktur der Metadaten verfügen kann.

Zu diesem Zweck werden sog. Repräsentationstypen eingeführt, deren Werte Repräsentationen konkreter Schemainformation sind. Durch Anwendungen von gewöhnlichen Funktionen auf solche Werte können Transformationen auf dem Schema durchgeführt werden. Auf diese Weise ist es möglich, Generatoren zu schreiben, die Schemabeschreibungen als Parameter erhalten und integritätserhaltende Operationen erzeugen.

Grafische Unterstützung: Eine mögliche Erweiterung des vorgestellten Ansatzes sind grafische Schnittstellen. Diese tragen zur Erhöhung der Benutzerfreundlichkeit bei. Es sind grafische Schnittstellen für den Schemaentwurf, wie sie etwa das System *Sidereus* [ACCP89] für die Datenbankprogrammiersprache *Galileo* [AGOO88] anbietet, denkbar. Das grafisch erstellte Schema könnte dann durch Aufrufe geeigneter Funktionen der Bibliothek in eine Schemadefinition in der Sprache *P-Quest* übersetzt werden. Dabei sollte die Übersetzung in beiden Richtungen möglich sein, um sich auch bereits existierende oder textuell modifizierte Schemata grafisch anzeigen lassen zu können. Definition und Übersetzung sollten inkrementell durchgeführt werden können.

Eine weitere mögliche grafische Unterstützung sind (generische) Datenbankbrowser, die eine Visualisierung des Datenbankinhalts erlauben. Im Rahmen einer Studienarbeit [KM92] an der Universität Hamburg wurde ein Ansatz für solche generischen Datenbankbrowser in der Sprache *Quest* implementiert.

Syntaktischer Erweiterbarkeit: Ein Vorteil von *Built-In*-Ansätzen gegenüber von *Add-On*-Ansätzen ist die benutzerfreundlichere Syntax, da bestimmte Konstrukte auf spezielle Zwecke zugeschnitten werden können.

Cardelli stellt in [Car92] ein System *F-sub* vor, das mit einem minimalistischen Typsystem auskommt. Das System enthält eine Komponente, die die inkrementelle, syntaktische Erweiterung von *F-sub* ermöglicht. Diese Komponente erlaubt es, in die Syntax von *F-sub* neue Konstrukte einzuführen und *F-sub*-Funktionen anzugeben, in die diese Konstrukte übersetzt werden.

Eine solche Komponente stellt eine ideale Ergänzung für Systeme dar, in denen *Add-On*-Ansätze realisiert werden sollen. Anstatt sprachliche Unterstützung für spezielle Konstrukte wie z.B. Klassen einzubauen, kann ausgehend von einem kleinen Sprachkern die Syntax gemäß den speziellen Bedürfnissen erweitert werden. Spezielle Syntax z.B. für die Definition von Transaktionen könnte dann als Teil der Bibliothek generischer Dienste angeboten und intern in Aufrufe von Funktionen der entsprechenden Schnittstelle übersetzt werden. Eine solche Komponente für syntaktische Erweiterungen könnte z.B. in das System TYCOON [Mat92], das die Sprache *TL* anbietet, aufgenommen werden.

Verweis auf den Anhang

Zu der vorliegenden Diplomarbeit gehört ein Anhang, der aus zwei Teilen besteht:

Anhang A: Datenmodellierungsbeispiel
: In diesem Teil des Anhangs findet sich eine zusammenfassende Darstellung des Stücklistenbeispiels, das in Kapitel 5 zur Veranschaulichung der vorgestellten Schnittstellen verwendet wird.

Anhang B: Ausgewählte Schnittstellen der Implementierung
: In diesem Teil des Anhangs werden die wichtigsten der in Kapitel 4 und 5 vorgestellten Schnittstellen aufgelistet.

Literaturverzeichnis

[AB87] M.P. Atkinson and P. Bunemann. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2), Juni 1987.

[ABM88] M.P. Atkinson, P. Buneman, and R. Morrison, editors. *Data Types and Persistence*. Topics in Information Systems. Springer-Verlag, 1988.

[ACCP89] L. Alfo, S. Coluccini, P. Corte, and D. Presenza. Manuale del Sistema Sidereus. Technical report, Dipartimento di Informatica, Università di Pisa, Italy, 1989.

[AGO89] A. Albano, G. Ghelli, and R. Orsini. Types for Databases: The Galileo Experience. In *Proc. of the 2nd Workshop on Database Programming Languages, Portland, Oregon*, Juni 1989.

[AGO91a] A. Albano, G. Ghelli, and R. Orsini. Objects and Classes for a Database Programming Language. Technical Report FIDE/91/16, Department of Computing Science, University of Glasgow, 1991.

[AGO91b] A. Albano, G. Ghelli, and R. Orsini. A Relationship mechanism for strongly typed object-oriented database programming languages. In G.M. Lohman, A. Sernadas, and R. Champs, editors, *Proceedings of the Seventeeth International Conference on Very Large Databases, Barcelona (Catalonia, Spain), September 3–6, 1991*, pages 565–576. Morgan Kaufmann Publishers, 1991. (also as FIDE report 91/17, Department of Computing Science, University of Glasgow).

[AGOO88] A. Albano, G. Ghelli, M.E. Occhiuto, and R. Orsini. Galileo Reference Manual, Version 2.0. Technical report, Dipartimento di Informatica, Università di Pisa, Februar 1988.

[AM87] M.P. Atkinson and R. Morrison. Polymorphic Names and Iterations. In *Proc. of the Workshop on Database Programming Languages, Roscoff, France*, September 1987.

[ART90] M.P. Atkinson, P. Richard, and P.W. Trinder. Bulk Types for Large Scale Programming. In *Information Systems*, 1990. (also as Rapport Technique Altair 60-90 nov. 1990, GIP Altair, France).

[BDRZ83] R.P. Brägger, A. Dudler, J. Rebsamen, and C.A. Zehnder. Gambit: An Interactive Database Design Tool for Data Structures, Integrity Constraints and Transactions. In C.A. Zehnder, editor, *Database Techniques for Professional Worskstations*, pages 65–96. Department Informatik, ETH Zürich, Switzerland, September 1983.

[Bla81] B.T. Blaustein. *Enforcing Database Assertions: Techniques and Applications.* PhD thesis, Harvard University, Cambridge, 1981.

[BMM92] F. Bry, R. Manthey, and B. Martens. Integrity Verification in Knowledge Bases. In A. Vronkov, editor, *Proceedings of the First Russian Conference on Logic Programming, Irkutsk, Russia, September 14–18, 1990*, volume 592 of *Lecture Notes in Artificial Intelligence*, pages 114–139. Springer-Verlag, 1992.

[BR84a] M.L. Brodie and D. Ridjanovic. On the Design and Specification of Database Transactions. In M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*, Topics in Information Systems. Springer-Verlag, 1984.

[BR84b] M.L. Brodie and D. Ridjanovic. A strict Database Transaction Design Methodology. *Computer Corporation of America, Cambridge*, 26(4), April 1984.

[Bro92] M.L. Brodie. Object-Oriented, Distributed Computing in Telecommunications: Opportunities and Challenges. (talk given at University of Hamburg, Database and Information Systems Group; also given at Telecommunication Networking Architecture Workshop '92, Narita, Japan, January 21–23, 1992), April 1992.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.

[Car89] L. Cardelli. Typeful Programming. Digital Systems Research Center Report No. 45, Digital Systems Research Center, Palo Alto, California, May 1989.

[Car90] L. Cardelli. The Quest Language and System (Tracking Draft). Digital Systems Research Center, DEC SRC Palo Alto, 1990. (shipped as part of the Quest V.12 system distribution).

[Car92] L. Cardelli. F-sub, the System. Digital Systems Research Center Report No., Digital Systems Research Center, Palo Alto, California, February 1992. (shipped as part of the Fsub 1.4 system distribution).

[CDF+86] M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita. The Architecture of the EXODUS Extensible DBMS. In K. Dittrich and U. Dayal, editors, *Proceedings of the First International Workshop on Object-Oriented Database Systems, Pacific Grove, California, September 23–26, 1986*, pages 52–65. The Institute of Electrical and Electronics Engineers, Inc., 1986.

[CDG+88] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 Report. Technical Report ORC-1, Olivetti Research Center, Menlo Park, California, 1988.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Boston, Massachusetts*, pages 316–325, Juni 1984.

[Cod70] E.F. Codd. A Relational Model of Data for Large Shared Databanks. *Communications of the ACM*, 13(6):377–387, June 1970.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintainance. Research Report RJ 7348 (68829) 3/1/90, IBM Almaden Research Center, San Jose, California, March 1990. (also in VLDB'90 but abreviated).

[Dat81] C.J. Date. Referential Integrity. In C. Zaniolo and C. Delobel, editors, *Proceedings of the Seventh International Conference on Very Large Databases, Cannes, France, September 1981*, pages 2–12. Morgan Kaufmann Publishers, 1981.

[Dat89] C.J. Date. *A Guide to the SQL Standard.* Addison-Wesley, second edition, 1989.

[Day89] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, Juni 1989.

[Deu90] O. Deux et al. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[Dij68] E.W. Dijkstra. Go To Statement Considered Harmful. *Communications of the ACM*, 3(11):147–148, March 1968.

[GOP90] K.E. Gorlen, S.A. Orlow, and P.S. Plexico. *Data Abstraction and Object-Oriented Programming in C++.* John Wiley & Sons, 1990.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, 1983.

[Gra81] J. Gray. The Transaction Concept: Virtues and Limitations. In C. Zaniolo and C. Delobel, editors, *Proceedings of the Seventh International Conference on Very Large Databases, Cannes, France, September 1981*, pages 144–154. Morgan Kaufmann Publishers, 1981.

[HL74] C.A.R Hoare and P.E. Lauer. Consistent and Complementary Formal Theories of the Semantics of Programming Languages. *Acta Informatica*, 3:135–153, 1974.

[HR83] T. Haerder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[HS78] M. Hammer and S.K. Sarin. Efficient Monitoring of Database Assertions. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, 1978.* Association for Computing Machinery, 1978.

[Hug91] J.G. Huges. *Object-Oriented Databases.* International Series in Computer Science. Prentice Hall, 1991.

[KC86] S. Khoshafian and G. Copeland. Object Identity. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Portland, Oregon*, Oktober 1986.

[Kis92] E. Kisicki. Datenintensive Anwendungen. (talk given at University of Hamburg, Database and Information Systems Group), April 1992.

[KM92] F. Kirch and S. Müßig. Entwicklung eines generischen Datenbankbrowsers in einer polymorphen Programmiersprache. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, April 1992.

[KMP+83] J. Koch, M. Mall, P. Putfarken, M. Reimer, J.W. Schmidt, and C.A. Zehnder. Modula/R Report, Lilith Version. Technical report, Department Informatik, ETH Zürich, Switzerland, Februar 1983.

[Knu74] D.E. Knuth. Structured Programming with goto Statements. *ACM Computing Surveys*, 6(4):261–301, December 1974.

[Kob84] I. Kobayashi. Validating Database Updates. *Information Systems*, 9(1):1–17, 1984.

[Kre88] T. Krebs. Optimierung und Synchronisation von Integritätstests. Diplomarbeit, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt/Main, Germany, June 1988.

[LH90] B. Lindsay and L. Haas. Extensibility in the STARBURST Experimantal Database System. In A. Blaser, editor, *Proceedings of the International Symposium on Database Systems of the 90s, Müggelsee, Berlin, Federal Republik of Germany, November 5–7, 1990*, volume 466 of *Lecture Notes in Computer Science*, pages 217–248. A. Blaser, 1990.

[LR84] T.-W. Ling and P. Rajagopalan. A Method to Eliminate Avoidable Checking of Integrity Constraints. In *Proceedings of the Trends & Applications Conference, Gaithersburg, Maryland, May 23–24, 1984*, pages 60–69. The Institute of Electrical and Electronics Engineers, Inc., 1984.

[LR89] C. Lécluse and P. Richard. The O_2 Database Programming Language. In P.M.G. Apers and G. Widerhold, editors, *Proceedings of the Fifteenth Internaltional Conference on Very Large Databases, Amsterdam, The Netherlands, August 22–25, 1989*, pages 411–422. Morgan Kaufmann Publishers, 1989. (also as Rapport Technique 26-89, GIP Altair, France).

[LRV88] C. Lecluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, Illinois*, pages 424–433. acm, 1988. SIGMOD RECORD Volume 17, Number 3, September.

[Mar90] V.M. Markowitz. Referential Integrity Revisited: An Object-Oriented Perspective. In D. McLeod and H. Schek R. Sacks-Davis, editors, *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane, Austalia, August 13–16, 1990*, pages 578–589. Morgan Kaufmann Publishers, August 1990.

[Mat92] F. Matthes. *Generische Datenbankprogrammierung: Sprachliche und architektonische Grundlangen.* Dissertation zur Erlangung des Doktorgrades der Naturwissenschaften, Fachbereich Informatik, Universität Hamburg, Germany, August 1992.

[MD89] D.R. McCarthey and U. Dayal. The architecture of an active database management system. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 – June 2, 1989*, pages 215–224. Association for Computing Machinery, 1989. SIGMOD RECORD Volume 18, Number 2, June.

[Mey88] B. Meyer. *Object-oriented Software Construction*. International Series in Computer Science. Prentice Hall, 1988.

[MJAP86] D. Maier, Stein J., Otis A., and A. Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Portland, Oregon*, Oktober 1986.

[MN91] P. Münnix and C. Niederée. Charakterisierung datenintensiver Anwendungen: Anforderungen, Ziele, Formalismen. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, March 1991.

[Mor83] M. Morgenstern. Active databases as a paradigm for enhanced computing enviroments. In M. Schkolnick and C. Thanos, editors, *Proceedings of the Ninth International Conference on Very Large Databases, Florence, Italy, October 31 – November 2, 1983*, pages 34–42. Officine Grafiche della Pacini Editore PISA, 1983.

[Mos81] J.E.B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. Technical Report MIT/LCS/TR-260, Massachusetts Institute of Technology, April 1981.

[Mos87] J.E.B. Moss. Log-Based Recovery for Nested Transactions. In P.M. Stocker, W. Kent, and P. Hammersley, editors, *Proceedings of the Thirteenth International Conference on Very Large Databases, Brighton, England, 1987*, pages 427–432. Morgan Kaufmann Publishers, 1987.

[MS90] F. Matthes and J.W. Schmidt. The Type System of DBPL. In R. Hull, R. Morrison, and D. Stemple, editors, *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon, June 4–8, 1989*, pages 255–260. Morgan Kaufmann Publishers, 1990. (also appeared as FIDE Technical Report 91/20, Department of Computing Science, University of Glasgow).

[MS92] F. Matthes and J.W. Schmidt. Bulk Types: Built-In or Add-On? In *Proceedings of the Third International Workshop on Database Programming Languages, Naflplion, Greece, August 27–30, 1991*, pages 33–54. Morgan Kaufmann Publishers, 1992. (also appeared as FIDE Technical Report 91/20, Department of Computing Science, University of Glasgow).

[Mü91] Rainer Müller. Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und -implementierung. Diplomarbeit, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt/Main, Germany, November 1991.

[Nic82] J.-M. Nicolas. Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18:227–253, 1982.

[OBBT89] A. Ohori, P. Buneman, and V. Brezeau-Tannen. Database Programming in Machiavelli - a Polymorphic Language with Static Type Inference. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 – June 2, 1989*, pages 46–57. Association for Computing Machinery, 1989. SIGMOD RECORD Volume 18, Number 2, June.

[Qia88] X. Qian. An Effective Method for Integrity Constraint Simplification. In *Proceedings of the IEEE Fourth International Conference on Data Engineering, Los Angeles Airport Hilton and Towers, Los Angeles, California, February 1–5, 1988*, pages 338–345. Computer Science Press, 1988.

[Reu87] A. Reuter. Maßnahmen zur Wahrung von Sicherheits- und Integritätsbedingungen. In P.C. Lockemann and J.W Schmidt, editors, *Datenbank-Handbuch*, chapter 4. Springer-Verlag, 1987.

[Ric83] K. Rich. *Artificial Intelligence*. McGraw-Hill Book Company, 1983. (chapter 5).

[Rum87] J. Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. In N. Meyrowitz, editor, *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Orlando, Florida, October 4–8, 1987*, pages 466–481. Association for Computing Machinery, 1987. Special Issue of the SIGPLAN Notices Volume 22, Number 12, December.

[SBK+88] J.W. Schmidt, M. Bittner, H. Klein, H. Eckhardt, and F. Matthes. DBPL System: The Prototype and its Architecture. Esprit Project 892 WP/IMP 3.2, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, November 1988.

[Sch84] P.M. Schwarz. *Transactions on Typed Objects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pennsylvania, 1984.

[Sch87] J.W. Schmidt. Datenbankmodelle. In P.C. Lockemann and J.W Schmidt, editors, *Datenbank-Handbuch*, chapter 1. Springer-Verlag, 1987.

[SDDS86] J.T. Schwartz, R.B.K. Dewar, E. Dubinski, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[SEM88] J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 112-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, 1988.

[Sha90] P. Shaw. Database Language Standards. In A. Blaser, editor, *Proceedings of the International Symposium on Database Systems of the 90s, Müggelsee, Berlin, Federal Republik of Germany, November 5–7, 1990*, volume 466 of *Lecture Notes in Computer Science*, pages 55–80. Springer-Verlag, 1990.

[SQL87] ISO. *Standard ISO 9075, Information processing systems - Database langauge SQL*, 1987.

[SS77] J.M. Smith and D.C.P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, 2(2):105–133, Juni 1977.

[SSS+92a] K.-D. Schewe, J.W. Schmidt, D. Stemple, B. Thalheim, and I. Wetzel. Generating Methods to Assure Global Integrity. (submitted to IEEE Data Engineering 1993), September 1992.

[SSS92b] D. Stemple, R.B. Stanton, and T. Sheard. Linguistic Reflection: A Bridge from Programming to Database Languages. In *Proccedings of the 25th Hawaii International Conference on System Sciences*, volume II, pages 844–855, 1992.

[Weg87] P. Wegner. Dimensions of object-based language design. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Orlando, Florida*, 1987.

[Wei88] G. Weikum. *Transaktionen in Datenbanksystemen.* Addison-Wesley, 1988.

[WF90] J. Widom and S.J. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In H. Garcia-Molina and H.V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Managment of Data, Atlantic City, NJ, May 23–25, 1990*, pages 259–270. Association for Computing Machinery, 1990. SIGMOD RECORD, Volume 19, Issue 2, June 1990.

[Wir85] N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1985.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. The IRIS Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[WSK83] W. Weber, W. Stucky, and J. Karszt. Integrity Checking in Data Base Systems. *Information Systems*, 8(2):125–136, 1983.