Konzepte für eine persistente Programmiersprache

Jürgen Schlegelmilch

Universität Rostock
Lehrstuhl für Datenbank- und Informationssysteme
http://wwwdb.informatik.uni-rostock.de/~schlegel/

Datenbanken speichern Daten, und mit Programmiersprachen kann man Anwendungen schreiben, die diese Daten manipulieren. Die Datenmodelle und Verarbeitungsparadigma beider Gebiete weichen aber stark von einander ab; dieser Umstand wird „impedance mismatch“ genannt. Auch die Objektorientierung schließt diese Lücke nicht: Die Klassifikation von Objekten ist zu unflexibel fr die langfristige Speicherung, Konsistenzsicherung und Transaktionen werden kaum unterstützt und Optimierung nicht betrachtet.
In dieser Arbeit stellen wir eine Menge von Konzepten vor, die diese Probleme überwindet und dabei sowohl den Anforderungen aus dem Datenbankbereich nach Mengenorientierung, Konsistenz , Flexibilität und Transaktionen als auch denen für Programmiersprachen nach Einfachheit, Typsicherheit, Modularisierung und Wiederverwendbarkeit gerecht wird.

1 Motivation und Überblick

Datenbanken speichern Daten für viele Anwendungen und über lange Zeiträume. Oft weichen die Datenschemata der Anwendungen voneinander ab, und spätere Schemaanpassungen sind nicht auszuschließen. Zudem muss u. U. der Typ einzelner Datenelemente angepasst werden, um die Realität korrekt wiederzuspiegeln. Die langen Zeiträume führen auch zu großen Datenmengen, die effizient verarbeitet werden müssen; Optimierung ist also obligatorisch, und diese setzt Mengenorientierung voraus, um effektiv zu sein. Sehr wichtig ist auch die Konsistenz der Daten, denn eine inkonsistente Datenbank ist praktisch wertlos. Anders als normale Programme kann man Datenbanken im Falle von Inkonsistenzen nicht einfach neu starten. All diese Aspekte müssen beim Entwurf einer Datenbankprogrammiersprache berücksichtigt werden.
Programmiersprachen orientieren sich mehr an Kriterien des Software Engineering, also für das Schreiben fehlerfreier, wartbarer und effizienter Programme. Bei ihnen legt man deshalb Wert auf eine kleine Menge von Konzepten, Typsicherheit, Wiederverwendbarkeit und Modularisierung. Die jüngste Entwicklung sind objektorientierte Programmiersprachen (OOPLs). Sie erreichen Modularisierung durch Klassen, die Schnittstellen bilden und Implementierungsdetails kapseln. Wiederverwendbarkeit wird durch Vererbung und Substituierbarkeit unterstützt, oft beschränkt durch eine Klassenhierarchie. Vererbung ermöglicht die Verwendung von Funktionen in mehreren Klassen, während Substituierbarkeit (auch Polymorphismus genannt) die Anwendung einer Funktion auf Argumente verschiedener Typen erlaubt. Können Eigenschaften von Routinen überhaupt spezifiziert werden, so soll das die Verifikation unterstützen, nicht die Optimierung; letzere bleibt – abgesehen von low-level-Optimierungen – dem Programmierer überlassen. Zugunsten all-

gemeiner Konstrukte verzichten viele Programmiersprachen auf direkte Unterstützung von Mengen und ihren Operationen.
Objektorientierte persistente Programmiersprachen sollen die Brücke bilden zwischen beiden Gebieten. Es reicht aber nicht aus, einfach die Obermenge von Konzepten zu bilden. Diese müssen aufeinander abgestimmt sein, um handhabbar zu bleiben.
Im folgenden stellen wir nun eine solche Menge eng kooperierender Konzepte vor, die allen Anforderungen von beiden Seiten genügt und die wir für die Zwecke dieser Präsentation PLEX[1] nennen. Zunächst gibt Kapitel 2 einen Überblick über PLEX, bevor Kapitel 3 das Datenmodell und Kapitel 4 das Verhaltensmodell vorstellen. In Kapitel 5 betrachten wir Transaktionen, die Persistenzfestlegung und die Konsistenzsicherung. Kapitel 6 geht kurz auf andere Ansätze ein, und Kapitel 7 fasst die vorgestellten Konzepte zusammen.

2 Überblick über die Konzepte

PLEX kombiniert ein sehr flexibles Datenmodell mit einem Verhaltensmodell, das in drei Schichten unterteilt ist. Das Datenmodell weicht von den in OOPLs üblichen stark ab, weil es Objekte mehreren Klassen zuordnen kann. Klassen sind nicht Typbeschreibungen und Namensräume für Funktionen, sondern Funktionen, die Objekten Teilzustände zuordnen. Diese Zustände sind Werte von ADTs und kapseln also Daten und Operationen. Verbindungen zwischen Objekten werden durch Beziehungen hergestellt, die im wesentlichen Relationen sind. Die Attribute dieser Relationen können beliebige ADTs, aber auch Klassentypen haben. Attribute mit Klassentypen werden Rollen genannt und sind die einzigen Elemente im Datenmodell, die Objektidentitäten speichern können[2]. Klassen und Beziehungen können entweder in der Datenbank abgelegt oder als Sichten durch Anfragen berechnet werden.
Die Klassenhierarchie ist in PLEX eine Beziehung zwischen Klassen mit einer Teilmengenbedingung: Unterklassen enthalten eine Teilmenge der Objekte ihrer Oberklassen. Dies ermöglicht Substituierbarkeit, ohne Vererbung zu erzwingen. Vererbung wird lediglich für ADTs angeboten, um Einkapselung und Wiederverwendung zu ermöglichen.
Methoden kapseln die Verteilung des Gesamtzustandes eines Objekts gegenüber der Außenwelt, indem sie die Operationen der Teilzustände koordinieren. Sie werden unterteilt in Anfrage- und Änderungsmethoden.
Migratoren und Demigratoren ändern die Klassenzugehörigkeit von Objekten, indem sie die Zustandszuordnung in den Klassen anpassen. Weil die Demigration die Typsicherheit gefährdet, muss ein spezieller Typprüfungsalgorithmus potenzielle Gefahrenquellen erkennen und sichern. Eine optionale dynamische Typprüfung ermöglicht sichere Programmierung in Fällen. die der Algorithmus als gefährlich bewertet.
Transaktionen sind spezielle Funktionen, die einen Datenbankzustand in einen anderen Datenbankzustand abbilden. Sie werden durch sequenzielle Komposition gebildet aus ein-

[1] für Persistent Language for EXTREM, weil EXTREM [HFW90] anfangs als Datenmodell diente.
[2] abgesehen von den gespeicherten Zustandsfunktionen von Klassen

zelnen Transaktionsschritten wie zum Beispiel Aufrufen von Änderungsmethoden, auch geschachtelte Transaktionen sind möglich.

Die Persistenz von Objekten ist definiert durch die Beziehungen, in denen sie auftreten. Die Rollen einer Beziehung werden dazu unterteilt in wichtige und unwichtige. Ein Objekt ist persistent, solange es am Ende einer Transaktion eine wichtige Rolle hat.

Sowohl Klassen als auch Beziehungen können Konsistenzbedingungen haben. die am Ende einer Transaktion zusammen mit der Persistenz geprüft werden. Inkonsistenzen führen zum Rücksetzen der Transaktion, wenn keine Reparatur möglich und gewünscht ist.

3 Das Datenmodell

OOPLs verstehen Klassen oft als Implementierungen von ADTs [Mey92] und Objekte als deren Instanzen, die eher zufällig identifiziert werden können über den Speicherplatz, in dem ihre Daten liegen. In PLEX sind Objekte wie in semantischen Datenmodellen [HFW90] reine Identifikatoren, denen ein Zustand zugeordnet wird. Diese Sichtweise erlaubt es, Objekten mehr Informationen zuzuordnen als nur einen Datensatz, und ist damit die Grundlage für die Flexibilitt von PLEX. Klassen sind in dieser Sichtweise Mengen von Objekten mit gleichen Eigenschaften; das ermöglicht Sichtklassen. Das letzte Element des Datenmodells bilden Beziehungen, die Objekte in Relation zueinander setzen können. Sie sind der semantisch reichere Ersatz für die in OOPLs verwendeten Referenzen.

3.1 Typen und Typkonstruktoren

PLEX verwendet ein rein funktionales Typsystem höherer Ordnung mit beschränktem, parametrischem Polymorphismus. Damit können ADTs definiert werden, inklusive Typhierarchie und Mehrfachvererbung, für die Kapselung von Daten und deren Operationen. Da es aber keine Subsumptionsregel gibt, können ADT-Operationen effizient gebunden werden. In PLEX haben ADTs zusätzlich eine Menge von Axiomen, die die Semantik der Operationen beschreiben und als Grundlage für die Optimierung dienen können.

Das Typsystem ermöglicht auch die Darstellung von Monaden [Wad95]; ein Monad ist ein Datentyp mit einem Typparameter und zwei Operatoren mit bestimmten Eigenschaften, die im wesentlichen die Konstruktion von Elementen des Typs sowie die sequentielle Komposition erlauben. Praktisch alle wichtigen Typkonstruktoren lassen sich durch Monade beschreiben [Gru99], außerdem wichtige Programmiersprachenkonstrukte wie Ausnahmen (Exceptions) oder Ein-/Ausgabeoperationen [Wad95]. In PLEX werden Monade für Typkonstruktoren und Transaktionen eingesetzt.

3.2 Objekte und Klassen

Objekte sind reine Identifikatoren, also global eindeutige Werte ohne weitere Semantik, dargestellt durch einen Typ `OId`. Objekte können nicht unabhängig existieren, sondern müssen nach ihrer Erzeugung wenigstens einer Klasse zugeordnet werden; dies geschieht durch Migration (vgl. Abschnitt 4.2).

Klassen bestehen aus einer Funktion, die Objekten einen Wert eines ADT als lokalen Zustand zuordnet, und einer Menge von Methoden (siehe Abschnitt 4.1). Bei Basisklassen wird die Funktion als Menge von Zuordnungen in der Datenbank gespeichert, bei Sichtklassen durch eine Anfrage berechnet. Der Typ des Zielbereichs der Funktion wird Zustandstyp der Klasse genannt. Der Vorbereich der Funktion ist die Menge der Objekte in dieser Klasse und wird Extension genannt. Er ist eine variable Teilmenge des Typs `OId`, die durch Migration und Demigration verändert werden kann, und kann als Typ (sogenannter Klassentyp) verwendet werden. Die sich daraus ergebenden Probleme werden in Abschnitt 4.2 diskutiert.

Die Klassenhierarchie ist eine binäre Beziehung zwischen Klassen mit der Konsistenzbedingung, dass die Extension der Unterklasse eine Teilmenge der Extension ihrer Oberklasse ist; eine Klasse kann beliebig viele Oberklassen haben. Die Konsistenzbedingung stellt sicher, dass ein Objekt einer Unterklasse für ein Objekt einer Oberklasse substituiert werden kann — es *ist* schließlich selbst ein Objekt der Oberklasse. So wird Polymorphismus erreicht, ohne eine Subsumptionsregel für Klassen im Typsystem zu benötigen. Außerdem wird die Substituierbarkeit von einem Typprüfungs- zu einem Konsistenzproblem und kann damit flexibler gehandhabt werden.

Anders als in OOPLs ist die Zuordnung von Objekten zu Klassen also mehrdeutig: Objekte können in beliebig vielen Klassen einen Zustand haben. Es muss auch keine eindeutige speziellste Klasse eines Objekts geben. Dies trägt sehr zur Flexibilität des Datenmodells bei, ist aber auch eine Herausforderung für das Verhaltensmodell.

3.3 Beziehungen

Beziehungen beschreiben Zuordnungen von Objekten durch eine Relation. Die Attribute der Relation können beliebige Typen haben; Attribute mit Typ `OId` nennen wir Rollen, da sie Objekte aufnehmen können. Die Tupel der Relation heißen Verbindungen und ordnen die Objekte in den Rollen einander zu. Bei Basisbeziehungen wird die Relation in der Datenbank gespeichert, bei Sichtbeziehungen durch eine Anfrage berechnet; dabei sind auch Nestung und Entnestung möglich.

Beziehungen sind ungerichtet: alle Rollen einer Verbindung sind gleichberechtigt, und über die Verbindung kann man zu jedem der beteiligten Objekte navigieren.

Die Kontrolle von Referenzen ist ein wichtiger Punkt im Datenmodell von PLEX. Beziehungen sind der einzige Weg, Objekte zu verbinden, und Rollen der einzige Platz für Referenzen. Für Basisklassen und -beziehungen gibt es deshalb eine Beschränkung: Der Zustandstyp einer Basisklasse darf keinen Klassentyp enthalten, und in Basisbeziehungen muss ein Attribut entweder eine Rolle sein oder sein Typ darf ebenfalls keinen Klassentyp enthalten. Sichtklassen und -beziehungen sind von dieser Einschränkung nicht betroffen.
Diese klare Trennung bringt alle Referenzen unter die direkte Kontrolle des Datenbanksystems und ist die Grundlage für eine effiziente Persistenzfeststellung.

Beziehungsklassen sind Sichtklassen, die aus einer Beziehung abgeleitet werden. Dazu muss nur die Relation in eine Funktion transformiert werden, zum Beispiel durch Nestung bezüglich einer Rolle[3]; dabei können eventuell andere Rollen im Zustandstyp der Beziehungsklasse auftreten. Sie können dann zur direkten Navigation von einem Objekt zu einem anderen Objekt benutzt werden, wie das in OOPLs mit Referenzen möglich ist.

4 Das Verhaltensmodell

Das Verhaltensmodell von PLEX ist rein funktional, kennt also keine Variablen im Sinne prozeduraler Programmiersprachen. Der Datenbankzustand ist eine solche Variable und wird deshalb in einen Monad gekapselt, der Transaktionen beschreibt und in Abschnitt 5.1 vorgestellt wird. Als Auswertungsstrategie wird die verzögerte Auswertung angenommen, da sie ein großes Optimierungspotenzial bietet: Nur benötigte Werte werden auch berechnet, Berechnungen können parallel und ressourcen-orientiert durchgeführt werden.
Das Verhaltensmodell besteht aus drei Schichten. Die unterste Ebene bilden die Operationen von ADTs, die außer zur Darstellung der üblichen Basisdatentypen im wesentlichen als Zustandstypen von Klassen auftreten. In die zweite Schicht sind die Methoden und Migratoren von Klassen einzuordnen, die die Zustandsfunktionen manipulieren. Die oberste Schicht schließlich besteht aus normalen Funktionen und Transaktionen.

4.1 Methoden

Objekte gehören zu mehreren Klassen und haben in jeder dieser Klassen einen Teilzustand mit Operationen. Diese interne Struktur kann sich durch Migration und Demigration ändern und muss deshalb gekapselt werden. Dies ist die Aufgabe von Methoden, die also die Operationen der Teilzustände koordinieren müssen.
Methodenaufrufe werden als Ereignisse verstanden, die an ein Objekt gerichtet werden. Ein eingehendes Ereignis wird an alle Klassen delegiert, in deren Extension das Objekt ist, und löst dort die Berechnung eines Ergebnisses durch eine Operation aus. Die Gesamtreaktion des Objekts ist dann zunächst eine Menge aus Tupeln, jeweils bestehend aus

[3] Alle anderen Attribute müssen außerdem in einen ADT gepackt werden.

einer Klasse und dem dort berechneten Teilergebnis. Anfrage- und Änderungsmethoden verwendet diese Menge unterschiedlich.

Anfragemethoden aggregieren die Teilergebnisse zu einem Gesamtergebnis mit Hilfe einer frei definierbaren Aggregationsfunktion. Diese Funktion kann zum Einen klassische Aggregationen auf den Teilergebnissen durchführen, zum Beispiel Summenbildung oder Wahl des Maximums oder beliebige Kombinationen, zum Anderen aber auch die Klassen heranziehen, in denen die Teilergebnisse berechnet wurden. Dies steigert die Mächtigkeit des Aggregationsansatzes signifikant:

- Die Auswahl des Teilergebnisses einer fest vorgegebenen Klasse entspricht dem statischen Binden von OOPLs.

- Die Auswahl des Teilergebnisses der bzgl. der Klassenhierarchie kleinsten Klasse bildet das späte Binden von OOPLs nach.

Wie oben bereits erwähnt, muss ein Objekt keine eindeutige speziellste Klasse haben, die Teilergebnis-Menge hat deshalb nicht notwendigerweise ein bzgl. der Klassenhierarchie kleinstes Element; für die Simulation des späten Bindens muss es jedoch existieren, sonst ist das Ergebnis nicht eindeutig. Dieses Problem wird durch Schemaanreicherung gelöst.

Der Algorithmus dazu wird nach der Schemadefinition angewendet und untersucht alle Mengen von Klassen, in denen ein Objekt gleichzeitig sein kann. Hat eine solche Menge mehrere kleinste Klassen, die für dieselbe Methode unterschiedliche Operationen zur Berechnung des Teilergebnisses verwenden, dann liegt ein Konflikt vor. Der Algorithmus fügt dann eine Sichtklasse als direkte Unterklasse dieser kleinsten Klassen in die Klassenhierarchie ein und lässt den Programmierer in der Sichtklasse ebenfalls ein Ergebnis für die fragliche Methode berechnen. Die Sichtklasse wird definiert als Schnittmenge der Extensionen ihrer Oberklassen; Objekte, die in all diesen Klassen sind und deren Methodenergebnis deshalb nicht eindeutig war, sind so auch in der Extension der Sichtklasse. Als Unterklasse der in Konflikt stehenden Klassen liefert sie also das eindeutige Minimum in der Menge der Teilergebnisse; damit ist das Methodenergebnis nun eindeutig.

Änderungsmethoden verwenden die Teilergebnis-Menge zur Änderung der Zustandsfunktionen. In jeder Klasse, die ein Teilergebnis liefert, wird dieses dem Objekt als neuer Zustand in dieser Klasse zugeordnet. Alle Berechnungen und danach auch die Änderungen können vollständig parallelisiert werden, da es keine Abhängigkeiten zwischen den Zustandsfunktionen gibt; sie dürfen ja keine Referenzen auf andere Objekte enthalten.
Sowohl bei Anfrage- als auch bei Änderungsmethoden werden alle Vorgänge durch je eine Methode der Metaebene koordiniert, die der Programmierer bei der Definition solcher Methoden verwenden muss, da die Zustandsfunktion selbst in einen ADT gekapselt ist. Typregeln stellen sicher, dass die Teilergebnisse jeweils die richtigen Typen haben (einen einheitlichen Typ für Anfrage- bzw. den jeweiligen Zustandstyp für Änderungsmethoden).

4.2 Migratoren und Demigratoren

Migratoren sind Funktionen, die die Zustandsfunktion einer Klasse um neue Zuordnungen erweitern. Dazu werden sie mit einem Objekt und einem Wert aufgerufen, der dem Objekt dann als Zustand in der Klasse zugeordnet wird. Demigratoren hingegen entfernen eine Zuordnung aus der Zustandsfunktion einer Klasse; weil dazu kein Argument notwendig ist, können sie implizit aufgerufen werden, zum Beispiel während der Persistenzfeststellung oder der Konsistenzsicherung (siehe Abschnitte 5.2 und 5.3).

Wie Methoden müssen auch Migratoren und Demigratoren mit Hilfe von je einer Methode der Metaebene definiert werden, die die Einhaltung der Teilmengenbedingung der Klassenhierarchie sicherstellen.

Typsicherheit Durch die Demigration eines Objekts aus einer Klasse C kann es zu Typfehlern kommen, wenn nachfolgend Methoden von C oder einer Unterklasse für das Objekt aufgerufen werden; dies muss also verhindert werden. Die Demigration kann explizit erfolgt sein, aber auch implizit, etwa bei einer Sicht in Folge einer Zustandsänderung.

In [Sch99] wird eine Lösung dieses Problems vorgestellt, die auf zwei Teillösungen basiert. Aufrufe von Methoden einer Klasse C können nur an Bezeichner gerichtet werden, deren deklarierter Typ C oder eine Unterklasse davon ist. In PLEX gibt es nur zwei Arten solcher Bezeichner:

- Rollen in Beziehungen sind Attribute mit dem Typ `OId` und der Konsistenzbedingung, dass der Wert in der Extension von C enthalten ist; eine Demigration kann also Verbindungen des Objekts inkonsistent machen, diese werden dann am Transaktionsende während der Konsistenzsicherung (siehe Abschnitt 5.3) gelöscht.

- Argumente von Funktionen können ebenfalls einen Klassentyp haben. Demigriert ein Objekt im Gültigkeitsbereich des Bezeichners aus der Klasse C, so darf der Bezeichner nicht mehr verwendet werden. Alle Funktionen, die – direkt oder indirekt – ein Objekt aus C demigrieren können, werden dazu in der Menge $critical_C$ zusammengefasst. Wenn nun eine Funktion mit Argument des Typs C eine Funktion aus $critical_C$ verwendet, hat das Argument danach den Typ C nicht mehr. Ein Typprüfungsalgorithmus erkennt diese Situation und verbietet alle nachfolgenden Methodenaufrufe, die ein Objekt der Klasse C voraussetzen. Der Algorithmus macht eine konservative Abschätzung, denn das Argument-Objekt muss nicht mit dem demigrierten identisch sein. Eine dynamische Typprüfung ermglicht dann dennoch Aufrufe von Methoden jeder gewünschten Klasse.

Diese Kombination erlaubt die freie Verwendung von Sichten und Klassen mit Konsistenzbedingungen im Datenbankschema; nur innerhalb von Funktionen unterliegen sie den oben geschilderten Einschränkungen.

4.3 Funktionsmethoden

Die oben beschriebenen, mit Meta-Methoden definierten Methoden einer Klasse nennen wir elementar. Daneben können andere Methoden als normale Funktionen definiert werden, die letztlich auf die elementaren Methoden zurückgreifen; solche Methoden nennen wir Funktionsmethoden. Sie können ohne Nachteile auch als freie Funktionen, also ohne Zuordnung zu einer Klasse, definiert werden, da sie nicht auf die Zustandsfunktion der Klasse zugreifen. Ihre Aufgabe ist die Koordination mehrerer Methodenaufrufe, für ein oder mehrere Objekte. Sie sind außerdem der einzige Weg, rekursive Methoden in PLEX zu definieren.

5 Transaktionen, Persistenz und Konsistenz

5.1 Transaktionen

Transaktionen sind spezielle Funktionen, die einen Datenbankzustand in einen anderen Datenbankzustand abbilden und so Änderungen modellieren. Ihr Typ wird mit einem Monad Tx beschrieben, der eine Kombination der Monads „State Transformer“ und „Exception“ aus [Wad95] ist, mit denen hier die Datenbankzustandsänderung bzw. das Auftreten eines Transaktionsabbruchs modelliert werden.
Die beiden Operatoren des Monads Tx dienen dem Bilden elementarer Transaktionsschritte und der sequentiellen Komposition dieser Schritte zu größeren Schritten. In PLEX sind alle Änderungsmethoden, Migratoren und Demigratoren sowie das Erzeugen und Löschen von Objekten Transaktionsschritte; außerdem können Subtransaktionen gebildet werden. Die Funktionen *begin*, *abort*, *commit* und *rollback* steuern Transaktionen wie gewohnt.
Der Kompositionsoperator von Tx führt implizit den aktuellen Datenbankzustand mit sowie ausreichend Information für das Rücksetzen[4]. Er verknüpft zwei Transaktionsschritte, indem er den ersten Schritt auf die Datenbank anwendet und prüft, ob es zu einem Transaktionsabbruch kam. Wenn ja, werden die bisherigen Änderungen zurückgesetzt, andernfalls der zweite Transaktionsschritt auf die Datenbank angewendet. So werden Änderungen automatisch serialisiert, und es kommt nicht zu Konflikten beim Schreiben.
Am Ende einer Transaktion wird immer die Persistenzfeststellung und die Konsistenzsicherung aufgerufen; auch bei diesen kann es noch zum Transaktionsabbruch kommen.

5.2 Die Persistenzfeststellung

Die Persistenzfeststellung geprüft, welche Objekte aus der Datenbank entfernt werden können. Die Prüfung kann nicht den Anwendungen überlassen werden, weil diese sich

[4] oder den Datenbankzustand vom Anfang der Transaktion sowie Informationen über die Änderungen

untereinander nicht notwendigerweise koordinieren. PLEX verwendet deshalb rollenbasierte Persistenz [Sch96], um die Lebensdauer von Objekten anwendungsunabhängig zu bestimmen. Dazu werden einige Rollen von Beziehungen als wichtig ausgezeichnet. Ein Objekt wird dann transient, wenn es keine wichtige Rolle in irgendeiner Beziehung hat; transiente Objekte werden am Transaktionsende gelöscht.

Beim Löschen eines Objekts werden automatisch auch alle Verbindungen gelöscht, die es zu anderen Objekten hatte. Es werden also Tupel aus den Relationen von Beziehungen gelöscht, die neben dem gelöschten Objekt in anderen Rollen auch weitere Objekte enthalten können. Dadurch können diese Objekte ihre letzte wichtige Rolle verlieren und müssen ebenfalls gelöscht werden. Der Prozess wird so lange wiederholt, bis alle Objekte noch wenigstens eine wichtige Rolle haben.

Da auch Sichtbeziehungen wichtige Rollen haben können, kann praktisch jedes als Anfrage formulierbare Kriterium für die Persistenzentscheidung genutzt werden. [Sch96] zeigt, dass rollenbasierte Persistenz jedes andere Persistenzkonzept simulieren kann.

5.3 Die Konsistenzsicherung

Basisklassen und Basisbeziehungen haben Konsistenzbedingungen, die am Transaktionsende geprüft werden. Dabei werden lokale und globale Bedingungen unterschieden. Lokale Bedingungen richten sich an eine Menge und können erfüllt werden, indem inkonsistente Elemente aus der Menge entfernt werden; bei Klassen bedeutet das die Demigration des Objekts, bei Beziehungen das Auftrennen der Verbindung. Inkonsistenzen wegen globaler Konsistenzbedingungen können so nicht repariert werden und führen deshalb immer zum Transaktionabbruch, während für lokale Bedingungen der Programmierer pro Klasse bzw. Beziehung wählen kann, ob eine Reparatur oder ein Transaktionsabbruch erfolgen soll.

Die Reparatur einer lokalen Inkonsistenz kann andere Objekte inkonsistent machen. Außerdem spielt die Reihenfolge eine Rolle, in der Klassen und Beziehungen geprüft werden; es gibt jedoch keine optimale Reihenfolge. Deshalb wird nach jeder Reparatur erneut die Konsistenz der Datenbank geprüft, bis entweder die Datenbank vollständig konsistent oder die Transaktion abgebrochen worden ist. Vor jeder Wiederholung der Konsistenzsicherung wird die Persistenzfeststellung durchgeführt, damit inkonsistente, aber transiente Objekte nicht unnötig zu Transaktionsabbrüchen führen.

6 Andere Ansätze

Datenbankmodelle wie Iris [FBC+90] sind PLEX sehr ähnlich, es fehlen aber Methoden zur Kapselung des verteilten Objektzustands. [SN88] verwendet aggregierende Methoden in föderierten Datenbanken, um verteilte Objektzustände zusammenzufassen. Normale OOPLs unterstützen mehrfache Klassenzugehörigkeit nur mit Rollenmodellen, z. B. [RS91]; Rollen sind hier Objekte, die die Teilzustände darstellen. Methoden sind hier im-

mer rollenspezifisch, spätes Binden wird nur mit starken Einschränkungen unterstützt. Sichten gibt es in OOPLs nur als Projektion in Form von Oberklassen . Viele Datenbankprogrammiersprachen wie Fibonacci [AGO95] unterstützen Sichten nicht als Klassen, sondern nur als Objektmengen; man kann also keine Methoden in Sichten definieren. Migration wird wegen der Typsicherheit meist nicht unterstützt. Fibonacci [AGO95] verbietet Demigration, BCOOL [Laa94] setzt ungültig gewordene Referenzen auf einen Nullwert.

7 Zusammenfassung

PLEX ist eine persistente Programmiersprache, die Anforderungen sowohl von Datenbanken als auch von Programmiersprachen erfüllt. Sie bietet Mengenorientierung im Datenmodell durch geeignete Darstellung der Typkonstruktoren wie auch im Verhaltensmodell durch Ausnutzung impliziter Parallelität. Die Konsistenzsicherung ist im Datenmodell verankert durch Konsistenzbedingungen in Klassen und Beziehungen und wird ergänzt durch Transaktionen im Verhaltensmodell. Das Datenmodell bietet große Flexibilität durch die Unterstützung von Objekten mit mehrfacher und wechselnder Klassenzugehörigkeit sowie Sichten; dazu passen die durch den Aggregationsansatz ebenso flexiblen Methoden und der Typprüfungsalgorithmus für die Migration. Rollenbasierte Persistenz erlaubt die anwendungsunabhängige Persistenzfeststellung. Modularisierung wird durch die Unterstützung von ADTs erreicht, Wiederverwendbarkeit durch parametrischen Polymorphismus und die Klassenhierarchie sowie Vererbung auf der Ebene der ADTs. Trotz dieser Fähigkeiten kommt PLEX mit rein funktionaler Grundlage und wenigen Konzepten aus.

Literaturverzeichnis

[AGO95] Albano, A.; Ghelli, G.; Orsini, R.: Fibonacci: A Programming Language for Object Databases. In The VLDB Journal, Bd. 4 (3):(1995), S. 403–444.

[FBC+90] Fishman, D. H.; Beech, D.; Cate, H. P.; Chow, E. C.; et al.: Iris: An Object-Oriented Database Management System. In Readings in Object-Oriented Database Systems, Morgan Kaufmann Publishers, San Mateo, CA, 1990, S. 216–226.

[Gru99] Grust, T.: Comprehending Queries. Dissertation, Universität Konstanz, Fakultät für Mathematik und Informatik, 1999.

[HFW90] Heuer, A.; Fuchs, J.; Wiebking, U.: OSCAR: An object-oriented database system with a nested relational kernel. In Proc. of the 9th Int. Conf. on Entity-Relationship Approach, Lausanne. Elsevier Science Publishers, 1990, S. 95–110.

[Laa94] Laasch, C.: Deskriptive Sprachen für Objekt-Datenbanken. Dissertation, Universität Ulm, Fakultät Informatik, 1994.

[Mey92] Meyer, B.: Eiffel: The Language. Prentice Hall, New York, NY, 1992.

[RS91] Richardson, J.; Schwarz, P.: Aspects: Extending Objects to Support Multiple, Independent Roles. In Proc. ACM SIGMOD Conference on Management of Data. ACM Press, 1991, Bd. 20 von ACM SIGMOD Record, S. 298–307.

[Sch96] Schlegelmilch, J.: Conflict Resolution using Derived Classes. In Proceedings of the 3rd International Conference on Object-Oriented Information Systems (OOIS'96). Springer-Verlag, Berlin, 1996, S. 267–279.

[Sch99] Schlegelmilch, J.: Typesafe Dynamic Classification. Preprint CS-04-99, Universität Rostock, Fachbereich Informatik, 1999.

[SN88] Schrefl, M.; Neuhold, E. J.: Object Class Definition by Generalization Using Upward Inheritance. In Proceedings of the Fourth International Conference on Data Engineering (ICDE'88). IEEE Computer Society, 1988, S. 4–13.

[Wad95] Wadler, P.: Monads for functional programming. In Advanced Functional Programming, Proceedings of the Båstad Spring School. Springer-Verlag, 1995, Nr. 925 in LNCS, S. 44–56.

Jürgen Schlegelmilch, geboren am 30. April 1967 in Osterode am Harz, Abitur 1986, Diplom in Informatik 1992, danach wissenschaftlicher Mitarbeiter an der Technischen Universität Clausthal, ab 1.4.1994 an der Universität Rostock. Seit dem 1.1.2000 Mitarbeiter am OFFIS e. V., Oldenburg.
Seine Forschungsinteressen liegen im Bereich der Informationssysteme, speziell der Datenbanken und Internet-Informationssysteme und ihrer Erstellung. Das Spektrum reicht von relationalen, post-relationalen und objektorientierten Datenbanken über Data Warehousing, datenbankgestützte Informationssysteme und Application Server auf Basis der Java 2-Plattform bis hin zu objektorientierten Programmiersprachen und Software-Entwicklungsprozessen.