ZRI der Universität Kaiserslautern, Bericht Nr. 5/89, 1989

Verarbeitung komplexer DB-Objekte in Ingenieuranwendungen

Christoph Hübel, Bernd Sutter
Universität Kaiserslautern
Erwin-Schrödingerstraße
6750 Kaiserslautern

Überblick

Nicht-Standard-Datenbanksysteme sollen den spezifischen Datenverwaltungsanforderungen speziell aus dem Bereich der Ingenieuranwendungen gerecht werden. Hierzu wurden in den vergangenen Jahren zahlreiche Ideen und Konzepte entwickelt, die zum einen die Architektur solcher Systeme, zum anderen deren Datenmodellierungsmächtigkeit betrafen. Um jedoch eine effiziente und anwendungsmoderate Bearbeitung der "nun" spezifizierbaren komplexen Objekte zu unterstützen, erscheinen angepaßte Verarbeitungskonzepte erforderlich.

Ausgehend von dem Prototyp eines Nicht-Standard-Datenbanksystems beschreiben wir ein Verarbeitungsschema für komplex-strukturierte Datenbank-Objekte und dessen Integration in die Umgebung konventioneller Programmiersprachen. Zur besseren Einordnung unseres Ansatzes erfolgt eine Abgrenzung mit konventionellen und alternativen Lösungsmöglichkeiten.

Abstract

Non-standard database systems should especialliy satisfy the specific data management requirements, which determine the field of engineering applications. Many ideas and concepts have been developed, concerning architecture as well as data modelling facilities for such new database systems during the last years. Nevertheless, adequate processing concepts seem to be additionally necessary in order to support efficient and application tailored processing of the complex objects, which can be described now by means of the new data models.

Starting from the prototype of a non-standard database system, we explain a processing scheme for complex-structured database objects as well as its integration in the environment of a conventional programming language. In order to valuate our own approach, we describe and classify conventional and alternative DB processing concepts.

1. Einleitung

Computer-Anwendungen auf dem Gebiet der Ingenieursysteme und die damit verbundenen Anforderungen an die Datentechnik entwickelten sich in den vergangenen Jahren zu einer der wichtigsten Herausforderungen auf dem Gebiet der Datenbankforschung [HR85, Lo85]. Bereits frühzeitig wurden die in verfügbaren Datenbanksystemen (DBS) angebotenen Konzepte als unzureichend erkannt. Ihr praktischer Einsatz resultierte in einem insgesamt sehr schlechten, nicht zu tolerierenden Leistungsverhalten.

Einen dominierenden Anteil an dieser Situation besitzt die sog. *"Modellierungsproblematik"*. Die komplex-strukturierten Objekte in den Ingenieuranwendungen können nur unzureichend und mit großem Aufwand auf die elementaren Datenstrukturen konventioneller Datenmodelle abgebildet werden. Eine ganzheitliche Handhabung der ein Anwendungsobjekt beschreibenden DB-Strukturen ist meist unmöglich. Zur Lösung dieser Modellierungsproblematik wurden u.a. eine Reihe von neueren Datenmodellen entwickelt, bzw. bestehende verfeinert und in ersten Prototypsystemen realisiert [LK84, SS86, PA86, Hä88].

Die recht einseitige Unterstützung der satzbezogenen Verarbeitung durch konventionelle DBS führt bei der Programmierung DB-gestützter Ingenieuranwendungen zu einer *"Verarbeitungsproblematik"*. So sind z.B. netzwerkorientierte DBS an ihrer Programmierschnittstelle durch eine "one-record-at-a-time"-Logik geprägt. Ebenso erlauben relationale Systeme lediglich die sukzessive Bearbeitung einer homogenen Tupel-Menge über einen darauf definierten Cursor [Co78, Da82]. Erforderlich sind dagegen Konzepte, die eine Bearbeitung der im Ingenieurbereich meist heterogen-und komplex-strukturierten "Verarbeitungsgegenstände" in einfacher und direkter Weise unterstützen. So konnte z.B. allein durch den Übergang von einer satz- zu einer strukturbezogenen Verarbeitungsphilosophie an der Datenhaltungsschnittstelle eines DB-gestützten 3D-Bauteilmodellierers eine signifikante Verbesserung des Leistungsverhaltens erzielt werden [HP89].

Als Antwort auf diese Probleme wird allgemein die Realisierung von sog. Nicht-Standard-Datenbanksystemen (NDBS) vorgeschlagen. Als eine aussichtsreiche Systemarchitektur für solche NDBS wird vielerorts [HR85, Lo85, PSSWD87] die NDBS-Kern-Architektur diskutiert (Abb. 1). Die Vorteile dieses zweigeteilten Architekturansatzes sind vor allem darin zu sehen, daß einerseits durch die Modellabbildung (MA) eine anwendungsbezogene Schnittstelle (Anwendungsmodell-Schnittstelle) mit den von der jeweiligen Anwendungsklasse benötigten Objekten und Operationen bereitgestellt werden kann und andererseits alle geeigneten, allgemein verwendbaren Darstellungs- und Zugriffstechniken sich im NDBS-Kern vereinigen und effizient implementieren lassen. Die Aufgabe der Systemkomponente MA besteht nun darin, ein konkretes Anwendungsmodell auf ein allgemeines, der Komplexität der Anwendungsob-

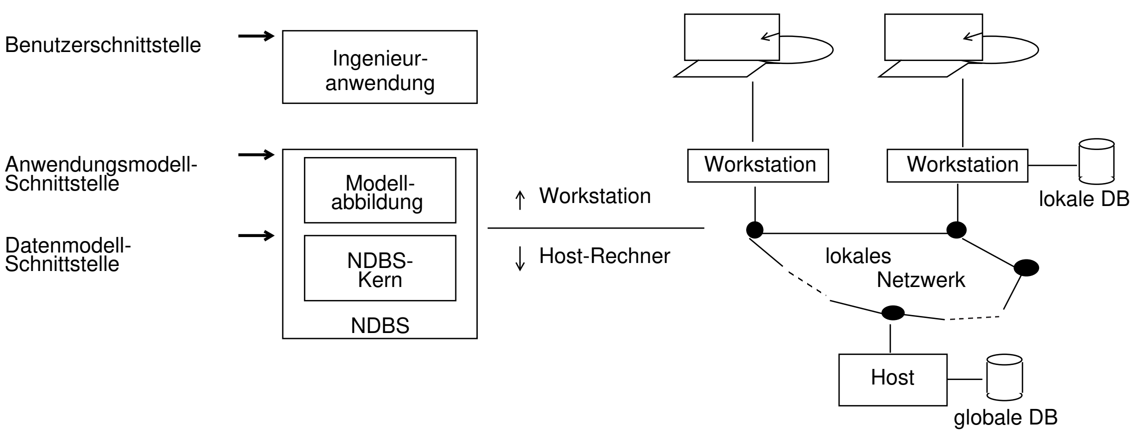

Abb. 1: Gesamtarchitektur eines NDBS-basierten Ingenieursystems

jekte angepaßtes Datenmodell des Kerns abzubilden. Eine Erweiterung von bestehenden bzw. ein Übergang zu neuen Anwendungsklassen ist durch die Erweiterung bzw. den Austausch der MA recht einfach zu erreichen.

In Abb. 1 ist neben der NDBS-Kern-Architektur eine mögliche Abbildung auf eine workstation-orientierte Hardware-Umgebung aufgezeigt. Aufgrund der Homogenität zwischen Hardware- und Software-Struktur sind die durch rechnerübergreifende Kommunikation zu erwartenden "Reibungsverluste" relativ gering. Dies gilt insbesondere, wenn es gelingt, die verarbeitungsinhärente Lokalität bzgl. der Datenreferenzen auf Workstation-Seite, also innerhalb der Modellabbildung, zu erhalten und auszunutzen. Die Lokalität und damit verbunden die Autonomie der einzelnen Systemkomponenten wird zusätzlich durch eine lokale Datenhaltung (lokale DB) unterstützt, die neben der globalen, von einem Host-Rechner kontrollierten Datenbank zum Einsatz kommt.

Die allgemeinen Konzepte sowie die konkrete Ausgestaltung der für die Systemleistung so relevanten Schnittstelle zwischen NDBS-Kern und Modellabbildung bestimmen den Gegenstand der vorliegenden Arbeit. Insbesondere werden Aspekte der Anbindung komplexer Strukturen an die Komponenten der Modellabbildung und, damit verbunden, deren Einbettung in die Laufzeitumgebung einer konventioneller Programmiersprache diskutiert.

2. Aspekte der DB-Verarbeitung

Mit zunehmender Komplexität der Algorithmen, deren Datenversorgung durch ein DBS zu realisieren ist, erscheint eine möglichst optimale Anpassung der DBS-Anwendungs-Programmierschnittstelle an die äußeren Gegebenheiten dringlicher denn je. Berücksichtigt man, daß in den hier zur Diskussion stehenden Anwendungsbereichen realistische Zugriffshäufigkeiten in einer Größenordnung von 10^6 Datenreferenzen pro Anwendungsoperation auftreten (z.B. bei typischen Operationen aus dem CAD-Bereich [Hä87]) und daß gerade im Entwurfsbereich die Toleranzgrenze bzgl. der Antwortzeit im Sekundenbereich liegt, so wird die gesamte Verarbeitungsproblematik bewußt.

Interne versus externe DB-Verarbeitung

Zunächst lassen sich zwei prinzipiell verschiedene Ansätze der DB-Verarbeitung unterscheiden. Im ersten Fall geschieht die eigentliche Verarbeitung "außerhalb" des DBS im Anwendungsprogramm. Hierbei übernimmt das DBS vorrangig die Aufgabe der Datenversorgung, d.h. also, es stellt die zu verarbeitenden Daten bereit und ist verantwortlich für das Einbringen (Propagieren) der vom Anwendungsprogramm durchgeführten Änderungen in den Datenbestand des DBS. Im zweiten Fall führt das Anwendungsprogramm selbst keinerlei Verarbeitung durch, sondern spezifiziert lediglich die Art der gewünschten Datenänderung, die dann schließlich "innerhalb" des DBS ausgeführt wird. Dieser Ansatz erscheint aus Sicht der Anwendung wesentlich angenehmer; allerdings ist zu berücksichtigen, daß verfügbare DBS i.a. nur eine geringe Mächtigkeit (einfache Ausdrücke) bei der Beschreibung der auszuführenden Modifikationen aufweisen. Neuere Vorschläge versprechen hier eine gewisse Verbesserung, indem sie das Einbringen von benutzer-definierten Operationen in das DBS erlauben [St87]. Aus Sicht der DBS stellt dieser Ansatz allerdings lediglich eine Verlagerung der Problematik dar: müssen doch auch die innerhalb des DBS abgearbeiteten Operationen mit den entsprechenden Daten versorgt werden. Daher wollen wir uns im folgenden hauptsächlich mit der Frage beschäftigen, wie eine effiziente Datenversorgung anwendungsspezifischer Algorithmen, die innerhalb bzw. außerhalb des DBS abgearbeitet werden, erreicht werden kann.

Allgemeines Verarbeitungsszenario

Der folgende Ablaufplan dient zur Verdeutlichung der prinzipiellen Vorgänge bei einer Datenanforderung durch ein Anwendungsprogramm (ähnliche Sequenzen treten jedoch auch bei Einfüge-, Lösch- und Änderungsoperationen auf; es ändert sich meist lediglich die Richtung des Datenflusses und die Reihenfolge der einzelnen Aktionen).

(1) Das Anwendungsprogramm setzt eine Datenanforderung an das DB-Laufzeitsystem ab.

(2) Die logische Datenanforderung wird umgesetzt in einen "seiten-bezogenen" Zugriff auf den DB-Systempuffer; gegebenenfalls wird der physische Transport entsprechender Seiten vom externen in den internen Speicherbereich des DBS abgewickelt.

(3) Die Daten im Systempuffer werden durch weitere Qualifikation und Projektion soweit verdichtet und in einem DB-Übergabebereich aufbereitet, daß sie den Anforderungen des Anwendungsprogramms genügen.

(4) Über spezielle Abrufanweisungen steuert das Anwendungsprogramm die konkrete Bindung dieser Daten in die Anwendungsumgebung. Hierbei wird i.a. ein Transport vom DB- in einen AP-Übergabebereich durchgeführt.

(5) Das Anwendungsprogramm verarbeitet die bereitgestellten Daten direkt im AP-Übergabebereich.

Im folgenden sollen nun unterschiedliche Ausgestaltungen dieses allgemeinen Verarbeitungsszenarios anhand konkreter DBS (bzw. der Datenmodelle, die sie realisieren) gegenübergestellt werden.

DB-Verarbeitung auf der Grundlage konventioneller Datenmodelle

Betrachtet man DBS mit netzwerk-orientierten Datenmodellen, so kann man feststellen, daß hier meist die DB- und AP-Übergabebereiche zusammenfallen. Dies liegt darin begründet, daß die DB-Datenstrukturen, die an der Datenmodell-Schnittstelle von Bedeutung sind, relativ einfach und homogen auf die Datenstrukturen des Anwendungsprogramms zu übertragen sind (z.B.: Record-Strukturen eines Netzwerk-Schemas können sehr einfach auf korrespondierende Strukturen in Cobol, PL/1 etc. umgesetzt werden). Etwas schwieriger gestaltet sich die Übergabe, wenn an der Datenmodell-Schnittstelle eine Treffermenge, z.B. durch Angabe nicht-eindeutiger Qualifikationsbedingungen, entsteht. In diesem Fall wird im DB-Übergabebereich die gesamte Treffermenge dargestellt. Die auf Anforderung sukzessive Bereitstellung der einzelnen Treffer-Records ist dann Aufgabe einer speziellen Bereitstellungskomponente, die die Verbindung zwischen DBS und Anwendung, d.h. die Einbettung von DB-Strukturen und DB-Operationen in die Anwendungsprogramme, realisiert.

Relational geprägte DBS sind durch den weitestgehend deskriptiven und mengenorientierten Charakter ihrer Datenmodell-Schnittstelle bestimmt. Die DB-Datenstrukturen sind Relationen, also Mengen homogen-strukturierter Tupel, die wiederum sukzessive (tupelweise) über den AP-Übergabebereich an die Anwendungsprogramme gebunden werden. Trotz mengenorientierter Datenmodell-Schnittstelle sind die Anwendungsprogramme i.a. auf eine satzorientierte Verarbeitung festgelegt. Allerdings besteht in einfachen Fällen die Möglichkeit, mengenbezogene Modifikationen durch das DBS selbst durchführen zulassen (z.B.: erhöhe das Gehalt aller Angestellten um 5 %). Diese Fähigkeit relationaler Systeme kann jedoch in Anwendungen aus dem Ingenieurbereich nur unzureichend eingesetzt werden, da hier häufig stark unterschiedliche und algorithmisch anspruchsvolle Änderungen auf meist heterogen-strukturierten Elementen notwendig sind.

Neuere Ansätze der DB-Verarbeitung

Eine gemeinsame Schwachstelle der Verarbeitungscharakteristik von netzwerk- und von relational-orientierten DBS ist darin zu sehen, daß es nur mit erheblichem Zusatzaufwand seitens der Anwendungsprogramme möglich ist, auf mehrere Elemente einer Treffermenge quasi "gleichzeitig" zuzugreifen, also eine Art Mehr-Tupel-Zugriff zu realisieren. Das in [SR84] vorgeschlagene Konzept der "Portals" zielt auf eine Entschärfung dieser Problemstellung ab, in dem den Anwendungsprogrammen ein Direktzugriff auf eine größere Anzahl von Trefferelementen ermöglicht wird. Ein wesentlicher Vorteil dieses Ansatzes liegt in der effizienten Nutzung der oftmals verarbeitungsinhärenten Lokalität bzgl. der Datenreferenzen: Bei wiederholten Zugriffen ist nur selten ein nochmaliges Bereitstellen im AP-Übergabebereich oder gar ein Zugriff auf den DB-Systempuffer bzw. den Externspeicher erforderlich. Die Beschränkung auf homogene Trefferelemente muß aus Sicht der Ingenieuranwendung allerdings als Nachteil angesehen werden.

Ein anderer Problemkreis wird angesprochen, wenn man sich den neueren Datenmodellen zuwendet, die eine Beschreibung und eine ganzheitliche Handhabung von komplexen Anwendungsobjekten ermöglichen. Hier wird von Anwendungsseite ein unmittelbarer Zugriff auf alle Komponenten der modellierten Datenstrukturen gefordert. Um dies zu erreichen sind Konzepte notwendig, die es erlauben, komplexe Datenstrukturen in ihrer Gesamtheit dem Anwendungsprogramm bereitzustellen bzw. von diesem entgegenzunehmen, also Konzepte, die eine strukturbezogene Verarbeitung ermöglichen. Der Vorgang des Herauslösens aus bzw. des Einbringens in den Datenbestand des DBS wird oftmals als *Check-out* bzw. *Check-in* bezeichnet [LP83].

In [KDG87] wird ein Verarbeitungsschema für ein NF^2-orientiertes Datenbanksystem [SS86, Da86] vorgestellt. Die Datenmodell-Schnittstelle und damit der DB-Übergabebereich ist hierbei durch nicht-normalisierte Relationen bestimmt, also durch Mengen, deren Elemente aus Tupeln mit relationen-wertigen Attributen bestehen (NF^2-Tupel). Der Datentransfer vom DB- zum AP-Übergabebereich erfolgt in zwei Schritten (vgl. Abb. 2a):

(1) Herauslösen eines Elementes aus der Ergebnismenge, d.h. Transport der entsprechenden Daten aus dem DB-Übergabebereich in einen sog. "Objektpuffer".

(2) Durch einen Cursor-bezogenen Zugriffsmechanismus [EW87] können dann Teile des im Objektpuffer angelegten NF^2-Tupels an den AP-Übergabebereich, der sich im vorgegebenen Fall aus einer Reihe von Programmvariablen zusammensetzt, gebunden werden.

Der Vorteil dieses Ansatzes ist darin begründet, daß zum einen über den Cursor-Mechanismus ein wiederholender Zugriff auf Teile des NF^2-Tupels möglich ist und zum anderen die Lokalität dieser Zugriffe im Objektpuffer genutzt werden kann. Ein Nachteil besteht in der Zweistufigkeit des Bindungsvorganges und dem damit verbundenen Datentransportaufwand. In neueren Ansätzen wird daher eine einstufige Lösung diskutiert, die eine Zusammenfassung von Objektpuffer und AP-Übergabebereich vorsieht. Dies geschieht dadurch, daß der AP-Übergabebereich von einfach-strukturierten Programmvariablen hin zu einer komplexen Datenstruktur erweitert wird, die unmittelbar ein entsprechend angepaßtes NF^2-Tupel aufneh-

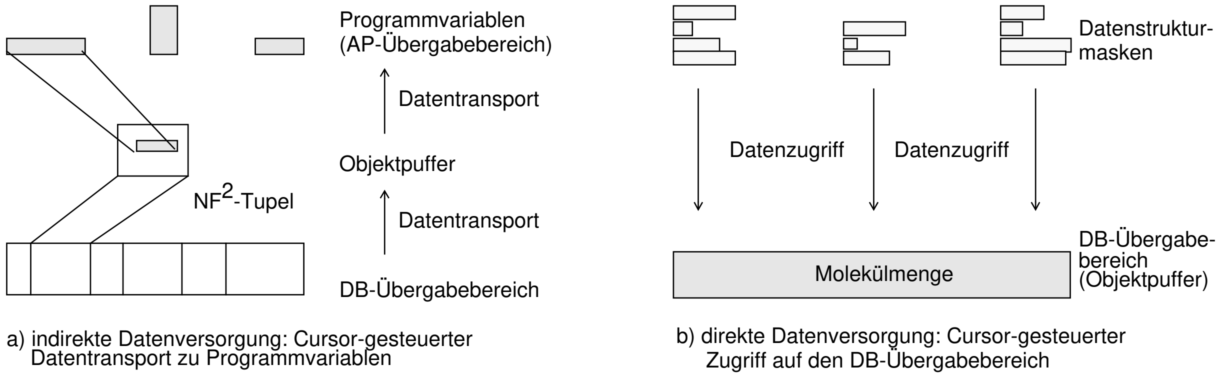

Abb. 2: Unterschiedliche Programmanbindung strukturierter Objekte

men kann. Das so an ein Anwendungsprogramm gebundene NF^2-Tupel kann nun sehr effizient unmittelbar mit den Mitteln der Anwendungs-Programmiersprache verarbeitet werden.

Verarbeitung netzwerkartiger Strukturen

Ausgangspunkt unserer eigenen Überlegungen hinsichtlich eines für Nicht-Standard-Anwendungen tauglichen Verarbeitunskonzeptes war das in [Mi88] vorgestellte Molekül-Atom-Datenmodell (MAD). Dieses Datenmodell und somit der DB-Übergabebereich ist durch den Begriff des Moleküls bzw. der Molekülmenge bestimmt. Moleküle können im Gegensatz zu den NF^2-Tupeln netzwerkartigstrukturiert sein und sich gegenseitig stark überlappen. Ein Vorgehen wie im oben beschriebenen NF^2-basierten Fall kann aus den folgenden Gründen nicht unmittelbar auf das Verarbeitungsschema netzwerkartiger Strukturen übertragen werden:

- Ein Herauslösen eines einzelnen Moleküls aus der im DB-Übergabebereich aufbereiteten Treffermenge würde dem Anwendungsprogramm gerade die "molekularen" Wechselwirkungen (Überlappungen) verbergen.
- Eine zu starke Einbindung der netzwerkartigen Strukturen an die Datenstrukturen des Anwendungsprogrammes bringt ebenfalls Probleme mit sich. Denkt man nur daran, daß diese Strukturen beliebig und unkontrolliert geändert werden können und, daß diese Modifikationen von dem DBS, bzw. der Bereitstellungskomponente erkannt werden müssen, wenn sie in der Datenbank nachvollzogen werden sollen.

Aufgrund dieser Argumente halten wir ein Verarbeitungsschema für angebracht, durch das dem Anwendungsprogramm ein direkter und zugleich kontrollierter Zugriff auf den DB-Übergabebereich ermöglicht wird (vgl. Abb. 2b):

- Das zugrundeliegende DBS (der NDBS-Kern) legt die gesamte Datenmenge (die das Resultat einer MAD-Anweisung darstellt) im DB-Übergabebereich in einer Form ab, die es den Anwendungsprogrammen erlaubt, über eine Art "Datenstrukturmaske" (eine Datenstrukturdefinition, die an unterschiedliche Hauptspeicheradressen gebunden werden kann) auf die Komponenten (atomare Elemente) der Ergebnismenge zuzugreifen.
- Der Zugriff auf die Struktur der Ergebnismenge, d.h. auf die Beziehungen zwischen den atomaren Elementen, geschieht in kontrollierter Form über einen noch näher zu erläuternden Cursor-Mechanismus. Ein Cursor identifiziert eine Komponente innerhalb des DB-Übergabebereiches und liefert dabei gerade die zum eigentlichen Zugriff benötigte Hauptspeicheradresse.

Da der DB-Übergabebereich also bereits die von den Anwendungsprogrammen benötigten Datenstrukturen und damit eine ganzheitliche Repräsentation der Anwendungsobjekte enthält, wollen wir ihn im folgenden ebenfalls als *Objektpuffer* bezeichnen.

Bei der Zuordnung von MAD-Anweisung und Objektpuffer existieren einige Freiheitsgrade. So ist ein schrittweises "Auffüllen" eines Objektpuffers durch mehrere MAD-Anfragen durchaus denkbar und aus Sicht der Anwendung oftmals wünschenswert. Daneben erlauben wir generell die Handhabung mehrerer Objektpuffer, wobei allerdings die wechselseitigen Beziehungen der Pufferinhalte durch das Anwendungsprogramm zu kontrollieren sind.

Die genauere Ausgestaltung der hier aufgeführten und insgesamt erforderlichen Konzepte und Mechanismen soll in den folgenden Kapiteln detailliert erläutert werden.

3. Ein Verarbeitungsmodell für komplex-strukturierte Objekte

Nach einer Diskussion der Verarbeitungskonzepte in verschiedenen Datenmodellen werden wir in diesem Kapitel die Verarbeitung komplex-strukturierter Objekte, die im MAD-Modell als molekulare Strukturen definierbar sind, genauer vorstellen. MAD ist als Datenmodell-Schnittstelle im NDBS-Kern PRIMA realisiert (**Pr**ototyp **I**mplementierung des **M**olekül-**A**tom-Datenmodells) [Hä88]. Zunächst werden die Konzepte des MAD-Modells anhand von Beispielen soweit erläutert, wie dies zum Verständnis des nachfolgend vorgestellten Verarbeitungsmodells notwendig ist. Eine detaillierte Diskussion des MAD-Modells findet sich in [Mi88].

3.1 Das Molekül-Atom-Datenmodell

Ähnlich wie eine Reihe von anderen Vorschlägen zur DB-seitigen Modellierung von komplexen Anwendungsgegenständen erlaubt auch das MAD-Modell die Spezifikation zusammengesetzter Objekte. Die in [LK84, PA86, SS86] aufgeführten Datenmodellierungskonzepte ermöglichen im wesentlichen die direkte Beschreibung von hierarchisch-strukturierten Objekten. Der MAD-Ansatz geht darüber hinaus und gestattet die Handhabung netzwerkartiger und rekursiver Strukturen. Die folgende Aufzählung faßt die wesenlichen Eigenschaften des MAD-Modells zusammen:

- Die komplexen Objekte werden als eine heterogene Menge von elementaren Objekten betrachtet.
- Diese können dynamisch definiert und abgeleitet werden.
- Die Beziehungen zwischen den elementaren Objekten werden direkt und symmetrisch beschrieben.
- Die Handhabung der komplexen Objekte erfolgt deskriptiv und mengenorientiert.

Die elementaren Objekte des MAD-Modells werden als Atome bezeichnet und bestehen aus einer Reihe von Attributen verschiedener Attributtypen (vergleichbar mit einem Tupel im Relationenmodell). Sie sind eindeutig identifizierbar und gehören zu einem entsprechenden Atomtyp. Die Attributtypen können einfache, einstufige Strukturen aufweisen, wie z.B. Feld, Liste oder Menge. Weiterhin unterscheidet MAD zwei spezielle Attributtypen: den Identifier-Typ und den Reference-Typ. Ein Attribut vom Identifier-Typ dient zur Aufnahme eines identifizierenden, systemvergebenen Schlüssels und wird als Surrogat verwendet. Der Reference-Typ erlaubt die Repräsentation logischer Referenzen zwischen Atomen und ermöglicht damit den Aufbau von Beziehungen. Diese Beziehungen werden als Links bezeichnet und sind ebenfalls typisiert. Ein Link-Typ ist durch ein Atomtypen-Paar bestimmt; der eigentliche Link (d.h. die Link-Ausprägung) durch zwei Atome. Ein Link ist bidirektional, so daß zu seiner Repräsentation stets ein Reference-Attribut-Paar (Referenz/Gegenreferenz) innerhalb der involvierten Atome erforderlich ist (entsprechend einer Primärschlüssel/Fremdschlüssel-Beziehung im Relationenmodell). Neben der Definition

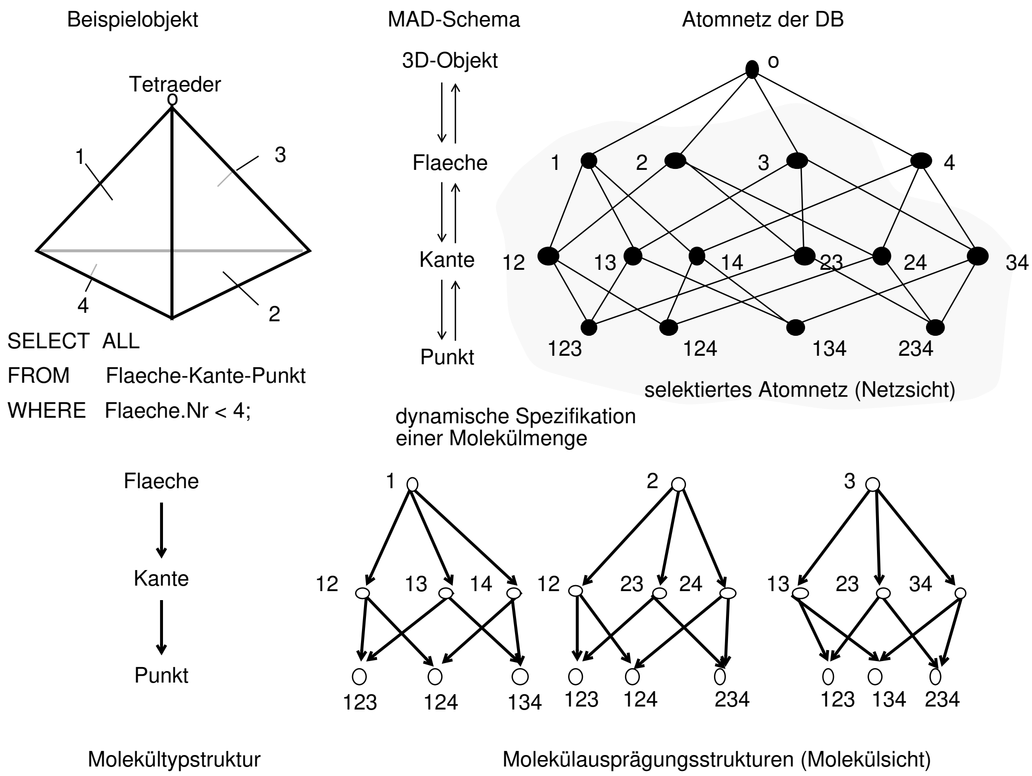

Abb. 3 : Objekte des Molekül-Atom-Datenmodells

von reflexiven Beziehungen erlaubt MAD über Reference-Listen eine direkte Darstellung der unterschiedlichsten Beziehungsarten (1:1, 1:n, n:m).

Die Atome und Links innerhalb einer DB bilden ein Atomnetz. Hierauf aufbauend können nun dynamisch die komplexen Objekte, in MAD Moleküle genannt, definiert und abgeleitet werden. Moleküle besitzen einen entsprechenden Molekültyp, der im wesentlichen die betroffenen Atom- und Reference-Typen angibt und damit die Molekültypstruktur festlegt. Ein spezieller Atomtyp spezifiziert die Molekülwurzel und damit die Startpunkte, von denen ausgehend alle Molekülkomponenten (die zugehörigen Atome) über die entsprechenden Referenzen ermittelt werden können.

Es wird deutlich, daß ein Molekül als eine Kombination aus einem heterogenen Atomnetz und einer Molekülausprägungsstruktur (kurz: Molekülstruktur) zu verstehen ist. Die Molekülstruktur ist dabei gerichtet und beschreibt die Abhängigkeiten bzgl. der Ableitungsreihenfolge der einzelnen Atome. Abb. 3 zeigt ein Beispiel eines Atomnetzes sowie eine Molekültyp- und -ausprägungsstruktur. Die Spezifikation und der Zugriff auf Moleküle kann in SQL-ähnlicher Notation formuliert werden. Hierzu bietet MAD die Sprache MQL (Molecule Query Language) an. Ähnlich wie in SQL gibt es auch in MQL drei zentrale Klauseln. Die FROM-Klausel enthält die Molekültyp-Definition. Die WHERE-Klausel legt Prädikate zur Einschränkung der zu betrachtenden Molekülmenge fest. Eine dritte Sprachklausel bestimmt die hierauf gewünschte Moleküloperation (SELECT, INSERT, DELETE, UPDATE). Ein besonderes Augenmerk gilt der SELECT-Klausel, da sie eine Art "qualifizierte" Projektion auf den durch FROM und WHERE bestimmten Molekülen erlaubt (hiermit wird eine wertabhängige Molekülgestaltung ermöglicht). Abb. 4 zeigt an einem Beispiel die Wirkungsweise. Daneben wird die Notwendigkeit von "Rollen" (P_1 für Anfangs- und P_2 für Endpunkt) verdeutlicht. Da der Punkt 123 als Endpunkt die Bedingung in der inneren WHERE-Klausel nicht erfüllt, wird er durch die qualifizierte Projektion in dieser Rolle aus der Molekülstruktur entfernt.

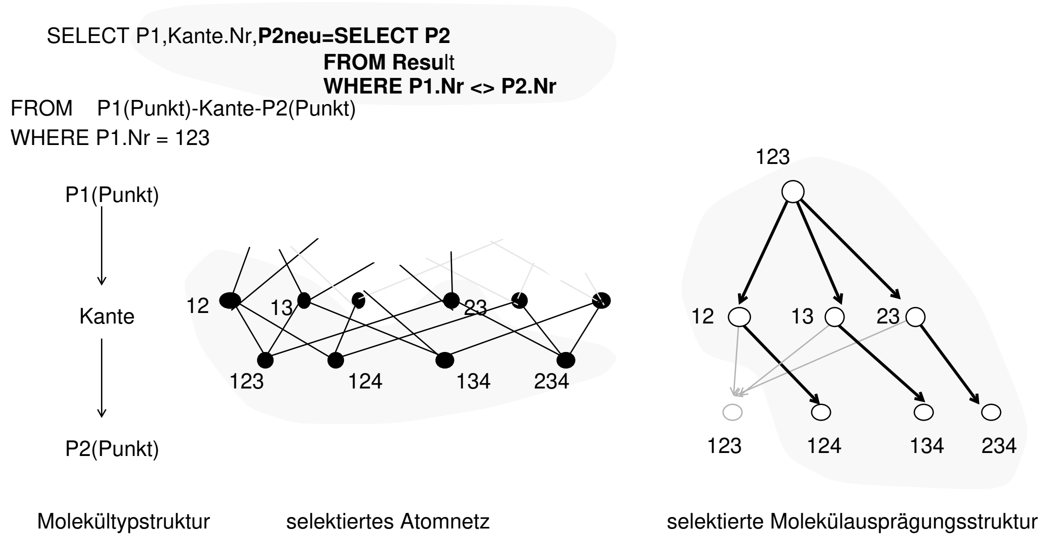

Abb. 4: Wertabhängige Molekülgestaltung durch qualifizierte Projektion

Im vorliegenden Fall werden sog. Rollennamen eingeführt, um die unterschiedliche Bedeutung festzuhalten, die Atome eines bestimmten Atomtyps innerhalb der Ergebnis-Moleküle annehmen können.

3.2 Verarbeitungskonzepte für molekulare Objekte

Abb. 5 zeigt die Architektur zur Anbindung des NDBS-Kerns PRIMA sowie des eingeführten Objektpuffers an ein Anwendungsprogramm. Wie bereits in der Einleitung motiviert, verspricht eine Zuordnung des NDBS-Kerns zu der Host-Seite und des Objektpuffers sowie der Anwendung zu der Workstation-Seite eine höhere Lokalität auf der Workstation und eine größere Autonomie der Systemkomponenten. Die Kommunikation mit dem Kern obliegt der Kern-Zugriffskomponente. Adressierungskomponente und Speicherverwaltung dienen zur Handhabung der Objektpuffer-Datenstrukturen. Aufgabe der Cursor-Verwaltung ist die Bereitstellung von Zugriffsmöglichkeiten auf die in den Objektpuffer eingelagerte Ergebnismenge. Über eine Anwendungs-Programmierschnittstelle kann schließlich ein Programm auf den Objektpuffer und die Datenmodell-Schnittstelle zugreifen.

Der prinzipielle Ablauf der Verarbeitung wird durch das Aktivieren von Check-out- und Check-in-Operationen bestimmt [LP83, KDG87]. Das Anstoßen einer Check-out-Operation bedeutet, daß die damit verbundene Retrieval-Anweisung im NDBS-Kern aktiviert, die Ergebnismenge evaluiert und im Objektpuffer vollständig abgelegt wird ((1) in Abb. 5). Nach dem deskriptiven Herauslösen des "Gegenstandes der Verarbeitung" können Änderungen lokal im Objektpuffer unter Benutzung eines dafür bereitgestellten Cursor-Konzeptes ausgeführt werden. Das Einbringen der Modifikationen in die DB wird durch eine Check-in-Operation angestoßen ((3) in Abb. 5). Daneben können vom Anwendungsprogramm die deskriptiven Modifikationsoperationen des MAD-Modells abgesetzt werden, die Änderungen direkt in der DB durchführen. Allerdings ist hiermit die Gefahr verbunden, daß der Inhalt eines Objektpuffers invalidiert: dies muß von der Anwendung, die ja im Prinzip über das notwendige Anwendungswissen verfügt, kontrolliert werden [HS89]. Vor einer genaueren Betrachtung der lokalen Verarbeitung ((2) in Abb. 5) wird zunächst die Strukturierung eines Objektpuffers an einem Beispiel näher vorgestellt.

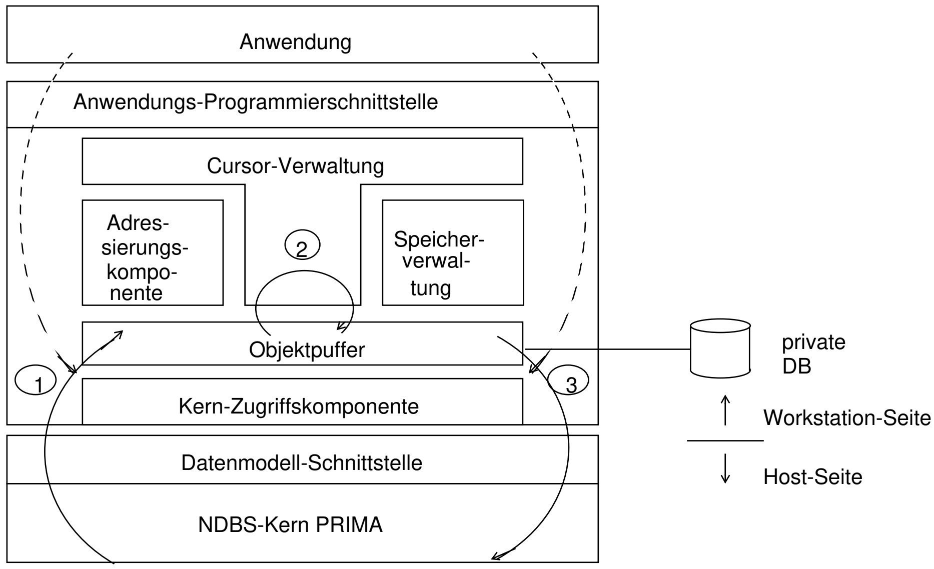

Abb. 5: Architektur der Objektpuffer-Einbettung

Datenstruktur zur Repräsentation einer Ergebnismenge

Das Resultat einer MAD-Anweisung wird vollständig im Hauptspeicher abgelegt. In Abb. 6 ist die Datenstruktur zur Repräsentation des Ergebnisses einer einfachen SELECT-Anweisung dargestellt. Ausgehend von einem Objektpuffer-Deskriptor lassen sich Strukturen zur Verwaltung des Hauptspeichers, Adressierungs- und Cursor-Verwaltungsstrukturen sowie Beschreibungsinformation über den Objektpuffer unterscheiden. Die Area-Tabelle und die Area-Blöcke werden durch die Speicherverwaltungskomponente (Abb. 5) kontrolliert. Die Area-Tabelle ist lediglich eine Verweisliste auf die Area-Blöcke, in denen schließlich die Atome als Basisbausteine einer Ergebnismenge abgespeichert sind. Ein Atom setzt sich aus einem Atomdeskriptor und den Atomattributen zusammen. Ein Atomdeskriptor besteht aus dem Atom-Identifier und einer Verweisliste auf die in der MAD-Anweisung projizierten Attribute sowie auf die durch die Molekültypstruktur bestimmten Reference-Attribute. Die Molekülstruktur-Information läßt sich auf sog. Strukturatome (diese sind in Abb. 6 aus Platzgründen weggelassen) abbilden, die den gleichen Aufbau wie gewöhnliche Atome besitzen, so daß zur Repräsentation der Molekülstruktur keine eigenen Datenstrukturen notwendig sind. Die Adressierungskomponente verwaltet die Atomtyp- sowie die Atomtabellen und ermöglicht den Zugriff auf ein Atom über dessen Identifier. Die Atomtyptabelle enthält für jeden in der Ergebnismenge auftretenden Atomtyp einen Verweis auf eine Atomtabelle, die als eine erweiterbare Hash-Struktur aufgebaut ist. Für jedes Atom existiert darin ein Eintrag, der die physische Adresse des Atoms und einen Index in der Area-Tabelle enthält, über den der zugehörige Area-Block erreicht wird. In einem Modifikationsflag werden darüber hinaus mögliche Änderungen auf dem Atom vermerkt, um ein späteres Einbringen in die DB zu erleichtern. Die zum Zugriff auf den Objektpuffer angelegten Cursor sind in der Cursor-Tabelle organisiert. Schließlich enthält die Puffer-Beschreibungsstruktur die für den Zugriff auf die Atome notwendige Metainformation.

Das angesprochene Area-Konzept wirkt zum einen einer Zerstückelung des Hauptspeichers entgegen und ermöglicht zum anderen eine einfache Verschiebbarkeit der Ergebnismenge, wodurch das Aus- bzw. Einlagern in die lokal auf Workstation-Seite vorhandene private DB erleichtert wird. Beim Auslagern wird

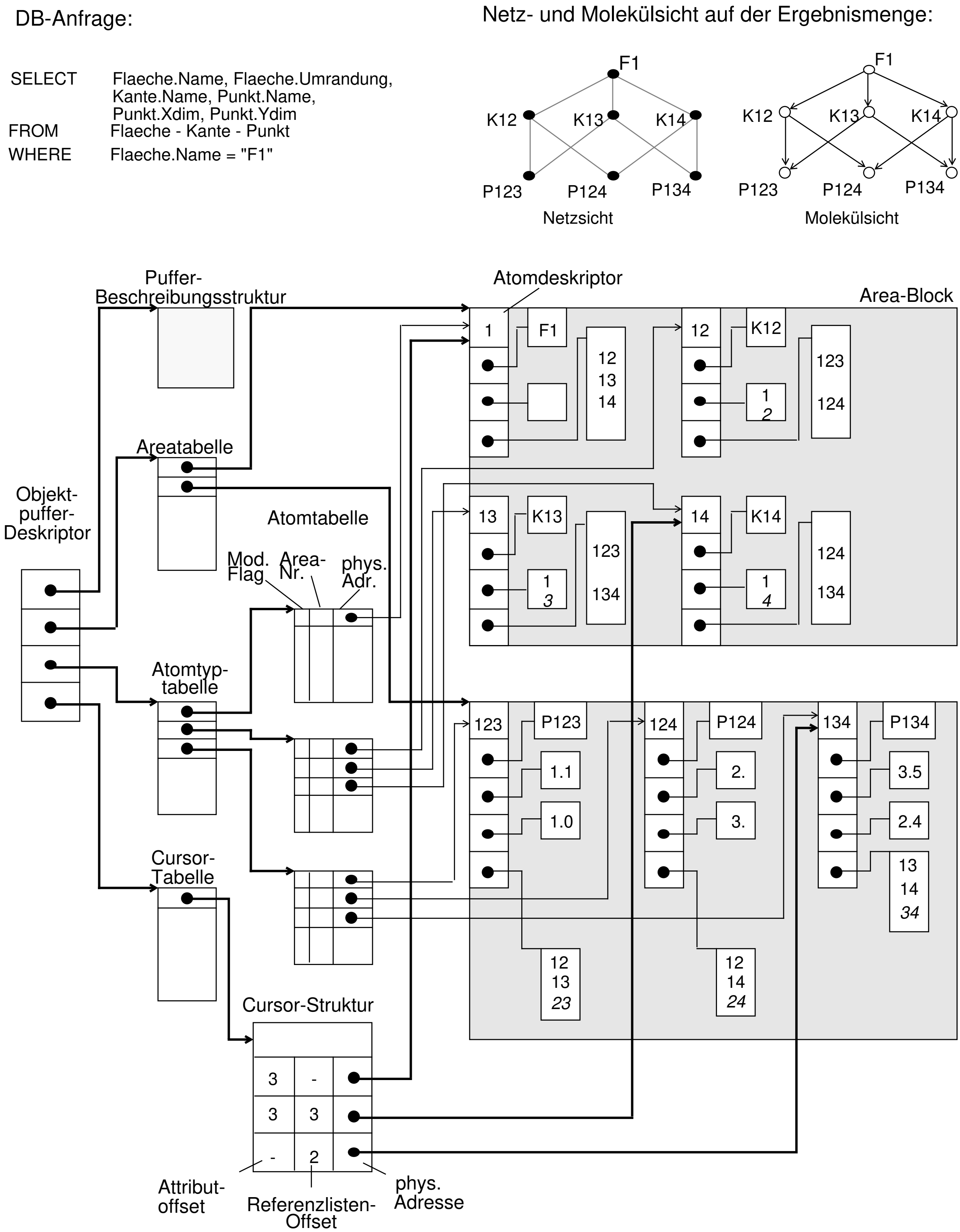

Abb. 6: Datenstruktur eines Objektpuffers (Netzsicht)

aus dem Area-Index und der physischen Atomadresse der Area-Offset eines Atomdeskriptors berechnet und in der Atomtabelle anstelle der physischen Adresse abgespeichert. Die Area-Blöcke und die Atomtabellen können somit ohne großen Aufwand in eine ortstransparente Struktur überführt werden.

Verarbeitung mit netzwerk- und molekülorientierter Sichtweise

In dem von uns vorgeschlagenen Verarbeitungsmodell wird eine Ergebnismenge in erster Linie als ein Atomnetz interpretiert, zusätzlich wird eine molekülorientierte Sicht angeboten. Die Anforderungen der Algorithmen bzgl. ihrer Datenversorgung weisen i.a. keine bevorzugte Verarbeitungsrichtung auf [Hä87, HHMM88], daher muß ein beliebiges Navigieren auf einer Ergebnismenge möglich sein. Hierbei ist eine *netzwerkorientierte Sichtweise* angebracht, bei der man von den bidirektionalen Beziehungen zwischen den Atomen ausgeht und die Molekülstruktur außer acht läßt. Eine *molekülorientierte Sichtweise* auf der Ergebnismenge ist durch die Molekülstruktur festgelegt (vgl. Kapitel 3.1), d.h., die mögliche Verarbeitungsrichtung wird in diesem Fall durch die Abhängigkeiten bzgl. der Ableitungsreihenfolge der betroffenen Atome bestimmt. Das Wissen über die Aufsuchreihenfolge ist vor allem bei einer Verarbeitung von Ergebnismengen von Bedeutung, die durch rekursive MAD-Anfragen oder Anfragen mit qualifizierter Projektion entstanden sind.

Ein Cursor-Konzept zur Unterstützung der lokalen Verarbeitung

Zur Unterstützung der Anwendungsalgorithmen wird ein mächtiges Cursor-Konzept bereitgestellt, das die explizite Definition flacher und hierarchischer Cursor erlaubt. Ein flacher Cursor ist über einen Atomtyp definiert. Der Cursor-Bereich, also die Menge der Atome, die über einen Cursor erreicht werden können, umfaßt bei einem flachen Cursor sämtliche Atome des Atomtyps. Auf der Aufsuchreihenfolge der Atome in einem Cursor-Bereich ist keine Ordnung definiert. Ein hierarchischer Cursor setzt sich aus einer Folge flacher, voneinander abhängiger, atomtyp-bezogener Cursor zusammen, die jeweils über eine in der Netz- bzw. Molekülsicht definierte Beziehung miteinander verbunden sind. Damit ist insbesondere eine Cursor-Hierarchie mit einer der Molekültypstruktur entgegengesetzten Richtung möglich. Der Cursor-Bereich auf einer Hierachiestufe ist immer durch die aktuellen Cursor-Positionen auf den übergeordneten Stufen bestimmt. Wird beispielsweise auf dem Atomnetz in Abb. 3 ein Cursor über die Atomtypen Flaeche-Kante-Punkt definiert und weist der Cursor auf Stufe "Flaeche" auf das Atom F_2 und auf der Stufe "Kante" auf das Atom K_{23}, so umfaßt der Cursor-Bereich auf Stufe "Punkt" die Atome P_{123} und P_{234}. Cursor-Bereiche können überlappend sein, d.h., ein Atom kann in mehreren Cursor-Bereichen vorkommen. Weist der Cursor auf die Atome F_2 und K_{24}, so setzt sich der Cursor-Bereich auf Stufe "Punkt" aus P_{124} und P_{234} zusammen.

Im folgenden sollen die wesentlichen Cursor-Eigenschaften stichpunktartig aufgezählt werden:

- Beim Aktivieren eines hierarchischen Cursors auf einer Ergebnismenge weist er zunächst auf jeder Cursor-Stufe auf ein Atom im aktuellen Cursor-Bereich. Für den Cursor auf der obersten Hierarchiestufe kann über einen zweiten aktiven Cursor ein Atom bestimmt werden, auf das der Cursor nach seiner Aktivierung weist (Abb. 7a). Dadurch wird ein Cursor auf einem vordefinierten Bereich eröffnet.
- Es wird ein navigierender Zugriff auf den Strukturen im Objektpuffer (Netz- und Molekülsicht) ermöglicht, wodurch gezielt auf die für den nächsten Verarbeitungsschritt benötigten Daten zugegriffen werden kann.
- Das Weiterschalten eines Cursors auf das nächste Atom einer Hierarchiestufe hat zur Folge, daß der Cursor auf allen tieferen Hierarchiestufen auf ein Atom des neu bestimmten Cursor-Bereichs gesetzt

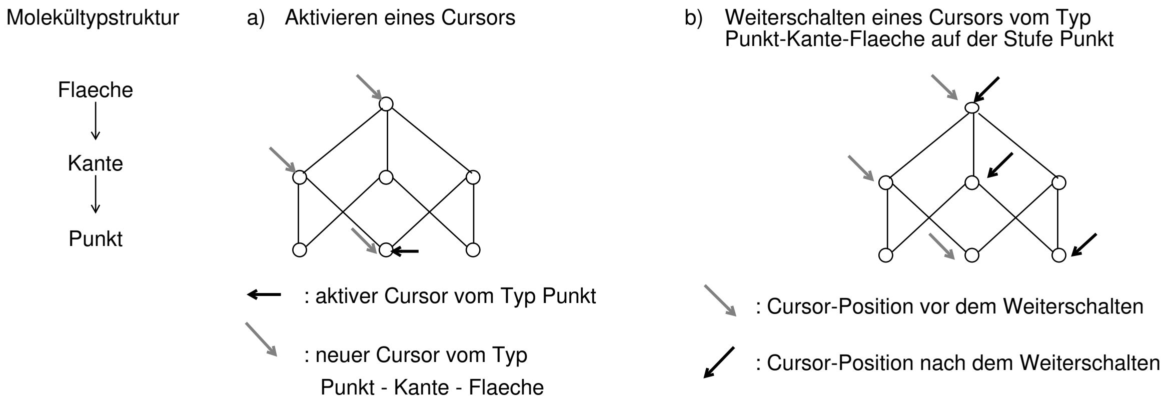

Abb. 7: Cursor-Aktivierung und Weiterschalten von Cursorn

wird (Abb. 7b). Auf den höheren Hierarchiestufen ändern sich die Cursor-Positionen nicht. Hat der Cursor bereits alle Atome überstrichen, so ist er auf dieser und allen abhängigen Stufen undefiniert.

In Abb. 6 sind die Datenstrukturen zur Verwaltung der Cursor aufgezeigt. Über eine Cursor-Tabelle, die für jeden auf dem Objektpuffer definierten Cursor einen Eintrag enthält, wird auf eine Cursor-Datenstruktur zugegriffen. Darin ist für jede Hierarchiestufe die physische Adresse des aktuellen Atoms abgespeichert. Weiter enthält ein Eintrag den Attribut-Offset des Reference-Attributs, in dem die Identifier der Atome der nächsten tieferen Hierarchiestufe abgelegt sind. Der Referenzlisten-Offset vermerkt den Index des Atom-Identifiers im Reference-Attribut des übergeordneten Atoms. Für den hierarchischen Cursor über "Flaeche-Kante-Punkte" in Abb. 6, der aktuell auf die Atome F_1, K_{14} und P_{134} weist, ist die Cursor-Struktur exemplarisch dargestellt. Der Zugriff auf ein Atomattribut läßt sich somit sehr effizient realisieren (zwei Dereferenzierungs-Vorgänge).

Änderungen der Daten im Objektpuffer

Sollen die im Objektpuffer durchgeführten Änderungen zu einem späteren Zeitpunkt in die Datenbank "durchgereicht" werden, so muß die Ergebnismenge bestimmte Bedingungen erfüllen: So darf in der DB-Anfrage keine "Verdichtung" der Daten spezifiziert werden (z.B. Built-in-Funktionen), da sonst im Objektpuffer Atomtypen und Attribute angelegt werden, die in der Datenbank nicht existieren.

Änderungen werden nur über Cursor ausgeführt. Einem Atom, auf das ein Cursor aktuell verweist, können neue Attributwerte zugewiesen werden. Es ist möglich, Beziehungen zwischen zwei Atomen im Objektpuffer einzufügen bzw. zu lösen. Dies geschieht über zwei Cursor, die die beiden Atome (und die Reference-Attribute) festlegen, zwischen denen eine Beziehung modifiziert werden soll. Ebenso ist das Einfügen und Löschen von Atomen in einem Objektpuffer möglich. Einem neu eingefügten Atom können nur die Attribute zugewiesen werden, die in der entsprechenden MAD-Anweisung projiziert wurden. Ein Atom wird nur gelöscht, wenn es keine Beziehungen (außer den über Cursor explizit angegebenen) zu anderen Atomen im Objektpuffer hat, ansonsten werden nur die spezifizierten Beziehungen gelöst.

Strukturelle Änderungen im Objektpuffer können zu einer Invalidierung von Cursorn führen, in deren aktuellem Cursor-Bereich die Änderung ausgeführt wurde. Durch die strukturellen Modifikationen verändert sich der Cursor-Bereich auf einer Hierarchiestufe, worauf beim Weiterschalten des Cursors geeignet reagiert werden muß. Das Einführen eines Cursor-Zustandes ermöglicht es, für einen Cursor auf einer Hierarchiestufe zu entscheiden, ob er auf ein Atom verweist, "zwischen" zwei Atomen des Cursor-Bereichs steht (z.B. wenn das referenzierte Atom über einen zweiten Cursor gelöscht wurde), oder ob der Cursor bereits den gesamten Bereich überstrichen hat. Eine genauere Untersuchung der Invalidierungsproblematik findet sich in [HRS89].

Ist die Bearbeitung der Ergebnismenge abgeschlossen, muß der Objektpufferinhalt in die Datenbank eingebracht werden. Dazu wird während der Verarbeitung im Modifikationsflag in der Atomtabelle (Abb. 6) für jedes Atom vermerkt, ob es verändert, gelöscht oder neu eingefügt wurde. In der Check-in-Operation werden die modifizierten Atome zusammengestellt, die dann durch die MAD-Modifikationsoperationen Delete, Update und Insert in die Datenbank eingebracht werden. Nach der Check-in-Operation wird die Speicherungsstruktur des Objektpuffers freigegeben.

3.3 Eine Programmierschnittstelle

Die Anwendungs-Programmierschnittstelle legt fest, wie das Verarbeitungsmodell an eine Programmiersprache angebunden wird, d.h., wie in einem Programm ein Objektpuffer angelegt, wie darauf über Cursor zugegriffen bzw. wie mit dem DBS kommuniziert werden kann. In der Literatur werden vier Ansätze zur sprachlichen Anbindung unterschieden [LaP83]:

- Eine *CALL-Schnittstelle* ist die einfachste Form der Anbindung. Sie setzt auf Anwenderseite meist eine sehr detaillierte Kenntnis der Parameterbelegung auf niedriger Abstraktionstufe voraus.
- Bei einer *Einbettung* der Datenbanksprache in eine konventionelle Programmiersprache (z.B. QUEL, SQL [Da87]) werden die DB-bezogenen Anweisungen von einem Vorübersetzer erkannt und in Aufrufe an ein DB-Laufzeitsystem transformiert.
- Eine *Erweiterung* der Wirtssprache (etwa CODASYL COBOL-DML, [Co78]) bedeutet, daß eine prozedurale Programmiersprache um DBS-spezifische Sprachkonstrukte erweitert wird.
- Der Ansatz zur *Sprachintegration* versucht, eine allgemeine Programmiersprache mit einem Datenmodell möglichst homogen zu vereinen (z.B. Pascal/R, [Sch77]).

Unser Vorschlag zur programmiersprachlichen Anbindung des MAD-Modells und des vorgestellten Verarbeitungsmodells kombiniert die Konzepte der Erweiterung und der Einbettung. Durch eine Erweiterung werden eine möglichst natürliche Integration der zur lokalen Verarbeitung erforderlichen Sprachkonstrukte in die Programmiersprache (es wurde Modula-2 gewählt, jede andere höhere Programmiersprache wäre auch denkbar) und ein homogener Zugriff auf Pro grammvariablen und Daten im Objektpuffer erreicht. Die Wirtssprache wird um die im MAD-Modell zusätzlich bekannten Datentypen und die darauf definierten Operationen erweitert (z.B List, Hull, Time, etc.; vgl. [Mi88]). Die eigentlichen MQL-Anweisungen werden in das Programm eingebettet und zur weiteren Bearbeitung an den MQL-Übersetzer des NDBS-Kernsystems übergeben.

Aufgabe und Funktionsweise des Vorübersetzers

Ein Vorübersetzer übernimmt die Aufgabe der Transformation eines mit den eingeführten Sprachkonstrukten erweiterten Programmes (eMQL-Programm) in ein "normales" Modula2-Programm (Abb. 8). Zunächst werden die DB-Anweisungen vom MQL-Übersetzer behandelt und als Zugriffsmodul in den Metadaten der Datenbank ablegt. Als Ergebnis liefert er dem Vorübersetzer anweisungsbezogene Metainformation zurück. Diese wird vom Vorübersetzer zur Umsetzung logischer Namen (etwa Atomtyp-, Attribut- oder Rollen-Name) in Indizes oder Offset-Angaben sowie zur Generierung von Datenstrukturmasken für den direkten Objektpufferzugriff verwendet. Insbesondere erfolgt eine Überprüfung der Typkompatibilität bei Zuweisungen und der Korrektheit von Cursor-Definitionen. Hierdurch ist es möglich, die eMQL-Anweisungen soweit vorzubearbeiten, daß der Interpretationsaufwand zur Laufzeit relativ gering ist. Im weit-

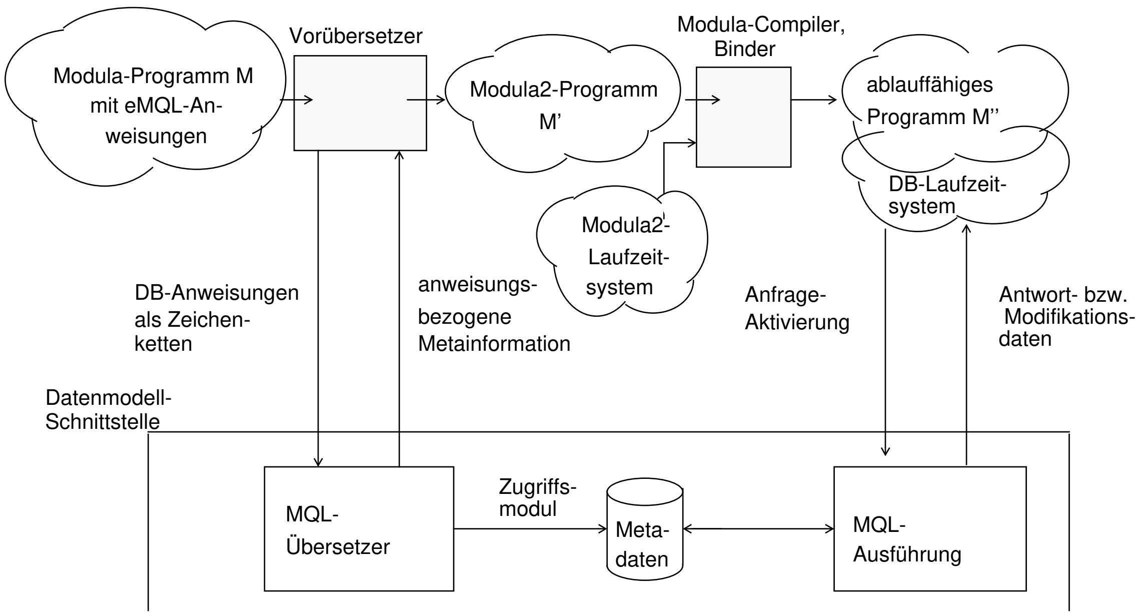

Abb. 8: Übersetzung von eMQL-Programmen

eren sollen nun die wichtigsten eMQL-Sprachkonstrukte an einem Beispielprogramm vorgestellt werden (vgl. Abb. 9).

Anweisungen zur Objektpuffer-Handhabung

Durch eine QUERY_TYPE-Anweisung erfolgt die Spezifikation einer MAD-Anweisung, der damit ein Typname zugeordnet wird. In der MAD-Anweisung auftretende Anweisungsparameter (durch das Symbol $ gekennzeichnet) werden am Ende der Typdefinition als Variablenliste (mit einer Typangabe) zusammengefaßt. Eine SELECT-Anweisung kann zusätzlich die Option FOR UPDATE enthalten, d.h., auf der Ergebnismenge werden Änderungen erlaubt; der MQL-Übersetzer überprüft das Änderungsrecht. Die Definition eines BUFFER_TYPEs bestimmt die Menge der Query-Typen, deren Ergebnismengen in einer Ausprägung dieses Objektpuffer-Typs eingelagert werden können. Eine Variablendeklaration legt eine Ausprägung eines Objektpuffer-Typs (gekennzeichnet durch das Schlüsselwort BUFFER_VAR) an, d.h., der Puffer kann das Ergebnis der angegebenen Anfrage-Typen aufnehmen. Die Aktivierung einer MAD-Anweisung erfolgt durch die EVAL-Anweisung. Bei einer SELECT-Anweisung werden die Werte für die Anweisungsparameter und der Name der Puffervariablen, in die die Ergebnismenge einzulagern ist, übergeben. Bei DB-Modifikationsoperationen (INSERT, UPDATE und DELETE) werden auch Anweisungsparameter und einzufügende DB-Daten übergeben (z.B. beim Einfügen neuer Atome). Die in einer Objektpuffervariablen durchgeführten Änderungen werden durch die PROPAGATE-Anweisung in die Datenbank eingebracht.

Cursor-Definition und Cursor-Operationen

Eine CURSOR_TYPE-Definition legt die Cursor-Hierarchie und den Objektpuffer-Typ fest, auf dem der Cursor definiert sein soll. Von einem Cursor-Typ werden durch eine CURSOR_VAR-Anweisung Ausprägungen angelegt. Ein ATTACH-Aufruf ordnet einen Cursor einer Objektpuffer-Variablen zu und aktiviert den Cursor. CLOSE deaktiviert einen Cursor, so daß über ihn nicht mehr auf den Objektpuffer zugegriffen werden kann. Die MOVE-Anweisung schaltet den Cursor auf der angegebenen Hierarchiestufe auf das

```
MODULE Flaeche;
QUERY_TYPE          Flaechen_Query_type=
                      "SELECTFOR UPDATE    Flaeche.Name,
                                           Flaeche.Umrandung, Kante.Name,
                                           Punkt.Name, Punkt.Xdim, Punkt.Ydim
                      FROM                 Flaeche-Kante-Punkt
                      WHERE                Flaeche.Name = $Fname1 OR
                                           Flaeche.Name = $Fname2"
                      (Fname1, Fname2: Byte_List);

BUFFER_TYPE    Flaechen_Buffer_type=(Flaechen_Query_type);
BUFFER_VAR     Flaechen_Buffer:Flaechen_Buffer_type;

CURSOR_TYPE    Flaechen_C_type=              Flaechen_Buffer_type@Flaeche
               FKP_C_type=                   Flaechen_Buffer_type@Flaeche[Kante[Punkt]];
CURSOR_VAR     F_C      :                    Flaechen_C_type;
               FKP_C    :                    FKP_C_type;

VARFname : Byte_List;

PROCEDURE Umrandung (FKP_C : FKP_C_type, VAR Laenge : REAL);
BEGIN
               Laenge := 0;
               MOVE FKP_C FIRST Kante;
/* Bestimme die Länge sämtlicher Kanten und addiere sie in "Laenge" auf; dazu läuft der Cursor über alle
Kanten der Flaeche; Zugriff auf Koordinaten über Cursor auf Stufe Punkt; implizites Weiterschalten des
Cursors auf Stufe Punkt (Punkt) */
             WHILE (NOT (NULL (FKP_CKante)) DO
                  Laenge:= Laenge + SQRT  SQR (FKP_CPunkt.Xdim  -  FKP_CPunkt1.Xdim) +
                           SQR (FKP_CPunkt.Ydim  -  FKP_CPunkt1.Ydim));
                  MOVE FKP_C NEXT Kante;
             END;
END Umrandung;

/*Beginn des Hauptprogrammes*/
BEGIN
/*Aktivieren der Query und des Flaechen-Cursors */
     EVAL (Flaechen_Query_TYPE ("F2", Fname), Flaechen_Buffer);
     ATTACH (Flaechen_Buffer, F_C);

/* Bestimme für jede Fläche die Länge der Umrandung, falls sie noch nicht berechnet ist*/
     WHILE   (NOT NULL (F_CFlaeche)) DO
             Umrandung  (F_C, Laenge);
             F_CFlaeche.Umrandung:= Laenge;
             MOVE F_C  NEXT Flaeche;
     END;

/*Einbringen des Objektpuffers in die Datenbank*/
     PROPAGATE (Flaechen_Buffer);

END Flaeche.
```

Abb. 9: Programmfragment mit eMQL-Sprachkonstrukten

QUERY_TYPE,BUFFER_TYPE, CURSOR_TYPE	Definition eines Anfragetyps, eines Objektpuffertyps bzw. eines Cursor-Typs
BUFFER_VAR, CUSROR_VAR	Deklaration von Objektpuffervariablen bzw. Cursor-Ausprägungen
EVAL, PROPAGATE	Ausführung der Check-out-/Check-in-Operationen
ATTACH, CLOSE	Aktivieren bzw. Schließen eines Cursors
MOVE	Weiterschalten eines Cursors
NULL	Testen, ob ein Cursor auf ein Atom weist
CONNECT, DISCONNECT,	Änderung der Beziehungen zwischen Atomen,
INSERT, DELETE	Einfügen, Löschen von Atomen

Abb. 10: Die wichtigsten Sprachkonstrukte von eMQL

nächste (bzw. das erste) Atom weiter. Durch einen Aufruf der Funktion NULL kann festgestellt werden, ob ein Cursor auf ein Atom weist oder nicht, da z.B. der Cursorbereich bereits vollständig abgearbeitet ist.

Ein Direktzugriff auf den Objektpuffer ist über einen aktiven Cursor möglich. Der Ausdruck <cursor_name><atomtyp>.<attribut> spezifiziert ein Attribut auf der Cusor-Stufe <atomtyp>, auf das nun ähnlich wie auf eine Programmvariable zugegriffen werden kann: Dem Attribut kann ein neuer Wert zugewiesen bzw. der Wert kann einer Programmvariablen übergeben werden, ebenso kann das referenzierte Attribut in einem Ausdruck des Programms auftreten. In Abb. 10 sind die wichtigsten Sprachkonstrukte nochmals aufgelistet.

4. Schlußfolgerungen

Die Modellierungs- und Verarbeitungsproblematik wurden als die wesentlichen Hindernisse beim DBS-Einsatz im Ingenieurbereich erkannt. Im Rahmen der allgemein diskutierten Kern-Architektur für Nicht-Standard-Datenbanksysteme wurde zur Lösung der Verarbeitungsproblematik ein spezielles Verarbeitungsschema vorgeschlagen. Dieses ermöglicht die Nutzung anwendungs- und verarbeitungsinhärenter Lokalität (bzgl. der Datenreferenzen) in "Anwendungsnähe". Anhand des MAD-Modells, das die Beschreibung netzwerkartig-strukturierter DB-Objekte gestattet, konnten dann Konzepte zur Einbettung einer strukturbezogenen Verarbeitung in eine mengenorientierte Datenmodell-Schnittstelle vorgestellt werden. Hierbei waren die wesentlichen Punkte:

- deskriptives Festlegen der "Verarbeitungsgegenstände",
- Speicherung der Verarbeitungsgegenstände in adäquter Form im anwendungsnahen Objektpuffer,
- Prozedurale Verarbeitung des Objektpuffer-Inhaltes mit Hilfe eines expliziten Cursor-Konzeptes (direkter, dennoch kontrollierbarer Zugriff),
- Ganzheitliches Einbringen der durchgeführten Modifikationen in den Datenbestand des DBS.

Die Konkretisierung dieser Ansätze hin zu einer programmiersprachlichen Einbettung wurde an einem Beispiel verdeutlicht. Insgesamt haben wir damit eine angemessene Grundlage für die DB-Verarbeitung (auch) im Ingenieurbereich vorgestellt.

Neben der Implementierung der diskutierten Konzepte liegt unsere zukünftige Arbeit in der Organisation der unmittelbaren Anwendungsprogramme, d.h., in der Gestaltung der NDBS-Modellabbildungsebene (vgl. Abb. 1). Hier verspricht das Konzept der abstrakten Datentypen (ADT) einige Vorteile, da durch die automatisch erreichte Modularisierung die Änderbarkeit, die Erweiterbarkeit, und die Anpaßbarkeit der Modellabbildung zunimmt. Zudem wird durch die Kapselung von Datenstrukturen und zugehörigen Operationen eine objektorientierte Sichtweise unterstützt, da die strukturelle und operationale Beschreibung eines Anwendungsobjektes als Einheit zusammengefaßt werden können.

Literatur

Co78 CODASYL Data Description Language Comittee Report, Information Systems, Vol. 3, No. 4, 1978, pp. 247-320.

Da82 Date, C.J.: An Introduction to Database Systems, third editon, Addison Wesley-Verlag, 1982.

Da86 Dadam,P.,et al.: A DBMS Prototype to Support Extended NF^2-Relations: An Integrated View on Flat Tables and Hierachies, in: Proc. ACM SIGMOD Conf., Washington D.C., 1986, pp. 356-367.

Da87 Date, C.J.: A Guide to Ingres, Addison Wesley-Verlag, 1987.

EW87 Erbe, R., Walch, G.: An Application Program Interface for an NF^2 Database Language or How to Transfer Complex Object Data Into Application Program, Technical Report TR.87.04.003, IBM wiss. Zentrum Heidelberg, April 1987.

Hä87 Härder, T.: Database Support for Engineering Applications, in: Proc. Int. Workshop on Information in Manufacturing Automation, Dresden, GDR, Juli 1987.

Hä88 Härder, T.: Overview of the PRIMA Project, in: The PRIMA Project, Design and Implementation of a Non-Standard Database System, T. Härder (ed.), Research Report No. 26/88, SFB 124, University Kaiserslautern.

HHMM88 Härder, T., Hübel, Chr., Meyer-Wegener, K., Mitschang, B.: Processing and Transaction Concepts for Cooperation of Engineering Workstations and a Database Server, in: Data and Knowledge Engineering, 3(1988), pp. 87-107, 1988.

HP88 Hübel, Chr., Pick, M.: Anwendungsnahe Pufferung komplex-strukturierter Objekte - ein Erfahrungsbericht, Universität Kaiserslautern, (in Vorbereitung).

HR85 Härder, T., Reuter, A.: Architektur von Datenbanksystemen für Non-Standardanwendungen, in: Tagungsband der GI-Fachtagung Datenbanken für Büro, Technik und Wissenschaft, Karlsruhe, 1985, pp. 253-286.

HRS89 Hübel, Chr., Ratajczak, B., Sutter, B.: Entwurf und Implementierung eines Laufzeitsystems zur objektpufferbasierten Molekülverarbeitung, technischer Bericht, Universität Kaiserslautern, 1989.

HS89 Hübel, Chr., Sutter, B.: Aspekte der Datenbank-Anbindung in workstation-orientierten Ingenieuranwendungen; erscheint in: Proc. der 19. Jahrestagung der Gesellschaft für Informatik, München, Oktober 1989.

KDG87 Küspert, K.,Dadam P., Günauer J.: Cooperative Object Buffer Management in the Advanced Information Prototype, Proc. VLDB '87, Brighton, U.K., Sept. 1987, pp. 483-492.

LaP83 Lacroix, M., Pirotte, A.: Comparison of Database Interfaces for Application Programming, in: Inf. Systems, Vol. 8, No. 3, 1983, pp. 217-229.

Lo85 Lockemann, P. C., et al.: Anforderungen technischer Anwendungen an Datenbanksysteme, in: Proc. GI-Fachtagung Datenbanken für Büro, Technik und Wissenschaft, Karlsruhe, 1985.

LK84 Lorie, R., Kim, W., et. al.: Supporting Complex Objects in a Relational System for Engineering Databases, IBM Research Laboratory, San Jose., CA, 1984.

LP83 Lorie, R., Plouffe, W.: Complex Objects and Their Use in Design Databases, in: Proc. Database Week 1983, IEEE Computer Society Press.

Mi88 Mitschang, B.: The Molecule-Atom Data Model in: The PRIMA Project, Design and Implementation of a Non-Standard Database System, T. Härder (ed.), Report Nr. 26/88, SFB 124, University Kaiserslautern, März 1988.

PA86 Pistor, P., Anderson, F.: Designing a Generalized NF2 Data Model with a SQL-Type Language Interface, Proc. 12th VLDB Conf., Kyoto, 1986.

PSSWD87 Paul, H.-B., Schek, H.-J., Scholl, M.H., Weikum, G., Deppisch, U.: Architecture and Implementation of the Darmstadt Database Kernel System, in: ACM SIGMOD Conf., San Francisco, 1987, pp. 196-207.

Sch77 Schmidt, J.W.: Some high level constructs for data of type relation, ACM Trans. Database Systems, 2, 3 (Sept.), pp. 247-261.

SR84 Stonebraker, M.,Rowe, Database Portals - A new Application Program Interface, in: Proc. 10th Int. Conf. on VLDB, Singapore,1984,pp.3-13.

SS86 Schek, H.-J., Scholl, M.H.: The Relational Model with Relation-Valued Attributes, in: Information Systems, Vol. 2, No. 2, 1986, pp. 137-147.

St87 Stonebraker, M., et al.: Extending a Database System with Procedures, ACM Trans. on Database Systems, Vol. 12, No. 3, Sept. 1987, pp. 350-376.