Daten- und Funktionsintegration durch Föderierte Datenbanksysteme

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation

von
Dipl.-Inf.

Klaudia Hergula

Dekan des Fachbereichs Informatik:
Prof. Dr. Hans Hagen

Promotionskommission:

Vorsitzender:	Prof. Dr. Stefan Deßloch
Berichterstatter:	Prof. Dr. Dr. Theo Härder
	Prof. Dr. Wolfgang Benn

Datum der wissenschaftlichen Aussprache:
10. Oktober 2003

D 386

.

Satz: LaTeX 2_ε

Geleitwort

Datenintegration zielt auf die Zusammenführung und Homogenisierung von Daten aus heterogenen Datenquellen ab, wobei idealerweise den zugreifenden Anwendungsprogrammen ein diese Datenquellen überspannendes Datenbankschema als Teil der Anwendungsprogrammierschnittstelle (API) zur Verfügung gestellt wird. Über ein solches DB-Schema sollen dann mit Hilfe einer generischen DB-Sprache (SQL) sowohl Anfrage- als auch Aktualisierungsoperationen abgewickelt werden, was schon wegen der semantischen Heterogenität der Daten und der (möglicherweise weitreichenden) Autonomie der Datenquellen immer noch eine große Herausforderung an die DB-Forschung und -Entwicklung darstellt.

Die Lösung dieser DB-Probleme durch Föderierte Datenbanksysteme (FDBS) ist jedoch in der Praxis nicht ausreichend, da in jüngster Zeit viele der zu integrierenden Datenquellen nicht direkt, d.h. nicht über generische Sprachen zugänglich sind. Vielmehr sind deren Daten durch Anwendungssysteme gekapselt, wobei deren API nur einen Zugriff über vordefinierte Funktionen (lokale Funktionen) gestattet. Weiterhin ist es im Sinne einer Anwendungsintegration erforderlich, Daten aus mehreren Anwendungssystemen zu verknüpfen und gemeinsam auszuwerten. Es sind also automatisch lokale Funktionen zu globalen Funktionen zusammenzufassen und deren Auswertungsergebnisse, die wiederum Daten repräsentieren, einem FDBS zuzuführen, das dann die Integration mit Daten aus weiteren Datenquellen bewerkstelligt. Diese Problemskizze charakterisiert das Thema der Arbeit von K. Hergula. Sie liefert damit wissenschaftliche Beiträge zu dem hoch aktuellen Thema der Informationsintegration, die als Verbindung von Daten- und Funktionsintegration angesehen werden kann. Da die reine Datenintegration in der Literatur schon breit abgehandelt wurde, die Funktionsintegration dagegen eher wissenschaftliches Neuland verkörpert, hat K. Hergula den Schwerpunkt ihrer Arbeit auf die Funktionsintegration gelegt und damit viele innovative Beiträge zur Entwicklung dieses Themenbereichs geleistet und auch – aus meiner Sicht erstmalig – ihre praktische Durchführbarkeit durch eine prototypische Implementierung gezeigt.

Der entwickelte Lösungsansatz zur Integration ist unabhängig von seiner technischen Umsetzung gehalten. Deshalb hat K. Hergula hier ausschließlich auf Konzeptebene ein Beschreibungs- und Ausführungsmodell für die zu integrierenden Funktionen entwickelt. Zentral für das Beschreibungsmodell für die Abbildung von lokalen Funktionen auf globale Funktionen ist die Beschreibung von Abhängigkeiten zwischen den Funktionsparametern. Auf diese Weise lässt sich die Ausführungsreihenfolge und ihre mögliche Parallelisierung automatisch durch eine topologische Sortierung festlegen. Das Ergebnis der Funktionsintegration, also das Ergebnis der Ausführung einer globalen Funktion, lässt sich als abstrakte Tabelle auffassen und kann so in konzeptionell einfacher Weise dem FDBS verfügbar gemacht werden, so dass aus Sicht des FDBS das Problem

zu einer reinen Datenintegration geworden ist. Nach dieser Abstraktion kann die technische Kopplung der Ausführungskomponente zur Integration von Funktionen (KIF) über die Wrapper-Technologie erfolgen. Dabei waren jedoch noch wichtige Fragen zur Anfrageoptimierung und Transaktionsunterstützung zu lösen.

K. Hergula hat sehr ausführlich die Möglichkeiten der Anfrageoptimierung untersucht, d.h., welche Operationen bereits in der KIF abgewickelt werden können, um so beispielsweise das FDBS zu entlasten und das Volumen des Datentransports zum FDBS zu minimieren. Ein neuartiges Kostenmodell beschreibt die anfallenden Anteile der Anfrageverarbeitung in den verschiedenen Komponenten und erlaubt so die Wirksamkeit der einzelnen Optimierungsschritte abzuschätzen.

Die Transaktionsverwaltung im Kontext der Arbeit wird besonders wichtig, wenn über den Mechanismus der Funktionsintegration die beteiligten Datenquellen auch aktualisiert werden sollen. Die bekannten Transaktionskonzepte greifen bei dieser Problemstellung und den Anforderungen der praktischen Anwendungen zu kurz, da für FDBS und KIF mit angeschlossenen Anwendungssystemen bisher noch kein geeignetes Modell gefunden wurde. Insbesondere stellen in dieser Umgebung globale Serialisierbarkeit, globale Atomarität und globale Deadlock-Erkennung offene und schwierig zu lösende Probleme dar. Einen Hinweis auf eine mögliche Lösung fand K. Hergula in der Dissertation von W. Schaad zur Transaktionsverwaltung in heterogenen, föderierten Datenbanksystemen. Darin wurde das Modell der Mehrebenen-Transaktion, die „vorsichtig offen“ ist und als ausgezeichnetes Merkmal die Ebenen-zu-Ebenen-Serialisierbarkeit gewährleistet, auf heterogene Systeme hin erweitert. K. Hergula baut auf diese Konzepte auf und wendet das Konzept der Mehrebenen-Transaktion auf das FDBS und seine angegliederten DBS inklusive KIF an.

Die modellhaften Untersuchungen und Entwicklungen hat K. Hergula prototypisch in Integrationsarchitekturen umgesetzt. Dabei wurden drei Architekturvarianten für die KIF auf der Basis von UDTFs (user-defined table functions), von Workflows und von Microflows implementiert und empirisch auf ihr Leistungsverhalten hin untersucht. Da die einzelnen Realisierungsvarianten ein unterschiedliches Funktionsspektrum unterstützen, konnte sie Szenarien identifizieren, in denen jeweils eine der drei Varianten die beste Lösung darstellt.

Diese kompakte Diskussion des Inhalts der Arbeit soll den interessierten Leser anregen, sich im Detail mit Fragen der Daten- und Funktionsintegration auseinanderzusetzen. Er wird in der vorliegenden Abhandlung nicht nur Ergebnisse von theoretischer Bedeutung finden, sondern viele systemnahe Konzeptdiskussionen und praktische Lösungen zum immer wichtiger werdenden Thema der Informationsintegration. Kurz zusammengefasst, alle wichtigen Aspekte insbesondere zur Funktionsintegration auf der Basis von Föderierten Datenbanksystemen werden in dieser Dissertation erörtert.

Kaiserslautern, November 2003 Theo Härder

Danksagung

Diese Arbeit entstand während meiner Tätigkeit als Doktorandin in der Forschungsabteilung „Prozesskette Produktentwicklung“ (FT3/EK) und später als Mitarbeiterin im Bereich Technologiemanagement der DaimlerChrysler AG sowie als Promotionstudentin in der Arbeitsgruppe „Datenbank- und Informationssysteme“ von Prof. Dr. Dr. Theo Härder an der Technischen Universität Kaiserslautern. Während dieser Zeit haben mich viele Personen unterstützt und begleitet, denen ich an dieser Stelle noch einmal ausdrücklich danken möchte.

Mein besonderer Dank gilt meinem Doktorvater und Mentor Prof. Dr. Dr. Theo Härder für seine Bereitschaft, meine Promotion zu betreuen und das bearbeitete Thema mitzugestalten. Trotz der großen geographischen Entfernung ermöglichte er eine intensive Zusammenarbeit, die durch großen Freiraum auf der einen Seite und richtungsweisende Ratschläge auf der anderen Seite geprägt war. Ein Bild ist mir seitdem immer vor Augen: Wenn ich an einem Punkt gelangt war, an dem ich nicht mehr weiterkam, stieß Prof. Härder in den Diskussionen die Tür zu einem Raum auf, in dem viele neue Ideen auf mich warteten. Dafür danke ich ihm von ganzem Herzen. Prof. Dr. Wolfgang Benn danke ich, dass er sich so kurzfristig bereit erklärte, die zweite Berichterstattung zu übernehmen. Er nahm eine nicht unerhebliche zeitliche Belastung auf sich und trug zum Gelingen des Verfahrens bei.

Ebenso geht mein Dank an Prof. Dr. Frank Leymann für seine begeisternden Diskussionen und seine Unterstützung hinsichtlich Flow-Technologien sowie an Prof. Dr. Bernhard Mitschang, Prof. Dr. Norbert Ritter und Prof. Dr. Stefan Deßloch für Ihre Anregungen und Ideen in unseren Doktoranden-Workshops.

Außerdem danke ich meinem ehemaligen Kollegen bei DaimlerChrysler Dr. Günter Sauter, der mich überzeugte, diese Arbeit anzugehen. Er unterstützte meine Arbeit und diskutierte meine Ideen kritisch mit mir. Meinen ehemaligen und derzeitigen Vorgesetzten Peter Schneider, Robert Winterstein, Dr. Hans-Joachim Dilling und Manfred Abrecht möchte ich für ihre Unterstützung durch die Gewährung des nötigen wissenschaftlichen Freiraums und ihr Vertrauen in meine Arbeit meinen Dank aussprechen.

Den früheren Studenten Markus Vogt, Klaus Wissmann, Niels Lisdorf und Ekkehard Steiss und den früheren und derzeitigen DaimlerChrysler-Kollegen Dr. Fernando de Ferreira Rezende, Dr. Jürgen Sellentin, Jochen Rütschlin, Dr. Hans-Peter Steiert, Gregor Lorenz, Ulrich Schäfer, Roland, Nagel, Ralf Sauter, Andreas Almer und Andreas Ulrich möchte ich für ihre Beiträge zu dieser Arbeit und für die gute Zusammenarbeit sehr danken. Den Kollegen der Arbeitsgruppe in Kaiserslautern und insbesondere Markus Bon danke ich, dass sie mir als „Externe“ immer das Gefühl gaben, eine von ihnen zu sein.

Schließlich möchte ich meinen Eltern danken, die mir ermöglichten, diesen Weg zu beschreiten. Insbesondere danke ich meiner Mutter, die mir in allen Lebenslagen zur Seite stand. Besonderer Dank gilt auch Jochen Rütschlin. Die Gestaltung unserer Freizeit und Urlaubszeit wurde durch mein Promotionsvorhaben stark beeinflusst.

Göppingen, November 2003 Klaudia Hergula

Inhaltsverzeichnis

Abkürzungsverzeichnis

2PC	Two Phase Commit
2PL	Two Phase Locking
A-UDTF	Access User-Defined Table Function
ACID	Atomicity, Consistency, Isolation, Durability
AOS	Abort of Subtransaction
AOT	Abort of Transaction
API	Application Programming Interface
BAPI	Business API
BOS	Begin of Subtransaction
BOT	Begin of Transaction
CORBA	Common Object Request Broker Architecture
COS	Commit of Subtransaction
COT	Commit of Transaction
DB	Datenbank
DBMS	Datenbankmanagementsystem
DBS	Datenbanksystem
DDL	Data Definition Language
DOM	Document Object Model
DTD	Document Type Definition
E/A	Eingabe/Ausgabe
EAI	Enterprise Application Integration
EDV	Elektronische Datenverarbeitung
FDBS	Föderiertes Datenbanksystem
FIX	Function Integration in XML
GST	Globale Subtransaktion
GT	Globale Transaktion
HTML	HyperText Markup Language
I-UDTF	Integration User-Defined Table Function
I/O	In/Out
ID	Identifier
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
J2EE	Java 2 Enterprise Edition
JSF	Join-Selektivitätsfaktor
JVM	Java Virtual Machine
KIF	Komponente zur Integration von Funktionen
LDBS	Lokales DBS

LT ..... Lokale Transaktion
MDBS ..... Multidatenbanksystem
MOM ..... Message-Oriented Middleware
ODMG ..... Object Database Management Group
OID ..... Object Identifier
OMG ..... Object Management Group
ORB ..... Object Request Broker
PDM ..... Produktdatenmanagement
PF ..... Projektionsfaktor
RMI ..... Remote Method Invocation
RPC ..... Remote Procedure Call
SAX ..... Simple API for XML
SF ..... Selektivitätsfaktor
SGML ..... Standard Generalized Markup Language
SOAP ..... Simple Object Access Protocol
SQL ..... Structured Query Language
SQL/MED ..... SQL/Management of External Data
SQL/PSM ..... SQL/Persistent Stored Modules
TAV ..... Transaktionsverwaltung
TP ..... Transaction Processing
UDDI ..... Universal Description, Discovery, and Integration
UDF ..... User-Defined Function
UDTF ..... User-Defined Table Function
URI ..... Uniform Resource Identification
W3C ..... World Wide Web Consortium
WfMC ..... Workflow Management Coalition
WfMS ..... Workflow-Managementsystem
WPDL ..... Workflow Process Definition Language
WSDL ..... Web Services Description Language
WWW ..... World Wide Web
XML ..... eXtensible Markup Language
XPDL ..... XML Process Definition Language
XSL ..... eXtensible Stylesheet Language
XSLT ..... eXtensible Stylesheet Language Transformation

Zusammenfassung

Unternehmen sind zunehmend mit der Integration ihrer Systeme beschäftigt, um alle ihre Informationen gemeinsam verarbeiten zu können. Viele dieser Systeme erlauben den Zugriff auf ihre Daten jedoch nicht über eine SQL-Schnittstelle. Stattdessen müssen Daten über bereitgestellte, vordefinierte Funktionen ausgelesen werden. Somit ist die reine Datenintegration nicht mehr ausreichend, sondern der Integrationsansatz muss um die Integration von Funktionen erweitert werden. Die vorliegende Arbeit stellt solch einen erweiterten Integrationsansatz vor, der auf existierenden Technologien und Standards aufbaut.

Zunächst widmet sich die Arbeit der reinen Funktionsintegration. Vergleichbar zur Datenintegration, werden bei der Funktionsintegration föderierte Funktionen auf lokale Funktionen abgebildet. Dabei werden die Heterogenitätsformen analysiert und klassifiziert, die bei der Funktionsabbildung auftreten können.

Der entwickelte Lösungsansatz teilt sich in ein Beschreibungs- und Ausführungsmodell, um die Definition der Integration unabhängig von der technischen Umsetzung zu halten. Die Abbildung von Funktionen wird durch Abhängigkeiten zwischen den Funktionsparametern beschrieben. Diese Vorgehensweise erlaubt eine einfache und einheitliche Definition unterschiedlichster Abbildungsszenarien. Die zugehörige Abbildungssprache basiert auf XML sowie verwandten Standards und erlaubt eine einfache Handhabung und Verarbeitung.

Das Ausführungsmodell bildet den Kern der vorliegenden Arbeit. Es wird eine Architektur zur Integration von Daten und Funktionen eingeführt, die aus einem föderierten Datenbanksystem und einer Komponente zur Funktionsintegration besteht. Diese Kombination erlaubt den integrierten Zugriff auf Daten und Funktionen. Die technische Kopplung erfolgt über die Wrapper-Technologie. Weitere Beiträge im Ausführungsmodell sind die Untersuchung der heterogenen Anfrageoptimierung und Transaktionsverwaltung. Die Arbeit zeigt auf, wie Zugriffe auf Funktionen transparent in die relationale Welt eingebunden und Verbesserungen in der Anfrageoptimierung erreicht werden können. Darüber hinaus wird ein Transaktionsmodell entwickelt, das lokale Systeme einbindet, die von sich aus nicht an übergreifenden Transaktionen teilnehmen können. Das Ausführungsmodell schließt mit einer Untersuchung möglicher Implementierungen der Komponente zur Funktionsintegration und erläutert die Wahl eines Workflow-Systems.

Die Implementierung und Untersuchung von drei verschiedenen Architekturalternativen runden die Arbeit ab. Die Abbildungsmächtigkeit sowie das Leistungsverhalten der Prototypen werden gegenübergestellt und bewertet. Die Untersuchungen zeigen, dass alle Architekturvarianten abhängig von den Anforderungen sinnvoll eingesetzt werden können.

Kapitel 1

Einleitung

1.1 Motivation und Problemstellung

Unternehmen sind zunehmend mit immer schneller anwachsenden Mengen an Daten und Informationen konfrontiert. Jüngsten Studien zufolge weisen geschäftsrelevante Informationen eine jährliche Wachstumsrate von rund 50 Prozent auf, wobei ein bis zwei Exabytes (10^{18}) an Informationen generiert werden [VL03]. Ein großer Teil dieser Informationen fällt in sog. *Packaged Software* oder *Packaged Applications* an (wie z. B. SAP R/3 [SAP03]). Um alle relevanten Informationen aus den Daten in den verschiedensten Systemen zu bekommen, sind Unternehmen zunehmend mit der Integration dieser Pakete beschäftigt.

Auch die Integrationsthematik hat inzwischen mehrere Gesichter bekommen. Spielte in der Vergangenheit vor allem die Datenintegration eine große Rolle, so haben sich inzwischen aufgrund unterschiedlicher Anforderungen weitere Formen der Integration entwickelt [JMP02]: Die *Portal-Integration* führt unterschiedliche Anwendungen über eine Web-Seite zusammen. Die *Geschäftsprozessintegration* baut Prozesse über mehrere Anwendungen hinweg auf. Die *Anwendungsintegration* lässt Anwendungen miteinander kommunizieren. Die *Informationsintegration* schließlich führt heterogene Daten zusammen, so dass Anwendungen auf alle relevanten Unternehmensdaten zugreifen können.

Die Informationsintegration hat dabei zwei grundlegende Aspekte [LR02]: *Datenintegration* und *Funktionsintegration*. Die Daten aus heterogenen Quellen werden über gemeinsame Schnittstellen und integrierte Schemata zugänglich gemacht, so dass sie dem Benutzer wie eine einzige Datenquelle erscheinen. Im Bereich der heterogenen Datenbanken gibt es bereits seit einigen Jahren zahlreiche Forschungsarbeiten. Im Mittelpunkt steht hierbei die Unterstützung von Interoperabilität zwischen heterogenen Datenbanksystemen. Dabei wurden Konzepte und Prototypen sog. Föderierter Datenbanksysteme (FDBS) und Multidatenbanksysteme (MDBS) entworfen, um Datenbanken mit unterschiedlichen Datenmodellen und Schemastrukturierungen integrieren zu können. Mittlerweile sind auch kommerzielle Produkte verfügbar, die als Datenbank-Gateways oder Datenbank-Middleware bezeichnet werden [RH98]. Für die essentiellen Probleme auf dem Gebiet der Datenbankintegration liegen somit inzwischen mächtige Lösungen vor, wenn auch einige Fragen noch offen sind [BE96, Con97, HBP94, Kim95, SL90].

Das Bild der heterogenen Datenbanklandschaft beginnt sich jedoch zu ändern. Haben sich die Unternehmen bisher bewusst für ein DBS entschieden und auch das Datenbankschema selbst festgelegt, so kommt es heute immer häufiger vor, dass eine Datenbank innerhalb eines Standardsoftware-Pakets mitgeliefert wird. Dabei werden das DBS sowie das zugehörige Anwendungsprogramm zusammengefasst und nach außen hin lediglich eine Programmierschnittstelle, ein sog. API (*Application Programming Interface*), bereitgestellt. Eine Datenbankschnittstelle ist somit nicht mehr verfügbar. Systeme, die dieses Konzept der Kapselung realisieren, werden im Folgenden *Anwendungssysteme* genannt. Hervorzuhebende Stellvertreter für Anwendungssysteme sind z. B. SAP R/3 [SAP03] oder PDM (Produktdatenmanagement)-Systeme wie Metaphase [SDR03]. Im Falle von SAP können demnach die darin verwalteten Daten nicht direkt mittels SQL aus der relationalen Datenbank ausgelesen werden. Stattdessen stellt SAP sog. BAPIs (*Business APIs*) zur Verfügung, die dem Benutzer den Datenzugang über vordefinierte Funktionen erlauben. Neben diesen kommerziellen Produkten gibt es häufig proprietäre, von den Unternehmen selbst implementierte Software-Lösungen, die ausschließlich über ein API zugänglich sind. Dies ist in der Tatsache begründet, dass die Einhaltung von Integritätsbedingungen als auch die Überwachung der Sicherheit (Autorisierung) in vielen Fällen im Anwendungsprogramm realisiert und nicht durch das DBS unterstützt werden. Überdies mangelt es oft an konzeptionellen Schemata und somit an ausdrucksstarken Bezeichnern. Um zu vermeiden, dass Schemaänderungen eventuell die Konsistenz, die Integrität und den Schutz der Daten gefährden, ist der Zugang zu den Daten (und zu der von der Anwendung bereitgestellten Funktionalität) nur über ein API gestattet. Deshalb besteht die Notwendigkeit, nicht nur die Datenbank- bzw. Schema-Integration, sondern in Ergänzung dazu auch die Anwendungssystem- bzw. Funktionsintegration zu unterstützen.

Ähnlich der Datenintegration soll die Integration von Funktionen lokale Funktionen verschiedener Systeme in einheitlicher Form bereitstellen. Dem Benutzer wird so eine homogene Menge an Funktionen an die Hand gegeben, über die er die Daten manipulieren kann, die von den verschiedenen Anwendungen gekapselt werden. Möchte ein Unternehmen alle relevanten Daten integriert zugängig machen, unabhängig davon, wie auf sie zugegriffen werden kann, so müssen Daten *und* Funktionen integriert werden. Im Gegensatz zur Datenintegration sind im Bereich der Funktionsintegration als auch der Kombination beider Integrationsformen bisher nur wenige Ansätze bekannt. Daher bildet die Funktionsintegration einen Schwerpunkt in der vorliegenden Arbeit.

Das folgende Beispiel soll verdeutlichen, welche Vorteile Funktionsintegration bietet. Es zeigt, wie Benutzer heutzutage mit Anwendungssystemen arbeiten. Unser Beispielszenario ist in der Einkaufsabteilung eines Unternehmens angesiedelt, findet sich aber in ähnlicher Form auch in jedem anderen Bereich. In dieser Abteilung hat ein Mitarbeiter zu entscheiden, ob ein neues Produkt eines dem Unternehmen bereits bekannten Zulieferers bestellt werden soll. Ein Einkaufssystem soll dem Mitarbeiter mit der Funktion `KaufEntscheid()` bei seiner Entscheidung helfen. Diese Funktion schlägt eine Entscheidung zum Kauf vor, die auf einem berechneten Qualitäts- und Zuverlässigkeitsgrad und der Nummer der zu betrachtenden Komponente basiert. Leider kennt der Mitarbeiter nur den Komponentennamen und die Nummer des Zulieferers. Da ihm die benötigten Eingabewerte nicht zur Verfügung stehen, muss der Mitarbeiter weitere Systeme heranziehen, um diese Werte zu ermitteln. Abbildung 1.1 illustriert die einzelnen Schritte,

die er dafür ausführen muss. Ausgehend von den Werten, die ihm bekannt sind, ruft er zunächst die Funktion `GetQualitaet()` des Lagerhaltungssystems und die Funktion `GetZuverlaessigkeit()` des Einkaufssystems jeweils mit der Zulieferernummer auf, um Qualität und Zuverlässigkeit des Zulieferers zu erfragen. Die resultierenden Ergebnisse werden anschließend als Eingabe für die Funktion `GetBewertung()` zur Berechnung des Grades eingesetzt, um den ersten Eingabewert der Funktion `KaufEntscheid()` zu erhalten. Außerdem ruft der Mitarbeiter die Funktion `GetKompNr()` des PDM-Systems zur Abfrage der zugehörigen Komponentennummer auf. Nun stehen ihm die Werte zur Verfügung, die für den Aufruf der Funktion `KaufEntscheid()` benötigt werden.

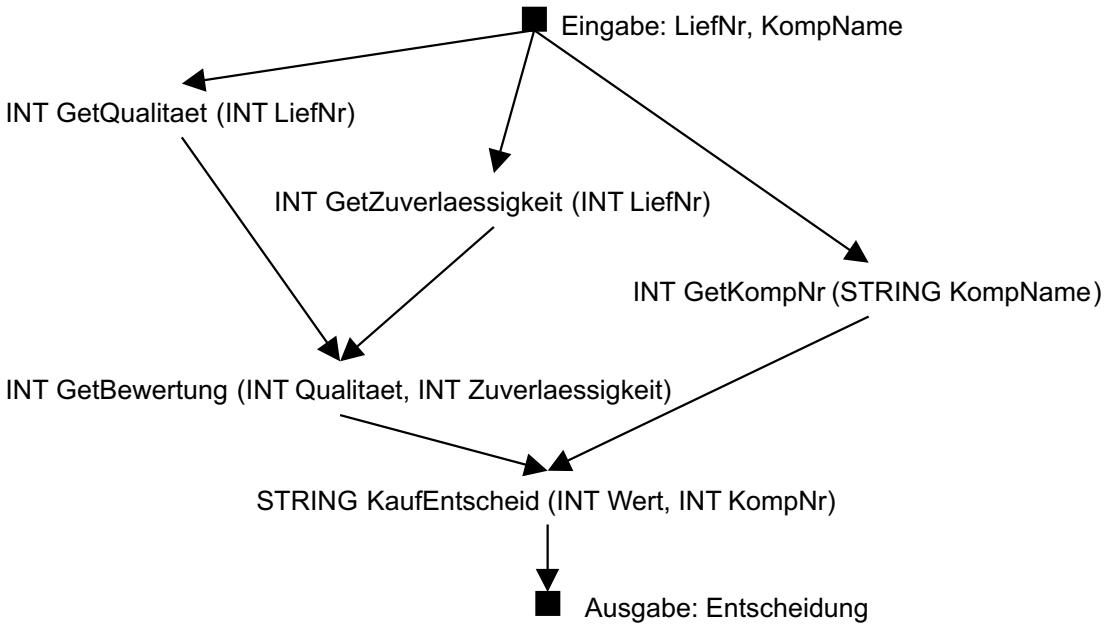

Abbildung 1.1: Der Ablauf der einzelnen Schritte des Benutzers bei der Arbeit mit mehreren Anwendungssystemen.

Während des Entscheidungsprozesses muss der Mitarbeiter mit drei unterschiedlichen Anwendungssystemen und demzufolge mit drei verschiedenen Benutzerschnittstellen arbeiten. Aus technischer Sicht setzt er von Hand eine Art Integration um, indem er die Funktionen der Anwendungssysteme aufruft und deren Ergebnisse zwischen den einzelnen Systemen hin- und herkopiert. Die Interaktion des Benutzers stellt daher die Verknüpfung der Anwendungssysteme dar. Ein solches Verfahren kostet den Benutzer aber zum einen sehr viel Zeit und zum anderen ist es fehleranfällig. Als typisches Beispiel hierfür mag dienen, dass oft alte Werte im Zwischenspeicher versehentlich als Eingabe in ein anderes System weitergereicht werden. Außerdem werden bestimmte Schritte immer wieder in derselben Reihenfolge durchgeführt. Diese Beobachtung führte zu der Idee, so genannte *föderierte Funktionen* anzubieten, welche die einzelnen Aufrufe der lokalen Funktionen der Anwendungssysteme implementieren und diese Schritte vor dem Benutzer verbergen. In unserem Beispiel muss der Benutzer lediglich eine föderierte Funktion `KaufLiefKomp()` statt der fünf lokalen Funktionen aufrufen.

Um diese Systemhilfe zu bieten, ist eine Integration von Funktionen bzw. Anwendungssystemen zu unterstützen. Da auch DB-Referenzen in einer Benutzeranfrage enthalten sein können, benötigt man sogar einen kombinierten Ansatz für den Daten- und Funktionszugriff. Hierbei sind mehrere Integrationsszenarien denkbar. Man kann den kombinierten Zugriff mit einem reinen Daten- oder einem reinem Funktionsintegrationsansatz angehen. Bei der reinen Datenintegration werden ausschließlich SQL-Datenquellen zusammengeführt, so dass die bekannten Konzepte der FDBS und MDBS greifen. Bei der reinen Funktionsintegration sind die zu integrierenden Anwendungssysteme in der Regel auf eine festgelegte Funktionalität zugeschnitten. Folglich sind keine lokalen Schemata verfügbar. Außer den nach außen offen gelegten Funktionen sind keine weiteren Informationen über die Daten vorhanden. Da solche Funktionen nicht die Datenunabhängigkeit und die Flexibilität bieten, die eine deklarative DB-Anfragesprache auszeichnen, ist auch ihre Nutzung eingeschränkt, d. h., Ad-hoc-Anfragen sind nicht möglich. Typisch hierfür ist das vorgestellte Anwendungsbeispiel.

Sollen nun Datenbank- und Anwendungssysteme in einer Architektur integriert werden, so ist die reine Datenintegration nicht mächtig genug. Stattdessen ist ein Lösungsansatz wünschenswert, der zusätzlich eine Funktionsintegration unterstützt. Über die Funktionsintegration werden föderierte Funktionen bereitgestellt, die mit den integrierten Daten aus dem DBS zusammengeführt werden können. Ein solcher Ansatz könnte eine spezielle Ausprägung der Informationsintegration sein, die den integrierten Zugriff auf alle relevanten Unternehmensdaten ermöglicht – unabhängig von der lokal bereitgestellten Schnittstelle.

1.2 Zielsetzung und Gliederung der Arbeit

Ziel dieser Arbeit ist die Entwicklung eines Integrationsansatzes, der neben der Datenintegration die Integration von Funktionen unterstützt. Vergleichbar mit einem föderierten Schema bei der Integration von Daten soll die Definition und Umsetzung föderierter Funktionen ermöglicht werden. Diese föderierten Funktionen werden auf lokal bereitgestellte Funktionen abgebildet und nutzen deren Funktionalität zur Ermittlung der angeforderten Daten. Darüber hinaus soll in globalen Anfragen die gemeinsame Auswertung von Daten und Funktionen unterstützt werden.

Implementierungen von individuellen Integrationslösungen haben in der Vergangenheit gezeigt, dass die Umsetzungen in den meisten Fällen sehr spezifisch sind und über kaum oder gar keine Dokumentation verfügen. Folglich sind diese Systeme nur schwer und aufwendig zu warten und an neue Anforderungen anzupassen. Daher streben wir einen generischen Ansatz an, der die Beschreibung der Integration von ihrer technischen Umsetzung trennt. Unser Lösungsansatz ist deshalb in ein Beschreibungs- und ein Ausführungsmodell aufgeteilt.

Im Fokus des Beschreibungsmodells liegt die Funktionsintegration, da hier die neuen Aspekte zu finden sind. Um die Abbildung föderierter Funktionen auf lokale Funktionen generisch beschreiben zu können, benötigen wir eine Abbildungssprache. Diese soll aber losgelöst von den zu beschreibenden Anwendungssystemen und ihren bereitgestellten Funktionen sein. Unabhängig von den in den lokalen Systemen eingesetzten Technologien (wie z. B. die Programmiersprache) soll die Beschreibung immer in derselben

Form möglich sein. Darüber hinaus sollte das Konzept der Abbildungssprache einfach nachvollziehbar und die Sprache selbst an bestehenden Standards ausgerichtet sein. Was man alles in einer solchen Abbildungssprache beschreiben können sollte, ermitteln wir durch eine Analyse der zu überwindenden Heterogenitätsformen bei der Integration von Funktionen.

Das Ausführungsmodell beinhaltet die Integrationsarchitektur, die schließlich die beschriebene Abbildung technisch realisiert. Eines der wichtigsten Ziele dieser Arbeit ist die Orientierung an bestehenden Technologien und Standards. Der Einsatz und die Anpassung existierender Produkte ist einer Neuimplementierung so weit wie möglich vorzuziehen. Die Umsetzung neuer Aspekte, die nicht vorhandene Funktionalität erfordern, sollte auf jeden Fall auf Standards aufbauen. Diese Vorgaben sind wichtig, um dem Einsatz unseres Ansatzes in der Industrie eine Chance zu geben. Nur wenn Produkte eingesetzt werden, bekommt man professionellen Support seitens des Herstellers, um beispielsweise den reibungslosen Betrieb sicherzustellen. Ähnliches gilt für den Einsatz von Standards. Nur wenn neue Konzepte auf Standards basieren, besteht die Möglichkeit, dass Hersteller diese in ihre Produkte aufnehmen.

Aus diesem Grund haben wir beschlossen, dass für die Datenintegration ein FDBS Teil unserer Integrationsarchitektur sein soll. Mehrere Gründe sprechen für diese Wahl. Mit einem FDBS setzen wir eine bestehende Technologie und einen Standard ein – SQL. Vor allem mit dem Einsatz von SQL an der globalen bzw. föderierten Schnittstelle unterstützen wir den Investitionsschutz in einem Unternehmen. Denn die meisten Entwickler kennen sich bereits mit SQL aus und auch die meisten datenverarbeitenden Produkte können die SQL-Schnittstelle bedienen. Außerdem geht es in erster Linie um die Integration und damit Verarbeitung von Daten, auch wenn sie über Funktionen abgerufen werden. Daher scheint ein DBS die am besten geeignete Ausführungskomponente zu sein, denn es verarbeitet Daten in effizienter und sicherer Weise. Doch wie sieht die Umsetzung der reinen Funktionsabbildung aus? Hier scheinen FDBS und SQL nicht die besten Kandidaten zu sein. Stattdessen suchen wir nach passenderen Technologien. Es soll die Frage untersucht werden, welche Alternativen für welche Anforderungen denkbar sind. Anhand einer der Technologien – in unserem Fall einem Workflow-Managementsystem (WfMS) – sollen weitere Aspekte des Ausführungsmodells betrachtet werden. Schließlich müssen FDBS und WfMS miteinander gekoppelt werden. Wie sieht dieser Kopplungsmechanismus aus und welche Funktionalität sollte er bereitstellen?

Auf diese und weitere Fragen und Anforderungen geht die vorliegende Arbeit ein. Sie ist wie folgt gegliedert.

Kapitel 2 betrachtet die verschiedenen Heterogenitätsformen, die bei einer Integration überwunden werden müssen. Es werden kurz die Heterogenitäten bei der Datenintegration wiederholt und anschließend die Heterogenitätsformen untersucht, die bei der Funktionsintegration und dabei insbesondere bei der Abbildung von föderierten Funktionen auf lokale Funktionen aufkommen. Hierzu haben wir eine Klassifikation mit aufsteigender Komplexität der einzelnen Heterogenitätsfälle aufgestellt. Das Kapitel schließt mit der Aufstellung weiterer Heterogenitäten, die bei der kombinierten Daten- und Funktionsintegration auftreten können.

Kapitel 3 gibt eine Einführung in all jene Themen, die grundlegend für das Verständnis der nachfolgenden Kapitel sind. Zu den behandelten Technologien gehören FDBS und

dabei insbesondere die Aspekte der verteilten und heterogenen Anfrageverarbeitung und Transaktionsverwaltung. Daran schließt eine Beschreibung der wichtigsten Eigenschaften von SQL:1999 an, wobei wir uns auf SQL-Funktionalität konzentrieren, die uns bei der Kopplung der beiden Komponenten unserer Integrationsarchitektur unterstützen können. Es folgt eine Einführung in Workflow-Systeme sowie XML und zugehörigen Standards. Das Kapitel schließt mit der Untersuchung verwandter Ansätze, wobei die Ansätze aufgeteilt sind in Lösungen für Datenintegration, Funktionsintegration und deren Kombination. Neben Arbeiten aus der Forschung betrachten wir auch Lösungen aus der Industrie und zeigen bestehende Standards auf.

In Kapitel 4 gehen wir auf den ersten Teil unseres Lösungsansatzes ein, dem Beschreibungsmodell. Dazu erläutern wir zunächst unser Konzept zur Beschreibung von Funktionsabbildungen, das auf der Definition von Parameterabhängigkeiten basiert. Das Ergebnis sind gerichtete, azyklische Graphen, die wir anhand einer selbst entwickelten Abbildungssprache festlegen können. Diese Sprache entwickeln wir auf Basis von XML, um eine leichtgewichtige und einfache Abbildungssprache zu erhalten.

Eingehende Betrachtungen zum Ausführungsmodell schließen sich in Kapitel 5 an. Wir untersuchen und bewerten zunächst mögliche Implementierungen des Kopplungsmechanismus von FDBS und WfMS. Anschließend beschreiben wir neue Aspekte der Anfrageoptimierung, wenn Funktionsaufrufe berücksichtigt werden müssen. Hierzu untersuchen wir, welche zusätzliche Funktionalität im Wrapper als Kopplungsmechanismus enthalten sein sollte und erarbeiten ein entsprechendes Kostenmodell. Ein weiterer wichtiger Punkt ist die Unterstützung eines erweiterten Transaktionsmodells, das verteilte Transaktionen über heterogene Systeme hinweg ermöglicht. Besonderer Knackpunkt sind hier die Anwendungssysteme, die in den meisten Fällen nicht an einer verteilten Transaktion teilnehmen können. Es folgt eine Diskussion möglicher Implementierungen des Ausführungsmodells. Dazu untersuchen und bewerten wir verfügbare Technologien und deren Einsatzmöglichkeiten und erläutern unsere Wahl – den Einsatz eines WfMS.

Nachdem wir uns für eine Integrationsarchitektur auf Basis von FDBS und WfMS entschieden haben, betrachten wir in Kapitel 6 alternative Implementierungen, in welchen Workflows durch Microflows bzw. benutzerdefinierte Tabellenfunktionen ersetzt werden. Anhand der in Kapitel 2 eingeführten Klassifikation von Heterogenitätsfällen werden föderierte Funktionen in drei Prototypen implementiert: Funktionsintegration auf Basis von Workflows, Microflows und Tabellenfunktionen. Wir untersuchen und vergleichen die Prototypen hinsichtlich der unterstützten Abbildungsmächtigkeit und der Performanz.

Die zentralen Ergebnisse der Arbeit werden abschließend in Kapitel 7 zusammengefasst. Darüber hinaus geben wir einen Ausblick auf zukünftige Themen und Entwicklungen in diesem Arbeitsgebiet.

Kapitel 2

Formen der Heterogenität

In der Einleitung haben wir erläutert, dass die angestrebte Architektur nicht nur Datenbank-, sondern auch Anwendungssysteme integrieren soll. In der Vergangenheit hat sich gezeigt, dass bereits bei der reinen Datenintegration unterschiedlichste Formen der Heterogenität auftreten können. Sollen darüber hinaus Funktionen integriert werden, so fächert sich das Spektrum an Heterogenitätsformen weiter auf. In den folgenden Kapiteln rekapitulieren wir zunächst Heterogenitätsformen der Datenintegration und analysieren anschließend neue Heterogenitätsaspekte, die mit der Integration von Anwendungssystemen und ihren Funktionen hinzukommen.

2.1 Heterogenitätsformen bei der Integration von Daten

Bei der Integration von Daten bzw. Datenbanksystemen können unterschiedliche Heterogenitätsformen in unterschiedlichen Schichten auftreten. Beginnt man bei der Plattform des DBS, so können sich Hardware, Betriebs- und Netzwerksystem unterscheiden. Betrachtet man die darüber liegenden Schichten, so können sich die DBS in ihrem Typ und ihrer Funktion und die darauf basierenden Anwendungen beispielsweise in der verwendeten Programmiersprache unterscheiden. All diese Heterogenitätsformen sollen jedoch nicht im Fokus der folgenden Abschnitte stehen. Stattdessen wird auf Konflikte eingegangen, die durch verschiedenartige Datenmodelle, unterschiedliche Schemata für denselben Sachverhalt oder auch sich widersprechende oder fehlende Daten entstehen. Mit dem Auftreten dieser Heterogenitäten ist zu rechnen, wenn zusammengehörige Daten zusammengeführt werden sollen, die in verschiedenartigen Datenmodellen oder auch unterschiedlichen Schemata modelliert sind. Grundsätzlich kann zwischen zwei Heterogenitätsformen unterschieden werden: der *semantischen* und der *strukturellen Heterogenität* [Sau98].

2.1.1 Semantische Heterogenität

Semantische Heterogenitätsformen findet man in erster Linie auf der Ebene der Schemata vor, da hier beim Entwurf anwendungsspezifisches Wissen einfließt und man nicht

davon ausgehen kann, dass ein gemeinsames Verständnis der modellierten Informationen vorliegt. Gefördert wird diese Problematik durch schlechte oder gar nicht vorhandene Dokumentation oder durch fehlende gemeinsame Begriffsbildung. Dies kann beispielsweise durch unterschiedliche Ausbildung oder unterschiedliches Hintergrundwissen der einzelnen Entwerfer gefördert werden.

In [Sau98] wird eine Klassifikation vorgestellt, welche die semantische Heterogenität in Unklarheiten einzelner Begriffe (Begriffsungenauigkeit) und zusammenhängender Gebilde (Unschärfe komplexer Gebilde) unterteilt. Bei Begriffsungenauigkeiten unterscheidet man wiederum zwischen ausdrucksstarken und ausdrucksschwachen Bezeichnungen. Ausdrucksstarke Bezeichnungen beinhalten Homonyme und Synonyme. In beiden Fällen liegt ein gewisses Grundverständnis vor, die Interpretation ist aber nicht exakt. Bei ausdrucksschwachen Bezeichnungen hingegen kann die Bedeutung der Begriffe nicht erfasst werden, da ein nicht ausreichend oder überhaupt nicht definierter Begriff, eine nicht eindeutige Abkürzung oder etwa eine unbekannte Sprache vorliegt. Abbildung 2.1 gibt einen Überblick über die Klassifikation der semantischen Heterogenität.

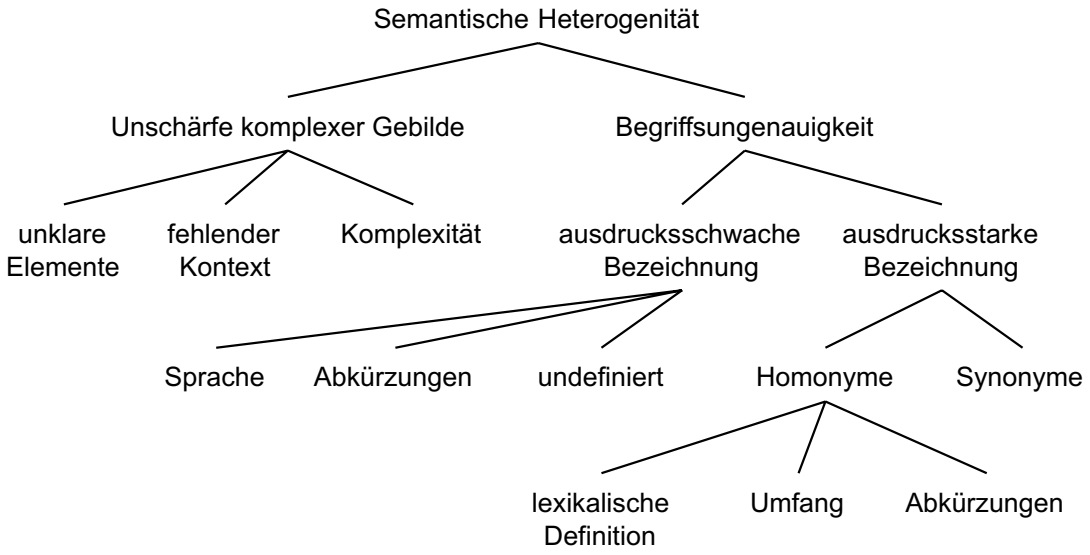

Abbildung 2.1: Klassifikation semantischer Heterogenität [Sau98].

Begriffsungenauigkeiten können geklärt werden, wenn sie im Kontext klarer Bezeichnungen auftreten. Problematischer ist hingegen das Verstehen von komplexen Gebilden aufgrund ungenauen Verständnisses zentraler Begriffe, fehlendem Kontext oder der Unüberschaubarkeit an sich.

Semantische Heterogenitäten können nicht automatisiert aufgelöst werden. Automatische Verfahren lassen sich maximal bei einfachen Problemen wie Begriffsungenauigkeiten als Unterstützung heranziehen. Um jegliche Missinterpretation der modellierten Informationen zu vermeiden, ist es unumgänglich, dass sich die involvierten Benutzergruppen die Bedeutung der Begriffe und damit der Daten erläutern und eine gemeinsame Begriffsdefinition erarbeiten.

2.1.2 Strukturelle Heterogenität

Strukturelle Heterogenität wird durch die unterschiedliche Mächtigkeit der Datenmodelle verursacht. Die Unterschiede ergeben sich beispielsweise durch vorhandene oder nicht vorhandene Unterstützung von Generalisierung, Klassifikation, Assoziation, Aggregation, komplexen Objekten usw. Die unterschiedliche Unterstützung führt zu unterschiedlicher Modellierung von Sachverhalten und dies wiederum führt zu verschiedenen Schemata und Instanzen. Sollen nun Daten aus heterogenen Systemen mit verschiedenen zugrunde liegenden Datenmodellen integriert werden, so müssen die strukturellen Heterogenitäten der Schemata überbrückt werden. Dabei geht man davon aus, dass ein Zielschema auf mehrere Quellschemata abgebildet wird. Alle Quellsysteme stellen einen Anwendungsbereich dar und die Abweichungen der einzelnen Darstellungen voneinander beeinflussen das Ausmaß der Heterogenität. Es können identische, teilweise überlappende oder auch disjunkte Anwendungsbereiche von den Systemen verwaltet werden. Ebenso spielt die unterschiedliche Repräsentation der Sachverhalte eine Rolle.

[Sau98] führt ein sehr detailliertes Schichtenmodell zur Klassifikation struktureller Heterogenität ein, das zahlreiche Faktoren aufführt, die zu Inkongruenzen bei der Abbildung eines Zielsystems auf ein Quellsystem führen. Um einen Überblick zu vermitteln, werden an dieser Stelle die wichtigsten Faktoren kurz beschrieben. Für eine tiefergehende Lektüre mit zahlreichen Beispielen verweisen wir den Leser auf [Sau98].

Zwei aufeinander abzubildende Systeme können folgende Unterschiede aufweisen, die in beliebiger Kombination zusammen auftreten können und im schwierigsten Fall sogar alle gemeinsam:

- Die Daten des Quellsystems sind semistrukturiert oder gar völlig unstrukturiert, d. h., ein formal beschriebenes (DB-)Schema liegt nicht vor.
- Die Schemata sind in heterogenen Datenmodellen beschrieben, z. B. relational und objektorientiert.
- Die Daten im Zielsystem werden persistent verwaltet. In diesem Fall muss die Konsistenz zwischen Quell- und Zieldaten überwacht werden. Dies gilt insbesondere für Änderungen.
- Ein Zielschema wird auf mehrere Quellschemata abgebildet ((1:n)-Verhältnis).
- Die beiden Systeme werden bidirektional verbunden, d. h., die Abbildungsdefinition wird in beiden Richtungen unterstützt.
- Es werden statische und dynamische Aspekte beider Systeme berücksichtigt, d. h., es existieren Funktionen, Trigger, spezifiziertes Verhalten usw.
- Die beiden Systeme beschreiben nicht den identischen Ausschnitt der (Mini-)Welt, sondern die Ausschnitte sind überlappend oder gar disjunkt.
- Es liegt eine partielle Replikation vor, wenn das Zielsystem beispielsweise nur eine Untermenge der Ausprägungen beinhaltet.
- Es werden unterschiedliche Bezeichner eingesetzt, so dass Probleme mit Synonymen und Homonymen auftreten können.

- Die aufeinander abzubildenden Objekte basieren auf unterschiedlichen Datentypen.
- Die Attribute der Schemata werden in einem (n:m)-Verhältnis abgebildet, so dass z. B. ein Attribut des Zielschemas nicht genau einem Attribut im Quellschema entspricht und andersherum.
- Auch die Objekte stehen in einem (n:m)-Verhältnis.
- Die Objekte des Quellschemas sind vernetzt, d. h., es liegen Primärschlüssel/Fremdschlüssel-Beziehungen oder Referenzen vor.
- Auch die Objekte des Zielschemas sind vernetzt, d. h., referentielle Integrität wird unterstützt.

Der Schwerpunkt dieser Analyse möglicher Konflikte zwischen Systemen liegt im Umfeld von DBS und betrachtet insbesondere Unterschiede in Datenmodellen, Schemata und Ausprägungen. In den folgenden Betrachtungen liegen diese Heterogenitätsformen weiter vor, doch kommen durch die starre Schnittstelle von Anwendungssystemen neue Aspekte hinzu.

2.2 Heterogenitätsformen bei der Integration von Funktionen

Sollen Anwendungssysteme statt DBS integriert werden, so muss analog zur Abbildung von Daten bzw. Schemata die Abbildung von Funktionen betrachtet werden. Ziel hierbei ist, eine föderierte Funktion oder auch Zielfunktion auf eine Sequenz von Funktionen der heterogenen Quellsysteme abzubilden, so dass vom Benutzer des integrierten Systems lediglich diese eine föderierte Funktion aufgerufen werden muss.

In den folgenden Abschnitten wird untersucht, welche Heterogenitäten bei einer Funktionsabbildung auftreten können. Dazu erläutern wir zunächst den Begriff der Funktion und die Abbildung von Funktionen an sich. Anschließend zeigen wir Heterogenitätsformen in ansteigender Komplexität auf.

2.2.1 Der Begriff der Funktion

Eine von einem Anwendungssystem zur Verfügung gestellte Schnittstelle besteht aus einer Menge von Funktionen. Der Begriff der Funktion kommt ursprünglich aus dem Bereich der Mathematik. Dort ist eine Funktion folgendermaßen definiert [MV93]:

Seien A und B zwei Mengen. Eine Funktion $f : A^n \subseteq D \mapsto B^m$ bzw. $x \mapsto f(x)$, ist eine Vorschrift, die jedem Vektor $x \epsilon D$ einen Vektor $f(x) \epsilon B^m$ zuordnet. Man bezeichnet dabei D als Definitionsbereich und B^m als Bildbereich.

Innerhalb von Programmiersprachen werden Funktionen eindeutig über ihre Signatur identifiziert. Die Signatur umfasst dabei den Namen der Funktion und die Datentypen sämtlicher Parameter sowie des optionalen Rückgabewerts. Die Parameter können dabei Eingabe- oder Ausgabeparameter oder auch beides sein. Der Rückgabewert ist

ein Spezialfall eines Ausgabeparameters. Die im folgenden Beispiel deklarierte Funktion trägt den Namen `GetKompNr()` und erhält den Eingabeparameter `KompName` vom Typ `STRING`. Der Ausgabeparameter liefert einen Wert vom Typ `INTEGER` zurück.

```
GetKompNr(IN KompName STRING, OUT KompNr INTEGER)
```
[1]

Neben den offensichtlichen Gemeinsamkeiten zu dem Funktionsbegriff der Mathematik ergeben sich einige Unterschiede. So ist es möglich, dass Funktionen in Programmiersprachen keine Eingabe- oder Rückgabeparameter besitzen. Des Weiteren können die Parameter einer Funktion verschiedene Datentypen besitzen.

Funktionen in Programmiersprachen können so genannte Seiteneffekte verursachen, d. h., sie haben die Möglichkeit, neben der Rückgabe von Werten auch den Zustand des Systems zu verändern. Beispiele hierfür sind der schreibende Zugriff auf eine Datenbank oder die Veränderung einer globalen Variable eines Systems (z. B. auch durch die Initialisierung oder die Anmeldung an einem System). Besitzt eine Funktion keine Rückgabeparameter, kann offensichtlich davon ausgegangen werden, dass sie Seiteneffekte verursachen wird.

Im Gegensatz zu Funktionen im mathematischen Sinn können Funktionen in der Informatik bei mehrfachem Aufruf mit denselben Parametern unterschiedliche Werte zurückgeben. Dies kann entweder durch Seiteneffekte innerhalb dieser oder anderer zwischenzeitlich ausgeführter Funktionen oder durch Änderungen „von außen“[2] an dem System verursacht worden sein.

2.2.2 Abbildung von Funktionen

Bevor wir die zu lösenden Heterogenitätsformen untersuchen, betrachten wir zunächst, was eine Funktionsabbildung darstellt. Vergleichbar zu einem globalen DB-Schema, das in der Datenintegration auf ein oder mehrere lokale Schemata abgebildet wird, sollen nun globale Funktionen auf lokale Funktionen abgebildet werden. Da es sich bei der Funktionsintegration letztlich um ein Pendant zur Datenintegration handelt, werden die globalen Funktionen föderierte Funktionen genannt, in Anlehnung an Föderierte Datenbanksysteme. Föderierte Funktionen implementieren nicht selbst eine Funktionalität, sondern leiten die Ausführung an bestehende Funktionen der lokalen Systeme weiter. Folglich enthält der Rumpf der föderierten Funktionen die Ausführungslogik und damit die Abbildungslogik auf die lokalen Systeme. Somit stellen föderierte Funktionen eine Art virtuelle Funktionen dar. Abbildung 2.2 zeigt eine Abbildung von einer föderierten auf eine lokale Funktion.

Die dargestellte Funktionsabbildung ist sehr einfach und bildet die föderierte Funktion `GetTeilNr()` auf die lokale Funktion `GetKompNr()` ab. Dabei sind von der Struktur her kaum Änderungen nötig, sondern es werden die Funktions- und Parameternamen sowie die Datentypen der Parameter aufeinander abgebildet. Solche Abbildungen machen

[1] Wir verwenden den Begriff der Funktion wie einen Eigennamen oder speziellen Mechanismus und setzen Funktions- und Prozedursignaturen gleichwertig ein. Wir werden zukünftig auch nicht zwischen Funktionen und Prozeduren unterscheiden, sondern sprechen allgemein von Funktionen.

[2] Dabei stellt die fortgeschrittene Uhrzeit zwischen zwei Funktionsaufrufen ebenfalls einen Änderung „von außen“ dar.

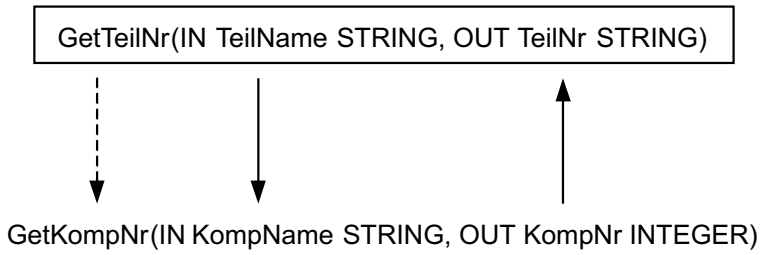

Abbildung 2.2: Abbildung einer Funktion.

beispielsweise Sinn, wenn auf föderierter Ebene andere Begrifflichkeiten vorliegen (z. B. Teil statt Komponente) und die modellierten Daten auf anderen Datentypen aufgebaut wurden (STRING statt INTEGER). Der gestrichelte Pfeil symbolisiert die Abbildung der Funktionen an sich, während die durchgezogenen Pfeile die detailliertere Abbildung der Funktionsparameter darstellen.

Bei der Funktionsintegration können beliebig komplexe Abbildungen auftreten. Die Komplexität und damit die Heterogenität steigt mit unterschiedlichen Namen, unterschiedlichen Datentypen, unterschiedlichen Signaturen bis hin zu Abbildungen auf mehrere lokale Funktionen. Die verschiedenen Ausprägungen von Heterogenität fassen wir in einer Klassifikation zusammen, die wir im nächsten Abschnitt vorstellen.

2.2.3 Klassifikation der Heterogenitätsformen

Im Folgenden klassifizieren wir die verschiedenen Fälle der Abbildung von Funktionen gemäß dem Ausmaß an Heterogenität. Gegeben seien eine Menge von föderierten Funktionen $\mathcal{F}$ und eine Menge von lokalen Funktionen $\mathcal{L}$. Die föderierten Funktionen sind paarweise verschieden, da sie zu einer Schnittstelle gehören. Lokale Funktionen hingegen können sowohl in ihrem Namen als auch in ihrer Struktur völlig identisch sein, da sie aus unterschiedlichen Systemen stammen können und eine Eindeutigkeit der Funktionen nicht gewährleistet werden kann.

In den folgenden Abschnitten wird die allgemeine Abbildung $\mathcal{F} \rightarrow \mathcal{L}$ also $f_1, \cdots, f_m \rightarrow l_1, \cdots, l_n$ von mehreren föderierten Funktionen auf mehrere lokale Funktionen betrachtet. Die lokalen Funktionen können von unterschiedlichen Systemen stammen. Bei Betrachtung der Abbildung kann man zwischen der Abbildung der Funktion und der Abbildung der Parameter unterscheiden. Die beschriebenen Fälle werden nach aufsteigender Komplexität der Abbildung besprochen.

2.2.3.1 Trivialer Fall

Der triviale Fall weist ein Minimum an Heterogenität auf. Es handelt sich um eine (1:1)-Abbildung, d. h., eine föderierte Funktion wird auf genau eine lokale Funktion abgebildet. Beide Funktionen sind hinsichtlich ihrer Funktionalität und Signaturstruktur vollkommen identisch. Lediglich die Namen der Funktionen bzw. der Parameter unterscheiden sich, so dass eine Umbenennung erfolgen muss. Abbildung 2.3 illustriert, wie die beiden Funktionen in Beziehung stehen. Beim Aufruf der föderierten Funktion wird in deren Rumpf die entsprechende lokale Funktion aufgerufen. Es sind bezüglich der

Parameterwerte keine speziellen Aufgaben notwendig, nur die Zuordnung von Funktion und Parametern muss stimmen.

Die beiden folgenden Funktionen stellen einen trivialen Fall dar:

f_1: `GetCompNo(IN CompName STRING, OUT CompNo INTEGER)`
l_1: `GetKompNr(IN KompName STRING, OUT KompNr INTEGER)`

Die Signaturen sind identisch. Es unterscheiden sich lediglich die Funktions- und Parameternamen.

2.2.3.2 Einfacher Fall

Der einfache Fall beinhaltet die Heterogenitäten des trivialen Falles und stellt ebenfalls eine (1:1)-Abbildung dar. Da nun unterschiedliche Signaturen möglich sind, kommen aufgrund der notwendigen Parameterabbildungen weitere Aspekte hinzu. In den folgenden Betrachtungen unterscheiden wir zwischen (1:1)-, (1:n)- und (n:1)-Abbildungen der Parameter.

(1:1)-Parameterabbildung

Im (1:1)-Fall ist zwar die Anzahl der Eingabe- und Ausgabeparameter identisch, doch können Datentyp und vor allem auch Semantik unterschiedlich sein. Stimmt die Semantik überein, so stellen unterschiedliche einfache Datentypen (wie integer, real, string usw.) der zugeordneten Parameter den einfachsten Fall dar. Hier reicht ein „Type Cast“, wie man ihn von Programmiersprachen her kennt, um die Heterogenität zu überbrücken.

Schwieriger wird es, wenn komplexe Datentypen vorliegen und beispielsweise eine Liste in eine Zeichenkette abgebildet werden muss. Ein Beispiel hierfür kann folgendermaßen aussehen: die lokale Funktion verarbeitet Namen in Listen mit zwei Feldern, die jeweils Vor- bzw. Nachname enthalten. Die föderierte Funktion hingegen verarbeitet Namen in einer Zeichenkette, die Vor- und Nachnamen enthält.

Weitere Heterogenität wird durch unterschiedliche Semantik verursacht. In solchen Fällen können sogar zusätzliche Schritte für z. B. Berechnungen notwendig werden. Es liegen folgende Funktionen vor:

f_1: `GetQualitaet(IN LiefNr INTEGER, OUT QualKlasse STRING)`
l_1: `GetQualitaet(IN LiefNr INTEGER, OUT QualNote INTEGER)`

In diesem Beispiel gibt die lokale Funktion l_1 die Qualitätsnote für einen Lieferanten zur gegebenen Lieferantennummer zurück. Die Bewertung der Lieferanten über die Qualitätsnote gleicht dem Schulnotensystem. Die föderierte Funktion f_1 soll hingegen eine Qualitätsklasse im Sinne einer Bewertung von „hoch“, „mittel“ und „niedrig“ liefern. Die Abbildung von Qualitätsnote auf Qualitätsklasse kann nicht mittels eines einfachen Cast umgesetzt werden, sondern muss durch eine Fallunterscheidung implementiert werden. Dabei müssen die Noten 1 und 2 auf die Klasse „hoch“, 3 und 4 auf „mittel“ und 5 und 6 auf „niedrig“ abgebildet werden.

Es zeigt sich, dass bei unterschiedlicher Semantik eine beliebig komplexe Abbildung notwendig sein kann und die Überwindung der Heterogenität dadurch enormen Aufwand verursacht.

(1:n)-Parameterabbildung

Die (1:n)-Abbildung deckt zwei Fälle ab: die Abbildung eines Eingabeparameters der föderierten Funktion auf mehrere Eingabeparameter der lokalen Funktion bzw. die Abbildung eines Ausgabeparameters der lokalen Funktion auf mehrere Ausgabeparameter der föderierten Funktion, d. h. den Abbildungspfeilen folgend. Es kommt nun zu den bisher bekannten Heterogenitäten hinzu, dass aus einem Parameterwert mehrere Werte für die Parameter der korrespondierenden Funktion erzeugt werden müssen.

Hierfür gibt es zwei Vorgehensweisen. Entweder können aus dem einen föderierten Eingabewert die benötigten Eingabewerte für die lokalen Eingabeparameter ermittelt werden, beispielsweise durch Berechnungen oder Fallunterscheidungen. Oder man arbeitet mit vorgegebenen Werten, da nicht alle Werte ermittelt werden können.

Der erste Weg funktioniert bei diesem Beispiel:

f_1: `GetAnsprechpartner(IN LiefNr INTEGER, OUT Vorname STRING,`
`OUT Nachname STRING)`
l_1: `GetAnsprechpartner(IN LiefNr INTEGER, OUT Name STRING)`

Die Quellfunktion l_1 gibt den Namen als eine zusammenhängende Zeichenkette zurück, während die föderierte Funktion Vor- und Nachnamen in zwei separaten Zeichenketten hält. Eine Abbildung wird mittels Zeichenkettenoperationen implementiert, mit welchen anhand vordefinierter Merkmale die Namensteile getrennt werden können. Beispielsweise markieren Leerzeichen die Trennung der einzelnen Namen. Technisch gesehen also kein Problem, doch ist dies nicht gezwungenermaßen eindeutig. Betrachtet man die beiden Namen „Ludwig van Beethoven“ und „Wolfgang Amadeus Mozart“, so ist hier die Aufteilung von Vor- und Nachnamen unterschiedlich und die Abbildung ist unter Umständen nicht richtig.

Bei dem folgenden Beispiel muss dagegen mit Default-Werten gearbeitet werden. Es liegen die folgenden Funktionen vor:

f_1: `GetAnzahl(IN KompNr, OUT Anzahl INTEGER)`
l_1: `GetAnzahl(IN LiefNr INTEGER, IN KompNr INTEGER, OUT Anzahl INTEGER)`

Die föderierte Funktion übergibt die Komponentennummer an die lokale Funktion, um die zugehörige Anzahl gelagerter Einheiten dieser Komponente zu ermitteln. Die lokale Funktion benötigt jedoch zusätzlich die Lieferantennummer, um das Ergebnis zu ermitteln. Aus der gegebenen Komponentennummer kann aber nicht immer die eindeutige Lieferantennummer geschlossen werden, da Komponenten von unterschiedlichen Herstellern geliefert werden. Es kann nun ein Default-Wert für die Lieferantennummer festgelegt werden, der an die lokale Funktion weitergereicht wird.

(n:1)-Abbildung

Die (n:1)-Abbildung stellt die Gegenrichtung des (1:n)-Falles dar. Es müssen Parameter nicht aufgeteilt, sondern zusammengefasst werden. Auch hier liegen wieder zwei Richtungen vor: Abbildung mehrerer Eingabewerte der föderierten Funktion auf einen Eingabeparameter der lokalen Funktion und die Abbildung mehrerer Ausgabewerte der

lokalen Funktion auf einen Ausgabeparameter der föderierten Funktion. Wie im umgekehrten Fall benötigt man Mechanismen, um die Parameterwerte in einem oder mehreren Zwischenschritten zu verarbeiten und die benötigten Werte zu berechnen.

Als Beispiel können die beiden Funktionen mit den Namen aus der (1:n)-Abbildung herangezogen werden. Möchte man die Abbildung umkehren, dann müssen die Zeichenketten für Vor- bzw. Nachname zu einer Zeichenkette konkateniert werden.

2.2.3.3 Unabhängiger Fall

Der unabhängige Fall stellt eine Erweiterung des einfachen Falles dar. Die föderierte Funktion wird nun auf mehrere lokale Funktionen abgebildet und damit liegt eine (1:n)-Funktionsabbildung vor. Die Bezeichnung „unabhängiger Fall“ ergibt sich aus der Tatsache, dass die lokalen Funktionen vollkommen unabhängig voneinander sind und damit parallel ausgeführt werden können (vgl. Abbildung 2.3 rechts). Folglich beeinflusst keine der lokalen Funktionen in irgendeiner Form eine andere lokale Funktion. Die Einzelergebnisse werden erst nach erfolgreicher Ausführung aller lokalen Funktionen für die Ausgabe der föderierten Funktion aufbereitet.

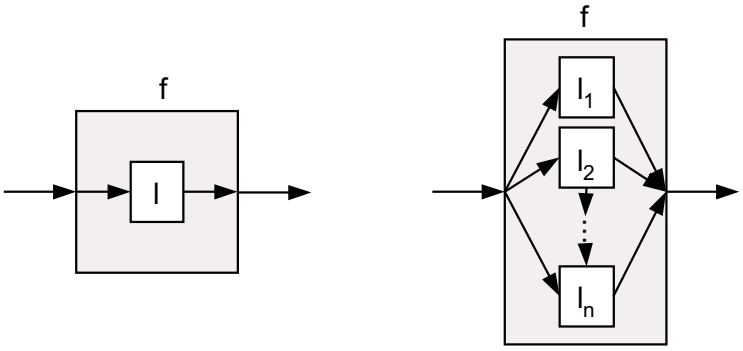

Abbildung 2.3: Der triviale/einfache (links) und der unabhängige Fall (rechts).

Da die föderierte Funktion auf mehrere lokale Funktionen aufgeteilt wird, müssen auch die Parameter über mehrere Funktionen verteilt werden. Da die Funktionen aus verschiedenen Systemen kommen, können sie gleiche Namen haben. Ähnliches gilt für die Parameternamen. Um eine eindeutige Zuordnung zu gewährleisten, müssen bei der Abbildung Parameter, Funktion und System angegeben werden. Darüber hinaus kommen aber keine neuen Heterogenitätsaspekte hinzu.

2.2.3.4 Abhängiger Fall

Auch dieser Fall ist eine (1:n)-Funktionsabbildung, bei der jetzt aber die lokalen Funktionen voneinander abhängig sind. Die Abhängigkeit entsteht in erster Linie durch Parameter, deren Eingabe in eine lokale Funktion von der Ausgabe einer anderen lokalen Funktion beeinflusst wird. Diese Abhängigkeit hat daher auch Einfluss auf die Ausführungsreihenfolge der Funktionen. Diese können deshalb auch nicht parallel ausgeführt werden. Stattdessen kann eine Funktion erst ausgeführt werden, wenn die be-

nötigten Eingabewerte von einer anderen Funktion erzeugt wurden. Daraus folgt eine sequentielle Ausführung der lokalen Funktionen (vgl. Abbildung 2.4).

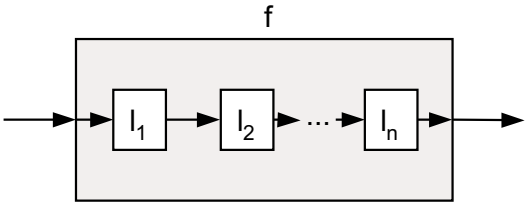

Abbildung 2.4: Der lineare abhängige Fall.

Offensichtlich werden die Eingabeparameterwerte von l_1 alle durch die Eingabewerte der föderierten Funktion f festgelegt und die Ausgabeparameterwerte von l_n bestimmen die Ausgabe der föderierten Funktion. In der Kette der lokalen Funktionen ist die Eingabe jeweils von der Ausgabe der vorherigen Funktion abhängig.

Die bisher betrachtete Abhängigkeit ist die einfachste Form. Im Folgenden werden weitere Formen vorgestellt.

Lineare Abhängigkeit

Der lineare Fall ist der einfachste Typ des unabhängigen Falles und wurde eben beschrieben. Alle lokalen Funktionen werden sequentiell ausgeführt, wie in Abbildung 2.4 dargestellt.

(1:n)-Abhängigkeit

Bei diesem Typ ist eine lokale Funktion von mehreren anderen lokalen Funktionen abhängig, die untereinander wiederum unabhängig sind (vgl. Abbildung 2.5 links). Somit werden die Eingaben der Funktionen $l_1, \ldots, l_i$ von den Eingaben der föderierten Funktion gespeist. Deren Ausgaben füttern die Eingabe von Funktion l_k, deren Ergebnis wiederum die Ausgabe der föderierten Funktion bestimmt. Da die Funktionen $l_1, \ldots, l_i$ unabhängig sind, können sie parallel ausgeführt werden. Die Funktion l_k kann erst aufgerufen werden, wenn alle Ergebnisse der restlichen lokalen Funktionen vorliegen.

(n:1)-Abhängigkeit

Dies ist die Umkehrung des vorangegangenen Falles, denn nun sind die voneinander unabhängigen Funktionen $l_2, \ldots, l_k$ von einer weiteren Funktion l_1 abhängig (vgl. Abbildung 2.5 Mitte). Die Funktionen $l_2, \ldots, l_k$ können erst nach erfolgreicher Ausführung von l_1 gestartet und parallel ausgeführt werden.

Iterative Abhängigkeit

Der iterative Fall lässt sich eigentlich als Konstrukt der bereits beschriebenen Fälle auffassen, soll aber aufgrund seiner Komplexität und neuer Aspekte als separater Typ behandelt werden. Zunächst stellt sich dieser Typ wie der lineare abhängige Typ dar,

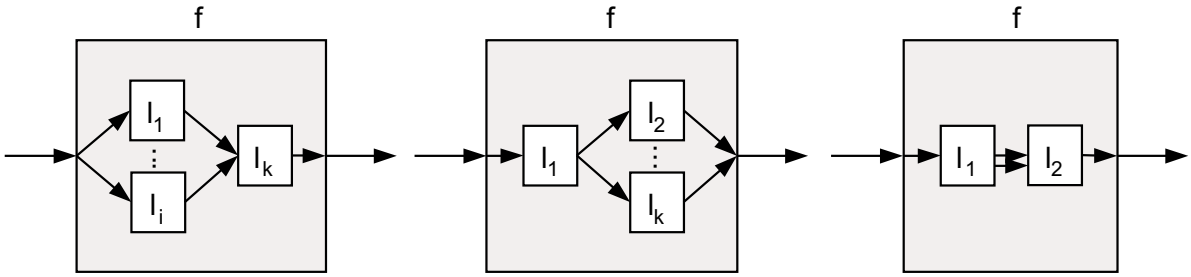

Abbildung 2.5: Der (1:n)-, (n:1)- und iterative abhängige Fall.

bei dem eine Funktion vom Ergebnis der vorherigen Funktion abhängig ist (vgl. Abbildung 2.5 rechts mit Abbildung 2.4). Man geht dabei davon aus, dass beide Funktionen genau einmal ausgeführt werden. Tritt jedoch der Fall ein, dass das Ergebnis der ersten Funktion eine Liste mit mehreren Werten ist, dann muss die abhängige Funktion gegebenenfalls für jedes Listenelement einmal ausgeführt werden (in Abbildung 2.5 symbolisiert durch den Doppelpfeil). Das folgende Beispiel verdeutlicht diesen Fall.

Eine föderierte Funktion soll bei Aufruf eine Liste von Namen der Komponenten ausgeben, die von einem gegebenen Lieferanten bereitgestellt werden:

f_1: `GetAlleKompNamen(IN LiefNr INTEGER, OUT KompNamen LIST[STRING])`

Die lokalen Systeme bieten hierfür keine direkte Funktion an, sondern es müssen die folgenden zwei Funktionen kombiniert werden:

l_1: `GetKompNr(IN LiefNr INTEGER, OUT KompNr LIST[INTEGER])`
l_2: `GetKompName(IN KompNr INTEGER, OUT KompName STRING)`

Zunächst liefert die Funktion l_1 eine Liste der Nummern der gesuchten Komponenten. Anschließend wird für jedes Element dieser Liste die Funktion l_2 aufgerufen, die den zugehörigen Komponentennamen liefert.

Die vier vorgestellten Typen (linear, (1:n), (n:1) und iterativ) bilden die Basis aller erdenklichen Funktionsabhängigkeiten, wenn sie beliebig kombiniert werden. Die bisher betrachteten Fälle zeigen, dass die Abbildung der Parameter in drei verschiedene Abbildungsrichtungen eingeteilt werden kann (siehe Abbildung 2.6). In den einfachen und unabhängigen Fällen werden Parameter der föderierten Funktion auf Parameter der lokalen Funktion abgebildet und umgekehrt. Dies sind die beiden vertikalen Richtungen. Beim abhängigen Fall kommt die horizontale Abbildungsrichtung zwischen den lokalen Funktionen hinzu.

2.2.3.5 Allgemeiner Fall

Der allgemeine Fall ist die beliebige Kombination aus allen bisher aufgeführten Fällen. Darüber hinaus sind mehrere Zielfunktionen möglich. Dieser Fall kann theoretisch unbetrachtet bleiben, da es sich bei den Zielfunktionen um Funktionen einer einzigen Schnittstelle handelt und man deshalb davon ausgehen kann, dass die Funktionen paarweise unabhängig sind.

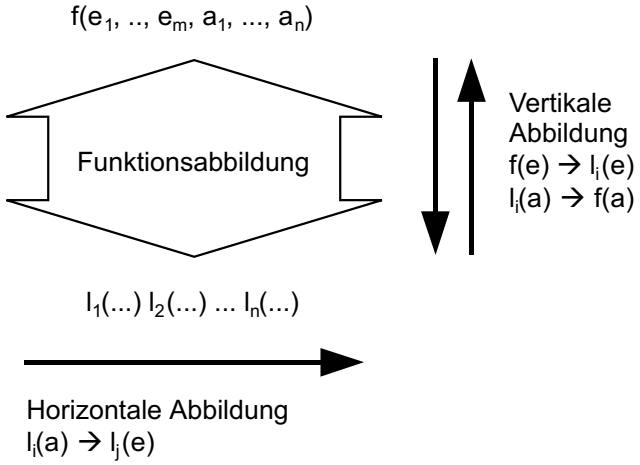

Abbildung 2.6: Mögliche Abbildungsrichtungen bei der Funktionsabbildung.

2.2.4 Übersicht

Abschließend gibt Abbildung 2.7 eine Übersicht zu den aufgeführten Fällen und ihre Beziehung zueinander. Die einzelnen Fälle bauen aufeinander auf, indem zunächst die (1:1)-Funktionsabbildung mit unterschiedlichen Parameterabbildungen betrachtet wird und diese über (1:n)- auf schließlich (n:m)-Funktionsabbildungen mit allen denkbaren Heterogenitätsformen erweitert wird.

2.3 Heterogenitätsformen bei der Integration von Daten und Funktionen

Nachdem in den vorangegangenen Abschnitten Heterogenitätsformen beschrieben wurden, die bei der Daten- bzw. Funktionsintegration auftreten können, betrachten wir in diesem Abschnitt Heterogenitäten, die hinzukommen, wenn Daten und Funktionen zusammen integriert werden sollen. Natürlich sind alle bisher erläuterten Formen vorzufinden. Darüber hinaus kommen Aspekte hinzu, die sich aus den grundlegend unterschiedlichen Modellen der Daten und Funktionen ergeben.

Zunächst klären wir, wo Daten bzw. Funktionen auftreten können. Hier sind mehrere Konstellationen möglich. Zum einen müssen Heterogenitäten überwunden werden, wenn auf föderierter Ebene ein relationales Modell unterstützt werden soll, während die zu integrierenden Systeme durchweg über eine reine Funktionsschnittstelle verfügen und umgekehrt. Zum anderen können die lokalen Systeme unterschiedliche Modelle unterstützen, so dass nicht nur die vertikale Abbildung zwischen föderiertem und lokalem System unterstützt werden muss, sondern auch eine horizontale zwischen den lokalen Systemen.

Im Folgenden werden einige Aspekte aufgeführt, welche die Abbildung erschweren. Mögliche Lösungen stellen wir in dieser Arbeit in den verbleibenden Kapiteln vor.

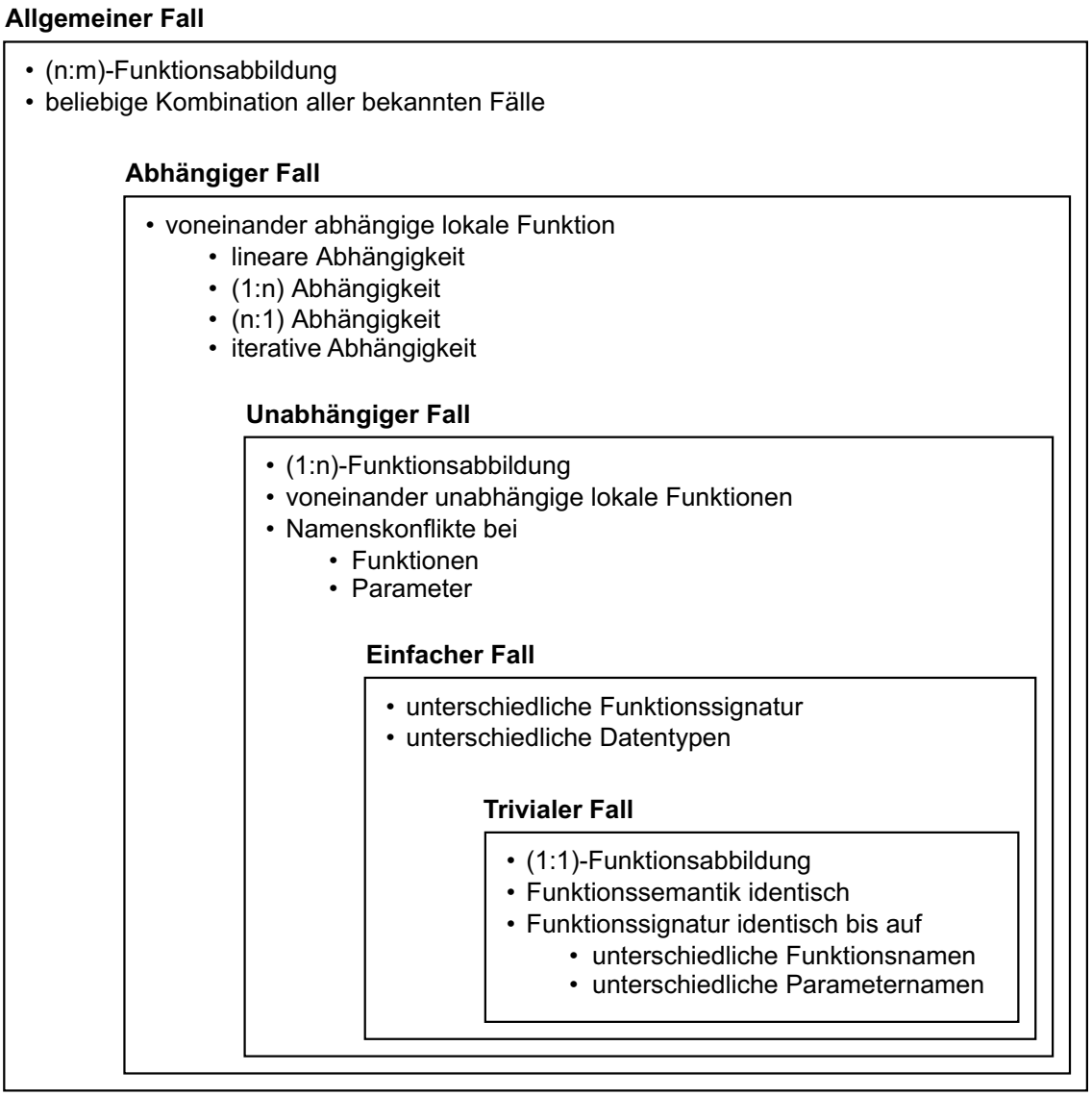

Abbildung 2.7: Übersicht der Heterogenitätsformen bei der Funktionsabbildung.

- *Unterschiedliche Datenmodelle*:
  Betrachtet man die Datenseite, so liegen zumeist Schemata eines bestimmten Datenmodells vor. In den meisten Fällen werden dies relationale Schemata sein, welche die Daten und ihre Beziehungen untereinander beschreiben. Stellt ein System lediglich eine vordefinierte Menge an Funktionen zur Verfügung, über die auf die Daten zugegriffen werden kann, so ist das Schema der zugrunde liegenden Datenbank häufig nicht bekannt. In vielen Fällen soll es auch gar nicht bekannt sein, da es nicht alle Informationen über die Daten abdeckt. Eine saubere Modellierung der Daten wird oft aus Performanzgründen aufgegeben mit der Folge, dass Mechanismen zur Konsistenzsicherung der Daten z. B. in die Anwendung verlegt werden. Für die Integration von Funktionen bedeutet das insbesondere, dass mit den veröffentlichten Funktionen nur eine Teilinformation über die Daten gegeben

wird. Selbst wenn das relationale Schema bekannt ist, kann durch die Funktionen ein anderes Verhalten der Daten auftreten, als dies anhand des Schemas abzuleiten wäre.

- *Unterschiedliche Anfragemöglichkeiten*:
  Das relationale Modell bietet mit SQL eine sehr mächtige und vor allem auch flexible Anfragesprache an. Wird auf föderierter Ebene SQL als Zugriffsschnittstelle angeboten, so steht dessen Flexibilität der fest vorgegebenen Funktionalität der reinen Funktionsschnittstelle gegenüber. Es stellt sich die Frage, ob sich die Mächtigkeit und damit die Anfragemöglichkeiten von SQL auf die vorhandene Menge an Funktionen abbilden lässt. Man muss untersuchen, ob und wie SQL-Anfragen auf Funktionen abgebildet werden können und wo eventuell Einschränkungen auftreten. Dieser Aspekt erklärt obige Aussage, dass ein bekanntes Schema einer Funktionsschnittstelle nicht weiterhilft. Betrachtet man es nämlich aus SQL-Sicht, so könnte die Abfragemächtigkeit der föderierten Schnittstelle auf die lokale Schnittstelle abgebildet werden. Die darüber liegenden Funktionen schränken dies jedoch unter Umständen stark ein.

- *Unterschiedliche Laufzeitumgebung*:
  Bei diesem Punkt ist der Blick vor allem auf die Transaktionsunterstützung gerichtet. Da DBS immer Teil eines umfassenderen Systems sind, bieten sie eine Schnittstelle an, über die Transaktionen gesteuert werden können. Insbesondere können inzwischen viele der kommerziellen DBS an einer verteilten Transaktion teilnehmen, indem sie ein Zwei-Phasen-Commit-Protokoll unterstützen. Anwendungssysteme hingegen stellen ein System dar, das zwar eine Schnittstelle zum Bearbeiten der Daten bereitstellt, grundsätzlich aber nicht darauf ausgelegt ist, in einem Verbund von Systemen mitzuwirken. Für die Transaktionsverwaltung auf föderierter Ebene bedeutet das, dass Transaktionen über transaktionale Systeme (DBS) als auch nicht-transaktionale Systeme (Anwendungssysteme) hinweg unterstützt werden müssen. Die Anwendungssysteme sind meistens jedoch wiederherstellbare Systeme, d. h., sie sind nach innen transaktional sind, können jedoch nicht an verteilten Transaktionen teilnehmen.

Bei der Integration von Daten und Funktionen müssen daher mehrere Fragen geklärt werden. Wie soll die föderierte Schnittstelle aussehen? Wie bildet man Daten auf Funktionen ab und umgekehrt? Wie integriere ich Funktionen in eine relationale Anfrageverarbeitung? Wie sieht eine übergreifende Transaktionsverwaltung aus? All diese Fragen lassen sich letztendlich in zwei große Fragen zusammenfassen: Wie beschreibt man die Abbildung und wie sieht die Laufzeitumgebung aus?

In dieser Arbeit gehen wir diesen Fragen nach und liefern Lösungsansätze.

2.4 Zusammenfassung

Zur Umsetzung von Integrationslösungen müssen unterschiedliche Heterogenitätsformen aufgelöst werden. Diese treten in unterschiedlichen Schichten auf und betreffen beispielsweise Hardware, Betriebssystem, Netzwerk und Programmiersprache. Wir haben uns in

diesem Kapitel vor allem auf Heterogenitäten konzentriert, die aufgrund verschiedenartiger Datenmodelle und unterschiedlicher Schemata entstehen.

Bei der Datenintegration kann man zwischen semantischer und struktureller Heterogenität unterscheiden. Semantische Heterogenität tritt vornehmlich auf der Ebene der Schemata auf. Sie entsteht aus dem Mangel an einem gemeinsamen Verständnis der modellierten Informationen und führt zu Missinterpretationen. Strukturelle Heterogenität wird durch unterschiedliche Mächtigkeit der Datenmodelle verursacht. Sachverhalte werden dadurch unterschiedlich modelliert und führen zu verschiedenen Schemata.

Bei der Integration von Funktionen kann wie bei der Integration von Daten semantische Heterogenität auftreten. In diesem Kapitel haben wir uns auf die Untersuchung derjeniger Heterogenitätsformen konzentriert, die bei der Abbildung einer föderierten Funktion auf eine oder mehrere lokale Funktionen aufkommen können. Wir haben dazu eine Klassifikation erarbeitet, die Heterogenitätsfälle in aufsteigender Komplexität beschreibt. Beim trivialen und einfachen Fall werden Heterogenitäten beschrieben, die bei einer (1:1)-Abbildung der Funktionen aufgelöst werden müssen. Die unabhängigen und abhängigen Fälle enthalten Heterogenitäten, die bei der Abbildung einer Funktion auf mehrere lokale Funktionen auftreten können. Der allgemeine Fall schließt diese Klassifikation nach oben ab. Er stellt eine beliebige Kombination aus allen anderen Fällen dar.

Sollen Daten und Funktionen integriert werden, so treten neben den bisher aufgeführten Heterogenitätsformen weitere Aspekte auf. Innerhalb einer Integrationslösung müssen unterschiedliche Datenmodelle zusammengeführt werden, die unter Umständen gar nicht vollständig bekannt sind. Während auf der Datenseite meistens Schemata vorliegen, sind diese bei einer Menge von bereitgestellten Funktionen häufig nicht bekannt. Ein weiterer Unterschied liegt in den Anfragemöglichkeiten. Auf der Datenseite findet man in den meisten Fällen mit SQL eine sehr mächtige und flexible Anfragesprache vor. Bei Funktionsschnittstellen muss man dagegen mit starren und eingeschränkten Anfragemöglichkeiten auskommen. Ein weiterer wichtiger Aspekt sind unterschiedliche Laufzeitumgebungen. Während die meisten DBS an verteilten Transaktionen teilnehmen können, bieten Anwendungssysteme hierfür keine Unterstützung an.

Dieses Kapitel hat gezeigt, dass die aufgeführten Heterogenitätsformen viele Fragen aufwerfen. Es ist entscheidend, wie die Abbildung beschrieben wird und wie die dazu passende Laufzeitumgebung aussieht. Die Antworten dazu geben wir in den Kapiteln 4 und 5.

Kapitel 3

Grundlegende Konzepte und verwandte Ansätze

In der Einleitung haben wir beschrieben, dass wir eine Lösung zur Integration von Daten und Funktionen entwickeln wollen. Die Verwendung von bestehenden Technologien und die Einhaltung von Standards sind ein wichtiger Aspekt unseres Ansatzes. Wir wollen daher zunächst die für uns relevanten Technologien und Standards in diesem Kapitel vorstellen, bevor wir in den darauf folgenden Kapiteln erläutern, wie wir sie einsetzen.

Da das FDBS das Kernstück unseres Ansatzes ist, gehen wir zunächst auf dessen wichtigste Aspekte ein: die verteilte Anfrageverarbeitung und die verteilte Transaktionsverwaltung. Der dazugehörige Standard SQL:1999 wird anschließend erläutert, wobei der Schwerpunkt unserer Betrachtungen auf Funktionalitäten liegt, welche die Integration unterstützen. Eine weitere wichtige Technologie sind Workflow-Systeme, die wir ebenfalls vorstellen werden. Abschließend gehen wir auf den Standard XML und dessen verwandte Standards ein.

Nach den grundlegenden Konzepten geben wir einen Überblick über verwandte Arbeiten. Unsere Untersuchungen teilen wir in Daten- und Funktionsintegration sowie deren Kombination auf. Für alle Teilgebiete zeigen wir verwandte Ansätze in Forschung und Industrie auf.

3.1 Föderierte Datenbanksysteme

Der Begriff der föderierten Datenbanksysteme (FDBS) wird zum ersten Mal in [SL90] aufgeführt. Dort wird der Oberbegriff Multidatenbanksysteme (MDBS) in föderierte und nicht-föderierte DBS aufgeteilt. Im Gegensatz zu zentralen und verteilten DBS, die jeweils ein einziges DBMS beinhalten, stellt ein MDBS einen Verbund von mehreren DBS dar. Ein MDBS wird heute primär dazu eingesetzt, um bestehende DBS zu integrieren. Neuen Anwendungen wird dadurch eine einheitliche Sicht auf die integrierten Daten bereitgestellt. Behalten die integrierten DBS einen gewissen Grad ihrer Selbständigkeit (Autonomie) bei, so spricht man von FDBS. Somit können in einem FDBS die lokalen DBS autonom, heterogen und verteilt sein.

Des Weiteren können FDBS lose oder eng gekoppelt sein. Bei einer losen Kopplung stellt der Benutzer die föderierte Sicht selbst zusammen und verwaltet diese auch eigenständig. Bei der engen Kopplung hingegen ist die Föderation vorgegeben. Folglich kann der Benutzer nicht selbst entscheiden, welche lokalen DBS in der globalen Sicht integriert sind. Dies hat aber auch den Vorteil, dass er sich nicht um die Umsetzung der Föderation kümmern muss.

Bei den eng gekoppelten FDBS kann man wiederum in einfache und mehrfache Föderation unterteilen. Die einfache Föderation sieht auf der globalen Schicht nur ein einziges föderiertes Schema vor. Auf Basis dieses einen Schemas werden die externen Sichten für die einzelnen Benutzer bereitgestellt. Bei der mehrfachen Föderation gibt es mehrere föderierte Schemata nebeneinander, die jeweils nur Teile der integrierten DBS abdecken und damit weniger umfangreich sind.

Die Kopplung und Verwaltung heterogener und verteilter DBS bedarf spezieller Funktionalitäten bei der Anfrageverarbeitung und Transaktionsverwaltung der FDBS. Diese Aspekte erläutern wir in den folgenden Abschnitten. Wir beschreiben zunächst jeweils die Vorgehensweise im zentralen Fall und zeigen anschließend die neuen Aspekte im heterogenen, verteilten Fall auf.

3.1.1 Verteilte Anfrageverarbeitung

Eine der Hauptaufgaben des DBMS ist die Anfrageverarbeitung, die den schnellen Zugriff auf die Daten ermöglicht. Wir beschreiben hierzu zunächst ihre Aufgaben und erläutern anschließend die Anfrageoptimierung und dazu notwendige Kostenmodelle. Die Betrachtung der Aspekte verteilter Anfrageverarbeitung schließen den Abschnitt ab.

3.1.1.1 Aufgaben der Anfrageverarbeitung

Wir beginnen unsere Betrachtungen zur Anfrageverarbeitung mit ihren Aufgaben. Die Ausführung einer SQL-Anweisung erfolgt in sechs Schritten:

1. *Lexikalische und syntaktische Analyse (Parsing):*
   In diesem Schritt wird die Anfrage auf sprachgrammatikalische Korrektheit überprüft und ein Anfragegraph erstellt, mit dem die folgenden Verarbeitungsschritte ausgeführt werden.

2. *Semantische Analyse:*
   Hier wird die Existenz und Gültigkeit der in der Anfrage referenzierten Datenbankobjekte (Relationen, Sichten, Attribute) anhand des DB-Katalogs überprüft. Außerdem werden die externen Namen durch interne Bezeichnungen ersetzt.

3. *Zugriffs- und Integritätskontrolle:*
   In diesem Schritt wird geprüft, ob für die Datenbankobjekte auch die entsprechenden Zugriffsrechte für den Benutzer vorliegen. Außerdem können zu diesem Zeitpunkt auch schon die Integritätskontrollen ausgeführt werden, die beispielsweise Format, Typ, Konversion und Zulässigkeit von Attributwerten betreffen.

4. *Standardisierung und Vereinfachung:*
   Die beliebig strukturierte globale Anfrage wird in eine Normalform gebracht und, wenn möglich, bereits vereinfacht, um eine effektive Fortsetzung der Übersetzung zu ermöglichen.

5. *Restrukturierung und Transformation:*
   Nun erfolgt eine Restrukturierung des Anfragegraphen, um eine globale Verbesserung der Anfrageauswertung zu erzielen. Dieser dient anschließend als Ausgangspunkt für die Generierung von Anfrageausführungsplänen.

6. *Code-Generierung / direkte Ausführung oder interpretative Ausführung:*
   Der vorliegende Ausführungsplan wird nun genutzt, um Code zu generieren oder eine interpretative Ausführung der Anfrage zu ermöglichen.

Die ersten drei Schritte werden oft als Vorbereitung der Anfrage zusammengefasst, während die Schritte 4 und 5 die Optimierung darstellen. Schritt 6 steht letztendlich für die Ausführung.

Es stehen zwei grundsätzliche Vorgehensweisen bezüglich des Bindungszeitpunktes zur Diskussion. Beim interpretativen Ansatz werden die einzelnen Schritte zur Laufzeit ausgeführt. Im Extremfall werden dabei für jede Anfrage alle sechs Phasen immer wieder durchlaufen. Dieses Vorgehen hat einerseits den Nachteil, dass die Ausführung sehr teuer ist, da bei jeder Anfrage mehrere Zugriffe auf den DB-Katalog erforderlich sind. Andererseits ist von Vorteil, dass durch den späten Bindezeitpunkt ein hoher Grad an Datenunabhängigkeit erzielt wird, da Änderungen bei den Daten und Zugriffspfaden bis zum tatsächlichen Ausführungszeitpunkt berücksichtigt werden können.

Im Gegensatz zur Interpretation werden bei der Übersetzungstechnik alle Verarbeitungsschritte zum Übersetzungszeitpunkt durchgeführt, so dass zur Laufzeit nur noch die direkte Ausführung des Codes für eine Anfrage erfolgen muss. Dies hat den Vorteil, dass die Ausführung der Anfrage zur Laufzeit schneller und günstiger ist. Auf der anderen Seite findet eine frühzeitige Bindung an vorhandene Daten- und Zugriffspfadstrukturen statt, so dass bei deren Änderung eine Wiederholung der Übersetzung erfolgen muss.

Da sich bei der Übersetzungstechnik Abhängigkeiten zum DB-Katalog ergeben, welche die Besprechung der Anfrageverarbeitung komplexer werden lassen, gehen wir im Folgenden zunächst von einem interpretativen Ansatz aus, bei welchem alle Verarbeitungsschritte zur Laufzeit durchgeführt werden.

In den folgenden Abschnitten konzentrieren wir uns auf die Optimierung der Anfrage. Die Vorbereitung und Code-Generierung werden nicht weiter vertieft.

3.1.1.2 Anfrageoptimierung

Zur internen Verarbeitung der Anweisung wird ein Anfragegraph aufgebaut. Dies geschieht in den Schritten 1 und 2, der Analysephase. Treten keine syntaktischen und semantischen Fehler auf, kann dieser Graph zur Erstellung des Ausführungsplans herangezogen werden. Dies ist ein möglicher Ausführungsplan, der aber durch ungünstige Operationsfolgen nicht der beste Ausführungsplan sein muss. Daher besteht die Anfrageoptimierung aus mehreren Verarbeitungsschritten, in denen der Anfragegraph schrittweise verbessert wird. Das Ergebnis ist ein zum ersten Ausführungsplan semantisch

äquivalenter Plan, der nach den vorgegebenen Kriterien optimal ist. Als Optimum strebt man entweder eine Antwortzeitminimierung für diese eine Anfrage oder die Durchsatzmaximierung für alle Anfragen an.

Die Anfrageoptimierung ist ein komplexer Vorgang. Zwar existieren einige Techniken und Verfahren zur Lösung der Optimierung, diese betreffen aber meistens nur Teilaspekte. Das Hauptproblem ist jedoch, dass die genaue Optimierung im Allgemeinen nicht berechenbar ist. Eigentlich müsste jeder erfolgversprechende Lösungsweg ausgerechnet werden, was aber nicht umsetzbar ist. Stattdessen setzt man Heuristiken ein, da alle Optimierungsentscheidungen auf einem einfachen DB-Modell mit wenigen statistischen Kenngrößen getroffen werden müssen. Da sehr große Lösungsräume zu evaluieren und zu reduzieren sind, muss eine strikte Systematik bei der Anfrageoptimierung eingehalten werden. Diese wird in vier Verarbeitungsschritte aufgeteilt: Standardisierung, Vereinfachung, Restrukturierung und Transformation. Wir gehen in den folgenden Abschnitten auf diese Schritte ein.

Standardisierung

Die standardisierte Darstellung einer Anfrage vereinfacht das Erkennen von Standardsituationen und deren Auswertung. Standardisierung erfolgt lokal auf der Ebene der Qualifikationsbedingungen und global auf der Anfrageebene.

Um Qualifikationsbedingungen zu standardisieren, können die disjunktive oder die konjunktive Normalform gewählt werden. Die konjunktive Normalform bevorzugt die Disjunktion von Prädikaten P_{ij}: $(P_{11} \vee \ldots \vee P_{1n}) \wedge \ldots \wedge (P_{m1} \vee \ldots \vee P_{mp})$. Die disjunktive Normalform dagegen zieht die Konjunktion vor: $(P_{11} \wedge \ldots \wedge P_{1q}) \vee \ldots \vee (P_{r1} \wedge \ldots \wedge P_{rs})$. Beide Normalformen können durch das wiederholte Anwenden der Umformungsregeln für Boolesche Ausdrücke (De Morgan'sche Regeln, Kommutativ-, Assoziativ- und Distributivregeln, Doppelnegationsregel) abgeleitet werden. Aufgrund der unterschiedlichen Vorlieben führen konjunktive Normalformen zuerst Vereinigungsoperationen aus, während disjunktive Normalformen Verbundoperationen in der Verarbeitung vorziehen.

Auf der Anfrageebene überführt die Standardisierung meistens geschachtelte Anfragen in eine symmetrische Notation oder ersetzt quantifizierte Unteranfragen durch äquivalente Verbundanfragen. Als Ergebnis befinden sich keine quantifizierten Terme im Qualifikationsteil einer Anfrage mehr. Die entsprechenden Umformungsregeln können in [JK84] nachgelesen werden.

Vereinfachung

Die Vereinfachung der Anfrage beschleunigt die spätere Auswertung und deckt frühzeitig Fehler auf. Der somit frühe Abbruch spart unnötige Arbeit ein. Die Vereinfachung findet ebenfalls lokal und global statt. Auf der globalen, also Anfrageebene fällt die Vereinfachung mit der Anfragerestrukturierung zusammen und wird dort von uns beschrieben. Auf der lokalen Ebene unterscheidet man bei den Qualifikationsbedingungen in der WHERE-Klausel mehrere Aspekte.

Durch die Formulierung des Benutzers oder das Ersetzen von Sichtendefinitionen können redundante Prädikate in der WHERE-Klausel auftreten, die überflüssige Operationen

veranlassen. Weitere Quellen für Redundanzen sind die Berücksichtigung von Integritätsbedingungen, wertabhängige Zugriffsrechte und von 4GL-Sprachen generierte Anfragen. Mit Idempotenzregeln für Boolesche Ausdrücke eliminiert man diese Redundanzen, indem beispielsweise der Ausdruck $A \wedge \neg(B \vee C) \wedge (B \vee C)$ umgeformt wird in $A \wedge false$ oder $false$. In diesem Beispiel wird eine leere Ergebnismenge erkannt und die weitere Verabreitung erübrigt sich somit.

Weitere Formen der Vereinfachung sind die Konstantenpropagierung und die Hüllenbildung der Qualifikationsprädikate [Mit95].

Restrukturierung

Die Anfragerestrukturierung (*query rewrite*) entspricht einer algebraischen Optimierung, da der Anfragegraph auf der Ebene der logischen Algebraoperatoren umgeformt wird. Dadurch sollen optimale Operatorreihenfolgen und möglichst kleine Zwischenergebnisse erreicht werden. Die Restrukturierung sucht nach der aus abstrakter Sicht günstigsten Auswertungsstruktur.

Die Restrukturierung erfolgt durch geeignete Regelmengen für die relationalen Basisoperatoren [Gra93]. Man versucht mit Heuristiken, die Anzahl der zu verarbeitenden Tupel und Attribute zu minimieren. Dabei werden physische Kriterien wie Seitenabbildung nicht beachtet. Die wichtigsten Regeln sind:

- Selektionen (σ) und Projektionen (π) ohne Duplikateliminierung sollten möglichst früh ausgeführt werden, da sie die zu verarbeitende Tupelmenge und Tupellänge erheblich verringern können.
- Aufeinanderfolgende unäre Operatoren (wie π und σ) auf einer Relation sollten zu einer Operation mit komplexerem Prädikat zusammengefasst werden.
- Selektionen und Projektionen, die ein und dieselbe Relation betreffen, sollten so zusammengefasst werden, dass jedes Tupel nur einmal verarbeitet werden muss.
- Bei aufeinanderfolgenden binären Operatoren (wie $\cap$, $\cup$, $-$, $\times$, $\bowtie$) sollte die Größe der Zwischenergebnisse minimiert werden.
- Gleiche Teile im Anfragegraphen sollten nur einmal ausgewertet werden.

Abhängig vom Datenmodell kann die Regelmenge weiter verfeinert werden. Die Anwendung der Regeln beschreiben wir in einem vereinfachten Algorithmus:

1. Zuerst werden komplexe Verbundprädikate so zerlegt, dass man sie binären Verbunden zuordnen kann.
2. Anschließend teilt man Selektionen mit mehreren Prädikatstermen in separate Selektionen mit jeweils einem Prädikatsterm auf.
3. Selektionen führt man frühestmöglich aus, indem sie zu den Blättern des Anfragegraphen hinuntergeschoben werden (*selection push-down*).
4. Einfache Selektionen fasst man zusammen, so dass aufeinanderfolgende Selektionen derselben Relation zu einer verknüpft werden können.

5. Anschließend führt man Projektionen ohne Duplikateliminierung frühestmöglich aus, indem sie so weit wie möglich zu den Blättern des Anfragegraphen hinuntergeschoben werden (*projection push-down*).

6. Abschließend fasst man einfache Projektionen derselben Relation zu einer Operation zusammen.

Die beschriebenen Schritte müssen auf das interne Darstellungsschema und seine Operatoren umgesetzt werden. In [Mit95] zeigen Beispiele, wie leistungsfähig die Anfragerestrukturierung sein kann. Obwohl DB-Größe und -Struktur und viele weitere Parameter das Ergebnis maßgeblich beeinflussen, ermöglicht diese Form der Optimierung einen deutlichen Leistungsgewinn.

Transformation

Während die Anfragerestrukturierung den Anfragegraphen gemäß einer abstrakten Sicht und nach Heuristiken für logische Operatoren umformt, betrachtet die Anfragetransformation zusätzlich die physischen Operatoren. Man bezeichnet sie daher als nichtalgebraische Optimierung. Diese Operatoren berücksichtigen alle Aspekte der Abbildung auf Seitenstrukturen und Zugriffspfade und ersetzen die logischen Operatoren im konkreten Ausführungsplan. Sie werden auch Planoperatoren genannt.

Logische Operatoren können häufig durch einen oder mehrere unterschiedliche Planoperatoren dargestellt werden und jeder Planoperator kann wiederum verschiedene Realisierungsalternativen besitzen. Folglich können meistens viele äquivalente Ausführungspläne erzeugt werden. Diese Ausführungspläne sind mit möglichst genauen Abschätzungen hinsichtlich ihrer Ausführungskosten zu bewerten, um den kostengünstigsten Ausführungsplan zu bestimmen.

Da wir in dieser Arbeit nicht weiter auf die Anfragetransformation eingehen werden, beschreiben wir die Planoperatoren und Plangenerierung nicht näher. Stattdessen verweisen wir den interessierten Leser auf [Mit95].

3.1.1.3 Kostenmodell

Um den kostengünstigsten Ausführungsplan zu ermitteln, reichen die Überlegungen zur Anfragerestrukturierung nicht aus. Die physische Datenabbildung hat großen Einfluss auf die Planoptimierung. Dies zeigt ein Erfahrungsbericht [CABG81], nach welchem ein Planoptimierer folgende Aspekte beachten muss:

- Neben den Zugriffen auf die Tupel sollten auch verborgene E/A- und CPU-Kosten für die Manipulation von TID-Listen berücksichtigt werden.

- Die Anzahl der physischen Seitenzugriffe ist ein zuverlässigeres Kostenmaß als die Anzahl der Tupelzugriffe. So können die Auswirkungen von Cluster-Eigenschaften und DB-Pufferersetzung besser einbezogen werden.

- Die Auswahlentscheidung sollte aufgrund einer gewichteten Funktion aus CPU-Zeit und E/A-Anzahl getroffen werden, deren Gewichte je nach Verarbeitungssituation und Rechnerkonfiguration angepasst werden können.

- Auf einfache Anfragen sollte besonders geachtet werden, um deren Pfadlänge gewissenhaft zu minimieren.

Nachdem wir die zentralen Problemaspekte kurz beschrieben haben, gehen wir in den folgenden Abschnitten genauer auf die Kostenberechnung ein.

Die Kostenabschätzung und damit die Planoptimierung gehen im Allgemeinen von den folgenden Grundannahmen über die Werteverteilung in der Datenbank aus:

- Die Werte aller Attribute sind gleichverteilt.
- Die Werte verschiedener Attribute sind unabhängig voneinander. Folglich wird die Unabhängigkeit aller Selektionsprädikate einer Anfrage unterstellt.

Mit diesen Annahmen können die erwarteten Treffer einer Anfrage durch Mittelwertbildung, Interpolation und arithmetische Verknüpfungen berechnet werden. Jedoch ist das Ergebnis in vielen Fällen nicht richtig. Obwohl die Abschätzung eines Ausführungsplans die Wirklichkeit weit verfehlen kann, erzielt man mit den beiden Grundannahmen im Allgemeinen gute Ergebnisse. Dies liegt vor allem daran, dass bei der Abschätzung nur relative Werte interessieren. Sind daher alle geschätzten Werte in gleicher Weise falsch, können die ermittelten Ergebnisse trotzdem zur Auswahlentscheidung herangezogen werden.

Parameter von Kostenmodellen

Im allgemeinen Fall sollten folgende Kostenarten bei der Berechnung von Anfragekosten einbezogen werden:

- Kommunikationskosten, welche die Anzahl der Nachrichten und die Menge der zu übertragenden Daten enthalten.
- Berechnungskosten, welche die CPU-Nutzung und die Pfadlänge der Anfrage berücksichtigen.
- E/A-Kosten, d. h. die Anzahl physischer Referenzen.
- Speicherungskosten, die durch temporäre Speicherbelegung im DB-Puffer, in speziellen Arbeitsbereichen und auf Externspeichern verursacht werden.

Verschiedenartige DBS-Algorithmen verknüpfen diese Kostenarten miteinander. Beispielsweise können die Speicherungskosten die Berechnungs- und E/A-Kosten erheblich beeinflussen. Außerdem müssen diese Kostenarten aufgrund technologischer Verbesserungen ständig überarbeitet werden. Folglich ist das Kostenmodell regelmäßig an die technologischen Gegebenheiten anzupassen.

Speicherungskosten werden aufgrund ihrer fehlenden Vergleichbarkeit mit Zugriffs- und Berechnungskosten häufig nicht in das Kostenmodell übernommen. Kommunikationskosten spielen vor allem bei verteilten DBS eine wichtige Rolle. Im Folgenden betrachten wir zunächst ein Kostenmodell, das in zentralisierten DB eingesetzt wird, bevor wir in Abschnitt 3.1.1.4 auf die Aspekte im verteilten Umfeld eingehen. Wir betrachten ein

Kostenmodell, das auf einer gewichteten Funktion von Berechnungs- und E/A-Kosten basiert. Zahlreiche praktische Erfahrungen [SAC+79, CABG81, Cha98] und Einsätze in kommerziellen DBS bestätigen diese Wahl.

Für jeden in die Planoptimierung einbezogenen Ausführungsplan wird ein Kostenvoranschlag nach folgender Kostenformel berechnet [SAC+79]:

$$Cost(T) = pagefetches(T) + W \times systemcalls(T)$$

In dem gewichteten Kostenmaß von physischer E/A und CPU-Belegung beschreibt die Funktion *pagefetches* die Anzahl der physischen Zugriffe auf externe Speichermedien (in den meisten Fällen den Platten). Die Funktion *systemcalls* charakterisiert die Anzahl der Aufrufe an das Zugriffssystem, die sich von der Anzahl der für die Anfrageverarbeitung benötigten Zeilen ableiten lässt. Folglich ist diese Funktion ein guter Indikator für die erwartete Prozessorlast.

Der Faktor W dient zur Berücksichtigung der Art der Systemauslastung. Ist das System CPU-lastig, so sollte W eher groß gewählt werden, um Ausführungspläne mit geringem CPU-Bedarf zu bevorzugen. Bei einem E/A-lastigen System sollte W hingegen eher klein gewählt werden, um eine Auswahl von E/A-intensiven Ausführungsplänen zu vermeiden.

Ermittlung der statistischen Kenngrößen

Die Berechnung der Anzahl der physischen Seitenzugriffe erfolgt für jeden Zugriffspfad durch detaillierte Formeln. Dabei gehen viele systemspezifische Größen und Annahmen ein. In [BE77, PI97, Cha98] wird die Ableitung solcher Formeln beschrieben. Wir beschränken uns an dieser Stelle auf die Erläuterung der grundsätzlichen Vorgehensweise.

Um die erwartete Anzahl von Seitenzugriffen abschätzen zu können, müssen Statistiken über die DB-Objekte in den DB-Katalogen geführt und gewartet werden. Die wichtigsten Parameter halten Informationen über die Datenseiten der Segmente, die Anzahl der Tupel einer Relation sowie deren Verteilung auf Seiten und die Größe und den Aufbau eines Indexes.

Obwohl sich einige dieser Parameter recht schnell ändern, ist eine periodische Aktualisierung der Statistiken der dynamischen vorzuziehen. Zwar erreicht man damit nicht die höchste Genauigkeit, man spart aber die zusätzlichen Schreib- und Log-Operationen und entlastet damit das System. Es scheint ausreichend, die statistischen Werte zu ermitteln, wenn Relationen und Indexstrukturen geladen oder generiert werden. Bei Reorganisationen oder auf Veranlassung des Benutzers kann eine Neubestimmung mit entsprechenden Anweisungen vorgenommen werden.

Abschätzung der Selektivität

Um die Kosten abschätzen zu können, müssen Selektivitätsfaktoren (SF) für den Selektionsausdruck einer Anfrage berechnet werden. Dieser Faktor gibt an, wie viel Prozent der Tupeln das Prädikat p des Selektionsausdruckes erfüllen. Somit ergibt sich für die Kardinalität der Ergebnismenge einer Selektion folgende Formel: $Card(\sigma_p(R)) =$

$SF(p) \cdot Card(R)$. In [SAC+79] werden für verschiedene Arten von Verbundtermen Selektivitätsfaktoren eingeführt. Die wichtigsten davon sind:

$$A_i = a_k : SF = \begin{cases} 1/j(I_i) & \textit{wenn Index auf Attribut } A_i \\ 1/10 & \textit{sonst} \end{cases}$$

$$A_i = A_k : SF = \begin{cases} 1/MAX(j(I_i), j(I_k)) & \textit{wenn Index auf beiden Attributen} \\ 1/j(I_i) & \textit{wenn Index nur auf } A_i \\ 1/j(I_k) & \textit{wenn Index auf } A_k \\ 1/10 & \textit{sonst} \end{cases}$$

$$A_i > a_i \textit{ oder } A_i \geq a_i : SF = \begin{cases} (a_{max} - a_i)/(a_{max} - a_{min}) & \textit{wenn Index auf } A_i \\ & \textit{und Wert interpolierbar} \\ 1/3 & \textit{sonst} \end{cases}$$

$$A_i \textit{ BETWEEN } a_i \textit{ AND } a_j : SF = \begin{cases} (a_j - a_i)/(a_{max} - a_{min}) & \textit{wenn Index auf } A_i \\ & \textit{und Wert interpolierbar} \\ 1/4 & \textit{sonst} \end{cases}$$

$$A_i \textit{ IN } (a_1, a_2, \ldots, a_r) : SF = \begin{cases} r/j(I_i) & \textit{wenn Index auf } A_i \textit{ und } SF < 0.5 \\ 1/2 & \textit{sonst} \end{cases}$$

Die konstanten Selektivitätsfaktoren sind Default-Werte und stellen Erfahrungswerte dar. Für die anderen Berechnungen wird die Gleichverteilung der Werte vorausgesetzt.

Für komplexere Ausdrücke mit Booleschen Operatoren gibt es einfache Regeln unter der Annahme, dass die qualifizierten Mengen stochastisch unabhängig sind:

$SF(p(A) \wedge p(B)) = SF(p(A)) \cdot SF(p(B))$
$SF(p(A) \vee p(B)) = SF(p(A)) + SF(p(B)) - SF(p(A)) \cdot SF(p(B))$
$SF(\neg p(A)) = 1 - SF(p(A))$.

Bei Verbundoperationen schätzt man die Größe der Ergebnisrelation ab. Diese wird als Basis für die Berechnung der Kosten nachfolgender Verbundoperationen genutzt. In [Dem80] wird der sog. Join-Selektivitätsfaktor JSF für den Gleichverbund von R und S folgendermaßen definiert:

$Card(R \bowtie S) = JSF \cdot Card(R) \cdot Card(S)$.

Whang et al. [WVZT90] untersuchen verschiedene Maßnahmen zur Ableitung von Join-Selektivitätsfaktoren. Einfache Zusammenhänge ergeben sich aber nur in speziellen Fällen.

Wie zu Beginn erläutert, basieren die aufgeführten Abschätzungen auf den Annahmen hinsichtlich Gleichverteilung und Unabhängigkeit der Attributwerte. Eine höhere Genauigkeit lässt sich durch Histogramme erreichen. Mit Histogrammen nähert man sich

der konkreten Verteilung der Attributwerte an, statt sie durch eine einzige Zahl zu beschreiben [Ioa93]. Dabei wird der Wertebereich in Intervalle aufgeteilt, für die jeweils die Häufigkeit der Attributwerte festgehalten wird. Weitere Verfeinerungen des Histogramm-Ansatzes sind in [MD88, SS94, PI97, Lyn88, GMP97] nachzulesen.

Mit den Selektivitätsausdrücken kann nun für jeden Ausführungsplan einer Anfrage die Anzahl der Seitenzugriffe (*pagefetches*) und die Anzahl der Aufrufe an das Zugriffssystem (*systemcalls*) abgeschätzt werden. Um die Genauigkeit zu erhöhen, können weitere Faktoren wie z. B. Cluster-Bildung und Sortier- und Mischkosten für Datensätze berücksichtigt werden.

Bestimmung des kostengünstigsten Plans

Für die verfügbaren Planoperatoren müssen detaillierte Kostenformeln vorliegen, damit der kostengünstigste Ausführungsplan ermittelt werden kann. Die Suche ist aufwendig, da nicht nur die Zulässigkeit eines Planoperators und die geeignete Verknüpfungsfolge bestimmt, sondern auch die kosteneffektivsten Planoperatoren ausgewählt werden müssen. In [SAC+79] wird für die Suche ein Lösungsbaum beschrieben, der mit geeigneten Heuristiken durchlaufen wird. Ein ausführliches Beispiel findet sich in [SAC+79].

3.1.1.4 Aspekte der verteilten heterogenen Anfrageverarbeitung

Ein FDBS muss eine verteilte heterogene Anfrageverarbeitung unterstützen, da die Ergebnisse globaler SQL-Anweisungen durch mehrere Systeme ermittelt werden. Wir werden zunächst den Ablauf einer globalen Anfrage betrachten und anschließend neue Aspekte in der Optimierung beschreiben.

Ablauf einer verteilten heterogenen Anfrageverarbeitung

Der prinzipielle Ablauf der Anfrageverarbeitung in einem föderierten System kann in verschiedene Phasen zerlegt werden [MY95]:

- *Zerlegung der globalen Anfrage in Teilanfragen*:
  Nachdem das FDBS die globale Anfrage analysiert hat, ermittelt es die betroffenen lokalen Systeme. Anhand der Information zu den lokalen Systemen erzeugt das FDBS Teilanfragen, die an die einzelnen Systeme weitergeleitet werden. Da die Funktionalität der Systeme sehr unterschiedlich ausfallen kann, ist nicht immer gegeben, dass die komplette Teilanfrage lokal verarbeitet werden kann. Das FDBS muss in solchen Fällen die fehlende Funktionalität kompensieren, indem diese Operationen im FDBS auf den angefragten Daten durchgeführt werden. Wird die Verarbeitung der globalen Anfrage über mehrere Systeme verteilt, muss das FDBS auch wissen, wie die einzelnen Zwischenergebnisse zum angeforderten Gesamtergebnis verknüpft werden.

- *Übersetzen der Teilanfragen in die lokalen Anfragesprachen:*
  Nachdem ermittelt wurde, wie die Teilanfragen für die lokalen Systeme aussehen, müssen diese in die entsprechenden Anfragesprachen übersetzt werden. Dies ist nötig, da die lokalen Systeme unterschiedlichen Datenbanktyps sein können, z. B.

relational, objektorientiert, hierarchisch usw. Selbst wenn alle Systeme beispielsweise auf relationalen DBS basieren, müssen die unterschiedlichen SQL-Dialekte der Hersteller berücksichtigt werden.

- *Lokale Bearbeitung der Teilanfragen*:
  Die lokalen DBS bearbeiten nun die Teilanfragen und liefern die Ergebnisse an das FDBS zurück. Hier ist zu beachten, dass die lokale Anfrageverarbeitung und -optimierung unabhängig von der globalen Ebene und auch den anderen lokalen Systemen erfolgt.

- *Zurückübersetzen der lokalen Anfrageergebnisse:*
  Hat das FDBS die Ergebnisse der lokalen Systeme erhalten, müssen diese eventuell wieder in das globale Datenmodell übersetzt werden. Bei rein relationalen FDBS ist diese Phase nicht sehr aufwendig.

- *Zusammensetzen des globalen Ergebnisses aus den lokalen Ergebnissen:*
  Liegen die Ergebnisse der Teilanfragen vor, müssen diese zum globalen Endergebnis zusammengesetzt werden. Außerdem müssen die Operationen durchgeführt werden, die nicht direkt vom lokalen System unterstützt wurden.

Nachdem wir den prinzipiellen Ablauf der verteilten heterogenen Anfrageverarbeitung aufgezeigt haben, skizzieren wir abschließend wichtige Optimierungsaspekte.

Verteilte heterogene Anfrageoptimierung

Wie in den vorangegangenen Abschnitten erläutert, sucht die Anfrageoptimierung nach einem möglichst effizienten Ausführungsplan für eine gegebene Anfrage. Dies ist auch in FDBS gewünscht, doch kommen hier aufgrund der Autonomie und Heterogenität der lokalen Systeme eine Reihe zusätzlicher Probleme hinzu. Optimierungsverfahren für homogene verteilte Datenbanksysteme lassen sich nicht übertragen, da sie von Voraussetzungen ausgehen, die normalerweise in föderierten Systemen nicht gegeben sind. Dazu gehören u. a. die Annahmen, dass keine Dateninkonsistenzen zwischen den lokalen Systemen vorliegen und dass charakteristische Eigenschaften der lokalen Systeme bekannt sind. Diese Annahmen sind jedoch für FDBS im Allgemeinen nicht gültig.

Betrachten wir die in Abschnitt 3.1.1.3 eingeführte Kostenformel, so muss diese um die Kommunikationskosten erweitert werden, da diese im verteilten Umfeld eine große Rolle spielen. Um diese Kostenart zu minimieren, muss das Ziel die Minimierung der zu übertragenden Datenmengen zwischen dem FDBS und den lokalen Systemen sein. Dies ist vor allem dann möglich, wenn die lokalen Systeme viele Operationen auf den Daten lokal durchführen können. Werden beispielsweise Selektionen und Projektionen lokal ausgeführt, ist das Zwischenergebnis deutlich kleiner und folglich die Kommunikationskosten geringer. Daher gilt grundsätzlich, dass das FDBS möglichst viele Informationen über die lokalen Systeme und deren Anfrageverarbeitung haben sollte. Vor allem die Mächtigkeit hinsichtlich unterstützter Operationen ist von Bedeutung, um zu entscheiden, welche Operationen lokal ausgeführt werden sollten.

3.1.2 Verteilte Transaktionsverwaltung

Neben der verteilten Anfrageverarbeitung ist die verteilte Transaktionsverwaltung und deren Unterstützung in dieser Arbeit von Belang. Dazu wird zunächst der klassische Transaktionsbegriff für zentrale Datenbanksysteme erläutert. Anschließend gehen wir auf weiterführende Aspekte ein, die zur Unterstützung von verteilten Transaktionen in FDBS wichtig sind.

3.1.2.1 Der klassische Transaktionsbegriff

Das Datenbanksystem garantiert eine sichere und konsistente Ausführung der DB-Zugriffe. Dies soll vor allem im Hinblick auf konkurrierende Zugriffe durch eine Vielzahl an Benutzern als auch Fehlersituationen wie Rechnerausfällen gewährleistet werden. Ein grundlegendes Konzept stellt das ACID-Paradigma dar [HR83], das wir im Folgenden näher erläutern.

Das ACID-Paradigma sieht vor, dass erfolgreich ausgeführte Transaktionen vier grundlegende Eigenschaften aufweisen, kurz ACID-Eigenschaften genannt. Das Akronym leitet sich von den englischen Begriffen *Atomicity*, *Consistency*, *Isolation* und *Durability* ab, die wie folgt definiert sind:

- *Atomarität (Atomicity)*:
  Hier gilt das „Alles oder Nichts"–Prinzip, d. h., eine Transaktion wird ganz oder gar nicht ausgeführt. Wird eine Transaktion erfolgreich beendet, so ist die Konsistenz der Daten sichergestellt. Tritt jedoch während der Ausführung ein Fehler auf, so dass die Transaktion nicht ordnungsgemäß fortgesetzt werden kann, sind Inkonsistenzen möglich. Um dies zu verhindern, müssen alle bis dahin bereits durchgeführten Änderungen wieder rückgängig gemacht werden, als ob diese nicht erfolgreiche Transaktion nie stattgefunden hätte.

- *Konsistenz (Consistency)*:
  Eine Transaktion muss die Datenbank von einem konsistenten Zustand in einen wiederum konsistenten Zustand überführen. Dazu wird am Ende der Transaktion geprüft, ob Integritätsbedingungen verletzt werden. Trifft dies zu, so muss die Transaktion abgebrochen und der konsistente Zustand zu Beginn der Transaktion wiederhergestellt werden.

- *Isolation*:
  Trotz der Unterstützung des gleichzeitigen Zugriffs durch eine Vielzahl an Benutzern muss für jeden der Eindruck bestehen, dass er alleine mit den Daten arbeitet. Folglich dürfen keine unerwünschten Nebenwirkungen auftreten, indem beispielsweise der Benutzer auf Daten zugreift, die ein anderer Benutzer gerade bearbeitet. Diese Isolation wird durch geeignete Synchronisationsmaßnahmen wie z. B. Sperrverfahren ermöglicht.

- *Dauerhaftigkeit (Durability)*:
  Diese Eigenschaft besagt, dass Änderungen erfolgreich durchgeführter Transaktionen persistent gemacht und somit für alle Benutzer sichtbar werden. Zudem überleben die Änderungen dieser Transaktionen zukünftige Systemfehler, d. h.,

dass der jüngste transaktionskonsistente Zustand der Datenbank wiederhergestellt werden kann.

Unterstützt die Transaktionsverwaltung diese vier Eigenschaften, so wird dem Anwendungsentwickler die Arbeit stark erleichtert. Die Atomarität garantiert, dass abgebrochene Transaktionen keine Spuren hinterlassen und die Isolation verhindert Anomalien durch Mehrbenutzerbetrieb. Zentrale Datenbanksysteme gewährleisten die Isolation durch das Kriterium der Serialisierbarkeit. Atomarität und Dauerhaftigkeit werden durch Recovery-Mechanismen sichergestellt.

In den folgenden Abschnitten beschreiben wir Synchronisation und Recovery näher.

Synchronisation

Da viele Benutzer gleichzeitig lesend und schreibend auf die Datenbank zugreifen, muss eine Synchronisation der Zugriffe erfolgen. So werden die konkurrierenden Zugriffe voneinander isoliert und die Konsistenz der Daten sichergestellt. Erfolgt keine Synchronisation, so können Konsistenzverletzungen, so genannte Anomalien auftreten. Dazu gehören verloren gegangene Änderungen (*lost updates*), Zugriffe auf schmutzige Daten (*dirty reads, dirty writes*), nicht-wiederholbares Lesen (*non-repeatable read*) und das Phantom-Problem [GR93]. Die Serialisierbarkeit schließt diese Anomalien aus und ist das allgemein akzeptierte Korrektheitskriterium der Synchronisation [EGLT76, BN97]. Die Ausführung konkurrierender Zugriffe ist korrekt, wenn das Ergebnis dem einer seriellen Abarbeitung der Transaktionen entspricht. Bei einer seriellen Ausführung gibt es keine zeitliche Überlappung der Transaktionen, sondern sie werden vollständig nacheinander ausgeführt. Die parallele Ausführung ist somit äquivalent zu einer seriellen, wenn sie die gleichen Ausgabewerte und den gleichen DB-Endzustand erzeugt. Die drei bekanntesten Klassen von Synchronisationsverfahren stellen Sperrprotokolle, Zeitmarkenverfahren und optimistische Algorithmen dar.

Sperrverfahren zeichnen sich dadurch aus, dass eine Transaktion erst dann auf Objekte zugreifen kann, nachdem sie Sperren für diese angemeldet und zugeteilt bekommen hat. Der Fundamentalsatz des Sperrens zeigt, dass bei Einhaltung der folgenden Bedingungen Serialisierbarkeit gewährleistet ist [EGLT76]:

- Jedes Objekt, das von der Transaktion referenziert werden soll, muss mit einer Sperre belegt sein.
- Ist ein Objekt bereits mit einer unverträglichen Sperre für eine andere Transaktion belegt, so muss auf deren Freigabe gewartet werden.
- Sperren, die eine Transaktion bereits besitzt, werden nicht erneut angefordert.
- Sperren werden in zwei Phasen, der Wachstums- und der Schrumpfungsphase, angefordert und freigegeben.
- Die Transaktion gibt spätestens an ihrem Ende alle Sperren frei.

Das zweiphasige Vorgehen bei der Sperranforderung und -freigabe gibt den *Zwei-Phasen-Sperrprotokollen* (*Two-Phase Locking*, 2PL) ihren Namen. In der Wachstumsphase werden die Sperren angefordert und in der Schrumpfungsphase wieder freigegeben. Da der

Fundamentalsatz aber von einer fehlerfreien Betriebsumgebung ausgeht, ergeben sich Probleme, wenn Transaktionen scheitern. Wird beispielsweise eine Transaktion in ihrer Schrumpfungsphase zurückgesetzt, wurden eventuell bereits einige Sperren freigegeben und somit waren Änderungen für andere Transaktionen schon sichtbar. Um die Konsistenz der Daten sicherzustellen, müssen Transaktionen, die solche Änderungen gesehen haben, ebenfalls zurückgesetzt werden. Aufgrund von möglichen Abhängigkeiten unter den Transaktionen kann dies zu kaskadierenden Rücksetzungen führen, was die Leistungsfähigkeit des Systems einschränken kann. Diese Nachteile werden vermieden, indem die Schrumpfungsphase atomar und vollständig zum Transaktionsende durchgeführt wird, wenn das erfolgreiche Durchführen der Transaktion beispielsweise durch entsprechendes Logging sichergestellt ist. Dieses Verfahren wird *striktes Zwei-Phasen-Sperrprotokoll* genannt und vermeidet die Rücksetzung von abhängigen Transaktionen. Abbildung 3.1 zeigt beide Versionen des Zwei-Phasen-Sperrprotokolls.

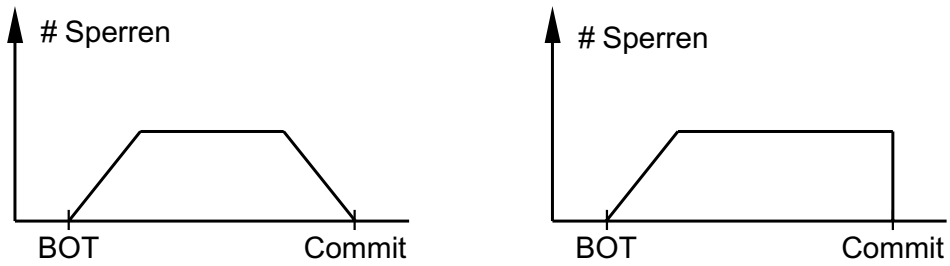

Abbildung 3.1: Sperranforderung und -freigabe beim Zwei-Phasen-Sperrprotokoll (links) und beim strikten Zwei-Phasen-Sperrprotokoll (rechts).

Das *RX-Sperrverfahren* stellt ein einfaches Sperrprotokoll dar, bei dem nur Lesesperren und Schreibsperren eingesetzt werden. Es kommt bei einem Objekt zu einem Sperrkonflikt, sobald eine Schreibsperre involviert ist. Folglich sind zwar Lesesperren mit sich selbst kompatibel, Schreibsperren sind jedoch weder mit Lesesperren noch mit sich selbst verträglich. Folglich werden bei z. B. gesetzter Schreibsperre alle weiteren Sperranforderungen abgelehnt. Kommt es zu einem Sperrkonflikt, muss die anfordernde Transaktion warten, bis die unverträgliche Sperre wieder freigegeben wird. Das RX-Verfahren erlaubt nicht das parallele Lesen und Schreiben eines Objektes, so dass eine Transaktion stets mit der aktuellsten Version eines Objektes arbeiten kann. Diese Eigenschaft ermöglicht eine chronologieerhaltende Serialisierbarkeit.

Neben den Sperren auf physischen Datenbankobjekten wie Sätzen, Indexeinträgen, Seiten oder Tabellen können auch *logische oder Prädikatsperren* angewendet werden. Bei der Sperranforderung $Lock(R, P, a)$ bezeichnet R die betroffene Relation, P ein logisches Prädikat und a den Zugriffswunsch. Mit einem Prädikat kann eine beliebige Menge von Objekten der Relation angesprochen werden. Ein Konflikt tritt genau dann ein, wenn mindestens eine der beiden Transaktionen schreiben möchte (also $a \neq Read$) und die Ergebnismengen der beiden Prädikate nicht disjunkt sind, d. h. eine Schnittmenge existieren könnte. Mit logischen Sperren kann das Phantom-Problem gelöst werden. Physische Sperrverfahren sperren nur tatsächlich vorhandene Objekte, aber nicht jene, die

Konsistenzstufe	Dirty Read	Non-Repeatable Read	Phantome
Read Uncommitted	+	+	+
Read Committed	-	+	+
Repeatable Read	-	-	+
Serializable	-	-	-

Tabelle 3.1: Die Konsistenzstufen von SQL:1999.

später eingefügt werden. Somit kann es nach der Einfügung bei einer erneuten Auswertung der Leseoperationen zu abweichenden Ergebnissen kommen, da nun die Phantome enthalten sind. Möchte man auch mit physischen Sperren Phantome vermeiden, muss man hierarchische Sperrverfahren einsetzen und die Sperre eine Hierarchiestufe höher anfordern. Dadurch werden die Objekte nicht einzeln, sondern als Menge gesperrt und Phantome sind mit enthalten.

Neben den (pessimistischen) Sperrverfahren gibt es auch *optimistische Synchronisationsverfahren* [KR81]. Sie gehen davon aus, dass Konflikte zwischen den Transaktionen eher selten auftreten und daher das vorbeugende Sperren von Objekten nicht notwendig ist. Statt die Transaktionen von Beginn an in ihrem Ablauf zu steuern, werden sie zunächst beliebig abgearbeitet. Erst bei Transaktionsende wird überprüft, ob es zu Konflikten mit anderen Transaktionen gekommen ist. Zu Transaktionsbeginn werden zunächst private Kopien von den zu bearbeitenden Objekten angelegt und auf diesen gearbeitet. Beim Commit wird überprüft, ob es zu einem Konflikt mit einer parallel laufenden Transaktion gekommen ist. Ist dies der Fall, werden die beteiligten Transaktionen zurückgesetzt. Wurde eine Änderungstransaktion erfolgreich ausgeführt, werden die Änderungen abschließend für andere Transaktionen sichtbar gemacht. Bei diesem Ansatz kommt es zu mehr Rücksetzungen als bei den Sperrverfahren; es können aber keine Deadlocks entstehen.

Ein weiteres Synchronisationsverfahren ist das *Zeitmarkenverfahren*. Hierbei werden den Transaktionen zu Beginn ihrer Ausführung eindeutige Zeitmarken zugeteilt, um die Serialisierbarkeit sicherzustellen. Bei der einfachsten Variante wird die Position einer Transaktion in der Serialisierungsreihenfolge von vornherein durch ihre Zeitmarke festgelegt [BN97]. Konflikte werden aufgelöst, indem Transaktionen in der Reihenfolge der Zeitmarken abgearbeitet werden. Auch bei diesen Verfahren sind keine Deadlocks möglich, da die Reihenfolge der Objektzugriffe aufgrund der Zeitmarken festgelegt ist. Dem stehen jedoch häufige Rücksetzungen gegenüber.

Zwar ist die Serialisierbarkeit aus Korrektheitsgründen wünschenswert, doch können Sperrverfahren zu häufigen Sperrkonflikten führen, die wiederum Leitungseinbußen verursachen. Daher unterstützen kommerzielle Datenbanksysteme schwächere Korrektheitskriterien als das der Serialisierbarkeit und nehmen bestimmte Mehrbenutzeranomalien in Kauf. Man unterscheidet mehrere Konsistenzstufen mit unterschiedlichen Anomalien.

Der SQL:1999-Standard unterscheidet vier Konsistenzstufen (*Isolation Level*). Die Definition der Konsistenzstufen erfolgt über die in Kauf genommenen Anomalien, um keine Vorgaben zur Implementierung zu treffen. Lost Updates sind generell zu verhindern. Tabelle 3.1 gibt einen Überblick über die einzelnen Stufen.

READ UNCOMMITTED ist die schwächste Konsistenzstufe und lässt Dirty Reads, Non-Repeatable Reads und Phantome zu. In SQL:1999 ist diese Stufe nur für Lesetransaktionen zulässig. READ COMMITTED verhindert neben Lost Updates auch Dirty Reads. Bei Repeatable Read sind nur noch Phantome möglich und in der restriktivsten Konsistenzstufe SERIALIZABLE sind sämtliche Anomalien ausgeschlossen.

Bei der Wahl der Konsistenzstufe ist SERIALIZABLE der Default-Wert. Unterstützt das DBS einen gewünschten Isolationsgrad nicht, ist die nächst-restriktivere Stufe zu wählen. Somit müssen SQL:1999-konforme DBS mindestens Serialisierbarkeit realisieren.

Beim Einsatz von Sperrverfahren ist immer mit Verklemmungen oder Deadlocks zu rechnen. Grund hierfür ist das Zusammentreffen von fünf Voraussetzungen:

- Mehrere Transaktionen greifen parallel auf Objekte zu.
- Es liegen exklusive Zugriffsanforderungen vor.
- Eine Transaktion, die bereits Sperren besitzt, fordert weitere Sperren an.
- Sperren werden nicht vorzeitig freigegeben.
- Es bestehen zyklische Wartebeziehungen zwischen mindestens zwei Transaktionen.

Insbesondere die zyklische Wartebeziehung macht den Deadlock aus und führt dazu, dass die beteiligten Transaktionen nicht weiterarbeiten können. Daher sollten Deadlocks möglichst schnell aufgelöst oder von vornherein verhindert werden. Es gibt vier generelle Ansätze zur Deadlock-Behandlung: Deadlock-Verhütung, Deadlock-Vermeidung, Timeout und Deadlock-Erkennung.

Die *Deadlock-Verhütung* verhindert die Entstehung von Verklemmungen, ohne dass spezielle Maßnahmen während der Abarbeitung der Transaktionen nötig sind. Dies ist beispielsweise durch ein *statisches* oder *Preclaiming-Sperrverfahren* möglich, bei dem eine Transaktion alle benötigten Sperren bereits bei Transaktionsbeginn anfordern muss. Sperrkonflikte können in zwei Varianten behandelt werden. Entweder werden bei einem Konflikt alle bereits erworbenen Sperren wieder freigegeben und zu einem späteren Zeitpunkt wieder angefordert oder es wird bei einem Konflikt auf die Freigabe der Sperre gewartet. Zyklische Wartebeziehungen werden dadurch vermieden, dass jede Transaktion die benötigten Sperren in einer auf den Objektbezeichnungen festgelegten Reihenfolge anfordert. Schwächen wie ein hoher Grad an Sperrkonflikten und lange Verzögerungen beim Erwerb von Sperren haben dazu geführt, dass dieser Ansatz praktisch keine Relevanz bei DBS hat. Bei den folgenden Ansätzen geht man daher wieder von dynamischen Sperrverfahren aus, d. h., die Sperren werden während der Transaktionsverarbeitung angefordert.

Um Deadlocks zu vermeiden, ist eine Laufzeitunterstützung notwendig, die bei einem Sperrkonflikt überprüft, ob die Blockierung der Transaktion eventuell einen Deadlock verursacht. Die *Deadlock-Vermeidung* wird durch Zurücksetzen einer der beteiligten Transaktionen bewerkstelligt. Hierfür gibt es mehrere Verfahren wie den *Immediate Restart*-Ansatz, das *Wait Depth Limited*-Verfahren [FRT92], die Verwendung von Transaktionszeitmarken [RSL78] und Zeitintervallen [Bay82, NW87]. Deadlock-Vermeidung

führt grundsätzlich zu mehr Rücksetzungen, da keine genaue Deadlock-Erkennung vorgenommen wird. Dadurch kommt es auch zu Rücksetzungen, ohne dass überhaupt ein Deadlock vorliegt.

Das *Timeout-Verfahren* ist durch eine sehr billige und einfache Vorgehensweise gekennzeichnet, indem es Transaktionen zurücksetzt, sobald ihre Wartezeit auf eine Sperre eine bestimmte Zeitdauer überschreitet. Problematisch ist jedoch die richtige Wahl des Timeout-Wertes. Bei einem zu hohen Wert bleiben Deadlocks zu lange bestehen und die betroffenen Transaktionen werden unnötig lang blockiert. Bei einem zu kleinen Wert kann es zu unnötig vielen Rücksetzungen kommen, ohne dass tatsächlich ein Deadlock vorliegt. Obwohl dieses Verfahren in den meisten Fällen zu vielen Rücksetzungen führt, ist es aufgrund seiner einfachen Realisierbarkeit beispielsweise problemlos zur Auflösung von Deadlocks in heterogenen Systemen geeignet.

Die *Deadlock-Erkennung* protokolliert alle Wartebeziehungen aktiver Transaktionen in einem Wartegraphen (*wait-for graph*) und erkennt Verklemmungen durch Zyklensuche in diesem Graphen. Die Auflösung eines Deadlocks geschieht auch hier durch das Zurücksetzen einer oder mehrerer Transaktionen. Von Vorteil ist, dass Rücksetzungen nur bei tatsächlich vorhandenen Deadlocks durchgeführt werden. Ihre Implementierung ist jedoch aufwendiger.

Recovery

Recovery-Mechanismen stellen die Atomarität und Dauerhaftigkeit der Transaktionen im Fehlerfall sicher. Dabei kann grundsätzlich zwischen Transaktionsfehlern, Systemfehlern und Geräte- bzw. Externspeicherfehlern unterschieden werden. Für jede dieser Fehlerarten sind unterschiedliche Recovery-Maßnahmen notwendig. Für alle Arten der Fehlerbehandlung wird eine Log-Datei benötigt, in der alle ausgeführten DB-Änderungen protokolliert werden. Am häufigsten kommen *Transaktionsfehler* vor, bei denen beispielsweise ein freiwilliger Transaktionsabbruch durch eine ROLLBACK-Anweisung seitens der Anwendung initiiert wird oder auch ein DBMS-seitiger Abbruch zur Auflösung von Verklemmungen stattfindet. Um die ACID-Eigenschaften und damit das „Alles oder Nichts“-Prinzip einzuhalten, werden solche Transaktionen mit Hilfe einer *Undo-Recovery* vollständig zurückgesetzt. Beim *Systemfehler* ist ein unterbrechungsfreier Betrieb des DBS nicht mehr möglich, da ein Hardware-, Software- oder Umgebungsfehler vorliegt. Allen gemeinsam ist, dass die Hauptspeicherinhalte verloren gegangen sind und damit nicht mehr von deren Korrektheit ausgegangen werden kann. Somit sind auch alle Änderungen, die zum Fehlerzeitpunkt nur im Hauptspeicher vorlagen, verloren und müssen rekonstruiert werden. In diesem Fall erfolgt die Fehlerbehandlung durch die *Crash-Recovery*, von der alle zum Fehlerzeitpunkt laufenden Transaktionen betroffen sind. Ebenso müssen die Änderungen all jener erfolgreich beendeten Transaktionen wiederholt werden, die noch nicht in die permanente Datenbank gelangten. *Externspeicherfehler* werden vor allem durch den Ausfall von Magnetplatten verursacht. Die Behandlung solcher Fehler muss dafür sorgen, dass die permanente Datenbank weitergenutzt werden kann und Datenverlust verhindert wird. Zu diesem Zweck wird eine *Redo-Recovery* durchgeführt, welche die verloren gegangenen Änderungen mit Hilfe von Archivkopien und Log-Dateien rekonstruiert. Eine weitere Variante stellt die

Katastrophen-Recovery dar, welche zum Einsatz kommt, wenn beispielsweise ganze Rechenzentren zerstört werden.

Für die Fehlerbehandlung sind Log-Dateien notwendig, welche die DB-Änderungen mitprotokollieren. In den Log-Dateien werden die durchgeführten Transformationen des Datenbankzustandes beschrieben und im Falle einer Undo-Recovery dazu verwendet, um den alten Zustand wiederherzustellen. Bei einer Redo-Recovery hingegen wird der neue Zustand vom ungeänderten DB-Zustand aus rekonstruiert.

Das Logging kann in mehreren Formen realisiert werden. Es kann als physisches oder logisches sowie physiologisches Logging umgesetzt werden. Während das physische Logging Informationen auf Ebene physischer Objekte Seiten oder Datenbanksätze verwaltet, protokolliert das logische Logging nicht die Objekte selbst, sondern die Informationen über die Änderungsoperationen und deren Parameter. Die Kombination von physischem Eintrags-Logging mit einem logischen Logging wird von [GR93] als physiologisches Logging bezeichnet. Wie bei physischem Logging werden die Log-Sätze auf Basis bestimmter Seiten geführt, die Änderungen in den Seiten werden jedoch als Operationen und damit logisch festgehalten.

3.1.2.2 Aspekte verteilter heterogener Transaktionsverwaltung

Nachdem wir den Transaktionsbegriff für zentrale Datenbanksysteme erläutert haben, erörtern wir nun, welche zusätzlichen Aspekte bei föderierten Datenbanksystemen hinzukommen. Bei verteilten Transaktionen wird die globale Transaktion in mehrere Subtransaktionen aufgeteilt, die von den zu integrierenden Systemen und deren Transaktionsverwaltungen ausgeführt werden. Die lokalen Systeme sind autonom und weisen heterogene Transaktionsverwaltungen auf, d. h., es werden unterschiedliche Scheduling-Verfahren, Sperrverfahren und nicht übereinstimmende Commit-Protokolle eingesetzt. Trotz der Heterogenitäten ist eine koordinierte Ausführung der globalen Transaktionen über die lokalen Systeme hinweg unter Einhaltung des ACID-Paradigmas erwünscht.

Grundsätzlich sind zwei Ansätze für eine globale Transaktionsverwaltung denkbar. Die erste Möglichkeit setzt voraus, dass die globale Transaktionsverwaltung Kenntnisse über die lokalen Transaktionsverwaltungen hat und dieses Wissen bei der Weitergabe einer Subtransaktion an das lokale System berücksichtigt. Da die globale Transaktionsverwaltung jedoch keine Informationen über die in Ausführung befindlichen lokalen Transaktionen hat, kann sie das Verhalten der lokalen Systeme nicht einschätzen. Beim zweiten Ansatz ist im Gegensatz dazu nichts von den lokalen Transaktionsverfahren bekannt. Es werden lediglich Annahmen über eine minimale Menge an Grundeigenschaften der lokalen Transaktionen getroffen. Alles, was darüber hinaus an Eigenschaften und Funktionalität benötigt wird, übernimmt die globale Transaktionsverwaltung.

Bevor Lösungsansätze für erweiterte Transaktionsmodelle betrachtet werden, geben wir einen Überblick über die Probleme heterogener Transaktionsverwaltung in einem FDBS. Bei den betrachteten Fällen handelt es sich um autonome Systeme, deren Leistungsfähigkeit und Funktionalität durch die Integration nicht beeinflusst werden sollen. In den meisten Fällen ist dem FDBS nur wenig über das System und dessen Eigenschaften bekannt. Änderungen als Folge der Integration sind nicht möglich, wie. z. B. das Erweitern der Systemschnittstellen oder Änderungen der lokalen Schemata. Die möglichen Probleme lassen sich in drei große Kategorien einteilen:

1. Globale Serialisierbarkeit,
2. globale Atomarität und
3. globale Deadlock-Erkennung und -Vermeidung.

In den folgenden Abschnitten erläutern wir diese Punkte näher.

Globale Serialisierbarkeit

Das FDBS greift auf heterogene DBS zu, die unterschiedliche Synchronisationsverfahren zur Serialisierung ihrer Transaktionen einsetzen können. Da der Großteil der Systeme autonom arbeitet und ursprünglich nicht dazu gedacht war, in einem Verbund mit anderen Systemen zu funktionieren, geben sie im Allgemeinen keine Kontrollinformationen nach außen. Folglich verfügt das FDBS über keine Informationen zum Verlauf der Transaktionen in den lokalen Systemen. Es können daher nicht die bekannten Verfahren für traditionelle DBS eingesetzt werden, weil diese homogene Mechanismen voraussetzen. Da das FDBS somit auch keinen Einfluss auf die Ausführungsreihenfolgen in den zu integrierenden DBS hat, können zwei lokale DBS zueinander widersprüchliche Ausführungsreihenfolgen ermitteln, die eine globale Serialisierung unmöglich machen, wie das folgende Beispiel zeigt [Con97].

Beispiel:

Ein FDBS besteht aus zwei lokalen DBS $LDBS_1$ und $LDBS_2$. Die Datenobjekte a und b werden von $LDBS_1$ verwaltet und die Objekte c und d von $LDBS_2$. Zwei globale Transaktion führen nun die folgenden Operationen aus:

GT_1: $r_1(a)$ $r_1(c)$
GT_2: $r_2(b)$ $r_2(d)$

Gleichzeitig werden zwei lokale Transaktionen ausgeführt – LT_3 auf $LDBS_1$ und LT_4 auf $LDBS_2$:

LT_3: $w_3(a)$ $w_3(b)$
LT_4: $w_4(c)$ $w_4(d)$

Die beiden möglichen lokalen Schedules gestalten sich wie folgt (mit c_i als Commit-Operation von GT_i oder LT_i):

S_1: $r_1(a)$ c_1 $w_3(a)$ $w_3(b)$ c_3 $r_2(b)$ c_2
S_2: $w_4(c)$ $r_2(d)$ c_2 $w_4(d)$ c_4 $r_1(c)$ c_1

Für diese Schedules ergeben sich die folgenden lokalen Ausführungsreihenfolgen:

$LDBS_1$: $GT_1 \rightarrow LT_3 \rightarrow GT_2$
$LDBS_2$: $GT_2 \rightarrow LT_4 \rightarrow GT_1$

Eine globale Serialisierung ist nicht möglich, da in $LDBS1$ GT_1 vor GT_2 ausgeführt wird und in $LDBS2$ gerade umgekehrt. Somit entsteht auf globaler Ebene ein nicht auflösbarer Konflikt.

Wie das Beispiel veranschaulicht, liegt das Problem darin, dass die Subtransaktionen zweier globaler Transaktionen in den DBS in unterschiedlicher Reihenfolge ausgeführt werden, da lokale Transaktionen bei der Synchronisation ebenfalls beachtet werden. Folglich kann auf globaler Ebene nicht serialisiert werden, weil es zu indirekten Konflikten kommt. Da das FDBS keine Kenntnisse über die lokalen Transaktionen hat, kann es die Synchronisation der globalen Transaktionen nicht entsprechend vornehmen.
Forschungsarbeiten vor allem in den neunziger Jahre haben gezeigt, dass die Neudefinition des Serialisierbarkeitskriteriums dieses Problem zu umgehen hilft. In [ZE93] werden mit Konfliktkettenserialisierbarkeit (*chain conflict serializability*), Teilungsserialisierbarkeit (*shared serializability*) und hybrider Serialisierbarkeit (*hybrid serializability*) drei neue Korrektheitskriterien vorgestellt, die eine globale Serialisierbarkeit garantieren, wenn die DBS lokal serialisierbare Ausführungsreihenfolgen erzeugen.

Globale Atomarität

Im FDBS-Umfeld bedeutet Atomarität, dass die Subtransaktionen der globalen Transaktionen entweder alle mit Commit oder alle mit Abort abschließen. Der Einsatz des Zwei-Phasen-Commit-Protokolls (2PC-Protokoll) scheint hier naheliegend, doch wird dabei vorausgesetzt, dass die lokalen Systeme über einen sichtbaren Prepare-to-Commit-Zustand verfügen. Bei der Heterogenität der lokalen Systeme kann jedoch nicht davon ausgegangen werden, dass sie einen solchen Zustand unterstützen. Vielmehr muss man damit rechnen, dass die Ausführungsautonomie zu beliebigen Abbrüchen führt. Das folgende Beispiel verdeutlicht die Problematik [Con97].

Beispiel:

Es liegt erneut ein FDBS mit zwei lokalen Systemen $LDBS_1$ und $LDBS_2$ vor, wobei $LDBS_1$ das Datenobjekt a verwaltet und $LDBS_2$ das Objekt c. Nun soll die folgende globale Transaktion ausgeführt werden:

GT_1: $r_1(a)$ $w_1(a)$ $w_1(c)$

Nach erfolgreicher Durchführung der einzelnen Operationen schickt das FDBS das Commit an die beiden lokalen Systeme. Während $LDBS_2$ die Subtransaktion von GT_1 erfolgreich abschließt, bricht $LDBS_1$ die Subtransaktion ab, bevor die globale Commit-Anforderung eintrifft. Der Abbruch führt dazu, dass die lokalen Effekte der Subtransaktion zurückgesetzt werden. Anschließend wird in $LDBS_1$ eine lokale Transaktion abgearbeitet:

LT_2: $r_2(a)$ $w_2(a)$

Die Atomarität der globalen Transaktion ist nun verletzt, da die Subtransaktion in $LDBS_2$ erfolgreich abgeschlossen wurde, während die in $LDBS_1$ abgebrochen wurde. Die erfolgreiche Subtransaktion kann nicht mehr abgebrochen werden, daher könnte das FDBS versuchen, die Subtransaktion in $LDBS_1$ zu wiederholen. Da hier jedoch die lokale Transaktion LT_2 das Datenobjekt a bereits gelesen und geschrieben hat, kann a nicht erneut von der zu wiederholenden Subtransaktion gelesen werden, weil dadurch eventuell das Ergebnis der bereits erfolgreichen Subtransaktion nachträglich geändert werden müsste. Somit bleibt nur der Weg, nicht

die ganze abgebrochene Subtransaktion zu wiederholen, sondern lediglich durch Wiederholung der Schreiboperationen zumindest den Effekt der Schreiboperation zu bewahren. Folglich muss im $LDBS_1$ die Operation $w_1(a)$ wiederholt werden. Dies geschieht aber durch eine neue Transaktion LT_3, da die ursprüngliche abgebrochen wurde. Im $LDBS_1$ ergibt sich folgender commit-abgeschlossener Schedule:

S_1: $r_2(a)\ w_2(a)\ c_2\ w_3(a)\ c_3$

Für das FDBS ist $w_3(a)$ aber Teil der globalen Transaktion GT_1, so dass sich der Ablauf in $LDBS_1$ wie folgt gestaltet:

$r_1(a)\ r_2(a)\ w_2(a)\ c_2\ w_3(a)\ c_1$

Somit ergibt sich in $LDBS_1$ eine Abhängigkeitsfolge $GT_1 \rightarrow LT_1 \rightarrow GT_1$, so dass die resultierende Ausführungsreihenfolge nicht serialisierbar ist.

Sind die Transaktionsverwaltungen des FDBS und der lokalen Systeme völlig losgelöst voneinander, können Atomarität und Serialisierbarkeit im FDBS nicht gleichzeitig vollständig durchgesetzt werden. Ein sichtbarer Prepare-to-Commit-Zustand in allen lokalen Systemen könnte das Problem entschärfen. Leider stellen gerade Legacy-Systeme solche Zustände nicht zur Verfügung. Andere Ansätze machen zur Problemlösung bestimmte Annahmen über die zu integrierenden Systeme, die auch häufig zutreffen. Man geht beispielsweise davon aus, dass in den lokalen Systemen das 2PL-Protokoll verwendet wird. Ein anderer Ansatz führt so genannte 2PC-Agenten ein [WV90], die für die lokalen Systeme einen Prepare-to-Commit-Zustand simulieren. Eine weitere Klasse von Ansätzen werden durch Mehrebenen-Transaktionen [Wei91] repräsentiert, auf die wir später näher eingehen.

Globale Deadlock-Erkennung und -Vermeidung

Zur Sicherstellung der Isolation von Transaktionen findet eine Sychronisation der Transaktionen statt, die in den meisten Fällen mit Hilfe eines Sperrprotokolls realisiert wird. Bei der Vergabe von Sperren kann es zu Verklemmungen oder Deadlocks kommen, wenn zwei oder mehr Prozesse oder Systeme aufeinander warten. Im Transaktionsumfeld bedeutet das, dass z. B. ein Prozess die Ressource a besitzt, zur weiteren Arbeit aber zusätzlich Ressource b benötigt. Diese besitzt momentan aber ein anderer Prozess, der aber wiederum auf die Freigabe von Ressource a wartet. Als Ergebnis warten beide Prozesse wechselseitig darauf, dass der andere Prozess die jeweils benötigte Ressource freigibt. Bei FDBS verschärft sich das Problem der Deadlocks. Zwar werden Verklemmungen in den lokalen Systemen erkannt und können behandelt werden, systemübergreifende Deadlocks sind aber unter Umständen nicht zu erkennen, wie das folgende Beispiel zeigt [Con97]:

Beispiel:

Ein FDBS besteht aus zwei lokalen DBS $LDBS_1$ und $LDBS_2$. Die Datenobjekte a und b werden von $LDBS_1$ verwaltet und die Objekte c und d von $LDBS_2$. Beide

Systeme verwenden 2PL-Protokolle, die lokale Serialisierbarkeit garantieren. Es sind folgende globale Transaktionen gegeben:

GT_1: $r_1(a)$ $r_1(d)$
GT_2: $r_2(c)$ $r_2(b)$

Außerdem werden folgende lokalen Transaktionen LT_3 und LT_4 in $LDBS_1$ bzw. $LDBS_2$ ausgeführt.

LT_3: $w_3(b)$ $w_3(a)$
LT_4: $w_4(d)$ $w_4(c)$

Mit diesen vier Transaktionen kann eine globale Verklemmung auftreten, und zwar genau dann, wenn die beiden globalen Transaktionen jeweils ihre erste Operation ausgeführt haben und anschließend LT_3 in $LDBS_1$ die Operation $w_3(b)$ und LT_4 in $LDBS_2$ die Operation $w_4(d)$ ausführt. In diesem Fall ergeben sich folgende Wartebeziehungen in den lokalen Systemen:

$LDBS_1$: $LT_3 \rightarrow GT_1$
$LDBS_2$: $LT_4 \rightarrow GT_2$

Werden im nächsten Schritt die jeweils zweiten Operationen der globalen Transaktionen ausgeführt, liegt eine globale Verklemmung vor, da nun folgende systemübergreifende Wartebeziehungen vorliegen:

$FDBS$: $LT_3 \rightarrow GT_1 \rightarrow LT_4 \rightarrow GT_2 \rightarrow LT_3$

Somit liegt eine zyklische Abhängigkeit vor, d. h., die zwei globalen und die zwei lokalen Transaktionen warten zyklisch aufeinander.

Dass der Deadlock nicht erkannt wird, liegt in erster Linie daran, dass die Transaktionsverwaltung des FDBS keine Informationen über die lokalen Transaktionen hat und somit auch nicht wissen kann, welche Sperren von den lokalen Transaktionen gehalten werden. Hauptursache sind fehlende Informationen in den einzelnen Systemen, um eine globale Verklemmung zu erkennen. Ein möglicher Lösungsansatz ist der Austausch der benötigten Informationen zwischen den beteiligten Systemen (z. B. in Form eines Wartegraphen). Da die lokalen Systeme jedoch autonom sind, ist dies kein gangbarer Weg. Andere Lösungsansätze arbeiten mit Timeout-Verfahren oder bauen einen potentiellen globalen Wartegraphen auf, der eine Obermenge des tatsächlichen globalen Wartegraphen darstellt und somit auf jeden Fall echte Deadlocks erkennt [BST90, BST92]. Da er jedoch gröber ist, beinhaltet er unter Umständen auch falsche Deadlocks und führt somit zu unnötigen Rücksetzungen von Transaktionen. Weitere Ansätze können in [EH88, OV91, GR93, ODV94] nachgelesen werden.

3.1.2.3 Erweiterte Transaktionskonzepte

Das klassische Transaktionskonzept hat sich für Datenbankanwendungen bewährt und wird zunehmend auch in anderen Anwendungsbereichen eingesetzt. Da es jedoch von flachen Transaktionen ausgeht, sind damit für einige Anwendungsfälle zu viele Beschränkungen verbunden. Zu diesen Beschränkungen gehört z. B. die Ausrichtung auf

kurze Transaktionen, was dazu führt, dass komplexe und länger andauernde Verarbeitungsvorgänge bei einem Fehler das vollständige Zurücksetzen der Transaktion und damit einen großen Arbeitsverlust fordern. Daher wurde eine Vielzahl an erweiterten Transaktionsmodellen entwickelt, wovon wir die geschachtelten Transaktionen und insbesondere die Mehrebenen-Transaktionen in den nächsten Abschnitten beschreiben.

Geschachtelte Transaktionen

Geschachtelte Transaktionen (*nested transactions*) sind intern in eine Hierarchie von Subtransaktionen untergliedert [Mos85]. Beim Aufruf der Transaktion können die Subtransaktionen als eigenständige Transaktionen von unterschiedlichen Systemen ausgeführt werden. Die Subtransaktionen sind jedoch nicht unabhängig, sondern hängen von der aufrufenden Transaktion ab.

Die Binnenstruktur einer Transaktion entspricht einem Transaktionsbaum, dessen Unterbäume selbst entweder geschachtelte Transaktionen oder flache Transaktionen (*flat transactions*) sind. Der Wurzelknoten heißt Top-Level-Transaktion, die inneren Knoten repräsentieren Subtransaktionen und die Blätter sind immer flache Transaktionen und müssen nicht auf einer Ebene liegen. Eine Subtransaktion kann erfolgreich abschließen oder zurückgesetzt werden. Ein Commit der Subtransaktion wird jedoch erst wirksam, wenn auch die Elterntransaktion ein Commit durchgeführt hat. Folglich kann jede Subtransaktion nur dann ein Commit durchführen, wenn auch die Top-Level-Transaktion mit einem Commit abschließt. Ein Rollback einer Transaktion verursacht immer ein Zurücksetzen aller ihrer Subtransaktionen.

Für die Zusammenarbeit in geschachtelten Transaktionen gelten die folgenden Regeln:

- *Commit-Regel*: Beim Commit einer Subtransaktion werden deren Ergebnisse nur der Vatertransaktion sichtbar; es ist somit ein vorläufiges Commit. Erst wenn alle übergeordneten Transaktionen bis hin zur Top-Level-Transaktion ebenfalls ein Commit durchgeführt haben, ist auch das Commit der Subtransaktion endgültig und damit dauerhaft.

- *Rollback-Regel*: Wenn eine Transaktion oder Subtransaktion zurückgesetzt wird, werden alle ihre Subtransaktionen ebenfalls zurückgesetzt, unabhängig davon, ob diese bereits erfolgreich mit Commit abgeschlossen haben. Die Rücksetzung einer Subtransaktion führt im Allgemeinen jedoch nicht zur Rücksetzung der Vatertransaktion. Kommunizieren Vater- und Subtransaktion über eine Konversationsschnittstelle, müssen eventuell auch übergeordnete Transaktionen abgebrochen werden, da die Vatertransaktion bereits Ergebnisse der Subtransaktion erhalten hat [HR87].

- *Sichtbarkeitsregel*: Alle Änderungen einer Subtransaktion werden bei ihrem Commit für die Vatertransaktion sichtbar gemacht. Alle Objekte, die von der Vatertransaktion gehalten werden, können den Subtransaktionen zugänglich gemacht werden. Änderungen einer Subtransaktion sind aber nicht für parallel dazu laufende Geschwistertransaktionen sichtbar.

Zusammenfassend halten wir für geschachtelte Transaktionen fest, dass die Subtransaktionen nicht ganz den klassischen flachen Transaktionen entsprechen, da ihre Gültigkeit

nur innerhalb der sie (direkt oder indirekt) aufrufenden Transaktionen definiert ist. Die Commit- und Rollback-Regeln schließen die Atomarität der Subtransaktionen ein und die Sichtbarkeitsregel sorgt für die Isolation hinsichtlich parallel laufender Subtransaktionen. Allerdings sind Änderungen von Subtransaktionen nicht dauerhaft, da ihr erfolgreicher Abschluss aufgrund ihrer Abhängigkeit von ihren Vorgängern ungültig gemacht werden kann. Die Eigenschaft der Konsistenz gilt nur für die von der Subtransaktion realisierten lokalen Funktion. Die Konsistenz der Datenbank wird erst mit Abschluss der Top-Level-Transaktion sichergestellt. Somit treffen für Subtransaktionen nur die Eigenschaften A und I zu.

Die bisher beschriebenen geschachtelten Transaktionen werden auch *geschlossen geschachtelte Transaktionen* (*closed nested transactions*) genannt, da der Abschluss und die Sichtbarkeit der Ergebnisse nur innerhalb der betreffenden Gesamttransaktion besteht. Gegenüber anderen Transaktionen sind die geschachtelten Transaktionen aber geschlossen. Im Gegensatz dazu ist der erfolgreiche Abschluss einer Subtransaktion bei *offen geschachtelten Transaktionen* (*open nested transactions*) unabhängig vom Abschluss übergeordneter Transaktionen. Folglich kann eine Subtransaktion erfolgreich abschließen und ihre Ergebnisse veröffentlichen, auch wenn die Vatertransaktion zurückgesetzt wird. Dies wiederum hat Auswirkungen auf Synchronisations- und Recovery-Verfahren.

Da Subtransaktionen ihre Ergebnisse vorzeitig bekannt geben, kommt es zu Mehrbenutzeranomalien und damit zu einer Verletzung der Serialisierbarkeit. Es sind daher zusätzliche Synchronisationsmaßnahmen notwendig. Zudem kann das Zurücksetzen einer Subtransaktion nicht mehr über eine zustandsorientierte Undo-Recovery erfolgen, sondern muss logisch mittels kompensierender Subtransaktionen verwirklicht werden. Mit diesem Verfahren wird nicht der ursprüngliche Zustand der Datenbank wiederhergestellt, sondern sie enthält lediglich nicht mehr die Änderungen der kompensierten Subtransaktion. Der Einsatz von Kompensationen bringt einige Probleme mit sich:

- Für jede Operation muss eine entsprechende Kompensationsoperation zur Verfügung gestellt werden. Dies verursacht zusätzlichen Aufwand bei der Erstellung von Anwendungen. Andererseits sind in vielen Anwendungen die entsprechenden Operationenpärchen bereits vorhanden.
- Die Ausführung einer Kompensation darf nicht scheitern, da sonst die Subtransaktion nicht zurückgesetzt werden kann. Scheitert die Kompensation auch nach mehreren Versuchen, so muss meistens eine manuelle Fehlerbehebung stattfinden.
- Es gibt Operationen, die nicht kompensiert werden können, da ihre Auswirkungen nicht umkehrbar sind (z. B. das Bohren eines Loches). Solche nicht kompensierbaren Operationen sollten daher immer am Ende einer Gesamttransaktion ausgeführt werden, wenn der Erfolg der Transaktion bis dahin sichergestellt ist.

Eine ausführliche Beschreibung von offen geschachtelten Transaktionen findet sich u. a. in [WS92, MRKN92].

Mehrebenen-Transaktionen

Eine Sonderform der offen geschachtelten Transaktionen sind die Mehrebenen-Transaktionen. Sie nutzen die semantischen Eigenschaften der Operationen, um die Isolation

von konkurrierenden Transaktionen zu lockern. Die Schachtelung der Transaktionen und ihrer Operationen erfolgt längs der Abbildungshierarchie einer Schichtenarchitektur [Wei91]. Bei einer festen Anzahl an Schichten werden Operationen der Ebene i durch Operationen der darunter liegenden Ebene $i - 1$ realisiert. Man kann daher die Ebenen der Abbildungshierarchie auch als verschiedene Ebenen der Abstraktion betrachten.

Bei dem Mehrebenen-Transaktionsmodell werden Operationen im Rahmen einer Subtransaktion atomar ausgeführt. Die Gesamttransaktion soll nach wie vor ACID-Eigenschaften aufweisen. Die Sperren werden jedoch nur für die Operationen der obersten Ebene bis zum Transaktionsende gehalten, während die Operationen tiefer liegender Schichten gemäß offen geschachtelter Transaktionen ihre Ergebnisse vorzeitig bereits beim Ende der Subtransaktion sichtbar machen. So werden die Sperrzeiten der Objekte der unteren Schichten relativ kurz gehalten und das Konfliktpotential erheblich reduziert. Die Serialisierbarkeit wird durch die gehaltenen Sperren auf höheren Ebenen trotzdem gewahrt.

Dass serialisierbare Transaktionsabläufe mit einem Minimum an Synchronisationskonflikten erreicht werden können, erweist sich als großer Vorteil dieses Modells. Im Gegensatz dazu stellt die Recovery einen Nachteil dar, weil auf jeder Ebene Logging- und Recovery-Funktionen vorhanden sein müssen, um die Atomarität der Subtransaktionen zu gewährleisten. Andererseits erlaubt dieses Vorgehen den Einsatz von heterogenen Transaktionsverwaltungen auf den unterschiedlichen Ebenen.

Das Zurücksetzen von Subtransaktionen ist wegen der vorzeitigen Sperrenfreigabe nur durch Kompensationen möglich, so dass für jede Operation eine kompensierende Gegenoperation vorhanden sein muss. Dies ist bei DBS-internen Operationen auf Satz- und Seitenebene relativ einfach möglich (z. B. Löschen und Einfügen). Auf dieser Ebene sind im Gegensatz zu Anwendungsfunktionen alle Operationen kompensierbar. Das Mehrebenen-Transaktionsmodell ist nicht nur in den Schichten der DBS-internen Abbildungshierarchie einsetzbar, sondern lässt sich prinzipiell auch auf anwendungsspezifische Funktionen erweitern. Um jedoch die Synchronisation zu ermöglichen, muss die Konfliktverträglichkeit der Funktionen definiert werden. Jedoch ist dies in der Regel äußerst schwierig, da eine fast unüberschaubare Zahl an Kompatibilitäten zwischen den Funktionen festgelegt werden müsste.

Wie geschlossen geschachtelte Transaktionen gewährleisten Mehrebenen-Transaktionen die ACID-Eigenschaften von Transaktionen. Mit offen geschachtelten Subtransaktionen werden darüber hinaus Verbesserungen hinsichtlich des Grades an Nebenläufigkeit der Transaktionen erreicht.

3.2 SQL:1999 und verwandte Standards

Die aktuelle Version des Standards SQL, SQL:1999 genannt, bringt eine ganze Reihe an neuen Funktionalitäten mit, die für diese Arbeit insbesondere hinsichtlich der standardkonformen Integration von heterogenen Datenquellen wichtig sind. Wir beschreiben zunächst die wichtigsten Neuerungen in SQL:1999 und geben anschließend einen Einblick in benutzerdefinierte Tabellenfunktionen und SQL/MED als Integrationsmechanismen.

3.2.1 SQL:1999

Vor dem Standard SQL:1999 wurde immer wieder bemängelt, dass die relationale Datenbanktechnologie nicht die Modellierung von komplexen Datenobjekten unterstützt und man durch umständliche Modellierung von beispielsweise geschachtelten Strukturen Konsistenzprobleme verschärft. Ebenso fehlt bei SQL der Anwendungsbezug, so dass zur Gewährleistung der Anwendungssemantik vieles davon in der Anwendungslogik nachgebildet und geprüft werden muss. Die Spezifikation der Anwendungssemantik kann nicht im DB-Schema spezifiziert und damit auch nicht vom DBMS überwacht werden.

SQL:1999 hat dazu beigetragen, diese Situation zu verbessern, indem der Funktionsumfang stark erweitert wurde. In diesem Abschnitt geben wir einen Überblick über die neuen Funktionen, die man in relationale und objektorientierte Eigenschaften aufteilt [EM99].

3.2.1.1 Relationale Eigenschaften

Die Erweiterungen der relationalen Eigenschaften von SQL lassen sich in fünf Gruppen aufteilen: neue Datentypen, neue Prädikate, erweiterte Semantik, zusätzliche Sicherheitskonzepte und aktive DB-Eigenschaften. Im Folgenden geben wir einen Überblick über diese fünf Erweiterungen.

Neue Datentypen

SQL:1999 definiert vier neue Datentypen. Der erste Typ ist der `LOB`-Datentyp (`LARGE OBJECT`), der in zwei Varianten unterstützt wird: `CHARACTER LOB` (`CLOB`) und `BINARY LOB` (`BLOB`). Mit Hilfe von `LOB`s können größere Datenmengen als ein Spaltenwert angelegt werden. Es gelten jedoch einige Einschränkungen bei der Verwendung von `LOB`s. So können sie z. B. nicht als Primary Key eingesetzt und nicht in beliebigen Vergleichsprädikaten verwendet werden. Eine weitere Besonderheit ist die Manipulation der `LOB`-Werte durch so genannte Locators, damit nicht der ganze `LOB`-Wert zwischen Client und Server transferiert werden muss.

Der Datentyp `BOOLEAN` ist ebenfalls neu und ermöglicht das direkte Ablegen der Werte `true`, `false` und `unknown`. Außerdem können komplexere Kombinationen von Prädikaten nun einfacher und damit benutzerfreundlicher ausgedrückt werden.

Des Weiteren gibt es zwei neue zusammengesetzte Datentypen: `ARRAY` und `ROW`. Mit dem `ARRAY`-Typ können Wertemengen in einer Tabellenspalte abgelegt werden. Mit Hilfe des `ROW`-Typs können strukturierte Werte wie z. B. eine Adresse in einer Tabellenspalte gespeichert werden. Bei diesen beiden Datentypen kam Kritik auf, dass durch sie die 1. Normalform nicht mehr unterstützt wird. Nachdem jedoch beide Typen zerlegt werden können, verletzen sie nicht wirklich den Charakter der 1. Normalform.

Die Einführung der *distinct*-Datentypen erlaubt die Definition von Datentypen auf bestehenden Basisdatentypen und erhöht die Typsicherheit. Dadurch ist es nicht erlaubt, verschiedene Typen in einem Ausdruck zu mischen, was bisher möglich war, wenn beide Werte z. B. vom Typ `Integer` waren.

Neue Prädikate

SQL:1999 hat drei neue Prädikate, wovon eines bei den objektorientierten Eigenschaften betrachtet wird. Die anderen beiden sind das `SIMILAR`- und das `DISTINCT`-Prädikat. Das Prädikat `SIMILAR` ermöglicht eine feinere Suche in Zeichenketten, als dies bisher möglich war. Man kann nun reguläre Ausdrücke spezifizieren. Das Prädikat `DISTINCT` ist ähnlich dem `UNIQUE`-Prädikat. Es prüft, ob zwei Werte gleich sind oder nicht. Sind die zu vergleichenden Werte beide `NULL`-Werte, dann sind sie nicht „distinct“. Auch wenn man nicht sagen kann, ob sie tatsächlich gleich sind oder nicht.

Neue Semantik

Zu den wichtigsten Funktionen, die neues Verhalten von SQL bewirken, gehört die Erweiterung der Sichten, welche direkt geändert werden können. Außerdem werden nun rekursive Anfragen unterstützt, um beispielsweise Stücklisten leichter verarbeiten zu können. Außerdem wurden Locators auch für `Arrays` eingeführt, da auch diese wie die `LOB`s zu groß werden, um sie zum Client zu übertragen. Abschließend sollen noch die *Savepoints* genannt werden, die als eine Art Subtransaktion betrachtet werden können. Sie unterstützen das Zurücksetzen einer Transaktion bis zu diesem Punkt, d. h. das partielle Zurücksetzen einer Transaktion.

Erweiterte Sicherheitsmechanismen

SQL:1999 beinhaltet das in den meisten Produkten bereits unterstützte Konzept der Rollen. Folglich können Rollen nun Zugriffsrechte vergeben werden, wie sie bisher einzelnen Benutzern zugeteilt wurden. Das Rollenkonzept erleichtert die Verwaltung der Zugriffskontrolle erheblich.

Aktive Datenbank

Die Idee von aktiven Datenbanken wird vor allem durch den Einsatz von Triggern gestützt, die von den Datenbank-Herstellern bereits lange implementiert werden. Der Standard zieht nun nach und nimmt Trigger mit auf. Trigger erlauben dem Entwickler die Datenbank anzuweisen, gewisse Operationen durchzuführen, immer wenn spezifizierte Operationen auf bestimmten Tabellen ausgeführt werden.

3.2.1.2 Objektorientierte Eigenschaften

Der Objektorientierung galt besondere Beachtung bei der Entwicklung von SQL:1999. Ziel war es, die Sprache um objektorientierte Konzepte zu erweitern, die wir in den folgenden Abschnitten kurz aufzeigen.

Strukturierte, benutzerdefinierte Typen

Eine der grundlegenden Erleichterungen sind die strukturierten, benutzerdefinierten Datentypen. Sie unterscheiden sich von den distinct-Datentypen darin, dass sie Strukturen enthalten und damit komplex sein können. Distinct-Datentypen sind hingegen auf eingebaute Basistypen beschränkt. Zu den wichtigsten Eigenschaften der strukturierten,

benutzerdefinierten Datentypen gehört, dass sie Attribute enthalten können, ihr Verhalten durch Methoden, Funktionen und Prozeduren bestimmt wird und sie in Typhierarchien enthalten sein können. Folglich wird Vererbung unterstützt, die aber auf einfache Vererbung beschränkt ist (vergleichbar mit dem Vererbungskonzept von Java).

Funktionen und Methoden

SQL:1999 beinhaltet benutzerdefinierte Methoden, Funktionen und Prozeduren, die in SQL oder anderen Programmiersprachen geschrieben sind (SQL/PSM [ISO02]). Man unterscheidet klar zwischen Funktionen und Methoden. Methoden sind Funktionen mit bestimmten Einschränkungen. Dazu gehört, dass Methoden an einen bestimmten benutzerdefinierten Typ gebunden sind und Methoden in demselben Schema wie der zugehörige benutzerdefinierte Typ abgelegt werden müssen.

Funktionale und „.“-Notationen

Der Zugriff auf Attribute eines strukturierten, benutzerdefinierten Typs erfolgt in zwei Varianten. Entweder wird das Attribut mit einem Punkt an den Spaltennamen angehängt (z. B. `mitarbeiter.gehalt`) oder über eine Funktionsnotation angesprochen (z. B. `gehalt(mitarbeiter)`). Methoden können aber nur mit der Punkt-Notation aufgerufen werden.

Objekte

Zur Unterstützung von Objekten definiert SQL:1999 getypte Tabellen (*typed tables*), deren Spaltendefinition von den Attributen eines strukturierten Typs abgeleitet ist. Dabei wird jedes Attribut auf eine Spalte abgebildet und die zugehörigen Methoden, Funktionen und Prozeduren arbeiten auf der Tabelle. Folglich entsprechen die Zeilen einer Tabelle den Instanzen eines Objekts. Außerdem erhält jede Zeile eine eindeutige Identität vergleichbar mit den OIDs (*object identifiers*). SQL:1999 definiert für diese ID den `REF`-Datentyp. Mit dem `REF`-Typ kann man mit der Pointer-Notation (->) auf Attribute eines strukturierten Typs verweisen.

3.2.2 Benutzerdefinierte Tabellenfunktionen

Neben den vordefinierten Funktionen ermöglicht SQL:1999 die Definition von benutzerdefinierten Funktionen (*user-defined functions*, UDFs), mit welchen die DB-Funktionalität erweitert werden kann. UDFs können in SQL und in verschiedenen Programmiersprachen implementiert werden. In SQL:1999 sind UDFs als skalare Funktionen definiert, die einen skalaren Wert als Ergebnis zurückliefern. Für die nächste Version SQL:200x sind zusätzlich Tabellenfunktionen geplant, die eine Tabelle als Ergebnis erzeugen [Mic01]. Das die Tabellenfunktion implementierende Programm gibt bei Aufruf immer eine Zeile der Ergebnistabelle zurück und kennzeichnet das Ende der Ergebnistabelle durch eine spezielle Meldung.

Tabellenfunktionen sind enorm mächtig, da mit ihrer Hilfe jegliche Datenquelle an das DBMS angebunden werden kann und ihre Daten wie eine Basistabelle dargestellt werden. Die Ergebnistabelle einer solchen Funktion kann in Verbunden, Gruppierungen,

Mengenoperationen und jeglichen anderen Operationen teilnehmen, die auch auf rein lesende Sichten angewendet werden können.

Die führenden DB-Hersteller wie IBM und Oracle unterstützen bereits Tabellenfunktionen. Wir greifen in unseren Beispielen auf die Syntax von IBM's DB2 zurück.

Eine Tabellenfunktion mit externem Programmiercode wird zunächst mit dem folgenden Befehl in der Datenbank registriert:

```
CREATE FUNCTION funktionname (datentyp, datentyp, ...)
RETURNS TABLE (spaltenname1 datentyp, spaltenname2 datentyp, ...)
EXTERNAL
LANGUAGE programmiersprache
PARAMETER STYLE DB2GENERAL
NO SQL
DISALLOW PARALLEL
```

Dabei wird mit `EXTERNAL` angezeigt, dass die Funktion auf einem Code in externer Programmiersprache basiert. Der Parameter `LANGUAGE` spezifiziert die Programmiersprache und `PARAMETER STYLE` die Art der Parameterübergabe. `NO SQL` und `DISALLOW PARALLEL` legen fest, dass die Funktion keine SQL-Befehle absetzen darf und sie nicht parallel über mehrere Partitionen ausgeführt werden kann.

Neben den hier aufgeführten Parametern gibt es noch weitere optionale Angaben, auf die wir nicht weiter eingehen. Wir verweisen den interessierten Leser auf die entsprechende DB2-Dokumentation [IBM01].

Nach ihrer Registrierung werden Tabellenfunktionen in der FROM-Klausel der SELECT-Anweisung mit dem Schlüsselwort `TABLE` und den Eingabewerten aufgerufen. Der folgende SQL-Befehl zeigt ein Beispiel:

```
SELECT *
FROM TABLE (funktionname(parameter1, parameter2)) AS fn
WHERE fn.spaltenname1 = 15
```

Mit dieser Abfrage werden alle die Ausgabewerte der Funktion ausgelesen, deren Wert für den Parameter oder die Spalte `spaltenname1` der Zahl 15 entspricht.

Zu beachten ist, dass Tabellenfunktionen nur in SELECT-Anweisungen und somit ausschließlich für den lesenden Zugriff eingesetzt werden können.

3.2.3 SQL/MED

Die Vielzahl an unterschiedlichen Systemen zwingt die Entwickler gegen unterschiedliche Schnittstellen zu programmieren, was sehr aufwendig und kostenintensiv ist. Allein der Zugriff auf relationale Datenbanken verschiedener Hersteller setzt das Wissen unterschiedlicher SQL-Dialekte voraus. Der zeitgleiche Zugriff auf relationale und Nicht-SQL-Daten verschärft die Situation zusätzlich. Die bekanntesten DB-Hersteller bieten bereits Lösungen für solche Fälle an, wie z. B. DataJoiner bzw. Information Integrator von IBM [IBM03a] oder Transparent Gateway von Oracle [Ora03].

Auch SQL wurde in Teil 9 SQL/MED (SQL – Part 9: Management of External Data, [FCD02]) entsprechend erweitert, um diesen Anforderungen gerecht zu werden und eine

standardisierte Schnittstelle vorzugeben. SQL/MED beinhaltet zwei Formen des Zugriffs auf externe Daten. Der erste Teil betrachtet die Möglichkeit, über die SQL-Schnittstelle auf Nicht-SQL-Daten zuzugreifen. Demnach führt eine Anwendung eine SQL-Anweisung aus, die auf Daten in mehreren Servern verweist. Diese Anweisung wird entsprechend aufgespaltet und die Fragmente an die betroffenen Quellen geschickt. Der Standard schreibt jedoch nicht vor, wie die Anweisung aufgeteilt wird, sondern spezifiziert die Interaktion zwischen dem SQL-Server und dem entsprechenden Wrapper.

Der zweite Teil betrachtet die Daten, die außerhalb des DBS in Dateien abgelegt werden und wie diese verwaltet werden können. CAD-Zeichnungen oder Bilder sind typische Beispiele solcher Daten, die direkt im Dateisystem gespeichert werden. Für die meisten Anwendungen müssen diese Daten direkt vom Dateisystem verwaltet werden. Häufig liegen aber zugehörige Daten in der Datenbank und es gestaltet sich schwierig, Daten und Datei synchron zu halten. SQL/MED führt hierfür den neuen Datentyp `DATALINK` ein, mit dem eine Referenz zu einer Datei außerhalb des DBS in einer Spalte abgelegt werden kann. Datalinks erweitern die DBS-Funktionalität damit um die Kontrolle von externen Dateien, ohne ihren Inhalt in der Datenbank speichern zu müssen. Folglich werden die Dateien bei Sicherstellung der referentiellen Integrität beachtet, sie sind Teil der Recovery-Maßnahmen und das DBS verwaltet auch die Zugriffsrechte auf die Dateien. Wir werden nicht weiter auf Datalinks eingehen. Weitere Details hierzu können in [FCD02], [MMJ+01] und [MMJ+02] nachgelesen werden.

3.2.3.1 Die Wrapper-Schnittstelle

Soll auf externe Daten über eine SQL-Schnittstelle zugegriffen werden, müssen diese Daten als relationale Tabellen dargestellt werden. SQL/MED führt den Begriff der fremden Tabellen (*foreign tables*) für Daten ein, die außerhalb eines SQL-Servers gespeichert sind. Diese fremden Tabellen werden von so genannten fremden Servern (*foreign servers*) verwaltet. Ein fremder Server könnte beispielsweise eine Web-Seite mit mehreren Preislisten sein, die als fremde Tabellen repräsentiert werden. Zerlegt ein SQL-Server die Anfrage in mehrere Fragmente, wird jedes Fragment zu dem entsprechenden fremden Server weitergeleitet, der für die referenzierte fremde Tabelle zuständig ist.

Da mehrere Datenquellen dieselben Schnittstellen haben können, ist der Einsatz eines einzigen Moduls sinnvoll, das den Zugriff zu diesen Quellen steuert. Jede Quelle wird als fremder Server dargestellt. Dieses gemeinsame Modul wird in SQL/MED als Wrapper (*foreign-data wrapper*) bezeichnet. Wrapper können unterschiedliche Konfigurationen aufweisen, die über generische Optionen definiert werden. Diese sind aber nicht von vornherein vorgegeben, sondern werden über Attribut-Wert-Paare bestimmt. Beispielsweise kann für einen Wrapper, der den Zugriff auf Dateien unterstützt, das Trennungszeichen zwischen den einzelnen Datensätzen in der Datei angegeben werden.

SQL/MED spezifiziert die Schnittstelle bestehend aus einer Menge an Funktionen zwischen SQL-Server und Wrapper. Die Funktionen auf Seiten des SQL-Servers werden als die SQL-Server-Routinen und die auf Seiten des Wrappers als Wrapper-Routinen der Wrapper-Schnittstelle bezeichnet.

Abbildung 3.2 zeigt die in SQL/MED spezifizierten Komponenten. Der SQL-Server und der Wrapper kommunizieren über die SQL/MED-Schnittstelle, müssen aber nicht

auf einem Rechner laufen. Die Kommunikation zwischen Wrapper und fremdem Server ist hingegen nicht durch SQL/MED festgelegt, sondern kann proprietär implementiert werden.

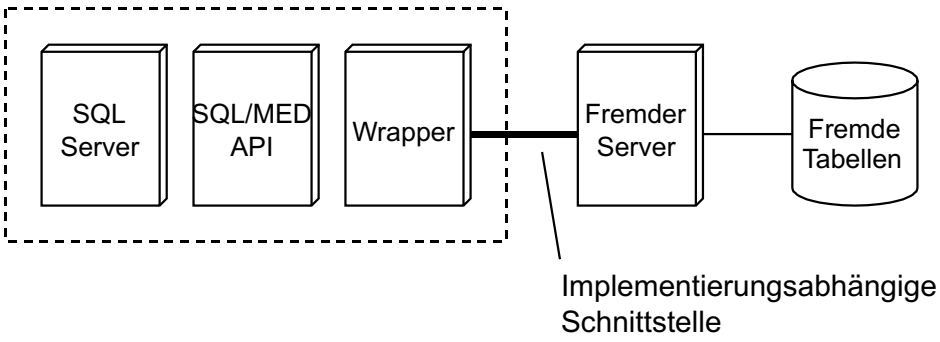

Abbildung 3.2: Die Komponenten von SQL/MED.

Die Definition von fremden Tabellen soll den transparenten Zugriff auf Daten ermöglichen, die nicht in einem lokalen SQL-Server gespeichert sind. Transparent bedeutet in diesem Fall, dass der Benutzer beim Zugriff auf diese Daten nicht merkt, dass sie nicht im lokalen SQL-Server abgelegt sind. Für den Benutzer sind alle Daten wie in Basistabellen verfügbar und können über SELECT-Anweisungen abgerufen werden.

Bevor auf die fremden Tabellen zugegriffen werden kann, müssen dem SQL-Server mehrere Informationen gegeben werden. Zunächst muss der zuständige Wrapper registriert werden. Dies ist über die folgende DDL-Anweisung möglich:

```
CREATE FOREIGN DATA WRAPPER wrap-name
[ AUTHORIZATION auth-id ]
[ LIBRARY library-name ]
LANGUAGE language-name
[ generic-options ]
```

Diese Informationen werden im Katalog des SQL-Servers abgelegt. Mit der `AUTHORIZATION`-Klausel wird der nominale Besitzer des Wrappers festgelegt. Die `LIBRARY`-Option spezifiziert die Software-Bibliothek, welche die Wrapper-Routinen implementiert und mit der `LANGUAGE`-Klausel gibt man die Programmiersprache an, in welcher die Bibliothek programmiert ist. Da die benötigten Informationen von Wrapper zu Wrapper variieren können, spezifiziert der Standard keine festen Konfigurationsattribute. Stattdessen können mit dem Konzept der generischen Optionen (`generic-options`) beliebige Informationen als Attribut-Wert-Paare übergeben werden.

Anschließend wird der fremde Server über die folgende Anweisung bekannt gegeben:

```
CREATE SERVER server-name
[ TYPE server-type ]
[ VERSION server-version ]
[ AUTHORIZATION auth-id ]
[ LIBRARY library-name ]
FOREIGN DATA WRAPPER wrapper-name
[ generic-options ]
```

Die `TYPE`- und `VERSION`-Optionen sind nur für bestimmte Implementierungen notwendig. Folglich sind keine gültigen Werte von SQL/MED vorgegeben. Beispielsweise könnte für einen bestimmten DB2-Server der Typ als „UDB“ und die Version als „7.2“ angegeben werden. Die `AUTHORIZATION`-Klausel definiert den Besitzer des fremden Servers. Auch diese Informationen sind als Metadaten im Katalog des SQL-Servers abgelegt.

Um schließlich auf die fremden Tabellen zugreifen zu können, müssen diese noch registriert werden mit dem folgenden Befehl:

```
CREATE FOREIGN TABLE table-name
[ ( col-def, col-def, ... ) ]
SERVER foreign-server-name
[ generic-options ]
```

Dieser Befehl erzeugt ein Objekt im Schema des SQL-Servers, das die fremde Tabelle repräsentiert. Optional können die Spalten der fremden Tabelle explizit angegeben werden. Außerdem wird mit `SERVER` derjenige Server angegeben, der die fremde Tabelle verwaltet. Nun kann über die fremde Tabelle auf Daten in externen Datenquellen zugegriffen werden.

3.2.3.2 Anfrageverarbeitung mit Wrappern

Es gibt zwei Varianten der Kommunikation zwischen SQL-Server und Wrapper: Zerlegungsmodus und Pass-Through-Modus. Beim Pass-Through-Modus wird die Anfrage unverändert als Zeichenkette an den Wrapper weitergeleitet und wird dort zusammen mit dem fremden Server analysiert und ausgeführt. Folglich muss auf Seiten des Wrapper und des fremden Servers ein Parsing der Anfrage durchgeführt werden. Dieser Modus ist vor allem dann sinnvoll, wenn der fremde Server ebenfalls eine relationale Datenbank ist.

Im Zerlegungsmodus wird die Anfrage im SQL-Server in Fragmente aufgespaltet, welche in den fremden Servern ausgeführt werden. Dieser Modus läuft in zwei Schritten ab.

1. In der *Anfrageplanungsphase* wird vom SQL-Server gemeinsam mit dem Wrapper ein Ausführungsplan erstellt.

2. In der *Anfrageausführungsphase* wird dieser Plan ausgeführt und die fremden Daten werden an den SQL-Server geliefert.

Damit die Implementierung in unterschiedlichen Programmiersprachen einfacher ist, nutzt SQL/MED eine funktionale Schnittstelle auf Basis von *Handles*. Anstatt Datenstrukturen für den Informationsaustausch zwischen SQL-Server und Wrapper vorzugeben, muss der SQL-Server laut SQL/MED ein Integer-Handle übergeben, der diese Datenstruktur repräsentiert. Wir gehen an dieser Stelle nicht weiter auf Handles ein, möchten aber darauf hinweisen, dass der Austausch von Datenstrukturen zwischen SQL-Server und Wrapper immer mittels Handles durchgeführt wird.

Anfrageplanungsphase

Die Interaktion zwischen SQL-Server und Wrapper geschieht nach dem Request/Reply-Paradigma. Der SQL-Server schickt einen Request mit dem Anfragefragment an den Wrapper, der es analysiert und mit einem Reply eine Beschreibung jener Teile der Anfrage zurückschickt, die vom fremden Server verarbeitet werden können. Jegliche Teile des Fragments, die nicht ausgeführt werden können, müssen seitens des SQL-Servers kompensiert werden.

Bevor die Ausführung beginnen kann, muss eine Verbindung zwischen dem SQL-Server und dem entsprechenden fremden Server aufgebaut werden. Dies geschieht über die Wrapper-Routine `ConnectServer()`. Mitgegebene Parameter enthalten den Namen des zu verbindenden fremden Servers sowie die Werte für die für diesen Server definierten generischen Optionen. Ebenso wird der Benutzername übermittelt, über den die Verbindung aufgebaut werden soll.

Als Nächstes führt der SQL-Server die Wrapper-Routine `InitRequest()` aus, um die Anfragefragmente in Form eines Requests an den Wrapper zu schicken. Ein Request ist eine Datenstruktur, die das Anfragefragment in abstrakter Weise beschreibt. Folglich wird die SQL-Anweisung nicht als Zeichenkette an den Wrapper geschickt. Dies hat zum Vorteil, dass der Wrapper kein Parsing von SQL unterstützen muss. Der Request beschreibt vielmehr die einzelnen Klauseln einer SQL-Anweisung, also die SELECT-, FROM- und WHERE-Klausel.

Die einzelnen Elemente der SELECT-Klausel werden als Wertausdrücke dargestellt, die Elemente der FROM-Klausel als Tabellenreferenzen und die Elemente der WHERE-Klausel als Boolean-Ausdrücke. Da der Standard mittlerweile komplexere Requests unterstützt, kann es durchaus sein, dass der Wrapper die Anfrage nicht vollständig ausführen kann. In solchen Fällen liefert der Wrapper die Basisdaten und der SQL-Server kompensiert die fehlende Funktionalität, indem beispielsweise eine Sortierung oder Filterung der Daten im SQL-Server durchgeführt wird. Erhält der Wrapper eine `InitRequest()`-Routine, dann wird die Anfrage mit Hilfe von SQL-Server-Routinen untersucht. Mit diesen Routinen kann der Wrapper beispielsweise die Tabellenreferenzen oder die Werte für die generischen Optionen abfragen. Wurde der Request analysiert, stellt der Wrapper einen SQL/MED-Reply zusammen, der zusammen mit dem Ausführungsplan an den SQL-Server zurückgeschickt wird. Dieser kann wiederum über Wrapper-Routinen den Reply untersuchen. Der Ausführungsplan wird jedoch nicht vom SQL-Server interpretiert, sondern dient lediglich zur Kapselung der Informationen, die vom Wrapper zur Ausführung des Anfragefragments benötigt werden. Wird die Ausführungsphase gestartet, übergibt der SQL-Server den Ausführungsplan wieder an den Wrapper.

Anfrageausführungsphase

Während der Ausführungsphase wird der Teil des Anfragefragments von Wrapper und Datenquelle ausgeführt, der im Reply angegeben wurde. Zum Start der Ausführung ruft der SQL-Server die Wrapper-Routine `Open()` mit dem Ausführungsplan auf. Das Ergebnis wird zeilenweise mit der Routine `Iterate()` abgerufen und mit `Close()` beendet. Nun kann der Wrapper nach der Ausführung aufräumen. Der Ausführungsplan kann bei

Bedarf erneut benutzt werden. Wenn er nicht mehr gebraucht wird, wird er durch die Routine `FreeExecutionHandle()` deallokiert, die der SQL-Server aufruft.

3.2.3.3 Die Zukunft von SQL/MED

Nachdem die erste Version von SQL/MED noch sehr eingeschränkt ist und nur die Ausführung einer Anfrage mit Verweis auf eine einzige Tabelle unterstützt, sind für die neue Version einige Erweiterungen angedacht. Im Wesentlichen können die Erweiterungen in drei Punkte gefasst werden.

1. Zukünftig können komplexe Anfragen an den Wrapper verschickt werden. Es werden die WHERE-Klausel und mehrere Tabellenreferenzen in der FROM-Klausel unterstützt. Außerdem ist es möglich, komplexe Wertausdrücke in der SELECT- und in der WHERE-Klausel zu definieren (z. B. `SELECT name || ' ' || phone FROM emp WHERE name = 'John Doe';`). Man kann benutzerdefinierte Funktionen dann auch in Wertausdrücken aufrufen (z. B. `SELECT info FROM emp WHERE func1(name, city-id) = 100;`).

2. Der SQL-Server kann den Anfragekontext an den Wrapper schicken. Dies ist vor allem dann hilfreich, wenn Teile einer Anfrage in mehreren Requests an den Wrappern geschickt werden, die möglicherweise dieselben Wertausdrücke und Prädikate dieser Anfrage referenzieren. Da die Analysen dieser Ausdrücke häufig unabhängig voneinander sind, können sie wiederverwendet werden.

3. Der SQL-Server kann beim Wrapper die Kosten der Anfrageausführung erfragen. Für die Anfrageplanung werden zumindest die Größe der Ergebnismenge und die Ausführungszeit benötigt. Aufgrund dieser Informationen kann der SQL-Server beispielsweise entscheiden, bestimmte Operationen selbst durchzuführen, auch wenn der fremde Server dazu in der Lage wäre. Wenn der SQL-Server dies viel schneller kann, kompensiert dieser Zeitgewinn die Übertragung einer größeren Ergebnismenge von Wrapper zu SQL-Server.

Für noch weiter in der Zukunft liegende Versionen ist neben dem lesenden auch der schreibende Zugriff von Interesse. Ebenso können zusätzliche Anfragefähigkeiten wie Gruppierung und Aggregatfunktionen Kandidaten zur Aufnahme in den Standard sein.

3.3 Workflow-System

Neben der Integration von Daten hat in den vergangenen Jahren auch die Vereinheitlichung und Integration von Arbeitsabläufen oder Prozessen an Bedeutung gewonnen. Hieraus haben sich die klassischen Workflow-Systeme entwickelt, die heutzutage als „people-driven“ Workflow-Systeme bezeichnet werden, da sie in erster Linie die Benutzer bei bestimmten Arbeitsschritten unterstützen sollten. Im Laufe der Zeit sind weitere Varianten und Einsatzgebiete der Workflow-Systeme aufgekommen, wie z. B. die Integration von Anwendungssystemen, indem ein automatisierter Datenaustausch zwischen diesen Systemen ohne jede Interaktion des Benutzers möglich ist. Solche Workflow-Systeme werden heute als Produktions-Workflows (*production workflows*) bezeichnet.

Workflow-Systeme sollen die Geschäftsprozesse unterstützen und gewinnen immer mehr an Bedeutung in den Unternehmen. Da sie auch zunehmend zur Integration von Systemen eingesetzt werden, beschreiben wir im folgenden Abschnitt die wichtigsten Aspekte von Workflows. Dazu gehören deren Definition, die verschiedenen Kategorien an Workflows, ihre Hauptbestandteile und ein Blick auf Standards in diesem Bereich.

3.3.1 Was sind Workflows?

In einer Firma finden sich eine Vielzahl an so genannten Geschäftsprozessen, die immer wieder in derselben Form ausgeführt werden. Typische Beispiele sind Zahlungen auf Konten oder Buchungen von Reisen. Das immer gleiche zugrunde liegende Muster nennt man Prozessmodell. Das Prozessmodell beschreibt den Aufbau des Geschäftsprozesses in der realen Welt. Es definiert alle möglichen Wege durch den Geschäftsprozess, einschließlich der Regeln, die den Weg bestimmen. Auch die auszuführenden Aktionen werden durch das Prozessmodell festgelegt.

Prozesse müssen nicht zwingend auf Rechnern ausgeführt werden. Das Spektrum an Prozessen ist groß. Sie können komplett ohne EDV-Unterstützung ablaufen und ebenso ganz ohne Benutzerinteraktion vollständig durch Computer ausgeführt werden. Dazwischen gibt es die Kombinationen, die Teile mit Computer- und Teile ohne Computer-Unterstützung enthalten. Die vom Rechner ausgeführten Teile werden Workflow-Modell genannt. Die Instanz eines Workflow-Modells ist der Workflow.

Workflows haben drei unabhängige Dimensionen, die man grafisch als Würfel darstellen kann. Die erste Dimension beschreibt die Prozesslogik. Folglich legt sie die Aktivitäten eines Workflows und ihre Ausführungsreihenfolge fest. Die zweite Dimension ist die Organisation. Sie beschreibt den organisatorischen Aufbau des Unternehmens in Abteilungen, Rollen und Personen. Mit diesen Informationen wird beschrieben, wer welche Aktivität ausführen soll. Wird eine Aktivität nicht durch einen Benutzer angestoßen, dann führt das Workflow-Managementsystem diese Aktivität automatisch aus. Die dritte Dimension bringt die IT-Infrastruktur hinzu. Man beschreibt die benötigte IT-Ressource. Dies kann beispielsweise ein Programm sein, das die Aktivität ausführt.

3.3.2 Kategorien von Workflows

Workflows können in vier Kategorien eingeteilt werden (siehe Abbildung 3.3 [GIG03]). Auf der y-Achse ist der Geschäftswert abgebildet, der die Wichtigkeit des Workflows für ein Unternehmen kennzeichnet. Ist der Geschäftswert hoch, handelt es sich um eine Kernkompetenz des Unternehmens. Die Wiederholung auf der x-Achse zeigt an, wie häufig ein bestimmter Prozess in derselben Form durchgeführt wird. Je häufiger ein Prozess durchgeführt wird, desto mehr lohnt dessen Automatisierung.

Auf Basis dieser beiden Eigenschaften kann man vier verschiedene Workflow-Typen unterscheiden.

- *Kollaborative Workflows* haben einen sehr hohen Geschäftswert, werden aber nur selten ausgeführt. Der zugrunde liegende Prozess ist sehr komplex und wird häufig verändert.

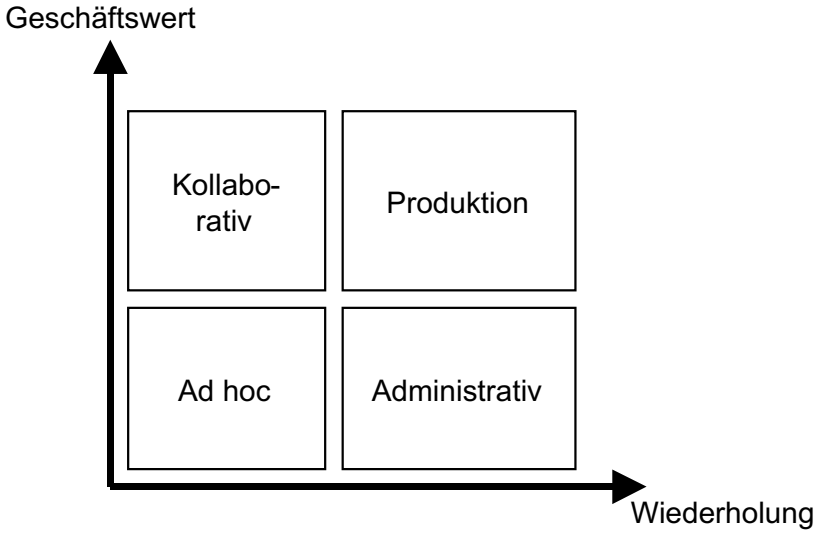

Abbildung 3.3: Klassifikation von Workflows hinsichtlich Geschäftwert und Wiederholung.

- *Ad-hoc-Workflows* haben einen geringen Geschäftswert und haben eine geringe Wiederholungsrate. Der Prozess hat keine definierte Struktur. Stattdessen legt der jeweilige Benutzer nach Bedarf den nächsten Schritt fest und führt ihn durch.
- *Administrative Workflows* haben ebenfalls einen geringen Geschäftswert, werden aber häufig durchgeführt. Diese Workflows sind typischerweise administrative Prozesse wie die Bearbeitung eines Auftrags.
- *Produktions-Workflows* haben einen hohen Geschäftswert und eine hohe Wiederholungsrate. Diese Workflows implementieren das Kerngeschäft eines Unternehmens und machen dessen Wettbewerbsvorteil aus.

Neben dem Geschäftswert und der Wiederholungsrate spielt der Grad der Automatisierung eines Workflows eine Rolle. Er ist ein Maß für die Benutzerinteraktion in einem Workflow, d. h., ob die Aktivitäten hauptsächlich durch Menschen oder durch das System ausgeführt werden. Ein hoch automatisierter Prozess ist rechenintensiv und integriert typischerweise heterogene und autonome Anwendungssysteme.

Da Prozessmodelle Unternehmensressourcen sind, müssen sie genauso sorgfältig wie Unternehmensdaten behandelt werden. Es muss gewährleistet werden, dass die Prozesse ohne Fehler genau so ablaufen, wie sie definiert wurden. In [LR00] werden hierzu Anforderungen an Workflow-Systeme und insbesondere an Produktions-Workflows aus Betriebs- und Unternehmenssicht beschrieben.

Ein Managementsystem für Produktions-Workflows muss folgende Anforderungen aus Betriebssicht erfüllen:

- Es muss globale Transaktionen unterstützen.

- Es muss zuverlässig sein, d. h., alle internen Operationen müssen als Transaktionen ausgeführt werden.
- Das System muss hohe Verfügbarkeit aufweisen, d. h. insbesondere, dass das Workflow-Managementsystem einen 24×7-Betrieb unterstützt, was impliziert, dass das System rund um die Uhr laufen muss.
- Es muss eine hohe Kapazität garantieren, d. h., es unterstützt eine große Zahl an Benutzern und Prozessen.
- Das System muss skalierbar sein, d. h., das Workflow-Managementsystem muss so ausgelegt sein, dass zusätzliche Ressourcen das System schneller machen oder höheren Durchsatz ermöglichen.
- Schließlich muss das Workflow-Managementsystem den Prozess verfolgen können. Auf diese Weise kann man das System beobachten und schneller auf aufkommende Fehler und Probleme reagieren.

Da Workflows meistens in einer heterogenen und verteilten Umgebung ablaufen, kommen weitere Anforderungen aus Unternehmenssicht hinzu:

- Das Workflow-Managementsystem muss mehrere Plattformen unterstützen, insbesondere verschiedene Betriebssysteme und Netzwerkprotokolle. Es muss nicht nur selbst auf unterschiedlichen Plattformen laufen, sondern auch Programme auf unterschiedlichen Plattformen aufrufen können. Nur so können unterschiedlichste Anwendungen integriert werden.
- Es muss sich in Programme für Systemmanagement einbinden lassen, damit beispielsweise die automatische Verteilung der Software unterstützt wird.
- Eine zentrale Administration ist ebenfalls wichtig, da sonst die verteilte Workflow-Umgebung nicht effizient verwaltet werden kann.
- Ein ausgeklügeltes Sicherheitssystem ist ein Muss für ein Workflow-Managementsystem. Um den Schutz des Workflow-Managementsystems zu gewährleisten, müssen alle Aspekte der Authentifizierung und Autorisierung als auch der Verschlüsselung der Workflow-Nachrichten unterstützt werden.

3.3.3 Komponenten eines Workflow-Managementsystems

Ein Workflow-Managementsystem besteht aus mehreren Komponenten (siehe Abbildung 3.4). Die wichtigsten Komponenten sind das Metamodell, die Buildtime- und die Runtime-Komponente sowie die Datenbank.

- Das Metamodel besteht aus Konstrukten, mit denen die Benutzer ihre Prozessmodelle, ihre organisatorischen Strukturen und die Topologie ihres Workflow-Managementsystems modellieren können. Zum Prozessmodell gehören die Eingabe- und Ausgabe-Container, welche die Daten und somit den Kontext für die Aktivitäten enthalten. Weitere wichtige Konstrukte sind die Aktivitäten und die Kontrollkonnektoren. Mit ihrer Hilfe legt man den Ablauf der Aktivitäten fest.

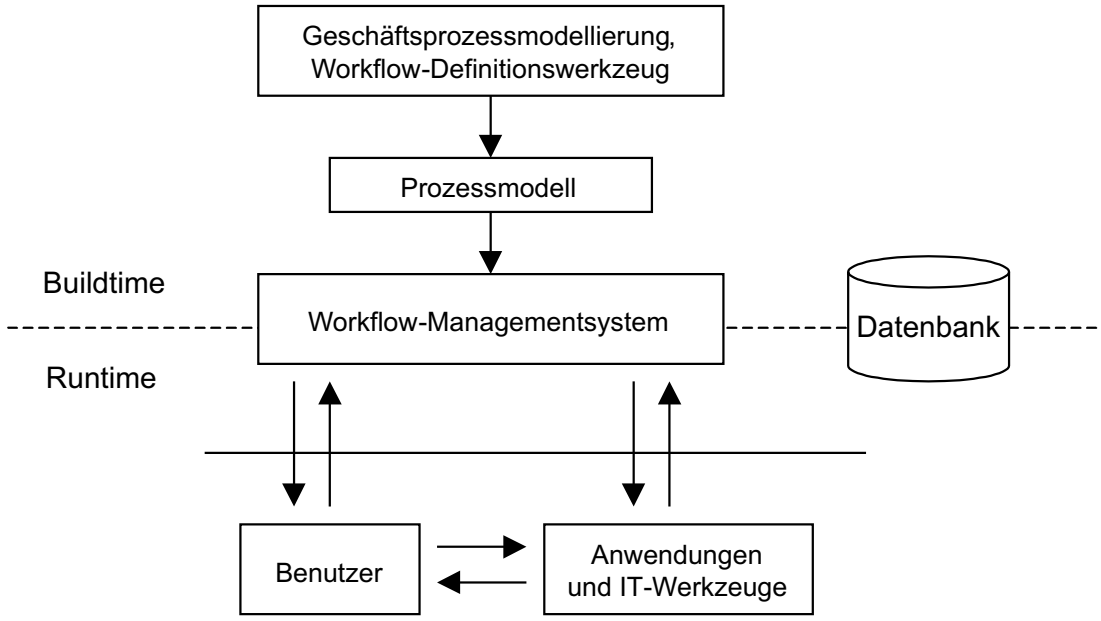

Abbildung 3.4: Die Hauptkomponenten eines Workflow-Systems [LR00].

- Mit der Buildtime-Komponente definiert man die Workflows. Dazu gehören die Prozessmodelle, die organisatorischen Strukturen und die Programme, welche die Aktivitäten implementieren. Diese Informationen kann man in unterschiedlicher Form festlegen. Zum einen kann der Entwickler mit der grafischen Benutzerschnittstelle den Workflow mit Icons und Pfeilen modellieren. Die Icons repräsentieren die Aktivitäten und die Pfeile symbolisieren den Daten- oder Kontrollfluss. Zum anderen kann der Workflow auf verschiedene Weise beschrieben werden, beispielsweise mit einer proprietären oder standardisierten Skriptsprache. Die Ablaufanweisung wird auch als Ergebnis der Buildtime-Komponente generiert.

- Die Runtime-Komponente führt die definierten Workflows aus. Dazu legt sie Prozesse an, navigiert durch diese und interagiert mit den Anwendungen und Benutzern. Wie die Runtime-Komponente mit den Aktivitäten umgeht, legen die Einstellungen für die Aktivität fest. Sind sie auf automatisch gesetzt, dann startet die Runtime-Komponente die Aktivität ohne Benutzerinteraktion. Ist sie auf manuell gesetzt, startet der Benutzer die Aktivität.

- Die Datenbank enthält all jene Informationen, die von der Buildtime- und Runtime-Komponente benötigt werden. Neben Informationen zu den definierten Workflows sind dies Informationen zu den aktuellen Prozessinstanzen.

3.3.4 Workflow-Standards

Die Workflow Management Coalition (WfMC) wurde 1993 von einigen Herstellern und Nutzern von Workflow-Managementsystemen gegründet. Die Organisation legt Standards für Workflow-Managementsysteme fest, welche die Interoperabilität zwischen he-

terogenen Workflow-Managementsystemen ermöglichen sollen. Halten sich die Hersteller an diese Standards, ist auch der Wechsel von einem Hersteller zum anderen leichter.

Die Workflow Management Coalition hat zwei Hauptaspekte von Workflow-Managementsystemen standardisiert. Man hat das zugrunde liegende Metamodell und damit die Konstrukte zur Modellierung eines Prozesses festgelegt. Außerdem hat man eine Menge von Schnittstellen standardisiert, welche die von den Systemen unterstützten Funktionen vorgibt.

Wir werden die Standards hier nicht ausführlich behandeln, sondern beschreiben die spezifizierte Architektur und die Schnittstellen. Den interessierten Leser verweisen wir auf die Homepage der Workflow Management Coalition [WfM03].

Die WfMC definiert ein Architekturmodell für ein Workflow-Managementsystem, das so genannte Workflow-Referenzmodell (*workflow reference model*). Eine bestimmte Instanz bzw. Installation eines Workflow-Managementsystems wird *Workflow Enactment Service* genannt. Dieser Service führt den Workflow aus. Teil dieses Service sind ein oder mehrere Workflow-Engines.

Das Referenzmodell definiert jedoch weder die Systemstruktur noch die zugrunde liegende Architektur eines standardkonformen Workflow Enactment Service oder Workflow-Managementsystems. Die Hersteller sind in ihren Implementierungen völlig frei, so lange die definierten Schnittstellen dem Standard entsprechen.

Fünf Schnittstellengruppen mit den Namen Schnittstelle 1 bis Schnittstelle 5 sind definiert. Wir beschreiben im Folgenden kurz die einzelnen Schnittstellen.

- *Schnittstelle 1* legt fest, wie unterschiedliche Systeme das Prozessmodell und organisatorische Informationen austauschen können. Die Schnittstelle ist als Tag-Sprache definiert und heißt *Workflow Process Definition Language (WPDL)*. Inzwischen wurde die WPDL auf Basis von XML Schema [TBMM01, BM01] aufgesetzt und wird nun XPDL (XML Process Definition Language) genannt.

- *Schnittstelle 2* definiert die dem Benutzer bereitgestellten Funktionen, mit welchen er mit dem Workflow-Managementsystem interagieren kann.

- *Schnittstelle 3* standardisiert den Aufruf von Aktivitäten. Es ermöglicht die Implementierung von Programmen, die als Aktivitätenimplementierungen in jedes Workflow-Managementsystem eingeklinkt und somit wieder verwendet werden können.

- *Schnittstelle 4* legt fest, wie Subprozesse zwischen verschiedenen Workflow-Managementsystemen gehandhabt werden. Dazu gehören Funktionen, um Subprozesse zu starten, um ihren Status abzufragen und um Änderungen an den Prozessdaten anzumelden.

- *Schnittstelle 5* definiert den Aufbau des Audit-Trails und verschiedene Einträge, die das Workflow-Managementsystem bereitstellen muss, um standardkonform zu sein.

Die Standards der WfMC haben sich leider bis heute nicht bei den Herstellern durchgesetzt. Es werden viel eher de-facto-Standards durch die gängigsten Workflow-Managementsysteme wie beispielsweise WebSphere MQ Workflow von IBM [IBM03b] geschaffen.

3.4 XML und verwandte Standards

Die *Extensible Markup Language* (kurz XML, [BPSM98]) beschreibt eine Klasse von Datenobjekten, den so genannten XML-Dokumenten. Darüber hinaus legt sie teilweise das Verhalten von Programmen fest, die XML-Dokumente verarbeiten. Da XML eine echte Untermenge von SGML (Standard Generalized Markup Language [Gol90]) darstellt, sind XML-Dokumente gleichzeitig SGML-konforme Dokumente.

XML-Dokumente setzen sich aus Speichereinheiten, den Entitäten, zusammen, die entweder Text oder Binärdaten enthalten. Während Textdaten analysiert werden, ist der Inhalt der Binärdaten nicht bekannt. Text besteht aus Zeichenketten und Markups, wobei die Markups die Speicheraufteilung und logische Struktur des Dokuments beschreiben. XML bietet einen Mechanismus an, um Einschränkungen bzgl. Aufteilung und logischer Struktur zu formulieren.

Ein Software-Modul, der XML-Prozessor, liest XML-Dokumente und erlaubt den Zugriff auf deren Inhalt und Struktur. Dieser XML-Prozessor ist Teil einer Anwendung, die mit XML-Dokumenten arbeitet. Die XML-Spezifikation definiert das notwendige Verhalten des XML-Prozessors, wie er XML-Daten einliest und welche Informationen er an die Anwendung weiterreicht.

XML wurde 1996 von einer Arbeitsgruppe unter der Schirmherrschaft des World Wide Web Consortium (W3C, [W3C03]) entwickelt. Die Entwurfsziele der Arbeitsgruppe sahen vor, dass XML auf einfache Weise im Internet genutzt werden kann. Trotz der Kompatibilität zu SGML sollte es eine sehr einfache und schlanke Sprache sein, mit der XML-Dokumente schnell erstellt werden können und zugehörige Anwendungen einfach zu entwickeln sind. Auf diese Weise sollte ein breites Spektrum an Anwendungen unterstützt werden.

In den nächsten Abschnitten beschreiben wir die Grundzüge von XML und für diese Arbeit relevante, verwandte XML-Standards.

3.4.1 Grundbegriffe

XML beschreibt eine Klasse von Datenobjekten, den so genannten XML-Dokumenten. XML-Dokumente implizieren aber nicht Dokumente im herkömmlichen Sinne. Das W3C definiert den Aufbau eines XML-Dokuments aus den folgenden Komponenten:

- Prolog (optional)
- DTD (optional)
- Wurzelelement und zugehöriger Baum
- Kommentare und Verarbeitungsanweisungen

Prolog

Der *Prolog* beschreibt die verwendete XML-Version (`version`) und den zugrunde liegenden Zeichensatz (`encoding`). Außerdem kann der Anwender mit dem Attribut `standalone` angeben, ob externe Deklarationen existieren, welche die Information des Dokumentes beeinflussen (z. B. Attribute mit default-Werten). Während die Version zwingend anzugeben ist, sind die anderen beiden Angaben optional. Der Prolog eines XML-Dokumentes kann wie folgt aussehen:

```
<? xml version="1" encoding="UTF-8" standalone="yes" ?>
```

DTD

Die *DTD* (*Document Type Definition*) beschreibt schematisch das XML-Dokument, indem seine Struktur mittels Elementen und ihren Attributen sowie Entitäten definiert wird. Die DTD kann außerhalb oder innerhalb eines XML-Dokumentes beschrieben werden. Hat man beides gewählt, dann überschreibt die interne Definition die externe.

Existiert eine DTD für ein Dokument und entspricht es der in der DTD definierten Struktur, dann ist es gültig (*valid*). Existiert keine DTD, ist das Dokument aber gemäß der XML Recommendation syntaktisch korrekt, dann ist es wohlgeformt (*well-formed*).

Ein XML-Dokument kann aus einer oder mehreren Speichereinheiten bestehen, die Entitäten (*entities*) heißen. Sie haben alle Inhalt und sind alle (abgesehen von der Dokumenten-Entität) durch einen Namen identifiziert. Jedes XML-Dokument besitzt eine Entität namens Document Entity, welche als Ausgangspunkt für den XML-Prozessor dient und das gesamte Dokument enthalten darf.

Es gibt unterschiedliche Entitäten. *Parameter-Entitäten* sind Konstrukte zum Textersatz innerhalb von DTDs. Um eine solche Entität in der DTD zu referenzieren, schreibt man das Zeichen '%' vor den Entitäts-Namen. Definiert man beispielsweise die Entität `children` mit `<!ENTITY % children "(A, B, C)">`, referenziert man sie mit `<!ELEMENT parent %children;>`. *Vordefinierte Entitäten* werden für jene Zeichen genutzt, die speziell für die Markierung des Dokuments gedacht sind. Dazu gehören beispielsweise die eckigen Klammern (`<, >`) und der Schrägstrich (`/`). Benötigt man diese Zeichen als Daten und nicht als Markierung, fügt man sie als Entities in den Text ein. Wenn man beispielsweise den Text „<PFLANZE>“ in einem Dokument einfügen will, muss man `&lt;PFLANZE&gt;` schreiben.

Neben den Entitäten spielen die Elemente eine wichtige Rolle in XML-Dokumenten. Auf Basis von Elementen definiert man die Baumstruktur des XML-Dokumentes. Jedes Element hat einen Start-Tag und einen End-Tag mit seinem Namen (z. B. `<Beispiel> Inhalt </Beispiel>`). Außerdem können Elemente optional Attribute enthalten: `<Beispiel Nr="1"> Inhalt </Beispiel>`. Ein leeres Element hat keinen Inhalt und kann verkürzt dargestellt werden mit `<LeeresBeispiel/>`. Der Inhalt eines Elementes besteht entweder aus weiteren Elementen oder er ist vom Typ PCDATA und kann somit eine beliebige Zeichenfolge beinhalten. Ebenso kann *mixed content* definiert werden, der aus PCDATA und Elementen besteht.

Bei der Definition der Elemente und dadurch der Baumstruktur des XML-Dokumentes stehen mehrere Möglichkeiten zur Auswahl. Das definierte Element enthält nur Elemente, die als Sequenz oder Auswahl festgelegt werden. Ebenso kann der Inhalt des

Elements gemischten Inhalt haben, also weitere Elemente und Zeichenketten (mixed content). Das Element kann aber auch ein Blatt des Baums sein und nur PCDATA beinhalten oder sogar leer sein. Eine sehr flexible Variante ist die Definition von `ANY`, d. h., diesem Element kann beliebiger Inhalt zugewiesen werden.

Wie häufig ein Element oder eine Elementgruppe vorkommt, bestimmt man mit Ober- und Untergrenzen. Das „`+`“ beschreibt eine Untergrenze. Das Element kommt mindestens einmal vor. „`*`“ besagt, dass es weder Unter- noch Obergrenzen gibt. Das Element kann somit beliebig häufig oder auch gar nicht vorkommen. „`?`“ ist eine Obergrenze und beschränkt das Vorkommen eines Elements auf höchstens einmal. Wird nichts angegeben, dann liegen Unter- und Obergrenze vor und das Element muss genau einmal vorkommen.

Pro Element kann es mehrere Attribute geben, deren Werte vorbelegt sein können (z. B. `<!ATTLIST Satz Sprache CDATA "Deutsch">`). Attribute können mit dem Schlüsselwort `#IMPLIED` als optional deklariert werden. Ebenso sind feste Attributwerte möglich, wie z. B. `<!ATTLIST Satz Sprache CDATA #FIXED "Deutsch">`.

Es gibt mehrere Attributtypen. Darf das Attribut nur Zeichenketten enthalten, dann ist es vom Typ CDATA. Ist das Attribut vom Typ Entity, referenziert es den Wert einer deklarierten Entität. Außerdem können Attributwerte auch als Identifier eingesetzt werden und sind vom Typ ID. Hier gilt insbesondere, dass die Werte der ID-Attribute in einem Dokument eindeutig sein müssen. Ergänzend zum ID-Typ gibt es den IDREF-Typ. Der Wert eines IDREF-Attributes muss ein Verweis auf eine anderswo im Dokument deklarierte ID sein. Ist ein Attribut vom Typ NMTOKEN, so ist der Attributwert auf ein einziges Wort beschränkt, das eine beliebige Zusammenstellung aus Buchstaben, Ziffern, Punkten, Doppelpunkten, Bindestrichen und Unterstrichen ist. Der Attributwert kann auch einem Wert einer vorgegebenen Aufzählung entsprechen wie in dem folgenden Beispiel: `<!ATTLIST Auto Farbe (rot|gelb|blau) "blau">`.

Zu Beginn dieses Kapitels haben wir erklärt, dass ein XML-Dokuments aus einem optionalen Prolog, einer optionalen DTD, dem Wurzelelement und zugehörigem Baum sowie Kommentaren und Verarbeitungsanweisungen (*processing instructions*) besteht. Fehlt also noch die Beschreibung der Kommentare und Verarbeitungsanweiungen.

Kommentare und Verarbeitungsanweisungen

Kommentare erscheinen zwischen Kommentar-Tags (`<!-- -->`) und können überall im Dokument vorkommen, müssen aber außerhalb eines Markups stehen. Sie werden nicht in der DTD deklariert und dürfen nicht geschachtelt werden. Ein Beispiel eines Kommentars sieht folgendermaßen aus: `<!-- Dies ist ein Kommentar -->`.

Verarbeitungsanweisungen liefern Anweisungen für die Anwendung, die das XML-Dokument verarbeitet. Sie können ebenfalls an beliebiger Stelle im Dokument platziert werden und werden nicht in der DTD deklariert. Die zugehörigen Tags sind `<?` und `?>`. Die bekannteste Verarbeitungsanweisung ist der Prolog eines XML-Dokuments, in dem unter anderem der zu verwendende Zeichensatz festgelegt wird.

Bis zu diesem Punkt haben wir die Sprache XML und den Aufbau von XML-Dokumenten beschrieben. Inzwischen haben sich neben XML weitere wichtige Standards entwickelt, die für die Arbeit mit XML notwendig sind.

3.4.2 XML-Namensräume

Der Einsatz von XML führt zu immer neuen XML-Strukturen, die zwangsläufig ähnliche oder sogar identische Problemstellungen bearbeiten. Dies führt zur Verwendung von identischen Bezeichnern in verschiedenen XML-Strukturen. Um zeitraubenden Mehrfachentwicklungen vorzubeugen, ist es wünschenswert, existierende Sprachfragmente in die eigene Sprache einzubetten. Jedoch tritt an diesem Punkt die kontextabhängige Elementeindeutigkeit zu Tage.

Zwei XML-Strukturen zu demselben Thema können einige Elemente und Attribute mit dem gleichen Inhalt verwenden. Diese weisen identische Teilbäume unter dem entsprechenden Elementknoten auf. Die Kindknoten der gleichen Elemente können sich aber auch hinsichtlich ihrer Struktureigenschaften unterscheiden. Solange die beiden Dokumente in unterschiedlichen Anwendungswelten eingesetzt werden, kommen keine Probleme auf. Sobald jedoch die beiden Dokumente gemischt werden, folgt das entstehende Zieldokument nicht mehr den Strukturierungsregeln eines der Ausgangsdokumente. Nach welchen Regeln werden nun die Elemente aufgebaut, die in beiden Ausgangsdokumenten definiert sind?

Folgende Gründe spielen eine wichtige Rolle für die Einführung von Namensräumen [Jec03]:

- Wiederverwendung bestehender XML-Strukturen in eigenen XML-Dokumenten.
- Nutzung bereits gesammelter Design-Erfahrung und dadurch Verringerung des eigenen Design-Aufwandes.
- Zusammenführung verschiedener XML-kodierter Inhalte.

Um diesen Anforderungen gerecht zu werden, stellen Namensräume eine XML-basierte Syntax zur Verfügung, um Element- und Attributnamen eines Vokabulars eindeutig zu identifizieren. So werden Bedeutungsüberschneidungen durch gleich benannte Elemente und Attribute in unterschiedlichen Vokabularen ausgeschlossen. XML-Namensräume unterstützen die freie und dezentrale Entwicklung eigener Vokabulare, die der Anwender später mischen kann.

Die Recommendation *Namespaces in XML* [BHL99] definiert die Syntax und Semantik der Namensräume. Die Element- und Attributnamen werden dabei so erweitert, dass eineindeutige Bezeichner entstehen – auch nach einer Vereinigung beliebiger Dokumente. Um einen unkoordinierten Einsatz solcher Namenserweiterungen auszuschließen, wurde das Namensschema der *Uniform Resource Identification* (URI, [BLFM03]) gewählt. Es kombiniert zentrale und dezentrale Elemente in der Handhabung und ermöglicht größtmögliche Flexibilität in der Anwendung.

Obwohl URIs XML-Namensräume identifizieren, handelt es sich dabei nicht um die Bezeichnung einer Internetquelle. Die verwendete Zeichenkette benennt ausschließlich die im Namensraum versammelten Elemente und Attribute.

Wie setzt man nun die Namensräume ein? Um die Lesbarkeit der Dokumente zu gewährleisten, geht man in zwei Schritten vor. Zunächst ordnet man den URIs Präfixe zu, was

als Bindung bezeichnet wird. Diese Präfixe werden anschließend Elementen und Attributen vorangestellt, um sie in bestimmte Namensräume zu übernehmen. Während der Verarbeitung eines XML-Dokumentes mit Namensräumen, ersetzt ein XML-Prozessor jedes auftretende Päfix durch die gebundene URI.

Das folgende Beispiel zeigt den Gebrauch von W3C-konformen XML-Namensräumen:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<myNS1:Rechnung xmlns:myNS1="http://www.xyz.com/sales">
   <myNS1:Kunde>
      <myNS1:KundenNr>4711</myNS1:KundenNr>
      <myNS1:Name>Max Mustermann</myNS1:Name>
      <myNS1:Anschrift>
         <myNS1:Straße>Musterplatz 1</myNS1:Straße>
         <myNS1:PLZ>12345</myNS1:PLZ>
         <myNS1:Ort>Musterstadt</myNS1:Ort>
      </myNS1:Anschrift>
   </myNS1:Kunde>
   <myNS1:Rechnungsposten>
<!--...-->
</myNS1:Rechnungsposten>
</myNS1:Rechnung>
```

In der Praxis hat es sich aus Gründen der Übersichtlichkeit durchgesetzt, alle in einem XML-Dokument benutzten Namensräume mit ihren Präfixen zu Beginn des Dokuments im Wurzelelement zu definieren.

3.4.3 XLink

Der Standard XLink (XML Linking Language, [DMO01]) schlägt für beliebige XML-Strukturen ein Vokabular mit zugehöriger Semantikdefinition vor, um beliebige Verweisstrukturen zu realisieren. Konzeptionell bildet XLink eine Obermenge des aus HTML bekannten `href`-Elements zur Verlinkung von HTML-Seiten.

Die XML Linking Spezifikation definiert in einem eigenen Namensraum Attribute mit zulässigen Werten, die man in eigenen XML-Strukturen verwenden kann. Indem sich die Spezifikation auf die Attributdefinition beschränkt, eröffnet sie dem Anwender einen zusätzlichen Freiheitsgrad gegenüber der bisherigen HTML-Verlinkung. Die XLink-Attribute können in beliebigen Elementen verwendet werden.

Die Spezifikation definiert folgende Attribute: `type`, `href`, `role`, `arcrole`, `title`, `show`, `actuate`, `label`, `from` und `to`. Die Attribute `show` und `actuate` spielen für die Darstellung in Browsern eine Rolle und sind damit für diese Arbeit nicht relevant. Die restlichen Attribute haben die folgenden Aufgaben:

- `type` kann die Werte `simple`, `extended`, `locator`, `arc`, `resource` und `title` haben. Sie stellen die verschiedenen von XLink definierten Link-Typen dar.
- `href` ist eine gültige URI und verweist auf die referenzierte Ressource.
- `role` ist eine gültige URI und verweist auf eine Ressource, welche die referenzierte Ressource beschreibt. Es wird für Links vom Typ `simple`, `extended`, `locator` oder `resource` verwendet.

- `arcrole` ist eine gültige URI und verweist auf eine Ressource. Es wird für Links vom Typ `simple` oder `arc` verwendet.

- `title` enthält freien Text und beschreibt die Bedeutung des Verweises in lesbarem Format.

- `label` ist ein zulässiger NCName (bestehend aus Buchstaben, Ziffern, Punkten, Bindestrichen und Unterstrichen) und definiert die textuelle Identifikation des XLinks.

- `from` ist ein zulässiger NCName, dessen Wert dem eines existierenden `label`-Attributs entsprechen muss. Das Attribut beschreibt die Sprungherkunft eines XLinks.

- `to` ist ein zulässiger NCName, dessen Wert dem eines existierenden `label`-Attributs entsprechen muss. Das Attribut beschreibt das Sprungziel eines XLinks.

Die simplen Links entsprechen größtenteils den von HTML bekannten Links. Die *extended links* oder erweiterten Verweise führen bisher nicht realisierbare Verweismöglichkeiten ein. Ein Link kann nun mehrere Ressourcen gebündelt referenzieren. Wir verdeutlichen dies mit dem folgenden Beispiel:

```
<Projekt
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:type="extended">
        <Person
                xlink:type="locator"
                xlink:href="urn:names:meier"
                xlink:label="ID001"
                xlink:title="Projektleiter"
                xlink:role="http://www.example.com/contracts/projectLeader" />
        <Person
                xlink:type="locator"
                xlink:href="urn:names:huber"
                xlink:label="ID002"
                xlink:title="Mitarbeiter"
                xlink:role="http://www.example.com/contracts/employee" />
        <Person
                xlink:type="locator"
                xlink:href="urn:names:müller"
                xlink:label="ID003"
                xlink:title="Mitarbeiter"
                xlink:role="http://www.example.com/contracts/employee" />
        <Leader
                xlink:type="arc"
                xlink:from="ID001"
                xlink:to="ID002"/>
        <Leader
                xlink:type="arc"
                xlink:from="ID001"
                xlink:to="ID003"/>
</Projekt>
```

Das Beispiel definiert einen dreigliedrigen *extended link*. Es wird zunächst das `type`-Attribut des Links des umschließenden Elements auf `extended` gesetzt. Es folgen die drei als `locator`-typisierten Verweise auf die verschiedenen Ressourcen. Für jedes `Person`-Element stellt man den `href`-Wert zur Referenzierung der Ressource, den `title`-Wert zur natürlichsprachlichen Beschreibung des Verweises und den `role`-Wert zur Referenzierung einer beschreibenden Ressource bereit. Zusätzlich identifiziert das Beispiel die Verweisteile innerhalb des extended Links eindeutig durch das `label`-Attribut.

Um die Traversierung der Verweise explizit anzugeben und damit die beliebige Navigation einzuschränken, definiert man die `arc`-Attribute `from` und `to`. In dem gezeigten Beispiel ist die Navigation so gewählt, dass die Beziehungsverfolgung von der `Projektleiter`-Ressource zu den `Mitarbeitern` möglich ist.

3.4.4 XSLT

XML trennt die Präsentation von den Daten. XML-Dokumente beschreiben nur die Daten. Doch wie wird die Präsentation festgelegt? Das W3C hat hierfür zunächst den Standard XSL (*Extensible Stylesheet Language*, [Adl01]) entwickelt. XSL ist eine Formatierungssprache und eine Anwendung von XML, so dass Struktur und Syntax in XSL XML entsprechen. Ein XSL-Prozessor erzeugt aus einem XML-Dokument und dem zugehörigen Stylesheet beispielsweise eine HTML- oder WAP-Datei.

XSLT (*XSL Transformations*, [Cla99]) wurde im Verlauf der Standardisierung von XSL von dieser abgetrennt und wird nun von einer eigenen W3C-Arbeitsgruppe vorangetrieben. XSLT wird getrennt betrachtet, weil es vielmehr eine Transformation der XML-Daten als ihre Präsentation betrachtet. XSLT ist eine Programmiersprache zur Transformation von wohlgeformten XML-Dokumenten in beliebige Unicode-Ströme und damit insbesondere wiederum in XML-Dokumente. Unter Transformation versteht man die Selektion einzelner Bestandteile des Quelldokuments, deren Umordnung sowie die Ableitung neuer Inhalte aus den bereits bestehenden.

Jedes XSLT-Stylesheet ist ein gültiges XML-Dokument, in dem alle Elemente der XSLT-Sprache im Namensraum `http://www.w3.org/1999/XSL/Transform` platziert sind. Der Namensraum wird üblicherweise an das Präfix XSL gebunden. Das Wurzelelement eines XSLT-Dokuments bildet der Knoten `stylesheet` oder alternativ `transform`. Außerdem bezeichnet jedes Stylesheet die verwendete XSLT-Version. Es folgen die Transformationsregeln, die auf Transformationsschablonen basieren. Eine Transformationsschablone besteht aus zwei Teilen: dem Lokalisierungspfad und dem Ersetzungsmuster.

Der Lokalisierungspfad liefert eine Knotenmenge. Im Rumpf des Musters definiert das Ersetzungsmuster diejenige Zeichenfolge, die statt jedem Element der lokalisierten Knotenmenge ausgegeben werden soll.

Das nachfolgende Beispiel liefert die Ausgabe `Person gefunden!` für jedes Auftreten des Knotens `Person` in der Eingabe.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:transform version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

        <xsl:template match="Person">
```

```
                <xsl:text>Person gefunden!</xsl:text>
        </xsl:template>
</xsl:transform>
```

Innerhalb des Ersetzungsmusters können im Allgemeinen beliebige Textsequenzen angegeben werden. Insbesondere ist die Verwendung wohlgeformter XML-Fragmente zugelassen. Textsequenzen werden hierbei durch direktes Anschreiben oder umschlossen durch das Element `xsl:text` definiert.

Wir gehen an dieser Stelle nicht näher auf XSLT ein, da es den Rahmen dieses Grundlagenkapitels sprengen würde. Es war vielmehr wichtig, XSLT und dessen grundlegende Funktionsweise zu beschreiben, um den Einsatz in dieser Arbeit aufzuzeigen.

3.4.5 XML Schema

Die DTD zur Definition von XML-Dokumenten weist einige Schwächen auf, die zur Entwicklung einer eigenständigen Schemasprache *XML Schema* [TBMM01, BM01] führten. Als die Metasprache SGML und damit auch die DTD entwickelt wurde, war das Anwendungsfeld klar umrissen. Zuvor papiergestützte Dokumentation sollte digitalisiert werden. Daher ist die DTD besonders zur Darstellung von Dokumentstrukturen geeignet. An eine datenorientierte Verwendung von SGML hat man zu diesem Zeitpunkt nicht gedacht. Die inzwischen zahlreichen XML-Strukturen stellen beliebigen Inhalt dar und haben mit den Beschränkungen und Unzulänglichkeiten des DTD-Mechanismus zu kämpfen. Zu den offenkundigen Beschränkungen gehören:

- Eine unzureichende Datentypunterstützung, da die DTD nur Datentypen auf Basis von Zeichenketten unterstützt. Anwender modellieren manchmal die benötigten Datentypen mit anwenderdefinierten Aufzählungstypen nach. Diese Vorgehensweise ist jedoch zeitaufwendig und fehleranfällig.

- Unzureichende Strukturierungsmöglichkeiten, da die DTD nur Operatoren zur Auftrittshäufigkeit unterstützt. Diese erlauben zwar die Kodierung beliebiger Kardinalitäten, jedoch sind sie umständlich in der Beschreibung und daher fehleranfällig. Zudem verliert die DTD an Lesbarkeit.

- Keine Unterstützung von Wiederverwendbarkeit, da Attribute immer an das umgebende Element gebunden sind. Die Nutzung in anderen DTDs ist nicht vorgesehen. Zwar lässt sich durch Definition von Entitäten etwas Ähnliches nachbilden, doch müssen diese bereits zum Erstellungszeitpunkt der DTD berücksichtigt werden.

- Ein starres Typsystem, das der Anwender nicht erweitern kann.

- Keine Unterstützung von Namensräumen, da sie in der DTD nicht angegeben werden können.

- Ein nur rudimentärer Referenzierungsmechanismus, der lediglich Verknüpfungen innerhalb eines Dokuments unterstützt.

- Die Definition der XML-Dokumente erfolgt nicht in XML. Folglich sind XML-Werkzeuge gezwungen, zusätzlich die DTD-Sprache zu implementieren.

Vor allem in Anbetracht des letzten Punktes scheint es naheliegend, eine XML-Struktur zur Inhaltsbeschreibung zu entwickeln. Da man mit XML beliebige Informationen beschreiben kann, ist auch die Beschreibung von XML-Strukturen denkbar. Somit sind sie für die Meta-Schemaebene selbstbeschreibend und das Schema eines Schemas kann durch sich selbst validiert werden.

Das W3C hat im Mai 2001 die Schemasprache *XML Schema* [TBMM01, BM01] als Recommendation verabschiedet. Zwar wird sie die DTD aus formalen Gründen nicht ersetzen, doch werden mittelfristig die neuen XML-Strukturen direkt Schemata statt DTDs definieren.

XML Schema ist vollständig in XML-Syntax formuliert und erlaubt die Formulierung beliebiger XML-Strukturen. Diese Strukturen basieren auf Elementen und Attributen. Entitäten oder Notationen können hingegen nicht durch Schemata ausgedrückt werden. XML Schema entwickelt XML implizit dahingehend weiter, dass ursprünglich von SGML übernommene – jedoch inzwischen als unpraktikabel oder potentiell fehlerträchtig angesehene – Sprachbestandteile nicht mehr durch den Grammatikmechanismus unterstützt werden. Zudem wurde, neben anderen Neuerungen, die Kommentarsyntax für Schemata neu definiert.

Die Spezifikation von XML Schema gliedert sich inhaltlich in zwei Teilbereiche. *Part 1: Structures* spezifiziert die Definition von Elementen, Attributstrukturen und wiederverwendbaren Strukturen. Mit *Part 2: Datatypes* werden Datentypen und konsistenzgarantierende Einschränkungen festgelegt. Konzeptionell rekonstruiert Part 1 die bekannte Mächtigkeit der DTD, um die evolutionäre Weiterentwicklung bestehender XML-Strukturen zu ermöglichen. Der zweite Teil der Spezifikation definiert ein eigenständiges Typsystem. Die angebotenen Typen sind an die verschiedenen verfügbaren Typsysteme aus den Programmiersprachen, Datenbanken und internationalen Standards angelehnt.

Alle durch XML Schema definierten Elemente befinden sich im Namensraum `http://www.w3.org/2001/XMLSchema`, der üblicherweise an das Präfix `xsd` gebunden wird.

Wir gehen an dieser Stelle nicht auf die Details von XML Schema ein, da wir in dieser Arbeit auf DTDs zurückgreifen. Dies geschieht aus zwei Gründen. Zum einen ist die DTD kompakter und somit leichter zu erlernen. Folglich sind unsere Beispiele für den Leser leichter nachzuvollziehen. Die Mächtigkeit von DTDs ist bis auf die mangelnde Datentypunterstützung ausreichend. Zum anderen sind Schema-Definitionen im Vergleich zu DTDs ungleich umfangreicher und würden den Rahmen dieser Arbeit sprengen. Wir ziehen daher die kompakte Schreibweise von DTDs vor.

Nichtsdestotrotz möchten wir an dieser Stelle betonen, dass in komplexeren Anwendungen Schemadefinitionen den DTDs vorzuziehen sind.

3.4.6 Verarbeitung von XML-Dokumenten

Abschließend betrachten wir, wie Anwendungen die XML-Dokumente verarbeiten können. Hierzu haben sich zwei Schnittstellen durchgesetzt. Neben der *Simple API for XML*,

einem einfach zu implementierenden Mechanismus, wird mit dem *Document Object Model* des World-Wide-Web-Konsortiums eine direkte Abbildung von XML-Dokumenten in Hauptspeicherstrukturen eingeführt. Im Folgenden beschreiben wir die grundlegenden Ansätze der beiden Schnittstellen.

Die *Simple API for XML* (kurz SAX, [SAX03]) ist ein einfacher, leichtgewichtiger Mechanismus für die Ereignis-basierte Verarbeitung von XML-Dokumenten. Die Charakterisierung als leichtgewichtiger Ansatz bezieht sich sowohl auf den Implementierungsaufwand der API selbst als auch auf ihren Integrationsaufwand in eigene Applikationen. Zunächst als Sammlung generischer Java-Schnittstellen für XML-Parser gedacht, gibt es für SAX inzwischen auch Implementierungen für C++, Python, Perl und Eiffel.

Ein SAX-basierter Parser definiert eine Reihe von Ereignissen, die durch Operationen (sog. *Callbacks*) behandelt werden. Der Aufruf der Operationen zum Zeitpunkt des Ereigniseintritts erfolgt immer durch den Parser. Der resultierende Programmcode weist daher keinen erkennbaren durchgängigen Kontrollfluss auf. Vielmehr ergibt sich dieser durch die serielle Aktivierung der verschiedenen Ereignisbehandlungsroutinen aus dem Eingabedokument.

SAX impliziert jedoch keinerlei Speicherungsstruktur zur Laufzeit. Darüber hinaus stellt SAX nur minimale Anforderungen an den verfügbaren Hauptspeicherausbau, der sich nach Art und Umfang der Übergabeparameter einer Callback-Operation richtet. Diese Art der ereignisgetriebenen Verarbeitung eines XML-Dokuments eignet sich daher besonders zur Verarbeitung von XML-Dokumenten, die den verfügbaren oder adressierbaren Hauptspeicher übersteigen.

Zusammengefasst zeigt sich, dass SAX eine leicht einzusetzende und für geeignete Anwendungsfälle sehr mächtige Schnittstelle ist. Insbesondere die geringen Hauptspeicheranforderungen sind für große XML-Dokumente geeignet. Dem stehen jedoch die fehlenden Navigationsmöglichkeiten gegenüber. Die Anwendung kann die Elemente nur in der durch das Dokument vorgegebenen Reihenfolge verarbeiten.

Die W3C-Spezifikation des *Document Object Model* (kurz DOM, [ABC+98, ABC+00]) definiert eine Menge abstrakter Schnittstellen zum lesenden und schreibenden Zugriff auf wohlgeformte XML-Dokumente sowie eine Reihe weiterer Formate. DOM ist eine Schnittstelle für beliebige XML-Dokumente. Hierzu stellt es eine Menge von Operationen bereit, mit denen Informationen aus dem Dokument extrahiert sowie speicherresidente Strukturen modifiziert und in ein XML-Dokument ausgegeben werden können.

Die verfügbaren DOM-Schnittstellen sind *Node*, *Document*, *Element*, *NodeList*, *Attr*, *ProcessingInstruction* und *Text*. Sie ermöglichen den navigierenden Zugriff auf die Baumstruktur eines XML-Dokuments.

Mit den im W3C Document Object Model versammelten Schnittstellen können XML-strukturierte Dokumente durch eine generische Schnittstelle vollständig im Hauptspeicher gehalten und manipuliert werden. Jedoch benötigt der Aufbau der DOM-Instanz beachtliche Speichermengen, was sich bei wachsender Dokumentgröße als Laufzeitproblem bemerkbar machen kann. Generell sollte man im praktischen Einsatz zwischen dem angestrebtem Modifikationsumfang und der Lesegeschwindigkeit abwägen. Während die Vorteile des DOM klar auf Seiten der Änderbarkeit – bis hin zur kompletten Erstellung vollständiger XML-Dokumente im Hauptspeicher – liegt, gewinnt SAX durch sein planbares Laufzeit- und Speicherplatzverhalten.

3.5 Verwandte Ansätze

Bevor wir in den nächsten Kapiteln auf unseren Ansatz zur Unterstützung der Daten- und Funktionsintegration eingehen, betrachten wir in diesem Abschnitt verwandte Ansätze zu dieser Thematik. Neben den von uns fokussierten Integrationsformen gibt es weitere Integrationsansätze auf unterschiedlichen Ebenen. Wir erläutern zunächst diese unterschiedlichen Integrationsformen. Anschließend konzentrieren wir uns auf die Daten- und Funktionsintegration sowie deren Kombination. Wir zeigen jeweils die Ansätze aus Forschung und Industrie auf und nennen bestehende Standards.

3.5.1 Integrationsformen

Wir beginnen mit einem Überblick über die verschiedenen Integrationsformen, die man heute vorfindet. Hier kann man zum einen unterscheiden, in welcher Tiefe eine Integration stattfindet. Zum anderen kann eine Integration auf unterschiedlichen Ebenen erfolgen.

Zunächst unterscheiden wir zwischen horizontaler und vertikaler Integration. Werden Systeme *horizontal integriert*, können sie gegenseitig ihre Funktionalität nutzen. Ein integriertes System kann eine Verbindung zu einem weiteren integrierten System aufbauen und die angebotene Funktionalität durch bestimmte Schnittstellen abrufen. Dies hat zum Vorteil, dass bereits bestehende Funktionalität wieder verwendet werden kann und nicht mehrmals implementiert werden muss. Außerdem können Geschäftsprozesse verbessert werden, wenn zwei Systeme direkt miteinander arbeiten können, statt einen Benutzer als Zwischenstation mit einzubinden. Dies ist vor allem beim Datenaustausch sehr hilfreich.

Bei der *vertikalen Integration* teilen die Systeme nicht nur gegenseitig ihre Funktionalität. Stattdessen werden die lokalen Funktionalitäten so zusammengeführt, dass auf ihrer Basis neue Funktionen entstehen. Diese werden eine Ebene höher, auf globaler Ebene für neue Anwendungen angeboten. Diese neue Funktionalität kann auch nur von den globalen Anwendungen genutzt werden, nicht von lokalen Systemen.

In unserer Arbeit streben wir die Integration von Daten und Funktionen an. Sowohl bei den Daten als auch bei den Funktionen handelt es sich um eine vertikale Integration. Wir integrieren die von den lokalen Systemen verwalteten Daten in der Form, dass wir auf globaler Ebene eine neue, föderierte Sicht auf die Daten anbieten können. Die lokalen Funktionen werden ebenfalls so zusammengeführt, dass als Ergebnis eine föderierte Funktion mit neuer Funktionalität auf globaler Ebene verfügbar ist. Die Daten und die föderierten Funktionen werden auf oberster Ebene vertikal integriert.

Integration kann in mehreren Schichten heutiger Systemarchitekturen stattfinden. Dies hat zu verschiedenen Formen der Integration geführt [JMP02]:

- Die Integration über *Portale* ist die einfachste Variante. Sie führt unterschiedliche Anwendungen über eine Web-Seite zusammen.
- Die Integration von *Geschäftsprozessen* baut die Prozesse über mehrere Anwendungen hinweg auf. Dies kann über Unternehmensgrenzen hinweg erfolgen. Hier gewinnen Web Services zunehmend an Wichtigkeit.

- Die *Anwendungsintegration* lässt Anwendungen miteinander kommunizieren, die ähnliche oder ergänzende Funktionalitäten vorweisen. Ihr Fokus liegt vor allem auf der Datentransformation und dem Message Queuing.
- Die *Informationsintegration* als tiefste Form der Integration führt heterogene Daten zusammen, so dass Anwendungen auf alle relevanten Unternehmensdaten zugreifen können.

Die Informationsintegration wiederum hat zwei grundlegende Aspekte [LR02]: Daten- und Funktionsintegration. Die Daten aus heterogenen Quellen werden über gemeinsame Schnittstellen und integrierte Schemata zugänglich gemacht, so dass sie dem Benutzer wie eine einzige Datenquelle erscheinen. Die Integration von Funktionen hingegen soll lokale Funktionen verschiedener Systeme in einheitlicher Form bereitstellen. Dem Benutzer wird so eine homogene Menge an Funktionen an die Hand gegeben, über die er die Daten manipulieren kann, die von den verschiedenen Systemen gekapselt werden. Möchte ein Unternehmen alle relevanten Daten integriert verfügbar machen, unabhängig davon, wie auf sie zugegriffen werden kann, so müssen Daten *und* Funktionen integriert werden.

Unsere Arbeit entspricht dem Fokus der Informationsintegration. Wir werden daher in den folgenden Abschnitten auf die Aspekte dieser Integrationsform eingehen: der Daten- bzw. Funktionsintegration und deren Kombination.

3.5.2 Datenintegration

Die Datenintegration ist bereits seit mehreren Jahren ein Thema in der Forschung und hat inzwischen mehrere Ansätze hervorgebracht. Vor allem in den Neunziger Jahren wurde ausgiebig an diesem Thema gearbeitet.

Viele Prototypen oder Projekte basieren auf einem objektorientierten Ansatz. Das *DISCO*-Projekt setzt Erweiterungen von ODMG und OQL [Cat94] als globales Datenmodell und als Anfragesprache ein [TRV96]. Es unterstützt auch nicht-verfügbare Informationsquellen und die transparente Hinzunahme von neuen Informationsquellen. Auch das IBM-Projekt *Garlic* setzt auf Erweiterungen von ODMG auf, um heterogene Datenquellen zu integrieren [RS97]. Das Projekt hat insbesondere neue Lösungen zur heterogenen Anfrageverarbeitung gefunden. Außerdem wurde ein Werkzeug entwickelt, mit welchem halbautomatisch Abbildungen zwischen den Schemata der Quellsysteme generiert werden können. Das *Distributed Object Management System (DOMS)* bietet eine objektorientierte Umgebung zur Integration von autonomen und heterogenen lokalen Systemen [MHG+92]. Die zu integrierenden Komponenten können neben DBMS auch Hypermedia-Systeme und Anwendungsprogramme sein. *IRO-DB* (*I*nteroperable *R*elational and *O*bject-oriented *D*ata *B*ases) integriert relationale und objektorientierte DBS [BFHK94]. Das globale Datenmodell basiert auf dem ODMG-Datenmodell und globale Anfragen werden in OQL formuliert. Das FDBS *MIND* (*M*ETU *IN*teroperable *D*atabase system) basiert auf der OMG-Architektur für verteiltes Objektmanagement (CORBA) [DDK+95]. Mit einem ODMG-konformen globalen Datenmodell und einer einheitlichen globalen Datenmanipulationssprache auf Basis eines erweiterten SQL soll eine einfache Integration objektorientierter und relationaler DBS ermöglicht werden.

Pegasus ist ein heterogenes Informationssystem, das eine objektorientierte Schnittstelle für globale Anwendungen bereitstellt und ihnen einen einheitlichen Zugriff auf autonome, heterogene und verteilte Informationssysteme ermöglicht [ADD+91]. Es basiert auf einem objektorientierten Datenmodell und integriert in erster Linie DBS. *Thor* ist ein objektorientiertes verteiltes DBMS, in dem heterogene verteilte Systeme genutzt werden können [LDS94]. Dies ermöglicht Entwicklern Programme in unterschiedlichen Programmiersprachen zu schreiben. Persistente Objekte bilden dabei die Basis. Thor ist kein FDBS, sonder vielmehr ein verteiltes DBMS, das heterogene Anwendungssysteme integriert, indem sie die von Thor verwalteten Objekte gemeinsam nutzen. Das *VODAK*-Projekt unterstützt die dynamische Integration von heterogenen und autonomen Informationssystemen [KDN91]. Das globale Datenmodell VML (VODAK Model Language) ist objektorientiert. Der Export der lokalen Schemata erfolgt über Metaklassen.

Andere Ansätze bringen spezielle Aspekte ein. Das *Carnot*-Projekt z. B. setzt Techniken der künstlichen Intelligenz ein, um eine logische Vereinigung von physisch verteilten und heterogenen Informationen umzusetzen [CHS91, HJK+92]. Mögliche Quellen sind DBS, Datenbankanwendungen, Expertensysteme und Wissensbasen. Das System *OMNIBASE* nutzt eine Wissensbasis zur Auflösung von Mehrdeutigkeiten von Tabellennamen, Inkompatibilitäten der DBMS und Inkonsistenzen der integrierten Datenbanken [REMC+88]. Die Wissensbasis enthält Informationen u. a. über die lokalen Datenbanken, Tabellenspalten, den Wertebereich der Spaltenwerte und Datentypen. Diese Informationen werden zur Durchführung globaler Anfragen benötigt. *TSIMMIS* ist ein System zur Integration von semistrukturierten Informationsquellen [PGMW95]. In diesem Projekt wurde u. a. die Anfragesprache LOREL entwickelt. Eine wichtige Funktionalität von TSIMMIS ist die automatische Generierung von Mediatoren.

Andere Ansätze fokussieren auf die Anwendungsumgebung. Das *Comandos Integration System (CIS)* ermöglicht die Integration verschiedener Anwendungsumgebungen [BNPS89]. Dabei können neben relationalen DBMS auch grafische oder öffentliche Datenbanken eingebunden werden. Das *Operational Integration System (OIS)* ermöglicht Anwendungsumgebungen Zugriff auf Daten in heterogenen Systemen über eine einheitliche Schnittstelle [GCO91]. Neben DBMS ist auch der Zugriff auf einfache Dateisysteme und spezielle Datenverwaltungssysteme möglich. OIS ist dem System CIS sehr ähnlich.

Inzwischen gibt es auch kommerzielle Lösungen für die Datenintegration. Diese kommen in erster Linie von den führenden Datenbankherstellern wie z. B. IBM und Oracle. Produkte wie IBM's DataJoiner bzw. dessen Nachfolger Relational Connect bzw. Information Integrator [IBM03a] und Oracle's Transparent Gateway [Ora03] ermöglichen den transparenten und integrierten Zugriff auf andere Datenbanktypen. Mit diesen Produkten können globale Anwendungen über eine einheitliche Schnittstelle auf unterschiedliche Datenbanken zugreifen. Außerdem werden fehlende Funktionalitäten auf Seiten der integrierten Datenbanken im Integrations-Server kompensiert. Nutzt man z. B. DB2-spezifische Funktionen an der Schnittstelle des Relational Connect und werden diese nicht in einer angebundenen Oracle-Datenbanken angeboten, dann führt Relational Connect diese Funktionen selbst aus.

Eine spezielle Unterstützung der Schemaintegration und der einfachen Verwaltung von implementierten Integrationsszenarien ist bei den Produkten allerdings nicht vorzufinden. Dies machen sich kleinere Anbieter zunutze und füllen diese Lücken mit ihren

Lösungen. MetaMatrix liefert seine Integrationslösung auf Basis von zwei Produkten aus: MetaMatrix Server und MetaBase [Met03]. Der MetaMatrix Server stellt die integrierten Daten als virtuelle Datenbank zur Verfügung und entspricht damit dem FDBS-Ansatz. MetaBase ist ein Metadaten-Repository, das alle Informationen zu den integrierten Systemen und den Abbildungen zwischen FDBS und den Quellen verwaltet. Es unterstützt ein Reihe von Standards wie MOF, CWM, XMI und UML.

Der Hersteller Nimble bietet eine Integrationslösung an, die auf XML basiert [Tec03]. Mit den Nimble-Produkten erstellt man eine XML-Sicht auf die integrierten Daten. Diese Sichten können wiederum in weiteren XML-Sichten genutzt werden. Der Zugriff auf die globalen Daten erfolgt in erster Linie über XML-Schnittstellen.

Die meisten Produkte verfügen bereits über eine gute heterogene Anfrageverarbeitung. Lücken finden sich jedoch in der verteilten, heterogenen Transaktionsverwaltung. Hier wird zwar im Allgemeinen ein Zwei-Phasen-Commit unterstützt, doch ist damit zunächst nur die globale Atomarität gewährleistet. Globale Serialisierbarkeit kann in der Regel jedoch nicht zugesichert werden.

Alle aufgeführten Prototypen und Produkte verwirklichen eine vertikale Integration.

Betrachtet man abschließend existierende Standards zur Datenintegration, so sind die SQL/MED-Wrapper des Standards SQL:1999 der bisher erste und einzige Schritt in Richtung Standardisierung von Datenintegration (vgl. Abschnitt 3.2.3). Da der Standard bisher aber nur den lesenden Zugriff abdeckt und hier noch nicht die ganze Mächtigkeit von SQL vorgibt, sind Erweiterungen in zukünftigen Versionen wünschenswert.

3.5.3 Funktionsintegration

Verwandte Ansätze hinsichtlich Funktionsintegration finden sich vor allem unter dem Begriff der Anwendungsintegration. Im Gegensatz zur Datenintegration gibt es zu diesem Thema bereits eine Vielzahl an Ansätzen und Lösungen in der Industrie und dazu passenden Standards. Wir gehen jedoch zunächst auf Arbeiten aus der Forschung ein.

Hier findet man Ansätze zweier unterschiedlicher Kategorien. Die meisten Arbeiten kommen ursprünglich aus der Datenintegration und wollen nun externe Funktionen referenzieren und damit integrieren. Diese Ansätze bauen jedoch keine föderierten Funktionen aus bestehenden lokalen Funktionen auf. Folglich wird keine vertikale Integration der Funktionen unterstützt. Chaudhuri et al. haben bereits sehr früh daran gearbeitet, wie Referenzen zu externen Funktionen in einer Anfragesprache ausgedrückt werden können [CS93]. Damit die Anfrageverarbeitung die Funktionen in der Optimierung beachten kann, werden Informationen über die Semantik der Funktionen und Regeln für ein *Rewrite* der SQL-Anweisungen bereitgestellt.

Allerdings wird nicht betrachtet, dass seitens der integrierten Funktionen Einschränkungen hinsichtlich der Zugriffsmuster bestehen. Beispielsweise müssen für den Funktionsaufruf und den benötigten Eingabewerten bestimmte Selektionskriterien in der WHERE-Klausel mitgegeben werden. Fehlen diese Werte, können die externen Funktionen nicht gestartet werden. Ansätze wie [FLMS99] erzwingen die Bereitstellung der Eingabewerte durch sog. *Binding Patterns*. Diese legen fest, für welche Spalten einer Tabelle Werte angegeben werden müssen, damit eine Menge von Zeilen ausgelesen werden kann.

Um die Beschreibung der Einschränkungen beim Zugriff externer Quellen kümmern sich Arbeiten wie [GMLY99] und [VP97]. Nicht alle integrierten Quellen bieten eine Schnittstelle mit der Mächtigkeit von SQL an. Daher muss die Anfrageverarbeitung des Integrations-Servers die Anfragen an die lokalen Systeme so formulieren, dass sie lokal verarbeitet werden können. Dazu werden die Beschreibungen der lokalen Schnittstellen und ihrer unterstützten Zugriffsfunktionalitäten benötigt.

Einer der wenigen und daher auch bekanntesten Ansätze zur reinen Funktionsintegration wird in [WWC92] vorgestellt. Das Konzept des Megaprogramming betrachtet die Zusammensetzung von Komponenten. Diese Komponenten werden Megamodule genannt und durch heterogene, autonome und verteilte Module als Methoden bereitgestellt. Ziel des Megaprogramming ist die Zusammenstellung der Methoden in der Form, dass neue Applikationen entwickelt werden. Gleichzeitig soll jedoch die Autonomie der Software-Module beibehalten werden. Dieser Ansatz verfolgt demnach eher die Idee der Bereitstellung von föderierten Funktionen. Jedoch fokussiert Megaprogramming auf die horizontale Integration, d. h. die Kombination und Verbindung von Komponenten statt der Umsetzung eines integrierten Zugriffs auf die ausgewählte Funktionalität der integrierten Quellen. Unser Ansatz hingegen implementiert eine vertikale Integration. Des Weiteren betrachtet Megaprogramming nicht die Möglichkeit, dass die verschiedenen Schemata der Quellsysteme überlappen können und damit Abhängigkeiten aufkommen.

Betrachten wir nun die Konzepte und Technologien, die sich inzwischen in Produkten wieder finden. Das bekannteste Schlagwort zum Thema Funktions- oder Anwendungsintegration ist *EAI* (Enterprise Application Integration). Es gibt verschiedene Definitionen von EAI, abhängig davon, ob man das Thema von einer geschäftlichen oder technischen Perspektive betrachtet. Aus technischer Sicht beschreibt EAI den Prozess der Integration von unterschiedlichen Anwendungen und Daten, damit Anwendungen Daten gemeinsam nutzen können und Geschäftsprozesse zwischen ihnen integriert werden. Dies soll ohne größere Änderungen an den bestehenden Systemen und Anwendungen erfolgen. Auch dieser Ansatz verfolgt eine horizontale Integration von Systemen, da er in erster Linie eine Integration und Verbindung der Systeme zum gegenseitigen Aufruf ermöglicht. Die Systeme werden nicht integriert, um damit neue, föderierte Funktionalität für weitere globale Anwendungen zu schaffen.

Produkte im Umfeld von EAI nutzen heute in den meisten Fällen *Message Broker* [Sch96b]. Message Broker basieren auf nachrichtenorientierter Middleware (*message-oriented middleware*, MOM) – besser bekannt als Message-Queuing-Systeme –, die eine asynchrone Kommunikation über Nachrichten zwischen Systemen ermöglicht. Diese Systeme können unterschiedlichsten Typs sein, wie z. B. DBMS, Legacy-, TP- oder PDM-Systeme. Der Message Broker stellt eine Art Vermittler dar, welcher den Transfer der Nachrichten zwischen den beteiligten Systemen verwaltet. Dabei übernimmt er mehrere Rollen: Er sorgt für die richtige Verteilung der Nachrichten (*message routing*) und nimmt notwendige Transformationen derselben vor. Außerdem unterstützt er die Definition eines Kontrollflusses der Nachrichten auf Basis eines Regelwerks. Mit Hilfe dieses Regelwerks kann die Verarbeitung und Verteilung der Nachrichten über Regeln wie Bedingungsüberprüfungen, mathematischen Funktionen oder auch Datentypkonvertierungen definiert werden.

Eine weitere Technologie zur Integration von Anwendungen kommt mit *J2EE-konformen Applikations-Servern* [Sun02]. Insbesondere die J2EE Connector Architecture [Sun00] ist hier von Interesse, da sie die Integration vereinfacht, indem vor allem der Zugriff auf die Systeme standardisiert ist und somit das Hinzufügen von neuen Systemen erleichtert wird. Dazu gehören u. a. Funktionalitäten wie Verbindungsverwaltung, Transaktionsverwaltung und Sicherheitsverwaltung.

Ein weiterer Ansatz, der vor allem in den letzten beiden Jahren große Beachtung gefunden hat, sind *Web Services* [W3C02]. Mit Hilfe von Web Services möchte man die Integration von Systemen nicht nur innerhalb einer Abteilung oder einer Organisation, sondern über Unternehmensgrenzen hinweg ermöglichen. Das Web erlaubt es auf einfachem Weg, dass Systeme ihre Funktionalität, oder auch Services, zur Verfügung stellen. Web Services basieren auf offenen Standards, wie z. B. SOAP [BEK+00], und nehmen Anfragen anderer Systeme entgegen. Diese Anfragen werden über eine leichtgewichtige und herstellerunabhängige Kommunikationstechnologie verschickt.

Object Request Broker (ORB) sind eine Middleware-Technologie, welche die Kommunikation zwischen verteilten Objekten oder Komponenten verwaltet und unterstützt. ORBs ermöglichen die nahtlose Interoperabilität zwischen den verteilten Objekten und Komponenten, ohne sich über die Details der Kommunikation Gedanken machen zu müssen. Dazu unterstützen sie die Transparenz hinsichtlich Rechnerknoten, Programmiersprache, Protokoll und Betriebssystem. Die Kommunikation zwischen den Objekten und Komponenten basiert in den meisten Fällen auf synchronen Schnittstellen. Die drei bekanntesten ORB-Standards sind OMG CORBA ORB [OMG03], Java RMI and RMI-IIOP [Sun03] und Microsoft COM/DCOM/COM+ [MS03].

Alle aufgeführten Technologien ermöglichen eine horizontale Integration. Sie unterstützen die gegenseitige Nutzung von Funktionen zwischen den integrierten Systemen, ermöglichen aber in ihrem ursprünglichen Sinne keine vertikale Integration. Folglich stellen sie keine föderierten Funktionen auf Basis der bestehenden Funktionen für weitere, neue Anwendungen bereit.

Die Technologie der Workflows geht eher in diese Richtung. Betrachtet man Produktions-Workflows [LR00] ohne Benutzerinteraktionen, so fügen diese lokale Anwendungen und deren Funktionalität so zusammen, dass mit einem Workflow eine neue Funktion bereitgestellt wird. Die heutigen Workflow-Systeme sind aber noch stark auf die Interaktion mit dem Benutzer ausgelegt, die für die Integration jedoch nicht benötigt wird.

Bei der Integration von Funktionen bzw. Anwendungen finden wir eine Reihe von Standards. Fast alle vorgestellten Technologien basieren auf existierenden Standards. Die J2EE Connector Architecture wurde von SUN als Standard definiert [Sun00]. Der Object Request Broker ist ein Standard der OMG [OMG03] bzw. von SUN [Sun03]. Die Standards für Web Services kommen größtenteils vom World Wide Web Consortium [W3C02, BEK+00, CCMW01]. Die Workflow Management Coalition schließlich stellt Standards für Workflow-Systeme bereit [WfM03].

3.5.4 Daten- und Funktionsintegration

In der Forschung wählen viele Ansätze zur Integration von Daten und Funktionen objektorientierte Ansätze. Dazu gehören alle objektorientierten Ansätze aus Abschnitt 3.5.2.

Diese Vorgehensweise ermöglicht neben der Beschreibung der strukturellen Eigenschaften einer Quelle auch die Beschreibung des Verhaltens der Instanzen durch Methoden und Funktionen. Dabei wird neben der strukturellen Abbildung der reinen Datenintegration auch eine operationale Abbildung beschrieben. Diese operationale Abbildung legt Übereinstimmungen zwischen Operationen auf unterschiedlichen Ebenen fest. Die operationale Integration erweitert somit das Anwendungsgebiet der Integration von der Wiederverwendung von Daten zur Wiederverwendung von Daten und Anwendungen.

Die referenzierten Ansätze verfahren jedoch nicht nach einer generellen Methode vergleichbar zu [SL90] für die Datenintegration. Stattdessen werden alle Plattformheterogenitäten durch proprietäre Implementierungen der föderierten Funktionen aufgelöst. Darüber hinaus gibt es keine oder nur wenige Mittel zur Modellierung von Semantik im föderierten Schema. Nichtsdestotrotz könnten Spezifikationen, die eher vollständig und deklarativ sind, den Integrationsprozess vereinfachen und dazu beitragen, die Abhängigkeiten zwischen den Systemen und Funktionen zu verstehen.

Einige der Produkte und Technologien, die bereits bei der Funktionsintegration aufgeführt wurden, können auch bis zu einem gewissen Grad in diesem Abschnitt genannt werden. Zumindest die Technologien EAI und J2EE Connector Architecture erheben den Anspruch, nicht nur Funktionen, sondern auch Daten zu integrieren. Allerdings liegt der Fokus klar auf den Funktionen der Anwendungsschnittstellen. Zudem sind zwar Verbindungen zu Datenbanken und Anwendungen möglich, doch geschieht dies separat. Folglich sind die Systeme nicht vertikal integriert, so dass mit einem globalen Zugriff nicht transparent auf mehrere Systeme zugegriffen werden kann.

Standards, die speziell Vorgaben zur Integration von Daten und Funktionen machen, liegen derzeit nicht vor.

3.5.5 Zusammenfassung

In den letzten Jahren haben sich unterschiedliche Integrationsformen entwickelt, die sich zum einen in der Tiefe der Integration und zum anderen in der Architekturschicht, in der die Integration stattfindet, unterscheiden. Man kann Integrationslösungen in horizontale und vertikale Integration einteilen. Die horizontale Integration ermöglicht die Kombination und Verbindung von Systemen, um angebotene Funktionalität über bestimmte Schnittstellen abzurufen und wieder zu verwenden. Die vertikale Integration strebt eine Zusammenführung lokaler Funktionalität in der Form an, dass daraus neue Funktionalität entsteht. Diese wird auf globaler Ebene von neuen Anwendungen genutzt.

Des Weiteren kann die Integration in verschiedenen Schichten einer Architektur erfolgen. Hier kann man zwischen Portal-, Geschäftsprozess-, Anwendungs- und Informationsintegration unterscheiden. Die von uns angestrebte Informationsintegration repräsentiert eine vertikale Integration, die sich aus Daten- und Funktionsintegration zusammensetzt. Ansätze dieser kombinierten Integrationsform sind momentan rar. Daher haben wir vor allem Arbeiten zur Daten- bzw. Funktionsintegration aufgeführt.

Da die Datenintegration bereits seit Beginn der neunziger Jahre ein viel beachtetes Thema in der Forschung ist, finden sich hier die meisten Ansätze. Diese Arbeiten bauen in den meisten Fällen auf objektorientierten Lösungen auf. Auch kommerzielle Lösungen

sind inzwischen vor allem von den führenden DB-Herstellern verfügbar. Diese konzentrieren sich aber primär auf die technische Umsetzung eines transparenten Zugriffs auf unterschiedliche DB-Typen. Die Unterstützung von Schemaintegration und der Verwaltung von Metadaten findet man hingegen bei kleineren Herstellern.

Für die Funktionsintegration haben sich vor allem Lösungen in der Industrie unter dem Begriff der Anwendungsintegration entwickelt. Zu den bekanntesten Ansätzen gehören EAI, Message Broker, J2EE-konforme Applikations-Server, Web Services, CORBA und Workflows. In allen Bereichen wurden entsprechende Standards entwickelt. In der Forschung findet man vor allem Vorschläge, die Referenzen auf Funktionen in die Datenintegration einbringen. Das Konzept des Megaprogramming ist einer der wenigen Ansätze zur reinen Funktionsintegration und strebt eine horizontale Integration an.

Zur Daten- und Funktionsintegration kann man alle objektorientierten Ansätze der Datenintegration aufführen. Sie beschreiben das Verhalten der Objektinstanzen durch Methoden und Funktionen. Allerdings verfahren sie nicht nach einer generellen Methode, sondern lösen Plattformheterogenitäten durch proprietäre Implementierungen der föderierten Funktionen auf.

Auf der Industrieseite können Lösungen für die Funktions- bzw. Anwendungsintegration auch für die kombinierte Integration eingesetzt werden. Dies gilt insbesondere für die Technologien EAI und J2EE Connector Architecture. Der Schwerpunkt liegt aber klar auf der Funktionsintegration.

Demnach zeigt sich, dass für eine Kombination von Daten- und Funktionsintegration, wie wir sie anstreben, momentan keine vergleichbaren Ansätze verfügbar sind. Es gibt jeweils für Daten- und Funktionsintegration vertikale Integrationslösungen, doch deren Kombination in einer weiteren vertikalen Integration ist nicht bekannt.

3.6 Zusammenfassung

In diesem Kapitel haben wir einige Grundlagen diskutiert, die wesentlich für das weitere Verständnis der vorliegenden Arbeit sind. Zunächst haben wir den Begriff von FDBS vorgestellt und haben den Schwerpunkt unserer Erläuterungen auf die Anfrageverarbeitung und Transaktionsverwaltung gelegt. Diese Aspekte haben wir insbesondere unter der Annahme einer verteilten und heterogenen Umgebung betrachtet. Daraufhin haben wir den Standard SQL:1999 beschrieben und haben vor allem diejenigen Funktionalitäten aufgezeigt, die uns bei der Umsetzung einer Integrationslösung mit einem FDBS unterstützen können. Eine weitere wichtige Technologie sind Workflow-Systeme, die wir ebenfalls in ihrem Aufbau und ihrer Funktionsweise erläutert haben. Wie in vielen anderen Arbeiten spielt auch bei uns XML eine wichtige Rolle. Daher haben wir XML und die für uns relevanten verwandten XML-Standards wie XLink und XSLT und ihre Einsatzmöglichkeiten beschrieben.

Das Kapitel schließt mit der Betrachtung verwandter Ansätze ab. Da es im Bereich der kombinierten Daten- und Funktionsintegration nur wenige Arbeiten gibt, haben wir die beiden Integrationsformen zunächst getrennt betrachtet. Dabei haben wir neben wissenschaftlichen Ansätzen aus der Forschung auch Lösungen aus der Industrie aufgezeigt.

Diese sind für uns interessant, da wir eine Lösung basierend auf bestehenden Technologien und Produkten anstreben. Abschließend haben wir Ansätze betrachtet, die ebenfalls Daten und Funktionen integrieren.

Kapitel 4

Beschreibungsmodell

Nachdem in den bisherigen Kapiteln Daten- und Funktionsintegration gemeinsam betrachtet wurden, fokussieren wir in diesem Kapitel auf die Funktionsintegration. Die Einleitung hat aufgezeigt, dass eine Integration von Funktionen benötigt wird, um den transparenten Zugriff auf alle Daten eines Unternehmens zu ermöglichen. Insbesondere werden Entwickler und Benutzer unterstützt, wenn föderierte Funktionen bereitgestellt werden. Rufen Entwickler oder Benutzer immer wieder dieselben Funktionen in derselben Reihenfolge auf, kann eine föderierte Funktion, die genau diese Aufrufsequenzen implementiert, die Arbeit erheblich erleichtern.

Wie kann eine föderierte Funktion realisiert werden? Man implementiert ihre Logik beispielsweise in einer Integrationsschicht, welche die notwendigen lokalen Funktionen aufruft und die Ergebnisse aufbereitet. Ein solches Vorgehen führt jedoch häufig dazu, dass die Lösung im Laufe der Zeit immer undurchsichtiger und schwieriger zu warten ist, da häufig nur der Entwickler selbst genau weiß, wie die Abbildung realisiert wurde. Ungenügende oder gar mangelnde Dokumentation verschlimmert das Problem. Und verlässt der Entwickler das Unternehmen, geht in den meisten Fällen auch das Wissen über die umgesetzte Lösung. Ab diesem Zeitpunkt kann die Realisierung nicht mehr ausreichend gewartet werden und an Erweiterungen oder größere Änderungen wagt man sich noch weniger.

Ein solcher Verlauf soll zukünftig vermieden werden, indem eine Trennung von Beschreibungs- und Ausführungsmodell eingeführt wird. Das Beschreibungsmodell ermöglicht die deskriptive Beschreibung der Abbildung von föderierten Funktionen auf Funktionen der lokalen Systeme. In der vorliegenden Arbeit konzentrieren wir uns auf die Abbildung der Funktionen. Abbildungsbeschreibungen bzw. -sprachen für Daten wurden bereits entwickelt, z. B. in [Sau98]. Wie die Abbildung technisch umgesetzt wird, ist zu diesem Zeitpunkt nicht von Bedeutung. Dies wird im Ausführungsmodell festgelegt. Folglich stellt das Beschreibungsmodell einen Mechanismus zur Verfügung, mit dem spezifiziert werden kann, welche lokalen Funktionen für eine föderierte Funktion aufgerufen werden, mit welchen Parametern und in welcher Reihenfolge. Dieser Mechanismus basiert auf den Funktionssignaturen, da nur diese bekannt sind. Darüber hinaus soll er die in Abschnitt 2.2 aufgezeigten Heterogenitätsformen überbrücken.

Die folgenden Vorteile sprechen für ein Beschreibungsmodell:

- *Technologie-Unabhängigkeit:*
  Die explizite Trennung vom Ausführungsmodell erlaubt den Einsatz von unterschiedlichen Technologien zur Umsetzung der Abbildung. Man muss sich nicht auf bestimmte Programmiersprachen oder Architekturen festlegen und kann von Fall zu Fall entscheiden, welche Implementierung sinnvoll ist.

- *Deklarativität:*
  Eine deklarative Beschreibung unterstützt das Verständnis der Abbildung. Sie erlaubt neben größerer Freiheit bei der Umsetzung auch eine bessere Dokumentation der Abbildung.

- *Konfliktlösung:*
  Das Beschreibungsmodell bietet Abbildungsmechanismen, welche die beschriebenen Konflikte bei der Funktionsintegration auflösen.

Diese Anforderungen stellen die Grundlage einer Abbildungssprache dar, die wir in den folgenden Abschnitten einführen. Mit der Sprache hat der Entwickler ein Werkzeug zur Hand, mit dem er spezifizieren kann, wie sich eine föderierte Funktion aus Aufrufen von lokalen Funktionen zusammensetzt und wie bestehende Heterogenitätsformen aufgelöst werden können. Die Abbildungssprache soll jedoch nicht nur die aufgeführten funktionalen Anforderungen erfüllen, sie soll vor allem auch leicht zu verstehen und einzusetzen sein. Nur dann kann gewährleistet werden, dass sie vom Entwickler in ihrer ganzen Breite genutzt wird und somit auch all ihre Stärken zum Einsatz kommen.

Der Aufbau dieses Kapitels gestaltet sich folgendermaßen. Zunächst stellen wir in den Abschnitten 4.1 und 4.2 das grundlegende Konzept zur Beschreibung der Funktionsabbildung vor und erläutern, wie ausgewählte Probleme gelöst werden. Anschließend wird in Abschnitt 4.3 die Abbildungssprache und ihr Einsatz beschrieben.

4.1 Das Konzept

Die Funktionsintegration bildet m föderierte Funktionen auf n lokale Funktionen ab. Da es sich um eine Abbildung handelt, kann man auch von Ziel- und Quellfunktionen oder -systemen sprechen. Wir werden diese Begriffe gleichbedeutend abwechselnd benutzen. Die föderierten Funktionen implementieren selbst keine Funktionalität, sondern fassen die Funktionalität der lokalen Funktionen zusammen. Die Beschreibung der Abbildung kann für jede föderierte Funktion separat betrachtet werden, da die Einzelabbildungen wieder zusammengefasst werden können. Wird eine föderierte Funktion auf eine oder mehrere lokale Funktionen abgebildet, müssen folgende Abbildungen auf Parameterebene stattfinden:

- Die Abbildung zwischen den Eingabe- und Ausgabeparametern des föderierten Systems und den lokalen Systemen.

- Die Abbildung der Ausgabeparameter von lokalen Funktionen auf Eingabeparameter anderer lokaler Funktionen.

Folglich sind vertikale und horizontale Abbildungen notwendig (vgl. auch Abbildung 2.6 in Abschnitt 2.2.3.4). Bis zu diesem Punkt ist zwar geklärt, wohin die Parameterwerte vermittelt werden müssen. Aber in welcher Reihenfolge müssen die lokalen Funktionen aufgerufen werden, wenn eine parallele Ausführung aufgrund bestehender Abhängigkeiten zwischen den Quellfunktionen nicht möglich ist? Die Antwort geben wir in der Beschreibung unseres Ansatzes.

In der Einleitung haben wir beschrieben, dass der Bedarf an Funktionsintegration aus dem Wunsch der Integration aller Unternehmensdaten entstanden ist und letztendlich auch zur Datenintegration genutzt wird. Daher scheint es naheliegend, eine Lösung auf Basis eines erweiterten SQL zu suchen. Die Abbildung könnte mit Sichten umgesetzt werden, so dass eine Sicht jeweils eine föderierte Funktion definiert. Erste Versuche in dieser Richtung haben aber gezeigt, dass eine einheitliche Beschreibung der unterschiedlichen Abbildungsrichtungen nicht möglich ist [Vog99]. Legt man zusätzlich die Ausführungsreihenfolge der lokalen Funktionen fest, wird die Abbildung sehr komplex und unübersichtlich. Dies ist vor allem darin zu begründen, dass SQL eine deklarative Sprache ist, während die Beschreibung der Ausführungsreihenfolge prozeduralen Charakter hat. Aus diesem Grund betrachten wir die Abbildung nun aus dem Blickwinkel der Parameter. Den Benutzer interessieren beim Aufruf einer föderierten Funktion hauptsächlich die Ausgabewerte. Wie erhält man diese Werte? Sie sind abhängig von Werten und/oder (Zwischen-)Ergebnissen, die wiederum von Funktionen erzeugt werden. Diese Funktionen benötigen unter Umständen Eingabewerte, um ein Ergebnis zu produzieren. Die so entstehende Kette muss bei den vorgegebenen Eingabeparametern der föderierten Funktion enden. Einige der Abhängigkeiten stehen von vornherein fest:

1. Die Ausgabeparameter der föderierten Funktion hängen immer von Ausgabeparametern der lokalen Funktion(en) ab.

2. Die Ausgabeparameter der lokalen Funktionen hängen immer von ihren Eingabeparametern ab (implizite Abhängigkeit).

3. Die Eingabeparameter einer lokalen Funktionen hängen entweder von Eingabeparametern der föderierten Funktion oder von Ausgabeparametern anderer lokaler Funktionen ab.

4. Die Werte der Eingabeparameter der föderierten Funktion sind vorgegeben.

Folglich kann die Abbildung vollständig mittels Abhängigkeiten beschrieben werden [HH99, Her00]. Abbildung 4.1 verdeutlicht dies mit Pfeilen, welche die Abhängigkeiten symbolisieren.

In diesem Beispiel wird die föderierte Funktion f_1 auf drei lokale Funktionen l_1, l_2 und l_3 abgebildet. Die Pfeile zeigen die Abhängigkeit der Parameter, wobei die Pfeilrichtung die Abbildungsrichtung und damit den Einfluss darstellt. l_3a_1 wird abgebildet auf f_1a_4, d. h., f_1a_4 ist abhängig von l_3a_1.

Betrachtet man die Parameter losgelöst von den Funktionen, erhält man mit den beschriebenen Abhängigkeiten einen gerichteten azyklischen Graphen, dessen Knoten die Parameter und dessen Kanten die Abhängigkeiten darstellen. Es fehlt noch die Definition der Ausführungsreihenfolge der lokalen Funktionen. Bei unserer Vorgehensweise muss

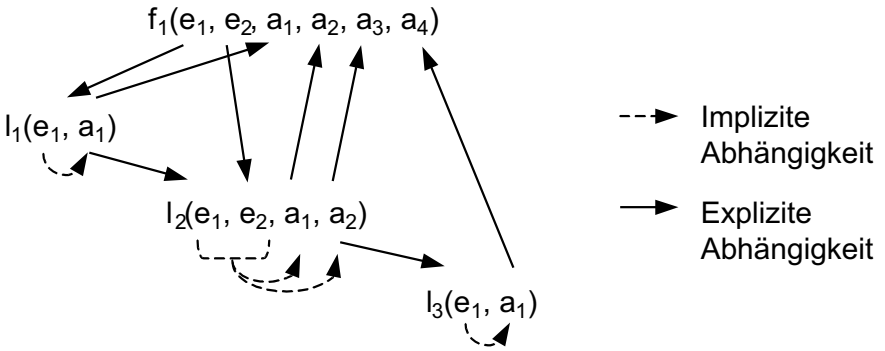

Abbildung 4.1: Funktionsabbildung dargestellt mit Abhängigkeiten.

sie aber nicht explizit festgelegt werden, da man auf die Graphentheorie zurückgreifen kann. Führt man auf dem gerichteten azyklischen Graphen eine topologische Sortierung[1] durch, wird eine mögliche Ausführungsreihenfolge ermittelt, da die Knoten aufgrund ihrer Abhängigkeiten voneinander sortiert werden. Können Teile der Abbildung parallel ausgeführt werden, so wird dies nicht beachtet und kann zu unterschiedlichen Ergebnissen der topologischen Sortierung führen. Es wird aber eindeutig bestimmt, welche lokalen Funktionen zwingend nacheinander ausgeführt werden müssen. Mit dieser Vorgehensweise ist es gelungen, die unterschiedlichen Abbildungsrichtungen sowie die Ausführungsreihenfolge der lokalen Funktionen in kohärenter Weise darzustellen.

Bisher wurde aber nur der Fall abgedeckt, in welchem das föderierte System als auch die lokalen Systeme hinsichtlich der Parameterbezeichnungen und -datentypen kongruent sind. Es fehlen Abbildungsinformationen zur Überwindung der Heterogenitäten, wie sie in Abschnitt 2.2 beschrieben wurden. Operationen wie das Konvertieren von Datentypen oder Konkatenieren von Zeichenketten werden ebenfalls als Funktionen betrachtet und in die Abbildungsbeschreibung eingebettet. Auf diese Weise bleibt die Darstellungsform der Abbildung weiterhin homogen. Abbildung 4.2 erweitert das in Abbildung 4.1 dargestellte Szenario um so genannte Hilfsfunktionen, welche die Parameter in bestimmter Form verarbeiten. Die dargestellte Abbildung verwendet die Hilfsfunktionen `cast()` und `concat()`, um eine Datentypkonvertierung und eine Konkatenierung zweier Parameterwerte vorzunehmen. Darüber hinaus ist es notwendig, Verweise auf globale Variablen oder Konstanten zu definieren. Auch dies kann mit Abhängigkeiten abgedeckt werden.

Der vorgestellte Ansatz weist folgende Vorteile auf:

1. Alle Abbildungsrichtungen werden in derselben Art und Weise beschrieben (homogenes Modell).

[1] Als *Graph* $G = (V, E)$ bezeichnet man eine Menge von *Knoten* V, die durch eine Menge von Kanten $E \subseteq V \times V$ verbunden werden. Weisen die Kanten eine Richtung auf, spricht man von einem *gerichteten Graphen*. Ein *Weg* in einem Graphen ist eine Folge von Knoten $v_0, v_1, \ldots, v_k$ aus V, die durch Kanten verbunden sind, so dass gilt $(v_{i-1}, v_i) \epsilon E$ für $i = (1, \ldots, k)$. Tritt innerhalb eines Weges kein Knoten mehrfach auf, wird dieser Weg als einfacher Weg bezeichnet. Ein einfacher Weg heißt *Zyklus*, falls $v_0 = v_k$. Ein Graph wird als azyklisch bezeichnet, falls in ihm kein Zyklus existiert. Die Knoten eines gerichteten azyklischen Graphen lassen sich durchnummerieren, so dass für jede Kante $(u, v) \epsilon E$ gilt: Nummer von $u \leq$ Nummer von v. Man nennt eine solche Nummerierung (oder Anordnung) der Knoten eine *topologische Sortierung*.

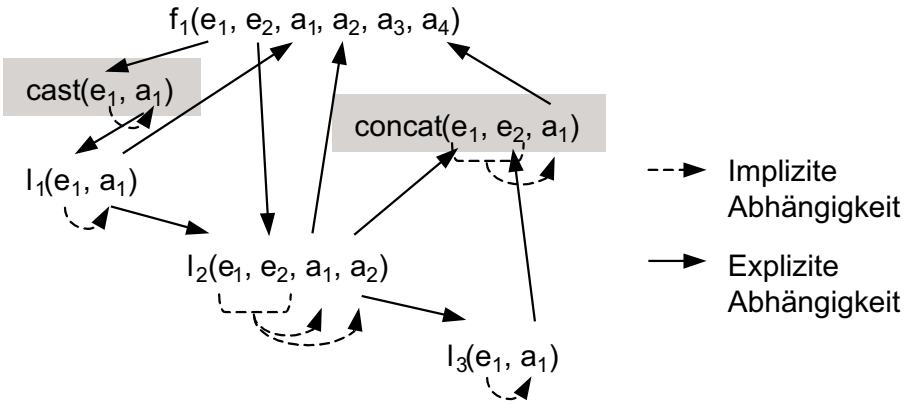

Abbildung 4.2: Funktionsabbildung mit Hilfsfunktionen (grau hinterlegt).

2. Mögliche Ablauffolgen der lokalen Funktionen müssen nicht explizit festgelegt werden, sondern werden anhand der beschriebenen Abhängigkeiten mittels einer topologischen Sortierung ermittelt.

3. Müssen Parameterwerte bearbeitet werden, sind die notwendigen Operationen als Funktionen in die Beschreibung einzubinden. Folglich bleibt die homogene Darstellung erhalten.

4. Man kann Abhängigkeiten nicht nur zwischen Funktionsparametern beschreiben, sondern auch zu Konstanten oder globalen Variablen.

5. Ein Teil der Abbildung kann anhand der Signaturen automatisch generiert werden, wie z. B. die Abhängigkeiten der Ausgabeparameter der lokalen Funktionen von ihren Eingabeparametern.

6. Der Lösungsansatz basiert auf der Graphentheorie und kann auf deren Erkenntnisse zurückgreifen.

Nachteilig erweist sich bei diesem Ansatz die intensive Dokumentation der Beschreibung. Daher ist die Bereitstellung eines grafischen Werkzeugs wünschenswert, das dem Entwickler die grafische Beschreibung der Abhängigkeiten mittels Pfeilen ermöglicht. Das Werkzeug generiert anschließend den Code der Abbildungsspezifikation.

Unsere grundlegende Idee zur Abbildungsbeschreibung betrachtet bisher lediglich die Abbildung der Parameter, deren Verarbeitung in Zwischenschritten sowie die Ablauffolge der Funktionen. Es liegen aber noch weitere Aspekte vor, die bei der Spezifikation einer Abbildung beschrieben werden müssen. Diese Aspekte untersuchen wir in den nächsten Abschnitten.

4.2 Aspekte der Funktionsabbildung

In folgenden Abschnitt gehen wir auf weitere Aspekte der Funktionsintegration ein. Wir zeigen auf, wie sie auf Basis unseres Grundkonzepts der Parameterabhängigkeiten modelliert werden können, um das Abbildungsmodell einfach zu halten. Ein Beispielszenario

dient als Grundlage zur Verdeutlichung der unterschiedlichen Fälle. Unser Beispiel umfasst zwei Werke einer imaginären Firma und deren Lieferanten. Da die EDV-Strukturen dieser Firma historisch gewachsen sind, werden in beiden Werken die verwalteten Bauteile auf sehr unterschiedliche Weise repräsentiert. In der Datenbank des Lieferanten, die bei der Integration mit einbezogen wird, werden die lieferbaren Bauteile wiederum in einer anderen Form gespeichert. Jede der drei Organisationen betreibt ein Anwendungssystem, das eine Schnittstelle für den Zugriff auf die Teiledaten bereitstellt.

4.2.1 Basisfunktionalität der Abbildung

Wir beginnen mit jenen Aspekten, die wir als Basisfunktionalität einer Abbildung einschätzen. Für die grafische Darstellung der Beispielabbildungen wurden folgende Darstellungen festgelegt: Föderierte Funktionen werden mit einen Rahmen dargestellt. Zwischen lokalen und Hilfsfunktionen werden keine Unterschiede gemacht. Die dargestellten Pfeile repräsentieren die (expliziten) Abhängigkeiten zwischen den Parametern bzw. Funktionen.

4.2.1.1 Hilfsfunktionen

Mit dem Beispiel in Abbildung 4.2 wurde bereits erläutert, dass Hilfsfunktionen benötigt werden, um Parameterwerte zu konvertieren oder in anderer Form zu bearbeiten, bevor sie an eine Funktion weitergegeben werden. Hilfsfunktionen müssen in separaten Systemen implementiert werden. Diese Hilfssysteme unterscheiden sich in ihren Eigenschaften nicht von lokalen Systemen. Innerhalb der Implementierung einer Hilfsfunktion dürfen keine Funktionen anderer, im Rahmen der Integration benutzter Systeme aufgerufen werden. Dadurch wird vermieden, dass die zur Integration notwendigen Maßnahmen in der Implementierung einer lokalen Funktion verborgen werden.

4.2.1.2 Ausführungsreihenfolge

In Abschnitt 4.1 haben wir gezeigt, dass eine explizite Modellierung der Ausführungsreihenfolge der lokalen Funktionen nicht notwendig ist, da sie sich mit einer topologischen Sortierung anhand der Parameterabhängigkeiten bestimmen lässt. Betrachtet man die modellierten Abhängigkeiten in Abbildung 4.3, so bestehen zwischen den beiden lokalen Funktionen `GetPart()` und `GetPreis()` keine Abhängigkeiten. Folglich können die beiden Funktionen parallel ausgeführt werden. Allgemein ist die parallele Ausführung von lokalen Funktionen immer dann möglich, wenn der Abhängigkeitsgraph keinen Weg zwischen ihren Parametern aufweist. Somit ist keine explizite Modellierung der parallelen Ausführung notwendig. Diese erfolgt, soweit es die Abhängigkeiten und die Ausführungszeiten der einzelnen Funktionen ermöglichen, automatisch.

Sind innerhalb eines Abhängigkeitsgraphen lokale Funktionen von den Seiteneffekten anderer lokaler Funktionen abhängig, ohne von deren Parameter abhängig zu sein, muss diese Tatsache im Abhängigkeitsgraphen durch eine explizite Abhängigkeit zwischen diesen beiden Funktionen modelliert werden. Ebenso können Abhängigkeiten zwischen Funktionen dazu benutzt werden, um parallele Ausführung zu verhindern (falls etwa ein System parallele Ausführung nicht oder nur unter erheblichen Geschwindigkeitseinbußen erlaubt).

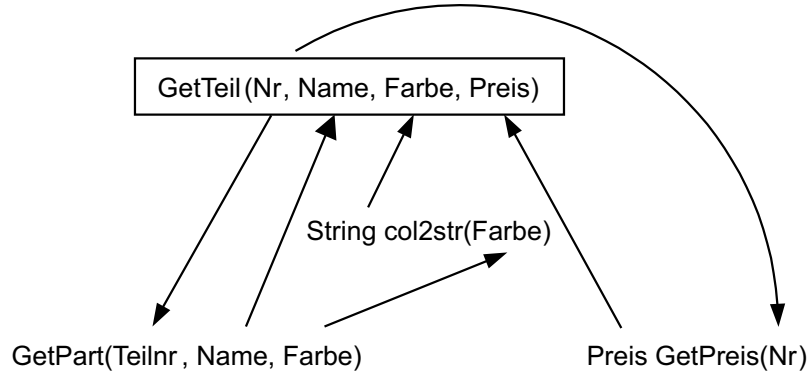

Abbildung 4.3: Die lokalen Funktionen `GetPart()` und `GetPreis()` können parallel ausgeführt werden, da keine Parameterabhängigkeiten zwischen ihnen bestehen.

Die Modellierung von Funktionsabhängigkeiten erfolgt analog zu Abhängigkeiten zwischen Parametern. Eine Abhängigkeit $funktion_a \rightarrow funktion_b$ hat die Bedeutung „$funktion_b$ wird erst aufgerufen, wenn $funktion_a$ korrekt ausgeführt wurde“. Abbildung 4.4 zeigt eine Modellierung der föderierten Funktion `GetTeil()` ohne parallele Ausführung.

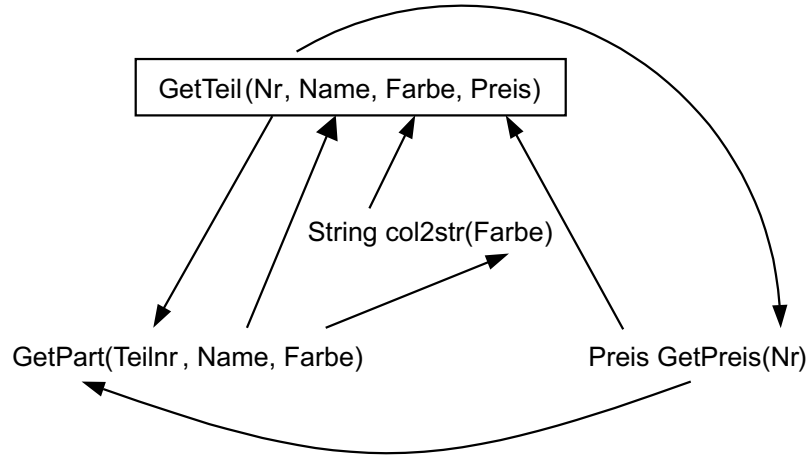

Abbildung 4.4: Vermeidung paralleler Ausführung der Funktionen `GetPart()` und `GetPreis()` durch Definition einer Funktionsabhängigkeit zwischen den beiden Funktionen.

Tabelle 4.1 fasst die bisher beschriebenen Abhängigkeitsrelationen zwischen Parametern und Funktionen und deren Bedeutungen zusammen.

4.2.1.3 Konstanten

Konstanten werden im Rahmen der Funktionsintegration benötigt, wenn für Eingabeparameter lokaler Funktionen keine Werte aus den Eingabeparametern der föderierten Funktion zur Verfügung stehen. Entweder sind die notwendigen Konstanten bereits im jeweiligen lokalen System definiert oder es müssen – analog zu den Hilfsfunktionen – so genannte Hilfskonstanten definiert werden.

Relation	Bedeutung
Parameter 1 → Parameter 2	Der Wert von Parameter 1 wird beim Aufruf der zu Parameter 2 gehörenden Funktion als Wert für Parameter 2 eingesetzt.
Funktion 1 → Funktion 2	Funktion 2 wird nur (und nur dann) ausgeführt, nachdem die Ausführung von Funktion 1 fehlerfrei abgeschlossen wurde.

Tabelle 4.1: Grundlegende Abhängigkeitsrelationen.

Den ersten Fall illustrieren wir am Beispiel der Funktion `fseek()` der Standardbibliothek von C [KR90] zum Setzen der Dateiposition. Als dritten Parameter erwartet sie den Ursprung, zu dem die neue Position relativ gesetzt werden soll. Als Werte kommen die in `stdio.h` (also im lokalen System) definierten `SEEK_SET`, `SEEK_CUR` oder `SEEK_END` in Frage.

Ein Beispiel für den zweiten Fall tritt bei der Modellierung der föderierten Funktion `GetTeile()` unter Verwendung der `GetAlleTeile()`-Funktion des Anwendungssystems von Werk 1 auf. Diese Funktion erwartet als Parameter die Angabe einer Sparte, die von der föderierten `GetTeile()`-Funktion allerdings nicht bereitgestellt wird. Da das föderierte System nur zur Integration der Automobil-Sparte dienen soll, muss der lokalen Funktion in jedem Fall der Wert `"KFZ"` übergeben werden. Abbildung 4.5 zeigt die zugehörige Abbildungsbeschreibung.

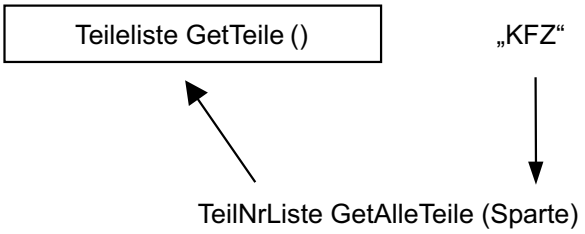

Abbildung 4.5: Der Wert „KFZ“ wird der lokalen Funktion `GetAlleTeile()` als Konstante übergeben.

Normalerweise erfolgt die Definition von Konstanten innerhalb der Schnittstellenbeschreibung des Anwendungssystems. Somit ist eigentlich keine Modellierung der Werte von Konstanten im Rahmen der Funktionsintegration notwendig. Soll aber die zu entwickelnde Abbildungssprache Funktionsintegration und Schnittstellenbeschreibung vereinigen, muss dies auch mit berücksichtigt werden.

Neben der beschriebenen Abhängigkeit *Konstante → Parameter* ist zusätzlich die Abhängigkeit *Funktion → Konstante* möglich. Tabelle 4.2 listet die in Verbindung mit Konstanten hinzugekommenen möglichen Abhängigkeitsrelationen und deren Bedeutungen auf.

Relation	Bedeutung
Konstante → Parameter	Der Wert der Konstanten wird beim Aufruf der zu dem Parameter gehörenden Funktion als dessen Wert gesetzt.
Funktion → Konstante	Die Funktion muss zuerst erfolgreich ausgeführt worden sein, bevor von der Konstanten ausgehende Abhängigkeitsrelationen aufgelöst werden können.

Tabelle 4.2: Konstanten in Abhängigkeitsrelationen.

4.2.1.4 Referenzierung von Teilen der Parameter

Parameter können auch komplexe Datentypen wie Felder oder Strukturen enthalten. Oft wird innerhalb eines Abhängigkeitsgraphen nur ein bestimmter Wert aus einer Struktur benötigt. Ebenso kann auf ein bestimmtes Element innerhalb eines Feldes zugegriffen werden. Ohne den Einsatz zusätzlicher Konstrukte kann man den Zugriff auf Komponenten von komplexen Datentypen mit Hilfsfunktionen modellieren. Das Beispiel aus Abbildung 4.6 zeigt die Realisierung der globalen Funktion `GetGroesse()` unter Verwendung der Funktion `GetPart()` aus Werk 2. Bei dem Parameter `Dim` handelt es sich um eine Struktur mit den drei Elementen `x`, `y` und `z`.

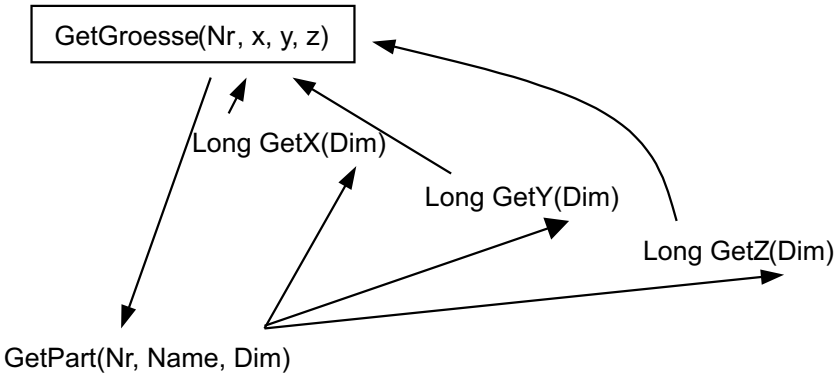

Abbildung 4.6: Verarbeitung der Strukturkomponente `Dim` durch Hilfsfunktionen.

Das Beispiel verdeutlicht die Nachteile dieses Vorgehens. Es wird eine große Anzahl verschiedener, nicht wieder verwendbarer Hilfsfunktionen benötigt. Daher braucht man ein Konstrukt, um direkt auf die Komponenten von komplexen Parametern zugreifen zu können. In Abschnitt 4.3.12 beschreiben wir die von uns gewählte Lösung auf Basis von einer Art „Pointer“.

4.2.1.5 Mehrfachinstanzierung

Wird in einer Abbildungsbeschreibung eine Funktion mehrfach benötigt (z. B. eine Konvertierungsfunktion), müssen die verschiedenen Instanzen als solche unterscheidbar sein. Dies kann beispielsweise durch angefügte Indizes realisiert werden. Wird nicht explizit

unterschieden, mag der Entwickler die Mehrfachinstanzierung zwar erkennen, ein Programm wird aber kaum zum gewünschten Ergebnis führen.

4.2.1.6 Textuelle Beschreibung

Manche Abbildungen können nicht oder nur unter erheblichem Aufwand durch Abhängigkeiten und Hilfsfunktionen beschrieben werden. In solchen Fällen ist eine textuelle Beschreibung der Abbildung wünschenswert. Wird aus der Abbildungsbeschreibung ein lauffähiges System erzeugt, muss ein Entwickler die textuell beschriebenen Teile „von Hand“ ausprogrammieren.

Textuelle Beschreibungen dürfen nicht losgelöst innerhalb einer Abhängigkeitsbeschreibung vorkommen, sondern müssen in den Graphen mit eingebunden werden. Nur dann werden bei der automatischen Generierung von föderierten Systemen die Platzhalter für den vom Entwickler einzufügenden Programmiercode an der richtigen Stelle im Code stehen.

Die textuelle Beschreibung wird so in den Graph eingebunden, dass sie von allen Parametern abhängig ist, die innerhalb des beschriebenen Bereichs verändert werden. Sämtliche nachfolgenden Parameter, die in diesem Bereich verändert oder erzeugt werden, sind wiederum von der textuellen Beschreibung abhängig. Soll z. B. die föderierte Funktion `GetFarbe()` ohne Verwendung von Hilfsfunktionen realisiert werden, kann die Abbildungsbeschreibung wie in Abbildung 4.7 aussehen.

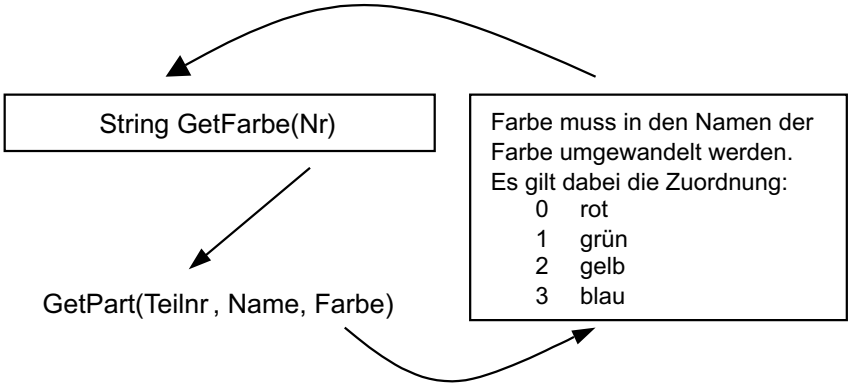

Abbildung 4.7: Textuelle Beschreibung der Umwandlung der Farbnummer in den entsprechenden Farbnamen.

Tabelle 4.3 zeigt die möglichen Verknüpfungen von textuellen Beschreibungen mit den bisher beschriebenen Parametern, Funktionen und Konstanten. Textuelle Beschreibungen können prinzipiell mit allen Knotentypen verknüpft werden.

4.2.2 Weitergehende Aspekte

Nachdem wir die grundlegenden Elemente des Abbildungsmodells beschrieben haben, stellen wir in diesem Abschnitt weiterführende Aspekte der Funktionsabbildung vor. Obwohl einige davon den deklarativen Charakter der Abbildungsbeschreibung aufweichen,

Relation	Bedeutung
Parameter → Text	Der Parameter wird innerhalb der textuellen Beschreibung verarbeitet.
Text → Parameter	Innerhalb der textuellen Beschreibung erfolgt eine Zuweisung zu diesem Parameter.
Funktion → Text	Die Funktion muss vor der textuellen Beschreibung ausgeführt werden.
Text → Funktion	Die Funktion darf erst nach der textuellen Beschreibung ausgeführt werden.
Text → Konstante	Die Abarbeitung der textuellen Beschreibung muss beendet sein, bevor von der Konstanten ausgehende Abhängigkeitsrelationen aufgelöst werden können.
Konstante → Text	Der Wert der Konstanten wird innerhalb der textuellen Beschreibung benutzt.

Tabelle 4.3: Verknüpfung textueller Beschreibungen.

ist ihre Einführung sinnvoll[2]. Zu Beginn dieser Arbeit haben wir erläutert, dass eine teilweise oder besser noch vollständige Generierung des Codes aus der Abbildungsbeschreibung heraus möglich sein soll. Belässt man die Beschreibung zu diesem Zeitpunkt, wie sie ist, können erhebliche Teile des Ausführungsmodells nicht automatisch generiert werden. Daher wird das Modell in den folgenden Abschnitten um Mechanismen u. a. zu den Punkten Objektorientierung, Kontrollflusssteuerung, Kompensationen und Aktualisierungsprobleme erweitert.

4.2.2.1 Objektorientierung

Schnittstellen von objektorientierten Systemen basieren im Gegensatz zu konventionellen Systemen nicht auf den exportierten Funktionen, sondern führen den Begriff eines Objekts ein. Ein Objekt kapselt Daten und Code und stellt seine Schnittstelle in Form von Methoden und Attributen bereit. Methoden gleichen dabei den bisher betrachteten Funktionen. Attribute verhalten sich wie Variablen einer Programmiersprache. Ein lesender oder schreibender Zugriff verändert jedoch nicht nur den Wert der Variablen, sondern kann auch weitere Aktionen zur Folge haben.

Außerdem können mehrere Objekte desselben Typs gleichzeitig instanziert werden, d. h., es existieren mehrere Versionen derselben Schnittstelle, die unter Umständen verschiedene Zustände besitzen und damit unterschiedliche Ergebnisse liefern. Unterstützt das jeweilige System nicht nur die entfernte Instanzierung von Objekten, sondern auch die Verwendung der Objekte innerhalb entfernter oder lokaler Methoden, muss auch innerhalb von Abhängigkeitsgraphen die Verwendung von Objekten als Parameter möglich sein.

[2] Auch SQL als bekanntester Vertreter einer deklarativen Sprache wurde vor allem mit SQL:1999 prozedural deutlich erweitert.

Lokale Attribute

Attribute verhalten sich gegenüber dem benutzenden Programm ähnlich wie globale Variablen. Es können Werte geschrieben und wieder ausgelesen werden. Ebenso kann man den Wert eines Attributs durch eine Methode verändern. Im Gegensatz zu Variablen können aber durch einen Zugriff auf ein Attribut weitere Aktionen ausgelöst werden.

Die Einbindung von Quellsystemattributen erfolgt ähnlich zu der von Parametern oder Konstanten. Ein Attribut kann dabei sowohl Ausgangspunkt (lesender Zugriff) als auch Ziel (schreibender Zugriff) einer Abhängigkeitsrelation sein. Außerdem können Attribute ähnlich wie Konstanten in Abhängigkeitsrelationen mit Funktionen auftreten.

Eine Abhängigkeitsrelation der Form Attribut → Parameter ist grundsätzlich jederzeit erfüllt (ein Attribut hat im Gegensatz zu einem Parameter immer einen Wert). Ist der Wert eines Attributs abhängig von einem Seiteneffekt einer vorangegangenen Funktion, muss eine zusätzliche Abhängigkeitsrelation Funktion → Attribut eingeführt werden, damit die dem Attribut folgenden Abhängigkeitsrelationen erst aufgelöst werden, wenn das Attribut den richtigen Wert besitzt.

Abbildung 4.8 zeigt die Modellierung der globalen Funktion `GetPreisEuro()`. Diese benutzt die Funktion `GetPreis()` des Lieferantensystems, welche abhängig vom Wert des lokalen Attributs `Waehrung` den Preis in der entsprechenden Währung zurückgibt.

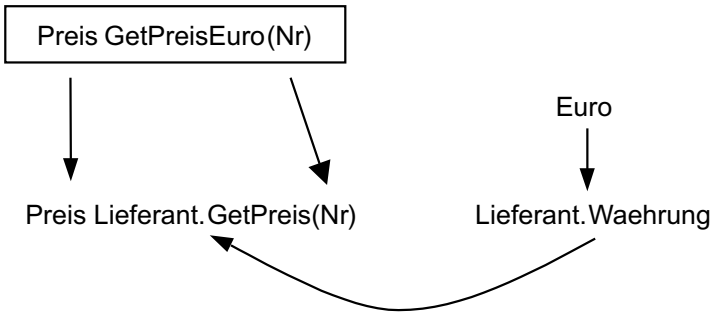

Abbildung 4.8: Das lokale Attribute `Waehrung` innerhalb eines Abbildungsgraphen.

Tabelle 4.4 zeigt die durch die Einführung von Attributen neu hinzugekommenen Möglichkeiten von Abhängigkeitsrelationen.

Globale Attribute

Zur Realisierung von globalen Attributen benötigt man die Modellierung von Abhängigkeitsgraphen für Attribute analog zu denen für föderierte Funktionen. Es muss jeweils ein Abhängigkeitsgraph für den lesenden und schreibenden Zugriff erstellt werden.

Innerhalb objektorientierter Programmiersprachen werden Attribute meist durch zwei (nach außen nicht sichtbare) Methoden implementiert, jeweils eine für den lesenden bzw. schreibenden Zugriff. So erfolgt die Repräsentation des globalen Attributs

```
Waehrung_t Waehrung
```

aus dem Beispiel durch die folgenden beiden Zugriffsfunktionen:

Relation	Bedeutung
Parameter → Attribut	Der Wert des Parameters wird in das Attribut geschrieben.
Konstante → Attribut	Der Wert der Konstanten wird in das Attribut geschrieben.
Attribut 1 → Attribut 2	Der Wert des ersten Attributs wird in das zweite Attribut geschrieben.
Attribut → Parameter	Der Wert des Attributs wird beim Aufruf der zu dem Parameter gehörenden Funktion als dessen Wert eingesetzt.
Attribut → Funktion	Die zu dem Attribut hinführenden Abhängigkeitsrelationen müssen erfolgreich aufgelöst worden sein, bevor die Funktion ausgeführt werden kann.
Attribut → Konstante	Die zu dem Attribut hinführenden Abhängigkeitsrelationen müssen erfolgreich aufgelöst worden sein, bevor die von der Konstanten ausgehenden Abhängigkeitsrelationen aufgelöst werden können.
Attribut → Text	Das Attribut wird innerhalb der textuellen Beschreibung verarbeitet.
Text → Attribut	Innerhalb der textuellen Beschreibung erfolgt eine Zuweisung zu diesem Attribut.

Tabelle 4.4: Abhängigkeiten in Verbindung mit Attributen.

```
Waehrung_t GetWaehrung();
SetWaehrung(IN Waehrung_t Waehrung);
```

Zur Realisierung von Attributen in föderierten Systemen sind also diese beiden Funktionen nötig. Abbildung 4.9 zeigt die mögliche Modellierung des globalen Attributs `Waehrung` durch das gleichnamige lokale Attribut des Lieferantensystems.

Die Abhängigkeitsrelation `Waehrung` → `Lieferant.Waehrung` aus Abbildung 4.9 rechts modelliert die eigentliche Wertzuweisung, während `Lieferant.Waehrung` → `SetWaehrung()` zur Vervollständigung des Abhängigkeitsgraphen dient, damit dieser von einer Ausführungskomponente richtig aufgelöst werden kann.

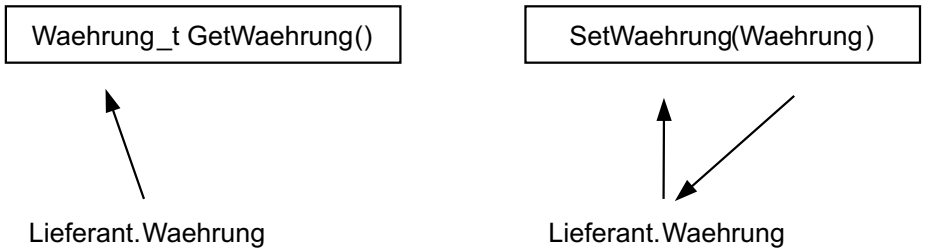

Abbildung 4.9: Modellierung von Zielsystemattributen: lesender Zugriff (links) und schreibender Zugriff (rechts).

Soll die Verwendung zusätzlicher Funktionen zum Setzen bzw. Auslesen des globalen Attributs vermieden werden, muss das globale Attribut selbst in den Abhängigkeitsgraphen integriert werden. Der lesende Zugriff erfolgt ähnlich zu der Ausführung einer Funktion. Das Attribut wird dabei analog zu dem Rückgabewert einer globalen Funktion in den Abhängigkeitsgraphen integriert (Abbildung 4.10 links).

Der schreibende Zugriff ist eine Besonderheit, weil das globale Attribut an zwei Stellen innerhalb des Abhängigkeitsgraphen auftaucht (Abbildung 4.10 rechts). Zum einen in der Abhängigkeitsrelation `Global.Waehrung` → `Lieferant.Waehrung` – hierbei erfolgt die Verwendung von `Global.Waehrung` analog zu einem Eingabeparameter einer Zielfunktion. Zum anderen in der umgekehrten Form `Lieferant.Waehrung` → `Global.Waehrung` stellvertretend für die Zielsystemfunktion selbst, um den Abhängigkeitsgraph zu vervollständigen. Diese zweite Abhängigkeit drückt dabei keine Wertübergabe o. Ä. aus, sondern dient allein dazu, dem Abhängigkeitsgraph das richtige Ziel zu geben.

Objekte als Parameter

Werden Objekte innerhalb eines Abhängigkeitsgraphen nur als Container für Methoden oder Attribute benutzt, d. h., werden sie nicht mehrfach instanziert oder als Parameter übergeben, können objektorientierte Systeme wie bisher beschrieben integriert werden. Es müssen keine neuen Konstrukte eingeführt werden. Ein Beispiel für diese Art des objektorientierten Ansatzes ist die Verwendung der Interfaces von CORBA.

Werden Objekte auch als Parameter benutzt (dazu zählt auch die Erzeugung eines Objekts beispielsweise durch einen Konstruktor), müssen Abhängigkeitsrelationen auf Teile

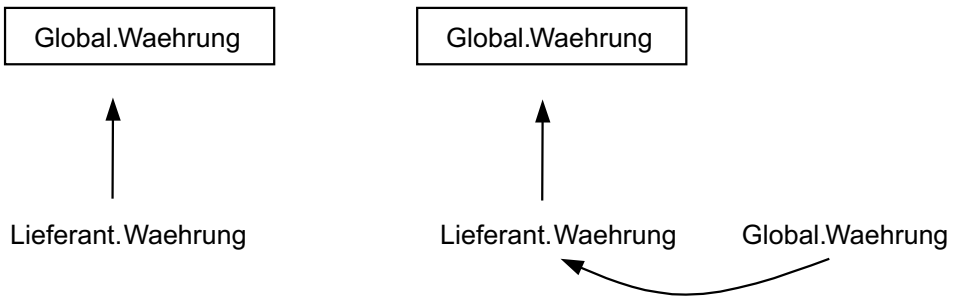

Abbildung 4.10: Fortsetzung der Modellierung von Zielsystemattributen: lesender Zugriff (links) und schreibender Zugriff (rechts).

Relation	Bedeutung
Konstruktor → Parameter	Konstruktoren besitzen i. Allg. keinen Rückgabewert. Deshalb muss in Abhängigkeitsrelationen auf den Konstruktor selbst verwiesen werden.

Tabelle 4.5: Abhängigkeitsrelation Konstruktor → Parameter.

von Parametern – in diesem Fall Methoden oder Parameter von Methoden – verweisen (ähnlich wie in Abschnitt 4.2.1.4 beschrieben).

Werden innerhalb eines Abhängigkeitsgraphen mehrere Instanzen eines Objekts benutzt, wird dies durch die Verwendung mehrerer Konstruktoren erreicht (die ihrerseits, wie in Abschnitt 4.2.1.5 beschrieben, unterschieden werden müssen). Da Konstruktoren im Allgemeinen nicht explizit mit einem Rückgabeparameter definiert werden, muss eine neue Abhängigkeitsrelation Konstruktor → Parameter eingeführt werden (siehe Tabelle 4.5). Ansonsten treten bei der Verwendung von Objekten als Parameter keine neuen Typen von Abhängigkeitsrelationen auf, da sie genau wie Parameter mit normalen Datentypen behandelt werden.

4.2.2.2 Kontrollflusssteuerung

In den bisher gezeigten Beispielen ergab sich die Reihenfolge, in der die modellierten Abhängigkeiten aufgelöst werden müssen, rein aus der Art und Weise wie Parameter oder Funktionen miteinander durch Abhängigkeitsrelationen verknüpft wurden. Die Ausführungsreihenfolge der lokalen Funktionen wurde dadurch von vornherein durch die Modellierung der Abhängigkeiten festgelegt. Mehrfache oder bedingte Ausführung von Teilen des Abhängigkeitsgraphen wurden nicht verwendet.

Programmiersprachen bieten zu diesem Zweck Konstrukte wie Schleifen oder bedingte Anweisungen an. Die Übertragung dieser Konstrukte auf Abhängigkeitsgraphen besprechen wir in den nächsten Abschnitten. Außerdem betrachten wir weitergehende Aspekte zur parallelen Ausführung.

Bedingte Ausführung

Bedingte Anweisungen werden in Programmiersprachen durch `if then`-Konstrukte realisiert. Diese bestehen aus einer Bedingung und dem Code, der ausgeführt werden soll, falls diese Bedingung zutrifft. Oft wird weitergehende Funktionalität durch `else`- oder `else if`-Klauseln oder durch Konstrukte wie die `case`-Anweisung geboten. Alle diese Konstrukte lassen sich aber auf die einfache Form der `if then`-Anweisung zurückführen (im Falle der `else`-Klausel z. B. durch die Verwendung einer zweiten `if then`-Anweisung mit negierter Bedingung).

Abbildung 4.11 zeigt die Abhängigkeiten, die zur Modellierung der föderierten Funktion `GetFarbe()` notwendig sind. Um kein neues Konstrukt für bedingte Anweisungen einführen zu müssen, wird sowohl der Bedingungs- als auch der Code-Teil einer bedingten Anweisung als Funktion modelliert. Dabei hängt die Code-Funktion von der Bedingungsfunktion ab. Abbildung 4.12 zeigt eine solche Modellierung von `GetFarbe()`. Die Funktion `IsEqual()` vergleicht die beiden übergebenen Werte. Stimmen die Werte überein, kehrt die Funktion zurück. Andernfalls bricht sie mit einem Fehler ab.

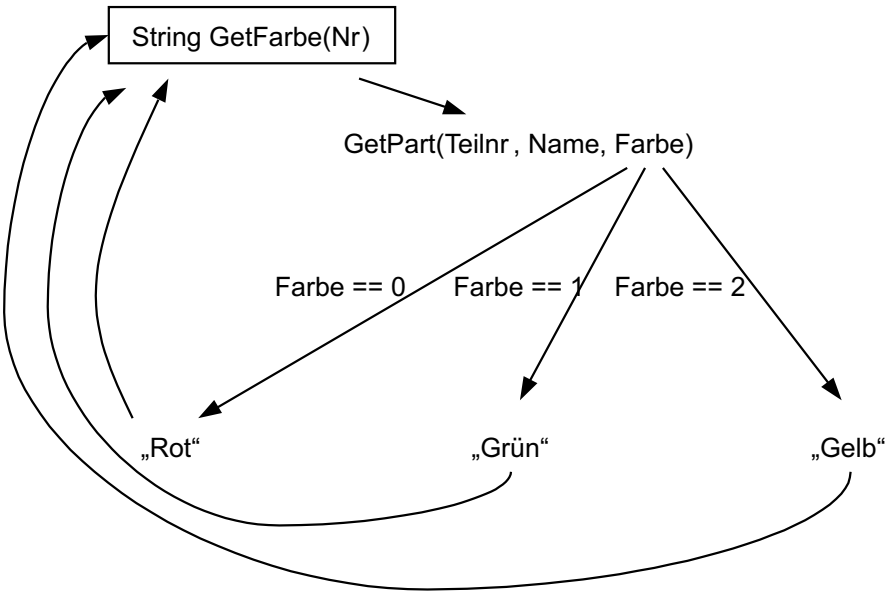

Abbildung 4.11: Die gewünschte Modellierung einer bedingten Ausführung.

Die drei Instanzen der Funktion `IsEqual()` können parallel ausgeführt werden. Schlagen dabei eine oder mehrere Funktionen fehl, wird die Ausführung des Abhängigkeitsgraphen nicht sofort abgebrochen. Der Abbruch erfolgt nur, wenn alle Instanzen der Funktion `IsEqual()` fehlschlagen.

Allgemein wird versucht, Abhängigkeiten zu Rückgabewerten, -parametern und der föderierten Funktion über jeden möglichen Weg zu erfüllen. Erst wenn dies nicht möglich ist, bricht die Ausführung des Abhängigkeitsgraphen mit einer Fehlermeldung ab.

Sämtliche lokalen Funktionen und auch die Hilfsfunktionen können grundsätzlich als Bedingungsfunktionen dienen. Vergleichsfunktionen (wie `IsEqual()`) werden dabei besonders häufig benötigt. Daher sollten solche Funktionen in einer Art Standardbibliothek vorhanden sein (siehe Abschnitt 4.2.2.5).

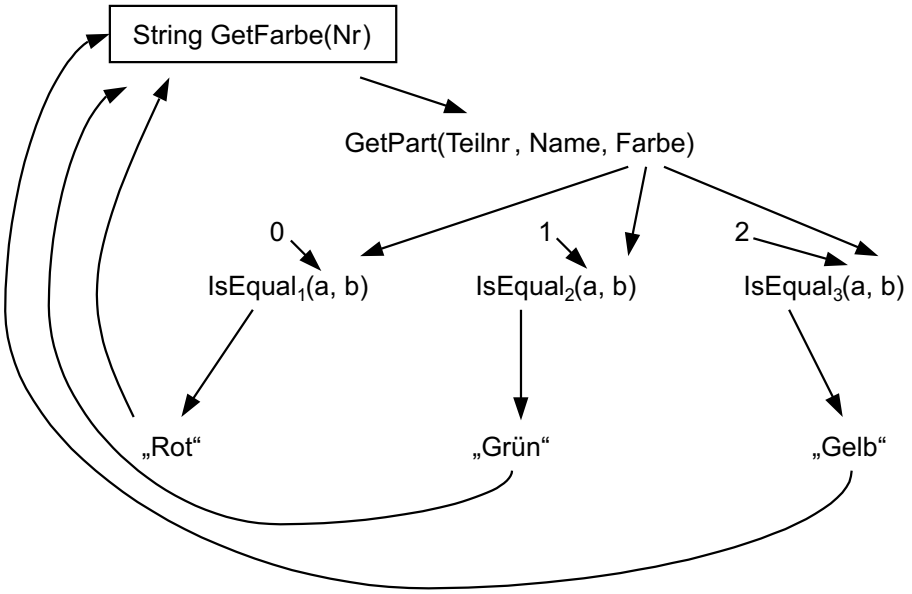

Abbildung 4.12: Eine bedingte Ausführung wird definiert, indem Bedingungs- und Code-Teil als Funktionen modelliert werden.

Iterativer Fall oder Schleifen

Schleifenkonstrukte in Programmiersprachen ermöglichen die wiederholte Ausführung bestimmter Programmteile. Ähnlich den bedingten Anweisungen bestehen Schleifen aus einer Bedingung und dem zu wiederholenden Code. Im Gegensatz zur bedingten Anweisung wird der Code aber nicht nur einmal ausgeführt. Vielmehr wird nach der Ausführung des Codes die Bedingung erneut ausgewertet und der Code gegebenenfalls noch einmal ausgeführt. Unter den verschiedenen Varianten von Schleifen trifft man die `while`-Schleife am häufigsten an. Weitere Ausprägungen sind die `repeat-` und `for`-Schleife. Grundsätzlich kann aber jede Schleife auf die einfache Form einer `while`-Schleife zurückgeführt werden.

Die Modellierung von Schleifen in einem Abhängigkeitsgraphen erfolgt ähnlich der Modellierung von bedingten Anweisungen. Die Bedingung wird als Aufruf einer lokalen (oder Hilfs-) Funktion modelliert und durch eine iterative Abhängigkeitsrelation (in den folgenden Beispielen durch einen Doppelpfeil symbolisiert) mit dem Code der Schleife verbunden. Dieser Code kann aus einem oder mehreren Funktionsaufrufen bestehen. Abbildung 4.13 verdeutlicht den Aufbau einer einfachen Schleife.

Eine iterative Abhängigkeit ist zunächst inaktiv. Führt die Ausführung der Funktion `condition()` aus Abbildung 4.13 links zu keiner Fehlermeldung, wird die Ausführung des Abhängigkeitsgraphen mit „weitere Ausführung“ fortgesetzt (Abbildung 4.13 Mitte). Tritt aber eine Fehlermeldung auf, wird die iterative Abhängigkeit aktiv und die Funktion, von der sie ausgeht, wird zur neuen, temporären Zielfunktion (Abbildung 4.13 rechts).

Die Realisierung der Schleife erfolgt in diesem Beispiel durch die beiden spezialisierten Hilfsfunktionen `condition()` und `code()`. Diese Funktionen können an keiner anderen Stelle wieder verwendet werden. Abbildung 4.14 links zeigt die Modellierung einer

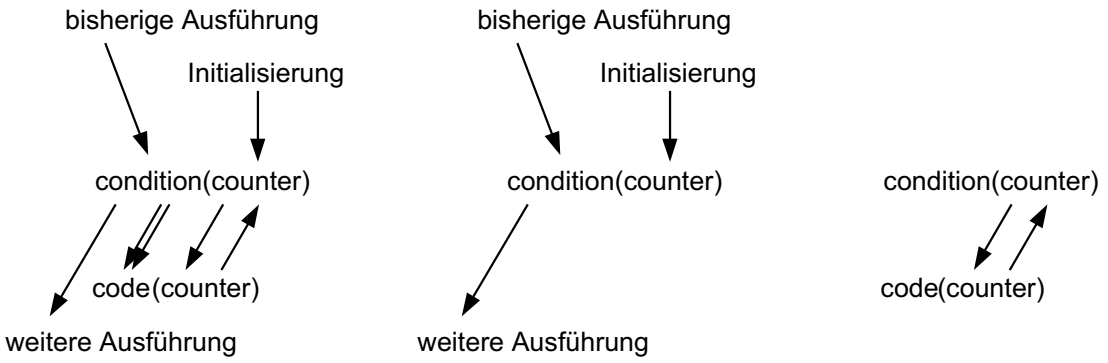

Abbildung 4.13: Einfache Schleife: Modellierung (links), Ausführung ohne Exception (Mitte), Ausführung nach Exception (rechts).

Schleife ohne die Verwendung spezialisierter Hilfsfunktionen. Auch in diesem Fall wird die iterative Abhängigkeitsrelation erst beim Auftreten einer Fehlermeldung aktiv und es ergibt sich der in Abbildung 4.14 rechts dargestellte neue Abhängigkeitsgraph mit der temporären Zielfunktion `eval()`.

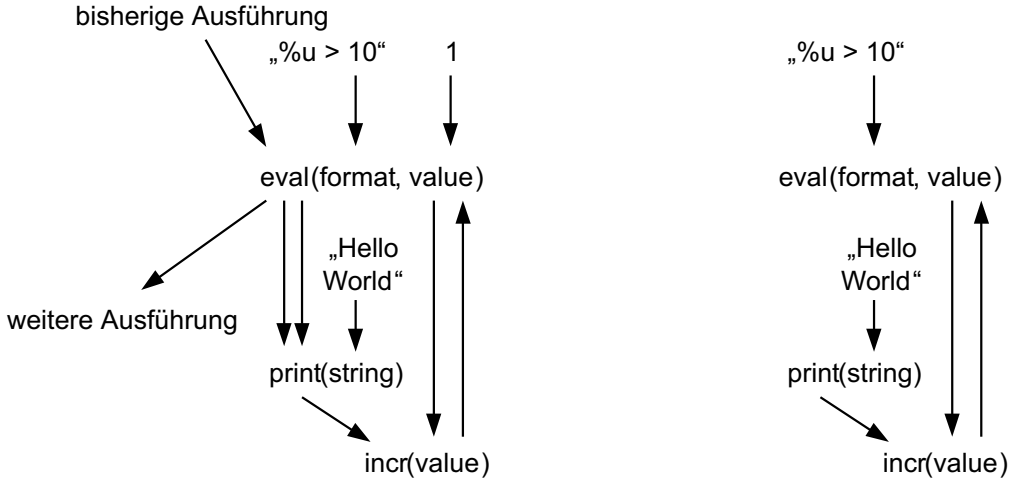

Abbildung 4.14: Einfache Schleife ohne spezialisierte lokale Funktionen: ihre Modellierung (links) und ihre Ausführung nach Exception (rechts).

Abbildung 4.14 macht deutlich, dass die Realisierung (auch einfacher) Schleifen durch Abhängigkeitsgraphen sehr aufwendig ist. Daher wurden die zwei in Abbildung 4.15 dargestellten Sonderfälle iterativer Abhängigkeiten definiert. Es werden eine Liste und ein einfacher Datentyp durch eine iterative Abhängigkeit verbunden. Dabei müssen die Datentypen der Listenelemente mit dem Datentyp des Parameters der anderen Funktion übereinstimmen. Der erste Sonderfall hat dabei die Bedeutung, dass `func2()` mit jedem Wert der Liste einmal aufgerufen wird. Im zweiten Fall wird `func1()` so oft ausgeführt, bis eine Fehlermeldung aufgetreten ist. Sämtliche zurückgegebenen Werte werden in einer Liste zusammengefasst und `func2()` als Parameter übergeben. Mit Hilfe dieser

Relation	Bedeutung
Funktion 1 ⇒ Funktion 2	Schlägt die Ausführung von Funktion 1 fehl, wird diese Abhängigkeit aktiv und Funktion 1 zur temporären Zielfunktion. Nach der Abarbeitung der Iteration wird Funktion 1 erneut ausgeführt.
Listenparameter ⇒ Parameter	Funktion des Zielparameters wird mit jedem Element der Liste einmal aufgerufen.
Parameter ⇒ Listenparameter	Ausgangsfunktion wird wiederholt aufgerufen, bis eine Fehlermeldung auftritt. Alle erhaltenen Werte werden als Liste der zweiten Funktion übergeben.

Tabelle 4.6: Iterative Abhängigkeiten.

beiden Konstrukte ist die Realisierung von vielen häufig auftretenden Problemen, wie beispielsweise die Berechnung des Durchschnitts der Werte einer Liste, einfach möglich.

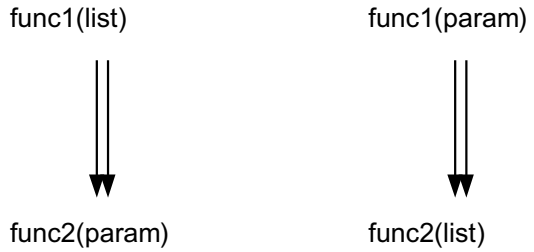

Abbildung 4.15: Iterative Sonderfälle.

Tabelle 4.6 fasst die im Rahmen des iterativen Falles möglichen Abhängigkeitsrelationen zusammen.

Redundante Ausführung

Manchmal können Werte von mehreren Funktionen ermittelt werden, da sie dieselbe Funktionalität implementieren. So ist die Realisierung der Funktion `GetName()` unter Verwendung der lokalen Funktion `GetPart()` sowohl von Werk 1 als auch von Werk 2 möglich. Werden im Rahmen einer föderierten Funktion beide lokale Funktionen aufgerufen, kann dadurch eine Kompensation des Ausfalls eines der beiden Systeme erfolgen. Außerdem wird die Ausführungsgeschwindigkeit optimiert, falls eines der beiden Quellsysteme im Augenblick der Ausführung überlastet ist. Eine mögliche Modellierung der Funktion `GetName()` illustriert Abbildung 4.16.

Hier können die beiden lokalen `GetPart()`-Funktionen gleichzeitig ausgeführt werden. Sobald eine der beiden zurückkehrt, kann der Rückgabewert der globalen Funktion ermittelt werden. Die Ausführung des jeweils anderen Teilgraphen wird (falls möglich) abgebrochen.

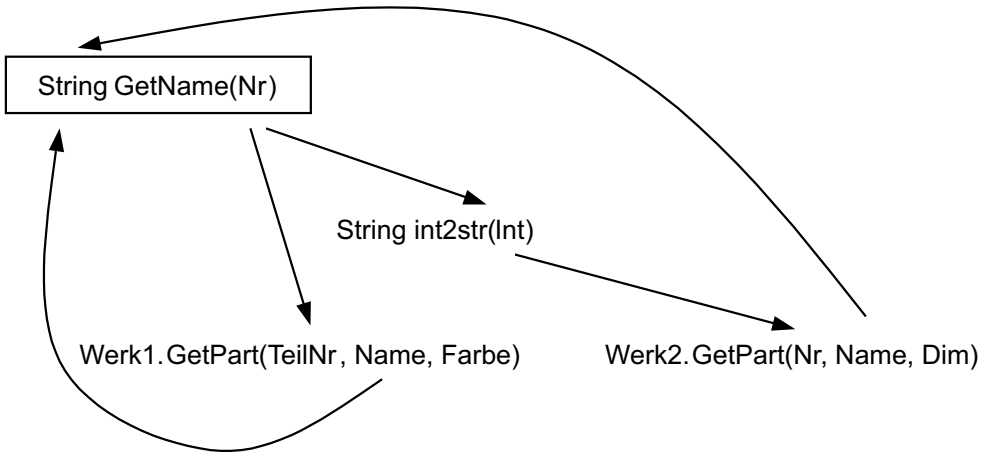

Abbildung 4.16: Redundante Ausführung der lokalen Funktionen.

Das gleiche Vorgehen ist auch bei einer Funktion möglich, die von mehreren Funktionen abhängig ist. Im Gegensatz zu den Parametern besteht aber in diesem Fall auch die Möglichkeit, dass eine Funktion erst aufgerufen werden soll, wenn alle Funktionen ausgeführt wurden, von denen sie abhängig ist. Abbildung 4.17 illustriert dies am Beispiel der Integration der lokalen `Init()`-Funktionen. In der Abbildung wird die zwingende Ausführung der beiden lokalen Funktionen durch einen Kreis um die Pfeile symbolisiert.

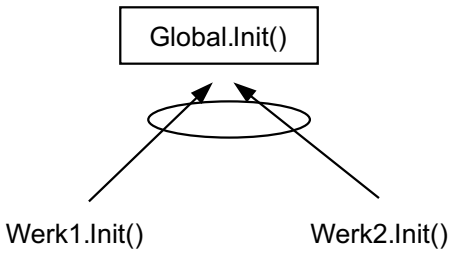

Abbildung 4.17: Der Kreis symbolisiert die erzwungene doppelte Ausführung der lokalen Funktionen.

Außerdem können Abhängigkeiten als optional markiert werden. Eine solche Abhängigkeit wird erst aktiv, falls jede andere, nicht zwingend erforderliche, zu demselben Ziel hinführende Abhängigkeit nicht erfüllt werden konnte. Abbildung 4.18 zeigt die Benutzung der (durch einen „runden“ Abschluss der Pfeilspitze dargestellten) optionalen Ausführung anhand der globalen Funktion `GetTeil()`. Führt der Lieferant ein Teil nicht, dann wird die lokale Funktion `GetPreis()` mit einer Fehlermeldung abgebrochen. Nur in diesem Fall tritt die optionale Abhängigkeit `0` → `Preis` in Kraft.

4.2.2.3 Kompensationen

Werden zur Modellierung einer föderierten Funktion lokale Funktionen verwendet, welche „Nebeneffekte“ verursachen, müssen im Fehlerfall vor Beendigung der föderierten Funktionen eventuell die an den lokalen Systemen vorgenommenen Zustandsänderungen

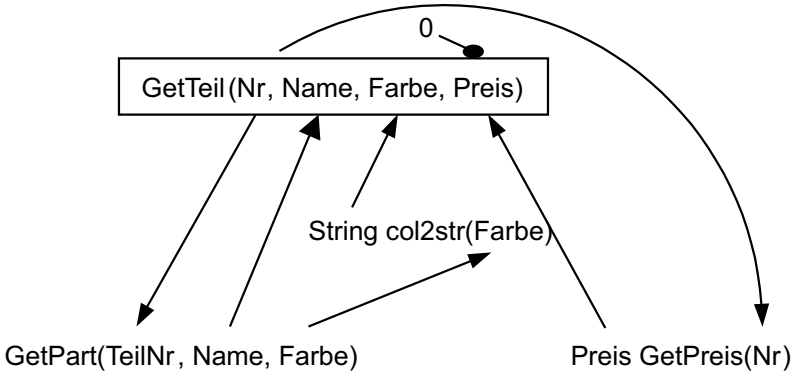

Abbildung 4.18: Optionale Ausführung.

wieder rückgängig gemacht werden. Dazu werden so genannte Kompensationsfunktionen eingeführt, welche die Effekte der ausgeführten lokalen Funktionen wieder rückgängig machen (siehe auch Kapitel 3). Welche oder wie viele dieser Kompensationsaktionen ausgeführt werden müssen, hängt von der Stelle ab, an der die normale Ausführung durch den Fehler unterbrochen wurde. Diese Kompensationsaktionen können jedoch nur modelliert werden, wenn das lokale System sie auch bereitstellt. Dies ist nicht immer der Fall.

Um Kompensationen in Abhängigkeit der bereits ausgeführten lokalen Funktionen aktivieren zu können, müssen diese in den Abhängigkeitsgraphen integriert werden. Innerhalb des Graphen werden Kompensationen ebenfalls durch ihre Parameter repräsentiert und stellen dadurch Knoten dar. Mit den Knoten der regulären lokalen Funktionsaufrufe werden sie mittels spezieller (Fehler-)Kanten verknüpft (in den folgenden Abbildungen immer durch gestrichelte Pfeile symbolisiert). Diese Fehlerkanten sind im Rahmen der normalen Ausführung der globalen Funktion inaktiv. Im Fall eines Fehlers werden alle Fehlerkanten aktiv, die von bereits ausgeführten Funktionen sowie der fehlgeschlagenen Funktion selbst ausgehen. Das Laufzeitsystem versucht nun wiederum alle noch bestehenden Abhängigkeiten aufzulösen.

Abbildung 4.19 zeigt die Modellierung der föderierten Funktion `AddTeil()`, die auf die Funktion `AddPart()` von Werk 2 mit der zugehörigen Kompensation abgebildet wird. Die in der Darstellung gestrichelte Abhängigkeit `Werk2.AddPart()` → `Werk2.DelPart()` bleibt während der normalen Ausführung des Abhängigkeitsgraphen inaktiv. Schlägt die Ausführung der Hilfsfunktion `str2int()` fehl, kann die Abhängigkeit `TeilNr` → `String` nicht mehr erfüllt werden. Stattdessen wird die Abhängigkeit `Werk2.AddPart()` → `Werk2.DelPart()` aktiv und die Ausführung kann über `Werk2.DelPart()` → `0` und `0` → `Long` abgeschlossen werden.

4.2.2.4 Aktualisierungsprobleme

DBMS bieten so genannte Views zur Definition von Sichten auf Tabellen. Während die Abbildung der originalen Tabellen auf die Sicht ohne Probleme möglich ist, bereitet die umgekehrte Richtung u. U. Schwierigkeiten, falls Änderungsoperationen auf der Sicht nicht eindeutig Änderungsoperationen auf den zugrunde liegenden Tabellen zugeordnet

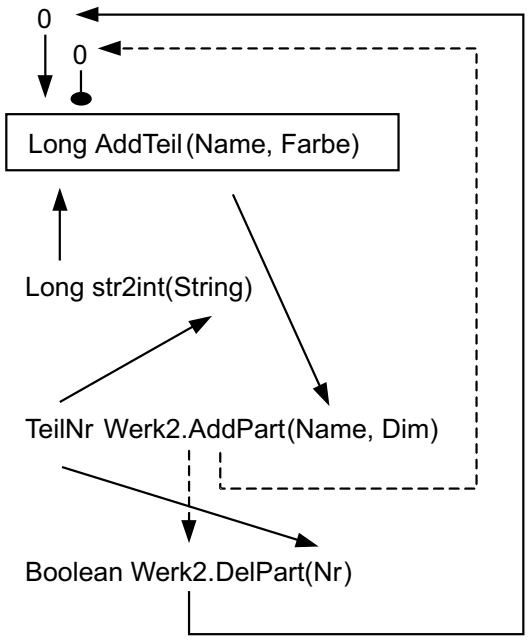

Abbildung 4.19: Modellierung der Kompensation `DelPart()` der Funktion `AddPart()`.

werden können. Man spricht dabei von dem Aktualisierungsproblem in Sichten (*view-update problem*).

View-Update-Probleme treten innerhalb relationaler DBMS beispielsweise bei der Projektion von Schlüsselattributen oder bei der Projektion des Verbundattributs bei einer Verbundoperation auf. Außerdem werden View-Update-Probleme durch in der Sicht berechnete Attribute verursacht. Im Rahmen der Funktionsintegration können ähnliche Probleme auftreten. Im Gegensatz zu den DBMS, welche die Abbildung von Sichten auf die zugrunde liegenden Tabellen automatisch aufgrund allgemeiner Regeln vornehmen müssen (und aus diesem Grund u. U. Schreiboperationen auf bestimmte Sichten nicht erlauben), können Aktualisierungsprobleme innerhalb der Funktionsintegration durch spezielle Abhängigkeitsgraphen für jeden Fall einzeln gelöst werden.

Fehlende Attribute

Werden innerhalb der föderierten Funktionen nicht alle Argumente der zu integrierenden lokalen Funktionen verwendet, kommt es bei der Aktualisierung bestehender Datensätze zu dem Problem, dass nicht alle notwendigen Daten von den globalen Funktionen zur Verfügung gestellt werden. Es müssen also durch eine vorangestellte Leseoperation die restlichen Daten aus dem bisherigen Datensatz ausgelesen werden. Abbildung 4.20 zeigt dies anhand der globalen Funktion `UpdTeil()`.

Informationsverlust

Ein Informationsverlust tritt auf, wenn keine bijektive Abbildung zwischen den entsprechenden Parametern der lokalen und föderierten Funktionen besteht. Dies kann

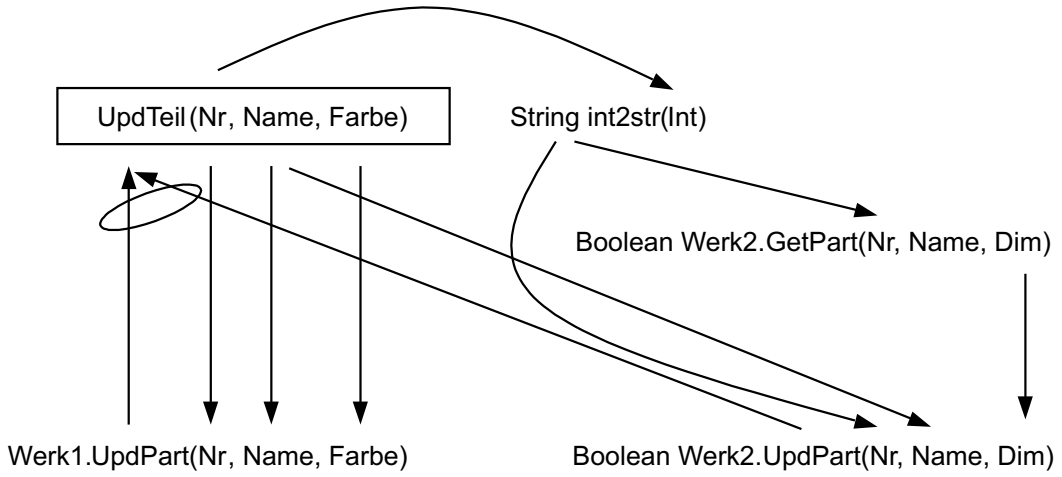

Abbildung 4.20: Ermittlung fehlender Werte durch vorangestellte Leseoperation.

z. B. auftreten, wenn zur Darstellung eines Werts in der lokalen Funktion ein Fließkommadatentyp verwendet wird, während der Wert innerhalb der föderierten Funktion durch einen Integer-Datentyp dargestellt wird. Außerdem können Aktualisierungsprobleme durch Informationsverlust bei berechneten Werten innerhalb der Zielfunktion auftreten.

Um dies zu verhindern, muss innerhalb eines Abhängigkeitsgraphen zunächst geprüft werden, ob sich die beiden Werte unterscheiden. Nur dann wird der Wert im lokalen System aktualisiert.

4.2.2.5 Verschiedenes

In diesem Abschnitt beschreiben wir zwei Aspekte, die keine weiteren Modellierungsaspekte für die Abbildungsbeschreibung einbringen, sondern Vereinfachungen für die Umsetzung an sich versprechen. Es handelt sich dabei um die Einführung einer Standardbibliothek mit häufig benötigten Hilfsfunktionen und der Verwendung von semantischen Attributen.

Standardbibliothek

Einige Typen von Hilfsfunktionen werden häufig zur Beschreibung der Funktionsintegration genutzt. Dies sind neben den oft benötigten Funktionen zur Konvertierung zwischen den verschiedenen Datentypen auch die für bedingte Anweisungen bzw. Schleifen benötigten Bedingungsfunktionen wie beispielsweise `IsEqual()`. Da diese Funktionen zumeist nicht Teil eines lokalen Systems sind, müssen sie durch ein Hilfssystem bereitgestellt werden. Damit diese Funktionen nicht bei jeder Integration neu und womöglich inkompatibel zueinander erstellt werden, soll eine Art Standardbibliothek definiert werden. Dies verringert den Aufwand der Integration und verbessert die Portabilität und Austauschbarkeit der erstellten Graphen. Folgende Kategorien von Funktionen sollten in einer Standardbibliothek vorhanden sein[3]:

[3] Abschnitt 4.3.13 enthält eine ausführlichere Beschreibung der hier aufgeführten Funktionen.

- *Bedingungsfunktionen:*
  Bedingungsfunktionen werden sehr häufig benötigt, da sie sowohl zur Realisierung von bedingten Anweisungen als auch von Schleifen genutzt werden. Sie müssen das in Abschnitt 4.2.2.2 beschriebene Verhalten aufweisen. Neben Funktionen in der Art des bereits in den Beispielen benutzten `IsEqual()` sollte eine allgemeine Bedingungsfunktion enthalten sein, welcher die gewünschte Bedingung als Argument übergeben werden kann.

- *Konvertierungsfunktionen:*
  Da das verwendete Modell statt eines Mechanismus zur Konvertierung von Datentypen Hilfsfunktionen vorsieht, sollten innerhalb einer Standardbibliothek grundlegende Konvertierungsfunktionen wie z. B. `str2int()` definiert sein.

- *Zeichenketten / Listen:*
  Im Rahmen der Funktionsintegration werden Operationen wie beispielsweise die Konkatenation von Zeichenketten sehr häufig benötigt. Ebenfalls ist ein Zugriff auf Felder oder Listen erforderlich. Daher sollte in der Standardbibliothek ein vollständiger Satz an Funktionen zur Bearbeitung von Zeichenketten und Listen vorhanden sein.

- *Gängige Berechnungen:*
  Die Aufnahme von Funktionen in der Art von `incr()` trägt ebenfalls zu einer Verringerung der Komplexität der entstehenden Abhängigkeitsgraphen und zur besseren Portabilität bei.

Semantische Attribute

Semantische Attribute kategorisieren sowohl lokale als auch globale Funktionen oder Attribute in der Form, dass ein grafisches Werkzeug Abhängigkeitsgraphen effizient darstellen und den Benutzer bei der Integration unterstützen kann. In der bisherigen Beschreibung der Funktionsintegration durch Abhängigkeitsgraphen wurde bewusst darauf geachtet, möglichst wenig verschiedene Konstrukte zu verwenden. So wurden Konvertierungen durch zusätzliche Hilfsfunktionen realisiert. Ebenso wurden die Bedingungen von bedingten Anweisungen und Schleifen als Funktionen realisiert. Dadurch kann eine entsprechende Beschreibungssprache relativ einfach gehalten werden. Dies führt aber u. U. auch zu sehr unübersichtlichen Abhängigkeitsgraphen mit einer großen Anzahl von Konvertierungs- bzw. Bedingungsfunktionen im Vergleich zu den verwendeten Quellsystemfunktionen. Werden Konvertierungs- und Bedingungsfunktionen als solche markiert, kann ein grafisches Werkzeug diese entsprechend darstellen oder – im Fall von Konvertierungsfunktionen – sie in bestimmten Ansichten ausblenden.

Eine weitere Möglichkeit der Kategorisierung bietet sich durch die Vergabe von semantischen Attributen, welche den Zweck einer Funktion genauer beschreiben. Beispiele dafür sind `read`, `add`, `update` oder `delete`. Sind die Funktionen der zu integrierenden lokalen Systeme derart gekennzeichnet, können Werkzeuge den Benutzer bei der Erstellung der Abhängigkeitsgraphen unterstützen, indem sie „passende“ lokale Funktionen zur behandelten föderierten Funktion heraussuchen. Passend bedeutet in diesem Zusammenhang, dass die Datentypen oder Namen von Parametern der lokalen Funktionen ganz oder

teilweise mit denen der föderierten Funktion übereinstimmen und dass die semantischen Attribute der Funktionen passen. Soll beispielsweise ein Abhängigkeitsgraph für die globale Funktion

```
void GetTeil(IN LONG Nr, OUT STRING Name, OUT STRING Farbe, OUT preis_t Preis);
```

(gekennzeichnet durch das semantische Attribut `read`) erstellt werden, so könnte ein grafisches Werkzeug die „passenden“ Funktionen

```
void GetTeil(IN teilnr_t TeilNr, OUT STRING Name, OUT farbe_t Farbe);
Preis GetPreis(IN LONG Nr);
```

des Systems von Werk 1 bzw. des Lieferanten heraussuchen. Zusätzlich würde auch die Konvertierungsfunktion

```
STRING col2str(IN farbe_t Farbe);
```

angezeigt werden.

Die Kategorisierung durch semantische Attribute ist nicht auf die hier vorgeschlagenen Werte `read`, `add`, `update` und `delete` beschränkt, sondern beliebig erweiterbar. Semantische Attribute können nur als Hilfestellung für den Benutzer bei der Erstellung eines Abhängigkeitsgraphen dienen. Eine vollständige Automatisierung durch semantische Attribute ist dabei nicht möglich.

4.3 Abbildungssprache auf Basis von XML

Im dritten Teil dieses Kapitels stellen wir die Beschreibungssprache FIX (Function Integration in XML) vor. FIX ist eine Sprache zur Modellierung der in den ersten beiden Teilen behandelten Konzepte. Folglich beschreiben wir mit FIX die Schnittstellen aller Funktionen und die Abbildungen der föderierten Funktionen auf die lokalen Funktionen mit Hilfe von Abhängigkeiten. Neben einer ausführlichen Syntaxbeschreibung enthält dieses Kapitel Beispiele zur Lösung der im vorangegangenen Kapitel beschriebenen Aspekte. Obwohl wir die Abbildungssprache ausführlich und mit vielen Beispielen erläutern, haben wir nicht zum Ziel, dem Leser die komplette Sprache in diesem Kapitel beizubringen. Zum vollen Verständnis sind XML-Vorkenntnisse notwendig, die wir in den folgenden Ausführungen voraussetzen. In [Wis00] findet sich eine sehr ausführliche Erläuterung aller Aspekte der Sprache FIX sowie deren Syntax.

4.3.1 Anforderungen an die Sprache

In den Abschnitten 4.1 und 4.2 erfolgte die Beschreibung der Abhängigkeiten durch Graphen. Dabei stellen Signaturen die Funktionen dar und Pfeile symbolisieren die Abhängigkeiten. Spezialfälle, wie z. B. der iterative Fall, werden durch weitere Symbole – in diesem Fall der Doppelpfeil – dargestellt. Diese visuelle Darstellung ist als formale und vor allem Computer-lesbare Beschreibung nicht geeignet. Hierfür bietet sich eine (Beschreibungs-)Sprache an.

Neben der Beschreibung der Abhängigkeiten muss mit der gewählten Beschreibungssprache die Definition der von den Systemen bereitgestellten Schnittstellen möglich

sein. Außerdem soll die Sprache auch zur Dokumentation der beschriebenen Systeme benutzt werden können. Die Schnittstellenbeschreibung enthält neben der Definition der bereitgestellten Funktionen zusätzlich die Definition der verwendeten Datentypen und Konstanten. Stellt ein System mehrere Schnittstellen zur Verfügung oder sind die verfügbaren Schnittstellen auf eine bestimmte Art und Weise (z.B. hierarchisch) strukturiert, müssen diese Strukturen ebenfalls in der Schnittstellenbeschreibung wiedergegeben werden. Benutzt ein zu beschreibendes System einen objektorientierten Ansatz, müssen dessen zusätzlichen Merkmale (etwa Vererbungsbeziehungen der exportierten Schnittstellen) ebenfalls wiedergegeben werden können.

Es existieren bereits mehrere Sprachen zur Schnittstellenbeschreibung. Von der im Rahmen des *Distributed Computing Environment* [Ope97] entwickelten *Interface Definition Language* (IDL) wurden inzwischen weitere Varianten, wie z. B. Microsoft COM IDL, sowie die von CORBA verwendete OMG IDL [OMG03] entwickelt. IDL benutzt eine an C++ angelehnte Syntax zur Beschreibung der Schnittstellen. Eine weitere Sprache zur Definition von Schnittstellen ist die an C angelehnte Sprache *eXternal Data Representation* [Sri95b], welche im Zusammenhang mit ONC RPC [Sri95a] benutzt wird. Da IDL unabhängig von Programmiersprachen ist, orientieren wir uns bei der Schnittstellenbeschreibung an dieser Sprache.

Die Systemdokumentation umfasst mehrere Teilaspekte. Zum einen müssen die zur Benutzung der exportierten Schnittstellen notwendigen technischen Daten definiert werden. Dazu gehören neben dem Namen bzw. der Netzwerkadresse des Systems das verwendete Kommunikationsprotokoll wie etwa ONC RPC oder IIOP und eventuell weitere protokollspezifische Details, z. B. die Port-Nummer bei einem auf TCP/IP basierenden Protokoll. Des Weiteren sollen die im Schnittstellenbeschreibungsteil vorgenommenen Funktions- bzw. Typdefinitionen dokumentiert werden können. Ebenso sollte man Anmerkungen zu den Abhängigkeitsbeschreibungen hinzufügen können.

4.3.2 Die Sprache FIX

Als Grundlage der Beschreibungssprache wählen wir XML [BPSM98]. XML stellt eine standardisierte Syntax zur Definition der Darstellung von Informationen zur Verfügung. Es ist plattformunabhängig und kann von Mensch und Maschine gelesen und manipuliert werden. XML bietet vor allem den Vorteil, dass – bedingt durch die Standardisierung und weite Verbreitung – eine große Anzahl an Software zur Verarbeitung von XML-Dokumenten existiert. Neben den für viele Systeme bzw. Programmiersprachen vorhandenen Parsern sind Editoren oder Tools zur Umwandlung von XML-Dokumenten in andere Formate erhältlich (siehe Abschnitt 4.3.14). Überdies führt die große Akzeptanz bei den Herstellern zu einer wachsenden Zahl an Produkten mit XML-Schnittstellen. Darüber hinaus sind immer mehr verwandte Standards verfügbar, welche die Einsatzfähigkeit von XML-Dokumenten verbessern und deren Verarbeitung zunehmend erleichtern.

FIX ist eine Anwendung des Standards XML 1.0. Folglich muss jedes FIX-Dokument wohlgeformt sein. FIX-Dokumente sollten gültig gegenüber der in Anhang A aufgeführten DTD bzw. bei Verwendung eines auf XML Schema [TBMM01] basierenden Parsers schemagültig gegenüber dem aufgeführten XML Schema sein.

Wir empfehlen darüber hinaus die Reservierung eines Media-Type bei der Internet Assigned Numbers Authority (IANA) für FIX-Dokumente. In Übereinstimmung mit [MLK00] wird dafür der Name `application/fix+xml` vorgeschlagen.

4.3.3 Namensräume

FIX definiert den Namensraum [BHL99] `http://dcx.com/fix`. Mit diesem Namensraum kann man u. a. die in FIX definierten Werte des `semantic`-Attributs (siehe Abschnitt 4.3.11) von denen anderer Namensräume unterscheiden. Außerdem wird dadurch beispielsweise die schemagültige Einbettung von FIX-Dokumenten in andere XML-Dokumente ermöglicht. In FIX-Dokumenten werden Attribute aus dem XLink-Namensraum verwendet. Um FIX-Dokumente gegenüber der verwendeten DTD validieren zu können, muss die Deklaration der Namensräume immer in der in Abschnitt 4.3.8.1 dargestellten Weise erfolgen.

Sollen Elemente anderer Namensräume innerhalb der Elemente `<desc>` bzw. `<expr>` oder zusätzliche Werte für das `semantic`-Attribut verwendet werden (siehe auch Abschnitt 4.3.11), müssen zusätzliche Namensräume deklariert werden. Im ersten Fall sollte dies innerhalb des ersten Elements des zusätzlich verwendeten Namensraums erfolgen, damit keine Probleme im Zusammenhang mit der Gültigkeit des Dokuments auftreten können. Im zweiten Fall hingegen ist dies nicht auf eine Art und Weise möglich, welche die Gültigkeit gegenüber dem hier verwendeten DTD erhält. Daher muss das FIX-Dokument eine interne DTD-Teilmenge mit einer zusätzlichen `ATTLIST`-Deklaration enthalten (siehe Abbildung 4.21).

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!DOCTYPE system SYSTEM "http://dcx.com/fix/fix.dtd" [
   <!ATTLIST system
              xmlns:myns CDATA #FIXED "http://example.com/myns">
]>
<system xmlns="http://dcx.com/fix" xmlns:fix="http://dcx.com/fix"
 xmlns:xlink="http://www.w3.org/1999/xlink"
 xmlns:myns="http://example.com/myns">
    ...
    <op semantic="myns:special">
    ...
    </op>
```

Abbildung 4.21: Definition zusätzlicher Namensräume.

4.3.4 XLink

FIX verwendet die XML Linking Language (XLink [DMO01], vgl. auch Abschnitt 3.4.3) zur Verknüpfung von verschiedenen Ressourcen. XLink definiert Attribute, mit denen

Verknüpfungen zwischen XML-Dokumenten realisiert werden können. Durch die Verwendung eines eigenen Namensraumes kann man die durch XLink spezifizierten Attribute beliebigen Elementen zuweisen.

XLink unterscheidet zwischen zwei Arten von Verknüpfungen. *Simple Links* dienen der Verknüpfung einer lokalen und einer entfernten Ressource. Die Verknüpfungsrichtung verläuft dabei immer von der lokalen zur entfernten Ressource. Simple Links ähneln damit dem aus HTML [RLJ99] bekannten Element `<a>`. In dem XML-Fragment in Abbildung 4.22 wird z.B. die lokale Ressource „Beispiel“ mit dem HTML-Dokument `verknuepft.html` verknüpft.

```
<a xmlns:xlink="http://www.w3.org/1999/xlink"
 xlink:type="simple" xlink:href="verknuepft.html">Beispiel</a>
```

Abbildung 4.22: Verknüpfung mit dem XLink Simple Link.

Mit *Extended Links* können mehrere, sowohl entfernte als auch lokale Ressourcen verknüpft werden. Somit können auch so genannte „out-of-line“-Links realisiert werden. Dies sind Verknüpfungen, an denen nur entfernte Ressourcen beteiligt sind. Extended Links erlauben weiterhin die Angabe, in welcher Richtung die Verknüpfung erfolgen soll. Abbildung 4.23 zeigt einen Extended Link, der drei Ressourcen (eine lokale Ressource und zwei HTML-Dokumente) miteinander verknüpft, wobei nur eine Verknüpfung von der lokalen Ressource bzw. dem Dokument `a.html` in Richtung `b.html` erfolgt, und nicht umgekehrt.

```
<ext xmlns:xlink="http://www.w3.org/1999/xlink xlink:type="extended">
   <remote xlink:type="locator" xlink:label="a" xlink:href="a.html"/>
   <remote xlink:type="locator" xlink:label="b" xlink:href="b.html"/>
   <local xlink:type="resource" xlink:role="c">
      lokal
   </local>
   <link xlink:type="arc" xlink:from="c" xlink:to="b"/>
   <link xlink:type="arc" xlink:from="a" xlink:to="b"/>
</ext>
```

Abbildung 4.23: Verknüpfungen mit dem XLink Extended Link.

Zum Verweis auf entfernte Ressourcen benutzt XLink die auf der XML Path Language (XPath) [CD99] basierende XML Pointer Language (XPointer) [DDJM99]. Mit XPointer können nicht nur komplette XML-Dokumente referenziert werden, sondern es kann auch auf einzelne Teile eines XML-Dokuments (bis hin zu einzelnen Zeichenfolgen) verwiesen werden. So zeigt das Element `<see>` aus Abbildung 4.24 auf das erste `<para>`-Element,

dessen Vater-Element die ID `intro` trägt. Neben der normalen XPointer-Syntax verwendet FIX zur Referenzierung von Teilen von Parametern (z.B. Elemente einer Struktur) eine erweiterte Syntax (siehe Abschnitt 4.3.12).

```
<see xmlns:xlink="http://www.w3.org/1999/xlink"
   xlink:type="simple"
   xlink:href="http://example.com/document.xml#xpointer(id("intro")/child::
 para[1]"/>
```

Abbildung 4.24: Der Gebrauch von XPointer.

Innerhalb von FIX werden XLink Simple Links beispielsweise innerhalb des `<type>`-Elements benutzt, um Datentypen zu referenzieren, welche an anderer Stelle definiert wurden. Simple Links werden ebenfalls zur Modellierung von Vererbungsbeziehungen benutzt. Die Abhängigkeitsgraphen werden in FIX durch XLink Extended Links beschrieben.

4.3.5 Parameter Entities

Parameter Entities sind Konstrukte zum Textersatz innerhalb von DTDs ähnlich den `#define`-Anweisungen des C-Präprozessors. Der in Abbildung 4.25 dargestellte Ausschnitt aus einer DTD definiert ein Parameter Entity. Nach dieser Definition wird jedes Vorkommen des Ausdrucks `%id;` innerhalb der DTD durch den Ausdruck `id ID #IMPLIED` ersetzt.

```
<!ENTITY % id
         "id ID #IMPLIED">
```

Abbildung 4.25: Definition eines Parameter Entity.

In der verwendeten DTD wird häufig von Parameter Entities Gebrauch gemacht. Um Wiederholungen zu vermeiden und die DTD kleiner und dadurch übersichtlicher zu gestalten, werden oft verwendete Element-Inhalte oder Attribute durch Parameter Entities ersetzt. Des Weiteren wurden Datentypen und – in Anlehnung an die IDL-Definition – eine Reihe von Typklassen als Parameter Entities definiert.

DTDs bieten nur sehr eingeschränkte Möglichkeiten zur Definition von Datentypen. Aus diesem Grund wurden innerhalb der hier benutzten DTD einige Entities zur Beschreibung der verschiedenen Datentypen definiert. Um eine einfache Umstellung auf XML Schema zu ermöglichen, wurden – soweit möglich – die Datentypen aus [BM01] benutzt (Abbildung 4.26).

```
<!ENTITY % string  "CDATA">
<!ENTITY % boolean  "(true | 1 | false | 0)">
<!ENTITY % uri  "CDATA">
<!ENTITY % language  "NMTOKEN">
<!ENTITY % Name  "NMTOKEN">
<!ENTITY % QName  "NMTOKEN">
<!ENTITY % NCName  "NMTOKEN">
<!ENTITY % integer  "CDATA">
<!ENTITY % non-negative-integer "CDATA">
<!ENTITY % positive-integer  "CDATA">
<!ENTITY % contenttype  "CDATA">
<!ENTITY % semantic  "NMTOKEN">
<!ENTITY % coords  "CDATA">
```

Abbildung 4.26: Parameter Entities für Datentypen.

Außerdem werden die in Abbildung 4.27 dargestellten Parameter Entities zur Beschreibung bestimmter Typklassen definiert. Diese werden analog zu den angegebenen Produktionen der IDL-Grammatik [OMG03] definiert.

4.3.6 Aufbau der Sprache

Neben den an verschiedenen Stellen benutzten allgemeinen Elementen sind FIX-Dokumente in drei Hauptteile gegliedert. Der erste Teil enthält die Systembeschreibung. Sie beinhaltet beispielsweise die Adresse und das verwendete Kommunikationssystem sowie weitergehende Dokumentation. Anschließend folgt die Definition der exportierten Schnittstellen. Diese orientiert sich stark an dem Ausdrucksvermögen von OMG IDL. An die Definition der Schnittstellen schließt sich der dritte, optionale Teil an, der die Abhängigkeitsbeschreibung erläutert.

In den folgenden Abschnitten beschreiben wir die einzelnen Elemente durch einen Auszug aus der DTD und einer Beschreibung der Verwendung des Elements und seiner Attribute. Obwohl die Verwendung von XML Schema eine genauere Beschreibung der verwendeten Elemente erlaubt, wurde auf DTD-Ausschnitte zurückgegriffen, da sie eine kompaktere und übersichtlichere Darstellung erlauben.

Anhand von Beispielen verdeutlichen wir die Verwendung der beschriebenen Elemente. Die Beispiele orientieren sich zumeist an den im Abschnitt 4.2 gezeigten Abhängigkeitsgraphen. In den Beispielen werden die Werte des `id`-Attributs zumeist gleich dem Namen des zugehörigen Elements gewählt; dies ist aber nicht zwingend notwendig (siehe dazu auch Abschnitt 4.3.9.2).

Nach der Beschreibung der in FIX verwendeten Elemente folgen Abschnitte über semantische Attribute, die erweiterte FIXPointer-Syntax sowie die Beschreibung von Konvertierungsmöglichkeiten in andere Formate durch XSLT Stylesheets.

```
<!-- IDL Grammatik Produktion 46 -->
<!ENTITY % base "(float | int | char | bool | octet |
                  any | object | valuebase)">

<!-- IDL Grammatik Produktion 47 -->
<!ENTITY % template "(sequence | string | fixed)">

<!-- IDL Grammatik Produktion 45 -->
<!ENTITY % simple "(%base; | %template; | type)">

<!-- IDL Grammatik Produktion 48 -->
<!ENTITY % constr "(struct | union | enum)">

<!-- IDL Grammatik Produktion 44 -->
<!ENTITY % datatype "(%simple; | %constr;)">

<!-- IDL Grammatik Produktion 95 -->
<!ENTITY % param "(%base; | string | type)">

<!-- IDL Grammatik Produktion 42 -->
<!ENTITY % type_dcl "(typedef | struct | union | enum | native)">

<!-- IDL Grammatik Produktion 2 -->
<!ENTITY % definition "(%type_dcl; | const | except |
                        interface | module | valuetype)">

<!-- IDL Grammatik Produktion 9 -->
<!ENTITY % export "(%type_dcl; | const | except | attr | op)">
```

Abbildung 4.27: Parameter Entities für Typklassen.

4.3.7 Allgemeine Elemente

In diesem Abschnitt erläutern wir Elemente, die an mehreren Stellen einer FIX-Datei verwendet und somit als allgemeine Elemente eingestuft werden. Wir definieren die Elemente `<title>`, `<desc>` und `<a>` als allgemeine Elemente. `<title>` und `<desc>` beschreiben das umgebende Vaterelement näher. Innerhalb des Elements `<desc>` können durch das `<a>`-Element Verknüpfungen zu anderen Dokumenten erfolgen.

4.3.7.1 `<title>`

Das Element `<title>` legt den Namen des Vaterelements fest. Dieser Name wird zur Darstellung des jeweiligen Objekts am Bildschirm benutzt. Außerdem wird der Inhalt des `<title>`-Elements an einigen Stellen bei der Konvertierung in eine IDL-Datei verwendet. Daher gelten für die benutzten Namen die IDL-Regeln zu „Scoped Names“ [OMG03].

Innerhalb des Elements `<system>` kann der Name frei gewählt werden, es muss sich nicht um den innerhalb des Elements `<address>` angegebenen Netzwerknamen handeln. Wir definieren das Element `<title>` wie folgt:

```
<!ELEMENT title (#PCDATA)>
<!ATTLIST title
          xlink:type (title) #FIXED "title">
```

Dabei ist das Attribut `xlink:type` auf den Wert `title` festgesetzt, um dieses Element als „XLink Title“-Element zu markieren. Dies ist notwendig, damit XLink-kompatible Software die innerhalb der Elemente `<graph>`, `<node>` oder `<dep>` auftretenden `<title>`-Elemente als Link-Titel interpretieren kann. Kommt das Element `<title>` an anderen Stellen vor, wird dieses Attribut ignoriert.

4.3.7.2 `<desc>`

Das Element `<desc>` dokumentiert das umgebende Vaterelement. Neben reinem Text kann es Verknüpfungen mit anderen Dokumenten (mittels `<a>`-Elementen) beinhalten. Mehrere `<desc>`-Elemente können hintereinander auftreten. Dies kann man zur mehrsprachigen Dokumentation mittels verschiedener `xml:lang`-Attribute verwenden.

Außerdem kann der innerhalb des Elements `<desc>` auftretende Text durch Elemente aus anderen Namensräumen ausgezeichnet werden (siehe Abbildung 4.28). Dies ist zwar durch das Schema aber nicht durch die DTD darstellbar[4]. Das Element `<desc>` ist in FIX folgendermaßen festgelegt:

```
<!ELEMENT desc (#PCDATA | a)*>
<!ATTLIST desc
          xml:lang %language; #IMPLIED>
```

Das Attribut `xml:lang` dient zur Identifikation der innerhalb des Elements `<desc>` verwendeten Sprache (siehe [BPSM98]).

4.3.7.3 `<a>`

Das Element `<a>` dient der Verknüpfung mit anderen Ressourcen. Die Verwendung erfolgt dabei analog zu [RLJ99]. Das Element `<a>` wurde als XLink Simple Link realisiert und ist wie folgt definiert:

```
<!ELEMENT a (#PCDATA)>
<!ATTLIST a
          xlink:type (simple) #FIXED "simple"
          xlink:href %uri; #REQUIRED>
```

Das Attribut `xlink:type` ist festgesetzt auf den Wert `simple`, um dieses Element als Simple Link zu markieren. `xlink:href` verweist bei XLink Simple Links und Locator-Elementen innerhalb von Extended Links auf die zu verknüpfende Ressource.

4.3.8 Systembeschreibung

Die Beschreibung eines Systems besteht aus dessen Namen, einer optionalen Beschreibung, seiner Adresse und dem Typ des verwendeten Kommunikationsprotokolls. Das Element `<system>` ist das Wurzelelement einer Systembeschreibung. Das `<address>`-Element spezifiziert die Netzwerkadresse. Das verwendete Kommunikationssystem wird durch das darauf folgende Element, z. B. `<corba>`, `<dcom>` oder `<rmi>`, festgelegt.

[4] Ein Elementinhalt von `ANY` erlaubt nur in der DTD tatsächlich definierte Elemente.

```
<system xmlns="http://dcx.com/fix" xmlns:fix="http://dcx.com/fix"
 xmlns:xlink="http://www.w3.org/1999/xlink">

   <title>Werk 1</title>
   <desc xml:lang="de">
      Schnittstelle von <a xlink:href="http://werk1.example.com">
      Werk 1</a>
   </desc>
   <desc xml:lang="en">
      Interface of <a xlink:href="http://werk1.example.com">
      plant 1 </a>
   </desc>

   ...

   <typedef id="TeileNr_t">
      <title>TeileNr_t</title>
      <desc>
         Bauteile werden über eine
         <html:b xmlns:html="http://www.w3.org/1999/xhtml">
          eindeutige</html:b>
         Teilenummer referenziert
      </desc>
      <int/>
   </typedef>

   ...

</system>
```

Abbildung 4.28: Verwendung der allgemeinen Elemente `<title>`, `<desc>` und `<a>`.

4.3.8.1 `<system>`

Das Element `<system>` ist das Wurzelelement einer Systembeschreibung. Es kann sowohl zur Beschreibung von lokalen als auch föderierten Systemen dienen. Neben den Elementen zur Dokumentation und zur Beschreibung von Netzwerkadresse und Kommunikationsprotokoll enthält es sämtliche Elemente zur Beschreibung der exportierten Schnittstellen.

Um weitestgehende Kompatibilität mit validierenden und nicht-validierenden Parsern zu erreichen, sollten die drei Namensraumdefinitionen in jedem FIX-Dokument innerhalb des Elements `<system>` mit denselben Präfixen enthalten sein (siehe Abbildung 4.29). Dies garantiert sowohl eine Gültigkeit gegenüber der verwendeten DTD als auch eine Interoperabilität mit nicht-validierenden Parsern. Zusätzliche Namensraumdefinitionen können, wie in Abschnitt 4.3.3 beschrieben, integriert werden. Die Definition von `<system>` gestaltet sich folgendermaßen:

```
<!ELEMENT system        (title, desc*, address?,
                         (dcom | iiop | local | oncrpc | rmi),
```

```
<system xmlns="http://dcx.com/fix" xmlns:fix="http://dcx.com/fix"
 xmlns:xlink="http://www.w3.org/1999/xlink">
   <title>Werk 2</title>
   <address>werk2.example.com</address>
   <iiop port="683"/>
   ...
</system>
```

Abbildung 4.29: Systembeschreibung.

```
                        (%definition;)*)>
<!ATTLIST system
          xmlns       %uri;  #FIXED "http://dcx.com/fix"
          xmlns:fix   %uri;  #FIXED "http://dcx.com/fix"
          xmlns:xlink %uri;  #FIXED "http://www.w3.org/1999/xlink">
```

Dabei setzt das Attribut `xmlns` den Default-Namensraum auf den FIX-Namensraum. `xmlns:fix` ordnet dem Präfix `fix:` ebenfalls den FIX-Namensraum zu, während das Attribut `xmlns:xlink` dem Präfix `xlink:` den XLink-Namensraum zuordnet.

4.3.8.2 `<address>`

Das Element `<address>` spezifiziert die Netzwerkadresse des Systems. Hierbei kann sowohl ein Host-Name als auch eine IP-Adresse angegeben werden.

```
<!ELEMENT address (#PCDATA)>
```

4.3.8.3 `<dcom>`, `<iiop>`, `<local>`, `<oncrpc>` und `<rmi>`

Das System kann seine Schnittstelle per DCOM, IIOP, ONCRPC oder RMI zur Verfügung stellen. Es kann auch lokal auf demselben Rechner liegen und benötigt daher kein spezielles Protokoll. Die Definition der Schnittstelle DCOM gestaltet sich wie folgt:

```
<!ELEMENT dcom EMPTY>
<!ATTLIST dcom
          uuid %string; #REQUIRED
          version %float;  #REQUIRED>
```

Das Attribut `uuid` steht für den Universal Unique Identifier der Schnittstelle. `version` enthält die Version der Schnittstelle.

Die Definition der anderen Protokolle ist sehr ähnlich. Wir werden sie daher in diesem Kapitel nicht weiter im Detail vorstellen, sondern verweisen den interessierten Leser auf Anhang A.

4.3.9 Schnittstellendefinition

Innerhalb der Schnittstellendefinition werden sowohl die verwendeten Datentypen als auch die exportierten Funktionen bzw. Attribute definiert. Es sind einfache und strukturierte Datentypen, Vorlagentypen, neu definierte Datentypen, Konstanten, Operationen und Attribute, Module und Interfaces sowie Value Types abgedeckt. Dieser Teil der Sprache wurde in Art und Umfang stark an OMG IDL angelehnt.

4.3.9.1 Einfache Datentypen

Zur Spezifikation von einfachen Datentypen stellt FIX die Elemente `<float>`, `<int>`, `<char>`, `<bool>`, `<octet>`, `<any>`, `<object>` und `<valuebase>` bereit. `<float>`, `<int>`, `<char>` und `<bool>` entsprechen den Datentypdefinitionen anderer Sprachen. Die verbleibenden Datentypen entsprechen den gleichnamigen Schlüsselworten von IDL.

Auch bei den einfachen Datentypen zeigen wir die Definition der entsprechenden Elemente in FIX nur für einen der aufgeführten Datentypen. Die Definitionen aller einfachen Datentypen kann im Anhang A nachgelesen werden.

<float>

`<float>` kennzeichnet Fließkommadatentypen. Es entspricht (durch die entsprechende Wahl des `type`-Attributs) den IDL-Typen `float`, `double` und `long double` und ist wie folgt definiert:

```
<!ELEMENT float EMPTY>
<!ATTLIST float
          type (float | double | long_double) "double">
```

Das Attribut `type` wird zur Auswahl des gewünschten Fließkommadatentyps benutzt. Mögliche Werte sind `float`, `double` und `long_double`.

4.3.9.2 Strukturierte Datentypen

Zur Beschreibung von strukturierten Datentypen stehen die Elemente `<struct>`, `<enum>` und `<union>` für Strukturen, Aufzählungen und Vereinigungen zur Verfügung. Auch diese Datentypen entsprechen den gleichnamigen Datentypen von IDL. Wir werden die Definition komplexer Datentypen an dem Element `<struct>` exemplarisch aufzeigen. Die Definition der verbleibenden Datentypen findet sich im Anhang A.

In FIX dürfen keine rekursiven Definitionen strukturierter Datentypen vorgenommen werden.

<struct>

Strukturen werden durch das Element `<struct>` beschrieben. Sie bestehen aus einer oder mehreren Komponenten, die durch das Element `<member>` beschrieben werden. Diese Komponenten können wiederum einfache oder komplexe Datentypen sein. `<struct>` ist wie folgt in FIX definiert:

```
<!ELEMENT struct (title, desc*, member+)>
<!ATTLIST struct
          id ID #IMPLIED>
```

Das Attribut `id` ist eine gültige XML ID. Über diese ID kann die Struktur von anderen Elementen referenziert werden (z.B. `<type>`). In den im Folgenden verwendeten Beispielen werden zumeist die Werte der ID-Attribute gleich dem Namen des zugehörigen Elements (Datentyp, Operation etc.) gewählt. Dies ist nicht zwingend erforderlich, sondern soll nur dem besseren Verständnis der Beispiele dienen. Für ID-Attribute bestehen keine über [BPSM98] hinausgehende Anforderungen.

<member>

`<member>` definiert die Elemente von Strukturen. Ein innerhalb eines `<member>`-Elements vorkommendes Element `<type>` darf nicht die übergeordnete Struktur referenzieren. Wir definieren `<member>` folgendermaßen:

```
<!ELEMENT member (title, desc*, %datatype;, array*)>
<!ATTLIST member
          id ID #IMPLIED>
```

Das Attribut `id` entspricht dem gleichnamigen Attribut in `<struct>`.

Das Beispiel in Abbildung 4.30 enthält die Definition einer Struktur `dimension_t` bestehend aus den drei Elementen `x`, `y` und `z`.

```
<typedef id="dimension_t">
   <title>dimension_t</title>
   <struct id="dimension">
      <title>dimension</title>
      <member id="dimension.x">
         <title>x</title>
         <float type="float"/>
      </member>
      <member id="dimension.y">
         <title>y</title>
         <float type="float"/>
      </member>
      <member id="dimension.z">
         <title>z</title>
         <float type="float"/>
      </member>
   </struct>
</typedef>
```

Abbildung 4.30: Definition einer Struktur.

4.3.9.3 Vorlagentypen

FIX stellt ebenso wie IDL drei so genannte Vorlagentypen (Template Types) zur Verfügung. Diese können zur Definition von Zeichenketten-, Listen- und Festkommadatentypen benutzt werden.

<sequence>

Das Element `<sequence>` beschreibt ein eindimensionales Feld variabler Länge. Neben dem Typ der Elemente kann deren Obergrenze angegeben werden. Das Element `<sequence>` definieren wir folgendermaßen:

```
<!ELEMENT sequence (float | int | char | bool | octet | any | object |
                    valuebase | sequence | string | fixed | type)>
<!ATTLIST sequence
          count %positive-integer; #IMPLIED>
```

Die Angabe des Attributs `count` ist optional. Mit `count` legt man eine Obergrenze der in dem jeweiligen Feldelement enthaltenen Elemente fest. Wird keine Obergrenze aufgeführt, ist die Länge der Sequenz unbeschränkt.

Abbildung 4.31 zeigt die Definition einer Sequenz.

```
<typedef id="Teileliste">
   <title>Teileliste</title>
   <sequence>
   <int/>
   </sequence>
</typedef>
```

Abbildung 4.31: Definition einer Sequenz.

<string>

Das Element `<string>` dient zur Vereinbarung von Zeichenketten und entspricht dabei einer Sequenz vom Typ `<char>`. Das Element `<string>` fasst die IDL-Typen `string` und `wstring` zusammen und ist in FIX wie folgt festgelegt:

```
<!ELEMENT string EMPTY>
<!ATTLIST string
          count %positive-integer; #IMPLIED
          wide  %boolean; "false">
```

Das Attribut `count` entspricht dem gleichnamigen Attribut des Elements `<sequence>` und ist ebenfalls optional. Das Attribut `wide` dient zur Vereinbarung von IDL `wide char`- bzw. `wide string`-Typen. Ist das Attribut `wide` auf `true` gesetzt, wird die `wide`-Variante verwendet.

In Abbildung 4.32 erfolgt die Definition eines Strings mit der maximalen Länge von sechs Zeichen.

```
<typedef id="TeileNr">
   <title>TeileNr</title>
   <string count="6"/>
</typedef>
```

Abbildung 4.32: Definition einer Zeichenkette.

<fixed>

Das Element `<fixed>` beschreibt einen Festkommatyp analog zum Datentyp `fixed` von IDL. Tritt ein Element `<fixed>` innerhalb eines Elements `<const>` auf, dürfen die Attribute `digits` bzw. `scale` nicht angegeben werden. In allen anderen Fällen müssen beiden Attributen Werte zugewiesen werden. `<fixed>` definieren wir als:

```
<!ELEMENT fixed EMPTY>
<!ATTLIST fixed
          digits %positive-integer; #IMPLIED
          scale  %positive-integer; #IMPLIED>
```

Das Attribut `digits` spezifiziert die Gesamtzahl der Ziffern. Das Attribut `scale` gibt dabei die Anzahl der Nachkommastellen an.

Abbildung 4.33 illustriert die Definition eines Festkommadatentyps.

```
<typedef id="Preis_t">
   <title>Preis_t</title>
   <fixed digits="7" scale="2"/>
</typedef>
```

Abbildung 4.33: Definition eines Festkommatyps.

4.3.9.4 Definition neuer Datentypen

FIX unterstützt alle aus OMG IDL bekannten einfachen und strukturierten Datentypen. Die Definition von neuen Datentypen erfolgt durch das Element `<typedef>`. Sie enthält, neben dem Namen und der Beschreibung des Datentyps, die Spezifikation eines einfachen oder komplexen Datentyps sowie einen oder mehrere Array-Deklaratoren.

Neu definierte Datentypen können wiederum bei der Definition weiterer Datentypen verwendet werden. Außerdem können sie bei der Definition von Operationen oder Attributen als Datentypen benutzt werden. Die Referenzierung neu definierter Datentypen erfolgt hierbei durch das Element `<type>`.

<typedef>

<typedef> entspricht weitestgehend dem IDL-Schlüsselwort typedef, mit der Ausnahme, dass Konstrukte in der Art von

```
typedef int stunden, minuten, sekunden;
```

nicht möglich sind. Diese müssen stattdessen in einzelne <typedef>-Konstrukte aufgelöst werden. Somit ist <typedef> wie folgt in FIX definiert:

```
<!ELEMENT typedef (title, desc*, (title, desc*, %datatype;, array*)>
<!ATTLIST typedef
          id ID #IMPLIED>
```

Die Definition des Attributs id entspricht der Definition in Abschnitt 4.3.9.2.

Abbildung 4.34 verdeutlicht die Verwendung des Elements <typedef>.

```
<typedef id="TeileNr_t">
   <title>TeileNr_t</title>
   <desc>
      Bauteile werden über eine
      <html:b xmlns:html="http://www.w3.org/1999/xhtml">eindeutige</html:b>
      Teilenummer referenziert
   </desc>
   <int/>
</typedef>
```

Abbildung 4.34: Definition eines neuen Datentyps.

<array>

Mit dem Element <array> können Felder definiert werden. Die Definition von mehrdimensionalen Feldern ist durch das mehrfache Vorkommen von <array>-Elementen möglich (siehe auch Abbildung 4.35). Wir definieren <array> folgendermaßen:

```
<!ELEMENT array EMPTY>
<!ATTLIST array
          count %positive-integer; #REQUIRED>
```

Das Attribut count muss bei einem <array>-Element angegeben werden, bei den Elementen <sequence> bzw. <string> ist die Angabe optional. count gibt eine Obergrenze der in dem jeweiligen Feldelement enthaltenen Elemente vor. Wird keine Obergrenze angegeben, ist die Länge der Zeichenkette bzw. der Sequenz unbeschränkt.

```
<typedef> typedef TeileNr_t array_t[20][10];
   <title>array_t</title>
   <type xlink:href="#TeileNr_t"/>
   <array count="20"/>
   <array count="10"/>
</typedef>
```

Abbildung 4.35: Definition eines Feldes.

<type>

Das Element `<type>` dient zur Referenzierung bereits durch `<typedef>`, `<struct>`, `<union>` bzw. `<enum>` definierter Typen. Es ist wie folgt in FIX festgelegt:

```
<!ELEMENT type EMPTY>
<!ATTLIST type
          xlink:type simple #FIXED "simple"
          xlink:href %uri; #REQUIRED>
```

Die Definition der Attribute `xlink:type` und `xlink:href` entspricht der Definition in Abschnitt 4.3.7.3.

4.3.9.5 Konstanten

Die Definition von Konstanten erfolgt ähnlich zur Definition von Datentypen. Sie besteht aus dem Namen der Konstanten, deren Datentyp und einer optionalen Wertzuweisung. Zur Modellierung der Abhängigkeiten wird der Wert einer Konstante nicht benötigt. Soll aus einem FIX-Dokument aber beispielsweise ein IDL-Dokument erzeugt werden, ist eine Konvertierung von Konstantendefinitionen ohne Wertzuweisung nicht möglich.

Wird innerhalb einer Konstantendefinition der Typ durch ein `<type>`-Element spezifiziert, darf dieses nur auf Typdefinitionen verweisen, deren Basistyp `<long>`, `<char>`, `<bool>`, `<float>`, `<string>`, `<octet>` oder `<enum>` ist. Wird innerhalb eines Elements `<const>` ein `<fixed>`-Element verwendet, darf dieses nicht die Attribute `digits` oder `scale` enthalten.

<const>

`<const>` definiert Konstanten und entspricht dabei dem Konstrukt `const` in IDL[5]. Neben der Angabe des Namens und eines Typs kann dabei optional mit dem Element `<expr>` der Wert der Konstanten festgelegt werden. Das Element `<const>` ist in FIX wie folgt festgelegt:

```
<!ELEMENT const (title, desc*, (int | char | bool | float |
                 string | fixed | type | octet), expr?)>
<!ATTLIST const
          id ID #IMPLIED>
```

[5] Auch Aufzählungstypen sind als Konstanten möglich.

Die Definition des Attributs `id` entspricht der Definition des gleichnamigen Attributs von Element `<struct>` in Abschnitt 4.3.9.2.

<expr>

Mit Hilfe des Elements `<expr>` ist die Modellierung von konstanten Ausdrücken für die Definition von Konstanten bzw. Unions möglich. Das Element `<expr>` definieren wir wie folgt:

```
<!ELEMENT expr ANY)>
<!ATTLIST expr
          type      %contenttype; #REQUIRED
          xml:space (default | preserve) "default">
```

Das Attribut `type` enthält den MIME-Typ [FB96] des innerhalb des Elements `<expr>` verwendeten Formats. Mögliche Werte sind u. a. `text/x-idl`, `text/mathml`, `text/x-openmath`, `text/plain` oder `application/fix+xml`. Mit dem Attribut `xml:space` kann speziell für die Kodierung `text/plain` das Verhalten des XML-Parser bzgl. Leerzeichen beeinflusst werden [BPSM98].

Die Modellierung kann in verschiedenen Formaten erfolgen:

- *IDL:*
  Die Beschreibung des Konstantenwerts erfolgt mit der normalen IDL-Notation direkt innerhalb des Elements `<expr>`. Um den Inhalt von `<expr>` als IDL zu kennzeichnen, muss das Attribut `type="text/x-idl"` angegeben werden.

- *MathML:*
  MathML [CIMP01] ist ebenfalls eine XML-basierte Sprache, die ähnlich LATEX zur Darstellung mathematischer Ausdrücke entwickelt wurde. Im Gegensatz zu LATEX erlaubt MathML auch die semantische Repräsentation von Ausdrücken. MathML ist eine Recommendation des World Wide Web Consortium, für die bereits Parser existieren. Die Modellierung der Wertzuweisung durch ein MathML-Objekt erfolgt durch die Einbettung des MathML-Wurzelelements `<math>` innerhalb des `<expr>`-Elements. Zusätzlich wird die Verwendung von MathML durch das Attribut `type="text/mathml"` gekennzeichnet. Abbildung 4.36 illustriert die Definition einer numerischen Konstante.

- *OpenMath:*
  OpenMath [OEC99] ist ein Standard zur semantischen Repräsentation mathematischer Ausdrücke. Analog zu MathML wird ein OpenMath-Objekt durch das OpenMath-Wurzelelement `<omobj>` und das Attribut `type="text/x-openmath"` gekennzeichnet.

- *Text:*
  Zur Darstellung von Zeichenkettenkonstanten ist keine weitere Sprache notwendig. Dies wird durch das Attribut `type="text/plain"` gekennzeichnet. In diesem Fall hängt die Behandlung des Inhalts des Elements `<expr>` auch von dem Attribut `xml:space` ab. Das Beispiel in Abbildung 4.37 enthält die Definition einer Zeichenkettenkonstante.

```
<const>
   <title>Version</title>
   <int/>
   <expr type="text/mathml">
      <math xmlns="http://www.w3.org/1998/Math/MathML">
         <cn>42</cn>
      </math>
   </expr>
</const>
```

Abbildung 4.36: Definition einer numerischen Konstanten mit MathML.

- *FIX:*
  Konstanten für Aufzählungen können durch ein `<type>`-Element innerhalb des Elements `<expr>` identifiziert werden, welches auf das entsprechende `<enumerator>`-Element verweist. In diesem Fall muss das Attribut `type` des Elements `<expr>` auf den Wert `"application/fix+xml"` gesetzt werden.

```
<const id="Name">
   <title>Name</title>
   <string/>
   <expr type="text/plain" xml:space="preserve">Werk 1</expr>
</const>
```

Abbildung 4.37: Definition einer String-Konstante.

4.3.9.6 Operationen und Attribute

Zur Definition von Operationen (also Funktionen bzw. Methoden) wird das Element `<op>` verwendet. Es beinhaltet die Definition der Parameter und des Rückgabewerts und die der Funktion zugeordneten Abhängigkeitsgraphen. Die Definition der Parameter und des Rückgabewerts einer Funktion erfolgt durch `<param>`-Elemente bzw. durch das Element `<return>`. Tritt kein `<return>`-Element auf, besitzt die beschriebene Operation keinen Rückgabewert.

Attribute werden durch das Element `<attr>` definiert. Eine Attributdefinition beinhaltet neben dem Namen und einer optionalen Beschreibung dessen Datentyp und die ihm zugeordneten Abbildungsgraphen.

`<op>`

Das Element `<op>` dient zur Definition einer Funktion oder Methode. Diese Definition erfolgt durch den Namen der Funktion, deren Signatur (durch die Elemente `<param>`

bzw. <return>) und durch die optionale Angabe eines Kontexts (siehe Abbildung 4.38) und auftretenden Nebeneffekten. Zusätzlich kann ein Element <op> ein oder mehrere <graph>-Elemente enthalten (siehe Abschnitt 4.3.10.1). In FIX ist <op> wie folgt festgelegt:

```
<!ELEMENT op (title, desc*, param*, return?, context*, effect*, graph*)>
<!ATTLIST op
          id       ID #IMPLIED
          oneway   %boolean; "false"
          semantic %semantic; "undefined">
```

Die Definition des Attributs id entspricht der Definition des gleichnamigen Attributs von Element <struct> in Abschnitt 4.3.9.2. Das Attribut oneway entspricht dem gleichnamigen Attribut in IDL. Es gelten die in [OMG03] beschriebenen Ausführungssemantiken. Ist oneway auf true gesetzt, darf das Element <op> kein <return>-Element enthalten, ebenso keine <param>-Elemente, deren Attribut type auf in bzw. inout gesetzt ist. Das Attribut semantic dient zur Angabe von semantischen Hinweisen für die grafische Oberfläche (siehe Abschnitt 4.2.2.5). Die möglichen Werte behandelt Abschnitt 4.3.11.

<param>

Das Element <param> beschreibt einen Parameter einer Funktion. Neben der Angabe des Namens und des Datentyps legt das Attribut type fest, ob es sich um einen Eingabe- oder Ausgabeparameter oder beides handelt. <param> definieren wir folgendermaßen:

```
<!ELEMENT param (title, desc*, %param;)>
<!ATTLIST param
          id   ID #IMPLIED
          type (in | out | inout) "in">
```

Die Definition des Attributs id entspricht der Definition des gleichnamigen Attributs von Element <struct> in Abschnitt 4.3.9.2. Das Attribut type gibt die Art des Parameters an. Mögliche Werte sind in für Eingabeparameter, out für Ausgabeparameter und inout für Eingabe- und Ausgabeparameter.

<return>

Das Element <return> dient zur Beschreibung des Rückgabeparameters einer Operation. Es entspricht dabei einem <param>-Element mit gesetztem Attribut type="out" und fehlendem Element <title>. Enthält ein Element <op> kein <return>-Element, entspricht dies der Deklaration einer Operation als void in IDL. <return> ist wie folgt in FIX definiert:

```
<!ELEMENT return (desc*, %param;)>
<!ATTLIST return
          id ID #IMPLIED>
```

Die Definition des Attributs id entspricht ebenfalls der Definition in Abschnitt 4.3.9.2.

```
<op id="GetPart">
   <title>GetPart</title>
   <param id="GetPart.Nr">
      <title>Nr</title>
      <type xlink:href="#TeileNr"/>
   </param>
   <param id="GetPart.Name" type="out">
      <title>Name</title>
      <string/>
   </param>
   <param id="GetPart.Dim" type="out">
      <title>Dim</title>
      <type xlink:href="#Dimension_t"/>
   </param>
   <return id="GetPart.return">
      <bool/>
   </return>
</op>
```

Abbildung 4.38: Definition einer Operation mit dem Element `<op>`.

<context>

Der Inhalt des Elements `<context>` wird der aufgerufenen Funktion als Kontext [OMG03] übergeben. Die Definition gestaltet sich wie folgt:

```
<!ELEMENT context (#PCDATA)>
```

<attr>

Das Element `<attr>` dient zur Definition der Attribute eines Interface. Neben Name und Typ des Attributs können in einem Zielsystem zwei `<graph>`-Elemente zur Beschreibung des schreibenden bzw. lesenden Zugriffs enthalten sein. FIX definiert das Element `<attr>` wie folgt:

```
<!ELEMENT attr (title, desc*, %param;, effect*, graph*)>
<!ATTLIST attr
          id       ID #IMPLIED
          readonly %boolean; "false">
```

Die Definition des Attributs `id` entspricht ebenfalls der Definition in Abschnitt 4.3.9.2. Ist das Attribut `readonly` auf `true` gesetzt, ist es nicht möglich, schreibend auf das vereinbarte Attribut zuzugreifen. Das `readonly`-Attribut entspricht dem gleichnamigen Schlüsselwort von IDL.

Abbildung 4.39 illustriert eine einfache Attributdefinition.

```
<attr id="num_parts" readonly="true"> readonly attribute long num_parts;
   <title>num_parts</title>
   <int/>
</attr>
```

Abbildung 4.39: Definition eines Attributs.

4.3.9.7 Module und Interfaces

Die bereitgestellten Schnittstellen eines Systems können zur Strukturierung in verschiedene Typen unterteilt werden. Dazu stehen die Elemente `<module>` und `<interface>` zur Verfügung.

Einzelne, durch `<interface>`-Elemente definierte Schnittstellen können untereinander in Vererbungsbeziehungen stehen. Eine Schnittstelle, die von einer anderen Schnittstelle erbt, implementiert zusätzlich zu ihren eigenen Operationen und Attributen die der beerbten Schnittstelle. Innerhalb eines Elements `<interface>` können zu beerbende Schnittstellen durch ein oder mehrere `<inherits>`-Elemente referenziert werden.

Abbildung 4.40 illustriert die Verwendung der hier beschriebenen Elemente.

<module>

Das Element `<module>` dient zur Unterteilung eines Systems in verschiedene Module. Bis auf die Elemente zur Beschreibung der Netzwerkadresse und des Kommunikationsprotokolls kann es die gleichen Definitionen enthalten wie das Element `<system>`. So können `<module>`-Elemente auch geschachtelt werden. Im Gegensatz zum Element `<interface>` können innerhalb eines `<module>`-Elements keine Operationen oder Attribute definiert werden. Das Element `<module>` entspricht im Wesentlichen dem Module in [OMG03] und ist in FIX wie folgt definiert:

```
<!ELEMENT module (title, desc*, (%definiton;)+)>
```

<interface>

Innerhalb eines Elements `<interface>` ist ebenfalls die Definition von Datentypen und Konstanten möglich. Zusätzlich können Operationen und Attribute definiert werden. Ein Element `<interface>` kann keine weiteren `<module>`- oder `<interface>`-Elemente beinhalten. Wir definieren `<interface>` als:

```
<!ELEMENT interface (title, desc*, inherits*, (%export;)*)>
<!ATTLIST interface
          id       ID #IMPLIED
          abstract %boolean; "false">
```

Die Definition des Attributs `id` entspricht der Definition des gleichnamigen Attributs von Element `<struct>` in Abschnitt 4.3.9.2. Das Attribut `abstract` wird zur Kennzeichnung von abstrakten Interfaces benutzt und entspricht dem gleichnamigen Schlüsselwort von IDL.

<inherits>

Das Element <inherits> dient zur Beschreibung von Vererbungsbeziehungen zwischen Interfaces oder Value Types (siehe Abschnitt 4.3.9.8). Es kann innerhalb der Elemente <interface> oder <valuetype> mehrfach auftreten und verweist jeweils auf ein weiteres <interface>- bzw. <valuetype>-Element, von dem das aktuelle Element erbt. Es gelten die Vererbungsregeln aus [OMG03]. Die Definition gestaltet sich folgendermaßen:

```
<!ELEMENT inherits EMPTY)>
<!ATTLIST inherits
          xlink:type (simple) #FIXED "simple"
          xlink:href %uri; #REQUIRED>
```

Die Attribute xlink:type und xlink:href entsprechen den gleichnamigen Attributen in Abschnitt 4.3.7.3.

```
<module>
   <title>Module</title>
   <interface id="a">
      <title>a</title>
      ...
   </interface>

   <interface>
      <title>b</title>
      <inherits xlink:href="#a"/>
   </interface>
</module>
```

Abbildung 4.40: Gliederung von Schnittstellen mit Hilfe der Elemente <module> und <interface>.

4.3.9.8 Value Types

Value Types entsprechen den gleichnamigen Konstrukten von CORBA. Durch sie können Objekte mit deren Methoden und Attributen definiert werden. Im Gegensatz zu Interfaces können Value Types auch als Parameter an andere Methoden übergeben werden.

<valuetype>

Das Element <valuetype> dient zur Definition eines Value Type. Zur Beschreibung der Vererbungsbeziehungen kann es die Elemente <inherits> bzw. <supports> beinhalten. Dabei dient das Element <inherits> zur Referenzierung anderer Value Types, während das Element <supports> Interfaces beschreibt, welche der Value Type implementiert.

Es gibt normale Value Types und so genannte Boxed Value Types. Während erstere neben sämtlichen innerhalb des Elements <interface> erlaubten Elementen zusätzlich

noch die Elemente <statemember> oder <factory> beinhalten können, dürfen Boxed Value Types nur ein einzelnes <boxed>-Element beinhalten. Der Value Type ist in FIX folgendermaßen festgelegt:

```
<!ELEMENT valuetype (title, desc*, inherits*, supports*,
                     ((%export; | statemember | factory)* | boxed))>
<!ATTLIST valuetype
          id          ID IMPLIED
          custom      %boolean; "false"
          abstract    %boolean; "false"
          truncatable %boolean; "false">
```

Die Definition der Attribute xlink:type und xlink:href entspricht der Definition in Abschnitt 4.3.7.3. Das Attribut custom gibt an, ob für diesen Value Type ein spezielles Marshalling[6] verwendet werden soll. Das Attribut abstract entspricht dem gleichnamigen Attribut des Elements interface in Abschnitt 4.3.9.7. Das Attribut truncatable entspricht dem gleichnamigen Schlüsselwort von CORBA (siehe auch [OMG03]).

<supports>

Das Element <supports> dient ähnlich dem Element <inherits> zur Beschreibung von Vererbungsbeziehungen. Im Gegensatz zu diesem kann es nur innerhalb eines Elements <valuetype> vorkommen. Es dient zur Referenzierung von <interface>-Elementen, welche durch den Value Type implementiert werden. Die Definition von <supports> lautet in FIX:

```
<!ELEMENT supports EMPTY>
<!ATTLIST supports
          xlink:type (simple) #FIXED "simple"
          xlink:href %uri; #REQUIRED>
```

Die Attribute xlink:type und xlink:href entsprechen den gleichnamigen Attributen in Abschnitt 4.3.7.3.

<boxed>

Innerhalb des Elements <valuetype> darf das Element <boxed> nur einmal auftreten. Das Element selbst darf nur einen (einfachen oder komplexen) Datentyp beinhalten und gestaltet sich wie folgt:

```
<!ELEMENT boxed (%datatype;)>
```

<statemember>

Ein Statemember definiert ein Element des Zustands, welches bei der Übergabe des Value Type als Parameter mit übergeben und bei Bedarf auch über das Netz übertragen wird. Statemembers können entweder privat oder öffentlich sein. Auf private Statemembers kann nur der Code der Implementierung des Value Type zugreifen. Die Verwendung

[6] Marshalling beschreibt den Prozess des Packens von Daten in einen Puffer, bevor dieser über eine Leitung übertragen wird. Dabei werden nicht nur Daten verschiedenen Typs gesammelt, sondern diese werden auch in eine Standard-Repräsentation umgewandelt, die auch der Empfänger versteht.

von <statemember>-Elementen erfolgt ähnlich zu der von <attr>-Elementen. Wir definieren <statemember> wie folgt:

```
<!ELEMENT statemember (title, desc*, %datatype;, array*, graph*)>
<!ATTLIST statemember
          id   ID #IMPLIED
          type (public | private) "public">
```

Die Definition des Attributs id entspricht der Definition des gleichnamigen Attributs von Element <struct> in Abschnitt 4.3.9.2. Das Attribut type dient zur Unterscheidung zwischen privaten und öffentlichen Statemembers.

<factory>

Factories werden durch das Element <factory> definiert. Die Verwendung des Elements <factory> erfolgt ähnlich zu der des Elements <op>. Die innerhalb des Elements <factory> enthaltenen <param>-Elemente müssen Eingabeparameter sein (type="in"), ebenso darf kein Rückgabewert definiert werden. FIX legt <factory> wie folgt fest:

```
<!ELEMENT factory (title, desc*, param*, graph)>
<!ATTLIST factory
          id       ID #IMPLIED
          semantic %semantic; "undefined">
```

Die Definition des Attributs id entspricht der Definition des gleichnamigen Attributs von Element <struct> in Abschnitt 4.3.9.2. Das Attribut semantic entspricht dem gleichnamigen Attribut des Elements <op> in Abschnitt 4.3.9.6.

4.3.10 Abhängigkeitsbeschreibung

Nachdem wir das System und die Schnittstellen beschreiben können, folgt nun die Definition der Abbildungsbeschreibung. Zunächst gehen wir auf die grundlegenden Elemente ein, die für die Beschreibung einer beliebigen Abhängigkeit oder Referenzierung benötigt werden. Es folgt die Definition spezieller Aspekte wie Abhängigkeiten zwischen Funktionen, deren mehrfache Instanzierung, Einbindung textueller Beschreibungen, Referenzierung von Objekten, Definition bedingter Ausführungen und Einbindung von externen Graphen.

4.3.10.1 Grundlagen

Abhängigkeiten werden in FIX mittels eines Extended Link von XLink [DMO01] beschrieben, repräsentiert durch das Element <graph>. Das Element <graph> kann sowohl in den Elementen <op> als auch <attr> auftreten. Innerhalb eines <op>-Elements dient es zur Modellierung einer föderierten Funktion, innerhalb eines <graph>-Elements wird es zur Beschreibung des lesenden bzw. schreibenden Zugriffs auf ein globales Attribut benutzt.

Innerhalb des Elements <graph> erfolgt die Beschreibung der Knoten des Abhängigkeitsgraphen durch <node>-Elemente. Die Kanten werden durch <dep>-Elemente repräsentiert.

Wert	Mögliche Vaterelemente	Erläuterung
exec	<op>, <factory>	Ausführung einer Operation
read	<attr>, <statemember>	lesender Zugriff auf Attribute
write	<attr>, <statemember>	schreibender Zugriff auf Attribute
xlink:external-linkset	<op>, <factory>, <attr>, <statemember>	Verweis auf externen Graphen

Tabelle 4.7: Mögliche Werte des Attributs xlink:role.

<graph>

Das Element <graph> leitet einen Abhängigkeitsgraphen ein. Den Typ des Abhängigkeitsgraphen bestimmt dabei das Attribut xlink:role. <graph>-Elemente können direkt innerhalb der Elemente <op>, <attr>, <factory> oder <statemember> oder als externe Graphen (Abschnitt 4.3.10.11) innerhalb eines <mapping>-Elements auftreten. <graph>-Elemente, die innerhalb eines Zielsystems auftreten (bzw. mit Elementen eines Zielsystems verknüpft sind), dienen zur Modellierung eines Abhängigkeitsgraphen für globale Funktionen und Attribute. Die Definition von <graph> lautet:

```
<!ELEMENT graph (desc*, ((node | todo | dep | done)* | ext))>
<!ATTLIST graph
          id         ID #IMPLIED
          xlink:type (extended) #FIXED "extended"
          xlink:role (exec | read | write| xlink:external-linkset) "exec">
```

Die Definition des Attributs id entspricht der Definition des gleichnamigen Attributs von Element <struct> in Abschnitt 4.3.9.2. Das Attribut xlink:type ist festgesetzt auf den Wert extended, um dieses Element als Extended Link zu markieren. Das Attribut xlink:role dient zur Kennzeichnung des Typs des Abhängigkeitsgraphen (vgl. Tabelle 4.7).

<node>

<node>-Elemente sind die Locator-Elemente des durch <graph> erzeugten Extended Link. Sie dienen zur Referenzierung der externen Ressourcen. Referenziert werden können dabei Parameter, Konstanten, Operationen und Attribute. <node>-Elemente repräsentieren die Knoten des Abhängigkeitsgraphen. Die möglichen Elemente, auf die ein Element <node> verweisen kann, sind: <param>, <return>, <op>, <factory>, <effect>, <attr>, <statemember> und <const>. Außerdem kann ein Element <node> auch auf ein Element <enumerator> verweisen, um dessen Wert als Konstante zu benutzen. Somit sieht die Definition von <node> folgendermaßen aus:

```
<!ELEMENT node (title, desc*)>
<!ATTLIST node
          xlink:type   (locator) #FIXED "locator"
          xlink:href   %uri; #REQUIRED
          xlink:label  %Name; #REQUIRED
          instance     %Name; #IMPLIED
          semantic     %semantic; "undefined"
          compensation %boolean; "true">
```

Das Attribut `xlink:type` ist festgesetzt auf den Wert `locator`, um dieses Element als Locator-Element eines Extended Link zu markieren. Innerhalb des Elements `<node>` kann für das Attribut `xlink:href` neben der normalen XPointer-Syntax auch die erweiterte FIXPointer-Syntax (siehe Abschnitt 4.3.12) benutzt werden. `xlink:label` wird zur Referenzierung innerhalb von `<dep>`-Elementen benutzt. Aus diesem Grund muss jedem Element `<node>` ein innerhalb des umgebenden `<graph>`-Elements eindeutiges Attribut `xlink:label` zugeordnet werden. Das Attribut `instance` dient zur Unterscheidung mehrerer Instanzen einer Funktion innerhalb eines Abhängigkeitsgraphen (siehe auch Abschnitt 4.2.1.5). `<node>`-Elemente können ebenfalls ein `semantic`-Attribut enthalten. Dies kann erfolgen, falls innerhalb des Quellsystems keine Semantik für diese Funktion angegeben wurde, oder um eine angegebene Semantik zu überschreiben, falls die Funktion innerhalb dieses Graphen zu einem anderen Zweck als dem ursprünglich vorgesehenen gebraucht wird. Siehe auch Abschnitte 4.3.9.6 und 4.3.11. Das Attribut `compensation` gibt an, ob eine innerhalb der Definition der Operation angegebene Default-Kompensation ausgeführt werden soll (Default) oder nicht.

<dep>

Das Element `<dep>` dient als Arc-Element zur Verknüpfung der Elemente `<node>` oder `<todo>` untereinander. Mit `<dep>`-Elementen werden Abhängigkeitsrelationen und Kompensationsabhängigkeiten realisiert. Wir definieren `<dep>` wie folgt:

```
<!ELEMENT dep (%descr;)>
<!ATTLIST dep
          xlink:type    (arc) #FIXED "arc"
          xlink:from    %Name; #REQUIRED
          xlink:to      %Name; #REQUIRED
          xlink:arcrole (implementation | compensation | exception)
                        "implementation"
          invocation    (single | cyclic) "single"
          completion    (required | sufficient | optional) #IMPLIED
          priority      %integer; "0">
```

Das Attribut `xlink:type` ist festgesetzt auf den Wert `arc`, um dieses Element als Arc-Element eines Extended Link zu markieren. `xlink:from` dient zur Spezifikation des Ausgangspunkts einer Abhängigkeitsrelation. Es wird dabei das Element `<node>` referenziert, dessen `xlink:role`-Attribut mit dem Wert von `xlink:from` übereinstimmt. Das Attribut `xlink:to` dient analog zu `xlink:from` zur Spezifikation des Ziels einer Abhängigkeitsrelation. `xlink:arcrole` spezifiziert den Typ der Abhängigkeitsrelation. Mögliche Werte sind `implementation` zur Kennzeichnung einer normalen Abhängigkeitsrelation und `compensation` zur Markierung einer Kompensationsabhängigkeit innerhalb des Abhängigkeitsgraphen (siehe Abschnitt 4.3.10.10). Bei der Ausführung wird nur bei einem auftretenden Fehler versucht, diese Abhängigkeitsrelation zu erfüllen. Ist das Attribut `invocation` auf den Wert `iterative` gesetzt, erfolgt eine iterative Ausführung wie in Abschnitt 4.3.10.8 beschrieben. Das Attribut `completion` gibt den Typ der Abhängigkeit in Bezug auf die redundante Ausführung an. Mögliche Werte sind `required`, `sufficient` oder `optional`. Die Bedeutung der Werte ist in Abschnitt 4.3.10.9 beschrieben. Das Attribut `priority` bestimmt die Priorität einer als

`completion="optional"` markierten Abhängigkeitsrelation. Dabei stehen größere Werte für eine höhere Priorität. Handelt es sich nicht um eine optionale Abhängigkeitsrelation, hat der Wert keine Bedeutung.

Wird durch ein Element `<dep>` eine Kopieraktion modelliert (also z. B. durch eine Abhängigkeit der Art Parameter → Parameter), so werden außer den innerhalb von CORBA definierten Datentypkonvertierungen (wie z. B. int → long) keinerlei Konvertierungen durchgeführt. Dies muss durch die Verwendung einer entsprechenden Hilfsfunktion erfolgen.

Abbildung 4.41 zeigt die Modellierung der globalen Funktion `GetName()`.

```
<op id="GetName" semantic="read">
   <title>GetName</title>
   <param id="GetName.Nr">
      <title>Nr</title>
      <int/>
   </param>
   <return id="GetName.return">
      <string/>
   </return>
   <graph>
      <node xlink:href="#GetName.Nr" xlink:role="GetName.Nr"/>
      <node xlink:href="#GetName.return" xlink:role="GetName.return"/>
      <node xlink:href="Werk1.xml#GetPart.TeileNr"
       xlink:role="GetPart.TeileNr"/>
      <node xlink:href="Werk1.xml#GetPart.Name"
       xlink:role="GetPart.Name"/>

      <dep xlink:from="GetName.Nr" xlink:to="GetPart.TeileNr"/>
      <dep xlink:from="GetPart.Name" xlink:to="GetName.return"/>
   </graph>
</op>
```

Abbildung 4.41: Definition eines Abhängigkeitsgraphen mit `<graph>`.

4.3.10.2 Abhängigkeiten zwischen Funktionen

Die in Abschnitt 4.2.1.2 eingeführte Modellierung der Ausführungsreihenfolge durch Abhängigkeiten zwischen Funktionen wurde innerhalb von FIX so erweitert, dass Nebeneffekte einzelner Funktionen explizit modelliert werden können. Andere Funktionen können in Abhängigkeit von diesen Nebeneffekten stehen. Nebeneffekte werden innerhalb der Elemente `<op>` oder `<attr>` durch `<effect>`-Elemente modelliert.

Zur Modellierung der Ausführungsreihenfolge kann das Attribut `xlink:from` des Elements `<node>` auf ein `<effect>`-Element und das Attribut `xlink:to` auf ein `<op>`-Element verweisen. Als Sonderfall besteht die Möglichkeit, das Attribut `xlink:from` auf ein `<node>`-Element verweisen zu lassen, welches auf ein Element `<op>` zeigt. Dies entspricht dem Verweis auf ein in dem `<op>`-Element enthaltenes Element `<effect>`

ohne Inhalt. Dieser Sonderfall ist nicht auf <attr>-Elemente erweiterbar, da in diesem Fall nicht klar ist, ob sich die Abhängigkeit aus einem lesenden oder schreibenden Zugriff ergibt.

<effect>

Das Element <effect> beschreibt einen auftretenden Nebeneffekt. Neben dem Namen des Nebeneffekts kann eine Beschreibung innerhalb eines oder mehrerer <desc>-Elemente angegeben werden. Die Definition von <effect> ist die folgende:

```
<!ELEMENT effect (title, desc*)>
<!ATTLIST effect
          id   ID #IMPLIED
          type (exec | read | write) "exec">
```

Die Definition des Attributs id entspricht der Definition des gleichnamigen Attributs von Element <struct> in Abschnitt 4.3.9.2. Das Attribut type gibt an, bei welcher Aktion der Nebeneffekt auftritt. Steht das Element <effect> innerhalb eines <op>-Elements, ist nur der Defaultwert exec zulässig. Innerhalb eines <attr>-Elements wird durch das Attribut type angegeben, ob der Nebeneffekt bei lesendem oder schreibendem Zugriff auftritt.

Die Definition von Nebeneffekten illustriert Abbildung 4.42 anhand der lokalen Funktion Init() von Werk 1. Abbildung 4.43 zeigt die Modellierung von Abhängigkeiten von Nebeneffekten anhand der föderierten Funktion Init().

```
<op id="Init">
   <title>Init</title>
   <effect id="Neustart">
      <title>Neustart</title>
      <desc xml:lang="de">
         Das System führt einen Neustart durch.
      </desc>
   </effect>
</op>
```

Abbildung 4.42: Definition von Nebeneffekten mit <effect>.

4.3.10.3 Mehrfache Instanzierung von Funktionen

Die in Abschnitt 4.2.1.5 beschriebene mehrfache Instanzierung von Funktionen erfolgt in FIX durch mehrere <node>-Elemente, die dasselbe Ziel referenzieren, aber verschiedene instance-Attribute besitzen. Solche Elemente müssen ebenfalls unterschiedliche Attribute xlink:role besitzen, da diese nicht doppelt vorkommen dürfen. <node>-Elemente, deren Attribute instance den gleichen Wert besitzen, gehören zur selben Instanz der Funktion.

```
<op id="Init">
   <title>Init</title>
   <graph>
      <node xlink:href="#Init" xlink:role="Global.Init"/>
      <node xlink:href="Werk1.xml#Neustart" xlink:role="Werk1.Neustart"/>
      <node xlink:href="Werk2.xml#Reboot" xlink:role="Werk2.Reboot"/>

      <dep xlink:from="Werk1.Neustart" xlink:to="Global.Init"
        completion="required"/>
      <dep xlink:from="Werk2.Reboot" xlink:to="Global.Init"
        completion="required"/>
   </graph>
</op>
```

Abbildung 4.43: Modellierung von Abhängigkeiten zwischen Funktionen.

Der Wert `fix:target` für das Attribut `instance` ist reserviert für den Einsatz im Rahmen der Modellierung von globalen Attributen (Abschnitt 4.3.10.5). Ansonsten kann die Wahl des Werts des Attributs `instance` frei erfolgen; er wird an keiner anderen Stelle verwendet.

4.3.10.4 Textuelle Beschreibung

Die in Abschnitt 4.2.1.6 beschriebene textuelle Notation von Aktionen wird in FIX durch das Element `<todo>` realisiert.

`<todo>`

Das Element `<todo>` beschreibt eine lokale Ressource des durch `<graph>` repräsentierten Extended Link. So kann es durch `<dep>`-Elemente mit anderen `<node>`- oder `<todo>`-Elementen verknüpft werden.

Bei einer eventuellen Konvertierung des FIX-Dokuments in eine Programmiersprache wird der Inhalt dieser Elemente in Kommentaren an den entsprechenden Stellen des Quellcodes (vorgegeben durch die Abhängigkeiten zu anderen Elementen `<node>` oder `<todo>`) gesetzt. Die Definition von `<todo>` gestaltet sich folgendermaßen:

```
<!ELEMENT todo (title, desc*)>
<!ATTLIST todo
          xlink:type (resource) #FIXED "resource"
          xlink:role %Name; #REQUIRED
```

Das Attribut `xlink:type` ist festgesetzt auf den Wert `resource`, um dieses Element als eine lokale Ressource des Extended Link zu markieren. Die Definition des Attributs `xlink:role` entspricht der Definition des gleichnamigen Attributs von Element `<graph>` in Abschnitt 4.3.10.1.

Abbildung 4.44 zeigt die Modellierung der föderierten Funktion `GetFarbe()`.

```
<graph>
   <node xlink:href="#GetFarbe.Nr" xlink:role="GetFarbe.Nr"/>
   <node xlink:href="#GetFarbe.return" xlink:role="GetFarbe.return"/>
   <node xlink:href="Werk1.xml#GetPart.TeileNr"
     xlink:role="GetPart.TeileNr"/>
   <node xlink:href="Werk1.xml#GetPart.Farbe"
     xlink:role="get_part.color"/>
   <todo xlink:role="color2string">
      <title>color2string</title>
      <desc xml:lang="de">
         Farbe muss in den Namen der
         Farbe umgewandelt werden.
         Es gilt dabei die Zuordnung
         0 rot
         1 grün
         2 gelb
         3 blau
      </desc>
   </todo>

   <dep xlink:from="GetFarbe.Nr" xlink:to="GetPart.TeileNr"/>
   <dep xlink:from="GetPart.Farbe" xlink:to="color2string"/>
   <dep xlink:from="color2string" xlink:to="GetFarbe.return"/>
</graph>
```

Abbildung 4.44: Textuelle Beschreibung innerhalb einer Abhängigkeitsbeschreibung mit `<todo>`.

4.3.10.5 Globale Attribute

Die Modellierung von globalen Attributen erfolgt mit FIX. Innerhalb der Definition eines globalen Attributs durch das Element `<attr>` müssen zwei `<graph>`-Elemente vorhanden sein, jeweils eins für den lesenden und den schreibenden Zugriff.

Die Modellierung des Beispiels von Abbildung 4.10 in FIX illustriert Abbildung 4.45.

4.3.10.6 Objekte als Parameter

Die in Abschnitt 4.2.2.1 beschriebene Referenzierung von Methoden innerhalb von instanzierten Objekten wird in FIX durch spezielle FIXPointer-Achsen ermöglicht (siehe auch Abschnitt 4.3.12). Mit Hilfe der Achse `member`, die auch zur Referenzierung von Komponenten einer Struktur dient, kann innerhalb eines FIXPointer auf Attribute, Methoden und Statemember eines Objekts verwiesen werden. Die Achse `parameter` kann zusätzlich zur Referenzierung eines Parameters der durch die vorangegangene Achse spezifizierten Methode benutzt werden. Ein XPointer verweist auf ein bestimmtes Objekt, in dem er auf die Stelle verweist, an der das Objekt erzeugt wurde. Dies kann entweder der Rückgabewert einer Funktion sein oder aber eine Factory. Der XPointer

```
id("factory")/member::method1
```

```
<attr id="Waehrung">
   <title>Waehrung</title>
   <type xlink:href="#Waehrung_t"/>
   <graph xlink:role="read">
      <node xlink:href="#Waehrung" xlink:role="Waehrung"/>
      <node xlink:href="Lieferant.xml#Waehrung"
        xlink:role="Lieferant.Waehrung"/>
      <dep xlink:from="Lieferant.Waehrung" xlink:to="Waehrung"/>
   </graph>
   <graph xlink:role="write">
      <node xlink:href="#Waehrung" xlink:role="Waehrung"
        instance="fix:target"/>
      <node xlink:href="#Waehrung" xlink:role="Waehrungsquelle"/>
      <node xlink:href="Lieferant.xml#Waehrung"
        xlink:role="Lieferant.Waehrung"/>
      <dep xlink:from="Waehrungsquelle" xlink:to="Lieferant.Waehrung"/>
      <dep xlink:from="Lieferant.Waehrung" xlink:to="Waehrung"/>
   </graph>
</attr>
```

Abbildung 4.45: Definition globaler Attribute.

verweist auf die Methode `method1()` des von `factory()` erzeugten Objekts, während

```
id("factory")/member::method1/parameter::value
```

auf deren Parameter `value` verweist.

4.3.10.7 Bedingte Ausführung

In Abschnitt 4.2.2.2 haben wir gezeigt, wie Bedingungen ebenfalls über Abhängigkeitsrelationen beschrieben werden können. Dabei ist die Code-Funktion abhängig von einer Bedingungsfunktion. Aufgrund dieser Modellierung sind innerhalb von FIX keine weitergehenden Sprachkonstrukte zur Modellierung von Bedingungen notwendig. Zur Realisierung häufig auftretender Bedingungen wurden in der Standardbibliothek Funktionen wie z.B. `eval()`, `evalseq()`, `istrue()` oder `isfalse()` definiert (siehe Abschnitt 4.3.13).

4.3.10.8 Schleifen

Schleifen werden in FIX durch zwei Funktionen realisiert, die Bedingungsfunktion und die Code-Funktion, welche durch ein Element `<dep>` mit gesetztem Attribut `invocation ="cyclic"` verknüpft sind. Das Attribut `invocation="cyclic"` entspricht dabei dem Doppelpfeil der Abbildungen in Abschnitt 4.2.2.2.

Ein solches Element `<dep>` verbindet dabei im allgemeinen Fall zwei `<node>`-Elemente, die jeweils auf Funktionen verweisen. Um die beschriebenen Sonderfälle zu realisieren, können solche `<dep>`-Elemente auch ein Feld oder eine Sequenz mit einem Parameter oder umgekehrt verbinden. Im Zusammenhang damit werden oft die in der Standardbibliothek vorhandenen Funktionen zur Listenverarbeitung benötigt.

4.3.10.9 Steuerung paralleler Ausführung

Verweisen mehrere `<dep>`-Elemente auf denselben Zielknoten, ist die Auflösung einer dieser Abhängigkeiten ausreichend. Ist die Auflösung aller zu einem Knoten führenden Abhängigkeiten notwendig, müssen diese als solche gekennzeichnet werden. Zu diesem Zweck stehen die Attribute `completion` und `priority` des Elements `<dep>` zur Verfügung. Die möglichen Werte für `completion` haben dabei folgende Bedeutung:

- Ist das Attribut auf den Wert `sufficient` gesetzt und wird die durch dieses `<dep>`-Element definierte Abhängigkeitsrelation bei der Ausführung erfüllt, muss keine weitere, zu dem unter `xlink:to` spezifizierten Knoten führende Abhängigkeitsrelation erfüllt werden. Die Ausführung mehrerer Abhängigkeitsrelationen vom Typ `sufficient` (oder `required`) sollte – soweit möglich – parallel erfolgen. `sufficient` ist der Default-Wert.

- Ist der Wert auf `required` gesetzt, muss die Abhängigkeitsrelation bei der Ausführung erfüllt werden. Kann sie nicht erfüllt werden, bricht die Verarbeitung des Graphen ab.

- Beim Wert `optional` wird erst dann versucht, diese Abhängigkeitsrelation zu erfüllen, wenn alle als `sufficient` markierten, zum selben Zielknoten führenden Abhängigkeitsrelationen nicht erfüllt werden konnten. Abhängigkeitsrelationen des Typs `optional` werden in der Reihenfolge ihrer Priorität ausgeführt.

Grundsätzlich gilt: Es müssen alle der als `required` markierten Abhängigkeiten und mindestens eine der als `sufficient` oder `optional` markierten Abhängigkeiten erfüllt werden.

Sind mehrere Abhängigkeitsrelationen als `optional` markiert, wird deren Attribut `priority` ausgewertet. Es wird versucht, die optionalen Abhängigkeitsrelationen in der Reihenfolge ihrer Priorität (je größer der Wert des `priority`-Attributs, desto höher die Priorität) nacheinander aufzulösen. Sobald eine Funktion erfolgreich ausgeführt werden konnte, wird nicht mehr versucht, weitere als `optional` markierte Abhängigkeitsrelationen aufzulösen.

Als Beispiel dient die globale Funktion `GetTeil()`. Zu deren Realisierung (siehe auch Abbildung 4.18) muss auf die Funktion `GetPreis()` des Lieferantensystems zurückgegriffen werden. Liefert diese keinen Wert zurück (etwa weil das referenzierte Bauteil nicht von diesem Lieferanten bezogen werden kann), soll `GetPart()` den Betrag `0` als Preis zurückgeben. Dies kann durch das in Abbildung 4.46 dargestellte FIX-Fragment realisiert werden.

4.3.10.10 Kompensationen

Die Realisierung von Kompensationen innerhalb eines Abhängigkeitsgraphen erfolgt in FIX wie in Abschnitt 4.2.2.3 beschrieben. Die so genannten Fehlerabhängigkeitsrelationen werden durch `<dep>`-Elemente mit dem Attribut `xlink:role="compensation"` modelliert. Abbildung 4.47 illustriert die FIX-Modellierung der Funktion `AddTeil()`.

```
<graph>
   <node xlink:href="#GetTeil.Nr" xlink:role="Global.Nr"/>
   <node xlink:href="#GetTei.Name" xlink:role="Global.Name"/>
   <node xlink:href="#GetFarbe.Farbe" xlink:role="Global.Farbe"/>
   <node xlink:href="#GetTeil.Preis" xlink:role="Global.Preis"/>
   <node xlink:href="Werk1.xml#GetPart.TeileNr"
         xlink:role="Werk1.TeileNr"/>
   <node xlink:href="Werk1.xml#GetPart.Name"
         xlink:role="Werk1.Name"/>
   <node xlink:href="Werk1.xml#GetPart.Farbe"
         xlink:role="Werk1.Farbe"/>
   <node xlink:href="helper.xml#col2str.color"
         xlink:role="col2str.color"/>
   <node xlink:href="helper.xml#col2str.return"
         xlink:role="col2str.return"/>
   <node xlink:href="Lieferant.xml#GetPreis.Nr"
         xlink:role="Lieferant.Nr"/>
   <node xlink:href="Lieferant.xml#GetPreis.return"
         xlink:role="Lieferant.Preis"/>
   <node xlink:href="helper.xml#null" xlink:role="null"/>

   <dep xlink:from="Global.Nr" xlink:to="Werk1.TeileNr"/>
   <dep xlink:from="Global.Nr" xlink:to="Lieferant.Nr"/>
   <dep xlink:from="Werk1.Name" xlink:to="Global.Name"/>
   <dep xlink:from="Werk1.Farbe" xlink:to="col2str.color"/>
   <dep xlink:from="col2str.return" xlink:to="Global.Farbe"/>
   <dep xlink:from="Lieferant.Preis" xlink:to="Global.Preis"/>
   <dep xlink:from="null" xlink:to="Global.Preis"
     completion="optional"/>
</graph>
```

Abbildung 4.46: Definition redundanter Ausführung.

4.3.10.11 Externe Graphen

Die Modellierung der Abhängigkeitsbeschreibung muss nicht innerhalb der Funktions- oder Attributdefinition erfolgen, sondern kann sich auch in einem externen Dokument befinden. Durch das Element `<ext>` innerhalb von `<graph>`-Elementen können extern definierte Graphen referenziert werden. Diese externen Graphen werden innerhalb eines Elements `<mapping>` definiert.

<ext>

Das Element `<ext>` dient zum Verweis auf externe Graphen. Ein Element `<ext>` kann nur innerhalb eines `<graph>`-Elements mit einem Attribut `xlink:role="xlink:external-linkset"` vorkommen. Es darf nur ein einziges Element `<ext>` innerhalb eines Elements `<graph>` vorkommen. Die Definition von `<ext>` gestaltet sich wie folgt:

```
<!ELEMENT ext EMPTY>
<!ATTLIST ext
```

```
<!-- "normale" Abhängigkeiten -->

<dep xlink:from="AddTeil.name" xlink:to="Werk2.AddPart.Name"/>
<dep xlink:from="AddTeil.Name" xlink:to="Werk1.AddPart.TeileNr"/>
<dep xlink:from="AddTeil.Farbe" xlink:to="Werk1.AddPart.Farbe"/>
<dep xlink:from="Werk2.AddPart.return" xlink:to="strtoint.str"/>
<dep xlink:from="strtoint.return" xlink:to="AddTeil.return"/>
<dep xlink:from="strtoint.return" xlink:to="Werk1.AddPart.TeileNr"/>
<dep xlink:from="Werk1.AddPart" xlink:to="AddTeil"/>

<!-- Fehlerabhängigkeitsrelationen -->

<dep xlink:from="Werk2.AddPart" xlink:to="Werk2.DelPart"
     xlink:role="compensation"/>
<dep xlink:from="Werk2.AddPart" xlink:to="null2"
     xlink:role="compensation"/>
<dep xlink:from="Werk2.AddPart.return" xlink:to="Werk2.DelPart.no"/>
<dep xlink:from="Werk2.DelPart" xlink:to="null1"
     completion="required"/>

<dep xlink:from="Werk1.AddPart" xlink:to="Werk1.DelPart"
     xlink:role="compensation"/>
<dep xlink:from="Werk1.AddPart" xlink:to="null2"
     xlink:role="compensation"/>
<dep xlink:from="strtoint.return" xlink:to="Werk1.DelPart.TeileNr"/>
<dep xlink:from="Werk1.DelPart" xlink:to="null"
     completion="required"/>

<dep xlink:from="null1" xlink:to="AddTeil.return"/>
<dep xlink:from="null2" xlink:to="AddTeil.return"
     completion="optional"/>
```

Abbildung 4.47: Definition von Kompensationen in FIX.

```
xlink:type (locator) #FIXED "locator"
xlink:href %uri; #REQUIRED>
```

Das Attribut `xlink:type` ist festgesetzt auf den Wert `locator`, um dieses Element als ein Locator-Element des Extended Link zu markieren. `xlink:href` verweist auf den externen Graphen.

<mapping>

Das Element `<mapping>` dient als Container für externe Graphen. `<mapping>` ist ein Wurzelelement in dem beliebig viele `<graph>`-Elemente vorkommen können. Die Definition von `<mapping>` ist einfach:

```
<!ELEMENT mapping (graph)*>
```

Abbildung 4.48 veranschaulicht die Verwendung der Elemente `<ext>` und `<mapping>`.

```
Datei "system.xml":

<op>
   <title>example</title>
   ...
   <graph xlink:role="xlink:external-linkset">
      <ext xlink:href="extgraphs.xml#example"/>
   </graph>
</op>

Datei "extgraphs.xml":

<mapping>
   <graph id="example">
   ...
   </graph>
</mapping>
```

Abbildung 4.48: Definition und Einbindung externer Graphen.

Wert	Bedeutung
`convert`	Konvertierungsfunktion
`condition`	Bedingung
`read`	lesender Zugriff
`add`	Hinzufügen eines Elements
`delete`	Löschen eines Elements
`update`	Verändern eines Elements
`other`	ein Nicht-Standard-Fall
`undefined`	keine Semantik angegeben

Tabelle 4.8: Mögliche Werte des Attributs `semantic`.

4.3.11 Semantische Attribute

Semantische Attribute (siehe auch Abschnitt 4.2.2.5) werden in FIX durch das Attribut `semantic` modelliert. Dies kann sowohl ein Element `<op>` als auch ein Element `<node>` näher beschreiben. Zeigt ein Element `<node>`, welches ein `semantic`-Attribut besitzt, auf ein Element `<op>`, das ebenfalls ein `semantic`-Attribut besitzt, so gilt der Wert des `<node>`-Elements.

Die von FIX definierten möglichen Werte für das Attribut `semantic` zeigt Tabelle 4.8. Die Implementierung des `semantic`-Attributs ist für FIX-kompatible Software optional. Wird das `semantic` durch ein bestimmtes Programm ausgewertet, so müssen die in Tabelle 4.8 definierten Werte auf jeden Fall unterstützt werden. Die dargestellten Werte sind innerhalb des FIX-Namensraums definiert. Sie müssen also mit den für das Element `<system>` vorgeschriebenen Namensraumdeklarationen sowohl mit als auch ohne vorangestelltem `fix:`-Präfix erkannt werden.

Es ist möglich, dass bestimmte FIX-Software zusätzliche Werte für das Attribut `semantic` definiert. Diese neuen Werte müssen innerhalb eines separaten Namensraums definiert werden, damit die Eindeutigkeit gewahrt bleibt. Der benutzte Namensraum muss dabei entsprechend im Element `<system>` deklariert werden (siehe Abschnitt 4.3.3).

Die Unterstützung von Werten für das `semantic`-Attribut, die in anderen Namensräumen definiert wurden und nicht den Werten in Tabelle 4.8 entsprechen, ist optional. Enthält ein Dokument einen Wert von `semantic`, der in einem Namensraum deklariert wurde, welcher von der Software nicht unterstützt wird, muss das Attribut `semantic` statt dessen den Wert `fix:other` annehmen.

4.3.12 FIXPointer

FIXPointer erweitern die normalen XPointer aus [DDJM99] um die drei neuen Achsen `member`, `element` und `parameter`. Mit Hilfe dieser neuen Achsen können Teile von Parametern durch `<node>`-Elemente referenziert werden. Um einen FIXPointer von einem normalen XPointer zu unterscheiden, wird er durch ein umschließendes `fixpointer()` gekennzeichnet.

element

Die Achse `element` dient zur Referenzierung eines bestimmten Elements innerhalb eines Feldes, einer Sequenz oder einer Zeichenkette. Als Node-Test muss die Nummer des Elements angegeben werden, wobei die Nummerierung bei 0 beginnt. Der FIXPointer aus Abbildung 4.49 referenziert beispielsweise das fünfte Element des Rückgabewerts der Funktion `AlleTeile()` von Werk 2:

```
<node xlink:href='Werk2.xml#fixpointer(id("AlleParts.return")/element::5)'
  xlink:role="element5"/>
```

Abbildung 4.49: Gebrauch der Achse `element`.

member

Durch die Achse `member` können einzelne Komponenten von Strukturen referenziert werden. Außerdem kann die Achse zur Referenzierung von Attributen, Operationen oder Statemember eines Value Type dienen. Als Node-Test muss dabei die ID der gewünschten Komponente des Parameters angegeben werden. In Abbildung 4.50 referenziert das Element `<node>` die Komponente `x` des Rückgabeparameters `Dim` der Funktion `GetPart()` von Werk 2:

parameter

Diese Achse dient zur Referenzierung von Parametern einer Methode. Sie findet bei der in Abschnitt 4.3.10.6 beschriebenen Integration objektorientierter Systeme Verwendung. Der FIXPointer aus Abbildung 4.51 referenziert das Element `<param>` mit der

```
<node
  xlink:href='Werk2.xml#fixpointer(id("GetPart.Dim")/
              member::dimension.x)'
  xlink:role="memberx"/>
```

Abbildung 4.50: Referenzierung mit der Achse `member`.

ID `convert.object` innerhalb des Elements `<op>` mit der ID `convert` des durch die Factory `factory2` erzeugten Objekts.

```
<node xlink:href="local.xml#fixpointer(id("factory2")/
  member::convert/parameter::convert.object)"
  xlink:role="factory2.convert.object"/>
```

Abbildung 4.51: Referenzierung mit der Achse `parameter`.

Gemeinsame Verarbeitung von XPointer und FIXPointer-Schemata

Um die Verarbeitung von FIX-Dokumenten auch durch generische XLink-Software zu ermöglichen, sollte bei der Verwendung von FIXPointern immer zusätzlich auch ein XPointer angegeben werden. Der XPointer sollte dabei auf den Parameter als Ganzes zeigen. Abbildung 4.52 zeigt das Beispiel aus Abbildung 4.50 mit einem zusätzlichen XPointer.

```
<node
  xlink:href='werk2.xml#fixpointer(id("GetPart.Dim")/
  member::dimsion.x)xpointer(id("GetPart.Dim"))'
  xlink:role="memberx"/>
```

Abbildung 4.52: Gemeinsame Verwendung von XPointer- und FIXPointer-Schemata.

4.3.13 Standardbibliothek

Die folgenden Funktionen müssen von einer FIX-konformen Software in der so genannten Standardbibliothek implementiert werden. In Abhängigkeitsbeschreibungen können diese innerhalb des Dokuments `http://dcx.com/fix/stdlib.xml` referenziert werden. Das FIX-Dokument `stdlib.xml` befindet sich in Anhang C.

Im Rahmen der Weiterentwicklung von FIX sollten vor allem die innerhalb von `stdlib.xml` definierten Funktionen wesentlich erweitert werden. Die folgenden Funktionen sollten den Mindestumfang der Standardbibliothek darstellen:

- `eval(format, value)` akzeptiert als Parameter einen Formatstring und einen Wert. Der Formatstring kann dabei ein Platzhalter in der Syntax eines `printf()`-Befehls von C sein. Nach dem Aufruf der Funktion wird der Platzhalter durch den übergebenen Wert ersetzt. Der sich ergebende Ausdruck wird ausgewertet.

- `evalseq(format, values)` ist eine Erweiterung der Funktion `eval()` auf mehrere Parameter. Als zweiter Parameter muss eine Sequenz übergeben werden. Die Anzahl der Elemente in der Sequenz muss mit der Anzahl der Platzhalter im Formatstring übereinstimmen.

- Wird der Funktion `istrue(value)` der Wert `TRUE` übergeben, führt sie keinerlei Operationen durch. Wird der Wert `FALSE` übergeben, kehrt sie mit einer Fehlermeldung zurück.

- `isfalse(value)` verhält sich komplementär zu `istrue()`, entspricht also dem Funktionsaufruf `istrue(!value)`.

- `isequal(a, b)` vergleicht die beiden übergebenen Parameter. Stimmen diese nicht überein, kehrt `isequal()` mit einer Fehlermeldung zurück.

- `isnotequal(a, b)` vergleicht ebenso die beiden übergebenen Parameter, kehrt aber mit einer Fehlermeldung zurück, falls die beiden Parameter übereinstimmen.

- `iszero(value)` löst eine Fehlermeldung aus, falls der übergebene Wert ungleich `0` ist.

- `isnotzero` löst eine Fehlermeldung aus, wenn der übergebene Wert gleich `0` ist.

- `loop(max, count)` realisiert eine einfache Schleife. Die Funktion erhöht den in `count` übergebenen Wert um 1. Ist der neue Wert größer als der in `max` übergebene, wird eine Fehlermeldung ausgelöst. Andernfalls wird der neu berechnete Wert in `count` zurückgegeben.

- `incr(value)` erhöht den übergebenen Wert um 1.

- `decr(value)` erniedrigt den übergebenen Wert um 1.

- `long str2int(str)` konvertiert den übergebenen String, soweit möglich, in einen Integer.

- `string int2str(int)` konvertiert den übergebenen Integer in einen String.

4.3.14 Konvertierungsmöglichkeiten

Eine Möglichkeit der Verarbeitung von FIX-Dokumenten ist die Konvertierung in andere Formate. Denkbar ist beispielsweise die Erzeugung von IDL-Definitionen aus FIX-Dokumenten oder die Konvertierung in HTML-Dokumente zur Generierung einer Online-Dokumentation. Durch die Verwendung von XML als Grundlage von FIX bietet sich der Einsatz bereits existierender Technologien zu diesem Zweck an.

Die XSL Transformations (XSLT [Cla99]) sind ein Teil der Extensible Stylesheet Language (XSL [Adl01]), einer Sprache zur Definition sogenannter Stylesheets, also Darstellungsanweisungen für XML-Dokumente. Mit Hilfe von XSLT ist die Beschreibung von Regeln zur Umwandlung von XML-Dokumenten in andere XML-Dokumente oder reine ASCII-Dokumente möglich. Mit Hilfe von XSLT können wir die Abbildung zwischen dem Beschreibungs- und dem Ausführungsmodell festlegen. Dabei kann grundsätzlich zwischen zwei Ansätzen unterschieden werden:

1. Man generiert Eingabeinformationen für bestehende Technologien und Produkte.
2. Man generiert Code-Fragmente für Eigenimplementierungen.

Darüber hinaus kann eine ausführliche Dokumentation der implementierten Integration automatisiert erstellt werden.

Das in Anhang B enthaltene XSLT-Stylesheet realisiert beispielhaft eine Konvertierung von FIX-Dokumenten in IDL-Definitionen. Das Stylesheet konvertiert die in einem FIX-Dokument definierte Schnittstelle in die IDL-Syntax. Die Elemente `<const>` werden dabei nur konvertiert, falls ein zugehöriges Element `<expr>` existiert und der enthaltene Wert im Format `text/plain` oder `text/x-idl` ist. Abhängigkeitsbeschreibungen innerhalb von `<graph>`-Elementen werden ignoriert.

4.4 Zusammenfassung

In diesem Kapitel haben wir uns auf einen Lösungsansatz für die Funktionsintegration konzentriert, den wir in ein Beschreibungs- und ein Ausführungsmodell aufteilen. Auf diese Weise kann die Beschreibung der Abbildung von föderierten Funktionen auf lokale Funktionen unabhängig von der technischen Umsetzung – dem Ausführungsmodell – dargestellt werden. Wie eine solche Beschreibung aussehen kann, zeigt das vorgestellte Beschreibungsmodell.

Wir haben zunächst das grundlegende Konzept der Abbildungsbeschreibung eingeführt. Es basiert auf der Beschreibung der Parameterabhängigkeiten zwischen den föderierten und lokalen Funktionen. Als Ergebnis erhält man einen gerichteten azyklischen Graph, der nach Durchführung einer topologischen Sortierung auch die möglichen Ausführungsreihenfolgen der integrierten Funktionen liefert. Die Vorgehensweise auf Basis von Abhängigkeitsbeschreibungen lässt sich für die wichtigsten Aspekte der Funktionsabbildung beibehalten. Zu diesen Aspekten gehören u. a. die Einbindung von Hilfsfunktionen und Konstanten, textuelle Beschreibungen von sehr komplexen Abbildungen, besondere Gesichtspunkte der Objektorientierung, die Kontrollflusssteuerung und Aktualisierungsprobleme. Damit können alle zu beschreibenden Aspekte in kohärenter Weise dargestellt werden.

Aufbauend auf diesem Konzept haben wir eine Abbildungssprache auf Basis von XML definiert. Die Sprache baut auf einem Standard auf, der mit zahlreichen verwandten Standards und entsprechenden Werkzeugen deren Verarbeitung vereinfacht. Die Abbildungssprache enthält im ersten Teil eine System- und Schnittstellenbeschreibung und im zweiten Teil die Beschreibung der Funktionsabbildung. Die Parameterabhängigkeiten werden mit dem Standard XLink definiert. Es ist gelungen, eine sehr schlanke Sprache zu entwickeln, die mit Hilfe des Standards XSLT in andere XML-Dokumente oder Textformate überführt werden kann. So kann eine Abbildung auf das Ausführungsmodell definiert werden, mit der entweder Eingaben oder Code-Fragmente für die Ausführungskomponente generiert werden können.

Kapitel 5

Ausführungsmodell

Die angestrebte Integrationsarchitektur setzt sich aus zwei Komponenten zusammen: einem FDBS und einer Komponente zur Integration von Funktionen (kurz KIF). Die KIF übernimmt die Umsetzung der in den letzten Kapiteln beschriebenen Funktionsintegration und damit der eingeführten Abbildungssprache. Als Ergebnis stellt sie föderierte Funktionen zur Verfügung, die wiederum von der zweiten Komponente, dem FDBS, integriert werden können. Damit ist das FDBS in einer Aufrufschicht über der KIF angesiedelt und setzt die Integration von Daten aus heterogenen DBS und föderierten Funktionen um.

Wir wählen das FDBS als Teil des Ausführungsmodells, da in erster Linie Daten integriert werden sollen. Die Integration der Anwendungssysteme und deren Funktionen wurde notwendig, weil nur über deren Funktionsschnittstellen die zu integrierenden Daten abzurufen sind. Daher scheint ein Ausführungssystem, dessen Stärke in der effizienten Verarbeitung von Daten liegt, das am besten geeignete für die Integration. Darüber hinaus spielen folgende Gründe eine wichtige Rolle für die Wahl des FDBS:

- Mit der Unterstützung von SQL steht eine mächtige, deklarative Anfragesprache zur Verfügung.
- In den allermeisten Fällen ist in Firmen SQL-Wissen vorhanden, so dass mit Einsatz des FDBS vorhandene Investitionen geschützt werden. Bestehende Entwicklungsteams können ohne zusätzliche Schulungen Anwendungen implementieren.
- Ähnliches gilt für den Betrieb und die Administration des FDBS. Da es sich hierbei im Wesentlichen um ein relationales Datenbanksystem handelt, können bestehende Konzepte für den Datenbankbetrieb übernommen werden.
- Nicht zu unterschätzen ist die weite Verbreitung und damit Unterstützung von SQL durch unabhängige Software-Hersteller. Somit ist gewährleistet, dass sich auf dem Markt befindliche Software-Produkte ohne größeren Anpassungsaufwand auf Basis der Integrationsarchitektur einsetzen lassen.

Wie sich die KIF im Detail gestaltet, erläutern wir erst gegen Ende dieses Kapitels, um zu verdeutlichen, dass der vorgestellte Ansatz sehr flexibel ist hinsichtlich der letztendlich gewählten Ausführungskomponente.

Das Zusammenspiel dieser beiden Komponenten erfordert, dass die KIF an das FDBS angebunden und von diesem angesprochen werden kann. Demnach muss das FDBS erkennen, welche Teile der Anfrage vom FDBS auszuführen sind und welche von der KIF übernommen werden müssen. Demzufolge ist ein Mechanismus einzusetzen, der es dem FDBS erlaubt, die KIF anzusteuern und ihre (Zwischen-)Ergebnisse zu übernehmen.

Neben der Anfrageverarbeitung spielt die Transaktionsverwaltung eine wichtige Rolle. Soll der schreibende Zugriff auf die integrierten Systeme unterstützt werden, müssen Aspekte von heterogenen Transaktionen untersucht werden, um die Konsistenz der Daten zu gewährleisten.

In diesem Kapitel betrachten wir mehrere Aspekte des Ausführungsmodells. Zunächst wird untersucht, wie die Kopplung von FDBS und KIF gestaltet werden kann. Anschließend erläutern wir die Umsetzung der Anfrageverarbeitung auf Basis der gewählten Kopplungsart. Nach der Vorstellung eines heterogenen Transaktionsmodells beschreiben wir Realisierungsalternativen für die Komponente zur Funktionsintegration.

5.1 Anbindung

Zur Integration von Daten und Funktionen müssen wir die beiden Hauptkomponenten unserer Integrationsarchitektur verbinden. Da das FDBS die oberste Schicht bildet, ist die Anbindung der KIF an das FDBS notwendig. In den nächsten Abschnitten werden wir mehrere Aspekte einer solchen Anbindung betrachten, die zwei grundlegende Fragen beantworten: Welcher Kopplungsmechanismus soll gewählt werden und welche Funktionalität sollte er unterstützen?

Welchen Anforderungen sollte der gewählte Mechanismus genügen?

- Der Mechanismus sollte standardkonform sein.
- Er sollte von kommerziell verfügbaren Produkten unterstützt werden.
- Selbst zu entwickelnde Komponenten sollten möglichst klein gehalten werden. So muss wenig Software-Code weiterentwickelt und gewartet werden, falls sich Schnittstellen der eingesetzten Produkte ändern.

Welche Funktionalität sollte der Mechanismus unterstützen?

- Die Mindestanforderung ist die bidirektionale Abbildung von Tabellen auf Funktionssignaturen und insbesondere von SQL-Anfragen auf Funktionsaufrufe. Der Mechanismus muss SQL-Anweisungen an das FDBS in die zugehörigen Funktionsaufrufe abbilden, um die angeforderten Daten auszulesen. Anschließend muss er das Ergebnis der Funktionsausführung in Tabellenform an das FDBS zurückliefern.
- Zusätzliche Funktionalität könnte die globale Anfrageverarbeitung und -optimierung maßgeblich unterstützen.

Da die Kopplung aus dem FDBS heraus stattfindet und wir eine standardkonforme Lösung suchen, ist der Mechanismus in der SQL-Welt zu suchen. Es stehen drei Implementierungsmöglichkeiten zur Wahl [HH00]:

LiefNr	KompNr	Lager	Order
1	11	5	20
1	13	10	10
2	11	2	10
2	12	3	10
2	13	0	15
3	12	6	15
3	13	7	10

Tabelle 5.1: Beispieldaten der föderierten Funktion `GetBestand()`.

- Gespeicherte Prozeduren
- Benutzerdefinierte Tabellenfunktionen
- Wrapper

Wie ein möglicher Einsatz dieser Alternativen aussieht und wie er zu bewerten ist, erläutern wir im folgenden Abschnitt.

5.1.1 Verfügbare Mechanismen

Wir betrachten den Einsatz der drei Kopplungsvarianten anhand eines Beispiels. Für jeden Mechanismus zeigen wir auf, wie die Abbildung von Tabelle auf Funktionssignatur bzw. SQL-Anweisung auf Funktionsaufruf umgesetzt wird und wie der Zugriff auf die integrierte Funktion erfolgt. Als Beispiel soll die Funktion `GetBestand(IN LiefNr, IN KompNr, OUT Lager, OUT Order)` über das FDBS eingebunden werden. Diese Funktion wird von der KIF als föderierte Funktion bereitgestellt und liefert für eine gegebene Lieferanten- und Komponentennummer (`LiefNr` und `KompNr`) den aktuellen Lagerbestand (`Lager`) und die Anzahl bestellter Komponenten (`Order`). Tabelle 5.1 zeigt die verfügbaren Werte in den lokalen Systemen.

Wie diese föderierte Funktion auf lokale Systeme abgebildet wird, ist an dieser Stelle nicht von Belang und wird daher nicht weiter betrachtet. Für alle Mechanismen gilt, dass sie eine Verbindung zur KIF herstellen, föderierte Funktionen aufrufen sowie das Ergebnis abrufen und an das FDBS zurückgeben können. Wir beschreiben jeden Mechanismus in knapper Form, erläutern die Abbildungsumsetzung anhand eines Beispiels und bewerten ihn.

5.1.1.1 Gespeicherte Prozeduren

Eine gespeicherte Prozedur ist ein Unterprogramm, das im Allgemeinen aus mehreren SQL-Anweisungen besteht und mit der CALL-Anweisung im Anwendungsprogramm gestartet wird. Der Einsatz von gespeicherten Prozeduren macht insbesondere dann Sinn, wenn Teile des Anwendungsprogramms längere Folgen von SQL-Anweisungen aufweisen, die keine Interaktion mit dem (End-)Benutzer verlangen. Werden die einzelnen SQL-Anweisungen in der Anwendung aufgerufen, so sind fortwährend zwischen Client-

und Server-Maschine Daten und Nachrichten zu übermitteln. Um den Kommunikationsaufwand zu reduzieren und damit das Leistungsverhalten zu verbessern, werden solche Folgen von SQL-Anweisungen in einer gespeicherten Prozedur zusammengefasst und auf Seiten des Servers gespeichert. Ihre Ausführung erfolgt nach dem Aufruf mit allen benötigten Eingabedaten vollständig auf der Server-Seite. Nachdem die Prozedur ganz ausgeführt wurde, wird das Ergebnis zum Client übermittelt. Während der Ausführung findet keine Interaktion zwischen Client und Server statt.

Es gibt zwei Varianten von gespeicherten Prozeduren: Sie können in einer vom SQL-Standard unterstützten Programmiersprache geschrieben werden, welche die SQL-Anweisungen enthält. Oder ihr Rumpf enthält nur die SQL-Anweisungen ohne Einbettung in eine Programmiersprache [ISO99, ISO02].

Setzen wir gespeicherte Prozeduren als Kopplungsmechanismus ein, erfolgt der Aufruf der föderierten Funktion in der KIF über die Prozedur. Eine Funktion entspricht technisch gesehen jedoch nicht exakt einer Prozedur, da die Funktion einen Rückgabewert hat, während die Prozedur Nebeneffekte erzeugt. Da sie aber fast den gleichen Aufbau haben, kann man eine Abbildung zwischen den beiden Formen festlegen. Dabei ist darauf zu achten, dass die Prozedur den Rückgabewert der Funktion in einen Ausgabeparameter schreibt und darüber hinaus keine weiteren Seiteneffekte entstehen. Beim Einsatz von Prozeduren sind grundsätzlich zwei Varianten denkbar.

In der ersten Variante wird eine (1:1)-Abbildung umgesetzt, indem für jede föderierte Funktion eine gespeicherte Prozedur erzeugt wird. Sie hat denselben Namen und dieselben Parameter. Damit ist der Name der aufzurufenden Funktion in der Prozedur fest verdrahtet.

Für unsere Beispielfunktion `GetBestand()` legen wir folgende gespeicherte Prozedur an:

```
CREATE PROCEDURE GetBestand (
        IN LiefNr INTEGER,
        IN KompNr INTEGER,
        OUT Lager INTEGER,
        OUT Order INTEGER)
   LANGUAGE JAVA
   PARAMETER STYLE GENERAL
   SPECIFIC GetBestand_Java
   DETERMINISTIC
   NO SQL
   DYNAMIC RESULT SETS 0
   EXTERNAL NAME '/sql/procedures/Lager.GetBestand'
```

Der Aufruf der Prozedur mit den Werten `1` und `13` für die Eingabeparameter `LiefNr` und `KompNr` gestaltet sich folgendermaßen:

```
CALL GetBestand (1, 13, :Anzahl, :Order)
```

Die zweite Variante basiert auf einer generischen Prozedur, die für den Aufruf aller föderierten Funktionen genutzt wird. Es werden dabei nicht nur die Parameter der Funktion übergeben, sondern es muss in einem weiteren Parameter der Name der aufzurufenden föderierten Funktion bekannt gegeben werden. Aufgrund dieser und weiterer Informationen stellt die generische Prozedur fest, welche Funktion in der KIF mit

welchen Werten angestoßen werden muss. Um den Prozeduraufruf korrekt auf einen Funktionsaufruf abzubilden, muss die Prozedur auf weitere Informationen zurückgreifen können. Dazu gehören die verfügbaren Funktionen und deren Funktionssignatur.

Die Definition dieser Prozedur lautet dann wie folgt:

```
CREATE PROCEDURE FuncCall (
        IN FuncName VARCHAR(50),
        IN InputNr INTEGER,
        IN InputValues VARCHAR(50) ARRAY[10],
        IN OutputNr INTEGER,
        OUT OutputValues VARCHAR(50) ARRAY[10])
   LANGUAGE JAVA
   PARAMETER STYLE GENERAL
   SPECIFIC FuncCall_Java
   DETERMINISTIC
   NO SQL
   DYNAMIC RESULT SETS 0
   EXTERNAL NAME '/sql/procedures/Generic.FuncCall'
```

Der Prozeduraufruf sieht nun wie folgt aus:

```
CALL FuncCall ('GetBestand', 2, ['1', '13'], 2, :Output)
```

Betrachtet man die erste Variante, ist die (1:1)-Abbildung von Prozedur auf Funktion klar von Vorteil, da die Funktionen nicht hinter Tabellen verborgen werden und damit auch keine Abbildung von Funktionssignatur auf Tabelle stattfinden muss. Gegenüber dem Benutzer wird die Transparenz hinsichtlich Tabellen- oder Funktionszugriff teilweise aufgegeben, da er sehen kann, dass er nicht mit Tabellen arbeitet.

Als Nachteil erweist sich jedoch die Handhabung der gespeicherten Prozeduren. Da sie in einer separaten CALL-Anweisung aufgerufen werden, können sie nicht in SELECT-Anweisungen mit anderen Tabellen kombiniert werden. Somit ist der kombinierte Zugriff innerhalb einer Anweisung auf DBS und föderierte Funktionen nicht möglich. Folglich wird eine Integration von Daten und Funktionen nicht direkt unterstützt. Ist dies jedoch ein Musskriterium, so bleibt nur der Weg über eine darüber liegende Prozedur, die innerhalb ihres Programmcodes einzelne Prozeduren aufrufen als auch SELECT-Anweisungen absetzen kann und die Einzelergebnisse zusammenfügt. Diese „Integrationsprozedur“ kann von der globalen Schnittstelle angestoßen werden. Allerdings ist nun fest vorgegeben, wie der integrierte Aufruf aussieht. Der Benutzer kann nicht mehr ad hoc entscheiden, welche Daten er in einer Anweisung referenzieren möchte.

Die Bewertung der zweiten Alternative auf Basis einer generischen Prozedur sieht ähnlich aus. Sie hat zusätzlich den Vorteil, dass weniger Prozeduren erstellt werden. Andererseits ist die Entwicklung dieser Prozedur aufwendiger, da sie so gestaltet sein muss, dass sie alle föderierten Funktionen aufrufen kann. Hinsichtlich der Integration von Daten und Funktionen gelten dieselben Nachteile wie für die erste Variante. Die Handhabung einer solchen generischen Prozedur ist nicht intuitiv und für den Benutzer eher umständlich. Daher verfolgen wir diesen Ansatz nicht weiter und beschränken unsere Betrachtungen auf die (1:1)-Abbildung.

5.1.1.2 Benutzerdefinierte Tabellenfunktionen

Benutzerdefinierte Tabellenfunktionen erlauben die Erweiterung der DB-Funktionalität durch benutzerdefinierte Funktionen (vgl. Abschnitt 3.2.2). Sie können in SQL und in verschiedenen Programmiersprachen implementiert werden. Benutzerdefinierte Tabellenfunktionen liefern eine Tabelle als Ergebnis zurück und können wie eine Basistabelle in der FROM-Klausel der SELECT-Anweisung referenziert werden. Leider unterstützen Tabellenfunktionen nur den lesenden Zugriff und können daher nur in einer SELECT-Anweisung genutzt werden, jedoch nicht in UPDATE-, DELETE- oder INSERT-Anweisungen.

Vergleichbar zu den gespeicherten Prozeduren erfolgt hier ebenfalls eine (1:1)-Abbildung der Tabellenfunktionen auf die föderierten Funktionen der KIF. Folglich wird für jede föderierte Funktion eine korrespondierende Tabellenfunktion mit demselben Namen und denselben Parametern im FDBS angelegt. Für unser Beispiel definieren wir zunächst eine benutzerdefinierte Tabellenfunktion `GetBestand()`:

```
CREATE FUNCTION GetBestand (INTEGER, INTEGER)
   RETURNS TABLE (Lager INTEGER, Order INTEGER)
   SPECIFIC GetBestand_Java
   EXTERNAL NAME '/sql/procedures/Lager.GetBestand'
   DETERMINISTIC
   EXTERNAL ACTION
   LANGUAGE JAVA
   NO SQL
   PARAMETER STYLE DB2SQL
   FINAL CALL
   DISALLOW PARALLEL
```

Der Aufruf der Prozedur mit den Werten `1` und `13` für die Eingabeparameter `LiefNr` und `KompNr` erfolgt in der FROM-Klausel des SQL-Befehls:

```
SELECT *
FROM TABLE (GetBestand(1, 13)) AS GB
```

Ebenso kann der Weg über eine generische Tabellenfunktion gewählt werden, die parametergesteuert die entsprechende föderierte Funktion aufruft. Aus denselben Gründen, wie bei den Prozeduren aufgeführt, werden wir diese Möglichkeit nicht weiter verfolgen.

Im Gegensatz zum Ansatz auf Basis von gespeicherten Prozeduren ist nun die Integration von Daten und Funktionen möglich. Da die Tabellenfunktionen wie Basistabellen referenziert und genutzt werden können, sind vielfältige Kombinationen der integrierten Daten möglich. Beispielsweise kann ein Verbund über einer föderierten Funktion mit einer integrierten Tabelle aus einem anderen DBS spezifiziert werden. Die Handhabung ist bei der (1:1)-Abbildung einfach. Die föderierten Funktionen sind durch die Tabellenfunktionen für den Benutzer sichtbar und können ähnlich wie Basistabellen verwendet werden.

Von großem Nachteil ist jedoch der rein lesende Zugriff auf Tabellenfunktionen. Folglich kann man keine schreibenden Zugriffe auf die integrierten Datenquellen implementieren. Trotzdem stellen die Tabellenfunktionen eine einfache und schnell umzusetzende Lösung zur Anbindung von verschiedenartigen Datenquellen dar.

5.1.1.3 Wrapper

Die in SQL/MED definierten Wrapper [FCD02] erlauben eine sehr enge Kopplung von FDBS und KIF. Wrapper ermöglichen den Zugriff über die SQL-Schnittstelle auf Nicht-SQL-Systeme (vgl. Abschnitt 3.2.3.1). Daten in fremden Tabellen, die von fremden Servern verwaltet werden, können mit einem Wrapper dem FDBS bekannt und zugängig gemacht werden. So können alle Daten beliebigen Formats mit SQL-Befehlen abgerufen werden.

SQL/MED spezifiziert eine Schnittstelle bestehend aus mehreren Funktionen zwischen Wrapper und SQL-Server, d. h. in unserem Fall dem FDBS. Diese Schnittstelle erlaubt auch eine sehr enge Integration bei der gemeinsamen Verarbeitung der Anfragen. Somit kann SQL-Funktionalität, die nicht vom fremden Server unterstützt wird, durch das FDBS transparent für den Benutzer kompensiert werden.

Die Kopplung von FDBS und KIF erfordert einen KIF-spezifischen Wrapper, über den alle föderierten Funktionen integriert und aufgerufen werden können. Da die fremden Daten innerhalb des FDBS als (fremde) Tabellen dargestellt werden, müssen die Funktionen auf Tabellen und umgekehrt abgebildet werden. Dies kann in einer (1:1)-Abbildung erfolgen, in der wir für jede föderierte Funktion eine korrespondierende fremde Tabelle erstellen. Diese hat denselben Namen wie die Funktion und ihre Spalten entsprechen den Funktionsparametern. Für unsere Beispielfunktion `GetBestand(IN LiefNr, IN KompNr, OUT Anzahl, OUT Order)` ergibt sich folgende Definition für die zugehörige fremde Tabelle (mit der Annahme, dass bereits ein fremder Server `kif-server` angelegt wurde):

```
CREATE FOREIGN TABLE GetBestand (
   LiefNr INTEGER,
   KompNr INTEGER,
   Anzahl INTEGER,
   Order INTEGER)
   SERVER kif-server
```

Um die föderierte Funktion mit den Eingabeparametern auf- und das Ergebnis abzurufen, können wir beispielsweise folgenden SQL-Befehl einsetzen:

```
SELECT Lager, Order
FROM GetBestand
WHERE LiefNr=1 AND KompNr=13
```

Für den Benutzer ist nicht ersichtlich, ob er auf Daten im FDBS oder auf föderierte Funktionen zugreift. Außerdem ist der Gebrauch von fremden Tabellen absolut identisch zu dem von Basistabellen. Folglich sind beliebige Kombinationen von Tabellenreferenzen (beispielsweise in Verbunden) möglich.

Wrapper erlauben die engste Kopplung von FDBS und KIF und die transparente Integration von Funktionen. Der Benutzer erkennt nicht mehr, wann er über Funktionen auf Daten zugreift. Zudem ermöglicht diese Technologie die Erweiterung der KIF-Funktionalität, indem zusätzliche Funktionalität im Wrapper implementiert oder fehlende Funktionalität im FDBS kompensiert wird.

Kriterium	Gespeicherte Prozeduren	Tabellenfunktionen	Wrapper
Standard-Konformität	ja, SQL:1999	nein, vorgeschlagen für SQL:200x	ja, SQL/MED
Hersteller-Unterstützung	ja	ja	teilweise
Eigenimplementierung	gering	gering	aufwendig
Abbildung Tabelle – Funktionsaufruf	(1:1)-Abbildung Prozedur–Funktion	(1:1)-Abbildung Funktion–Funktion	(1:1)-Abbildung Tabelle–Funktion
Zugriffstransparenz	Funktion sichtbar	Funktion sichtbar	transparent über fremde Tabellen
Integrationsbreite	keine Kombination mit Daten	Daten und Funktionen	Daten und Funktionen
Integrationsgrad	gering	mittel	groß, inklusive globaler Anfrageverarbeitung
Schreibzugriff	ja	nein	zukünftig ja

Tabelle 5.2: Auswahlkriterien für den Kopplungsmechanismus.

Im Gegensatz dazu steht die komplexere und aufwendigere Implementierung von Wrappern. Da pro Quellentyp ein Wrapper benötigt wird, muss für die vorgestellte Integrationsarchitektur lediglich ein Wrapper implementiert werden. Da momentan die Wrapper-Schnittstelle standardisiert wird, ist zukünftig damit zu rechnen, dass alle führenden Datenbankhersteller die Wrapper-Technologie unterstützen werden. Die Entwicklung von eigenen Wrappern sollte dann durch die von den Herstellern gelieferten Entwicklungs-Kits deutlich einfacher und schneller sein. Wird zudem der Standard von den DB-Herstellern weitgehend eingehalten, ist auch der Austausch der Wrapper zwischen unterschiedlicher DB-Middleware mit geringem Aufwand möglich.

Wie in Abschnitt 3.2.3 beschrieben, legt der Standard derzeit nur den lesenden Zugriff fest. Jedoch ist für zukünftige Versionen auch der schreibende Zugriff vorgesehen, so dass eine voll funktionsfähige Anbindung unterstützt werden kann.

5.1.1.4 Auswahl des Kopplungsmechanismus

Für die verbleibenden Betrachtungen des Ausführungsmodells wählen wir eine der vorgestellten Kopplungsmöglichkeiten aus. Tabelle 5.2 gibt einen Überblick über die einzelnen Auswahlkriterien und deren Unterstützung durch die einzelnen Mechanismen.

Die gespeicherten Prozeduren werden zwar von allen führenden Herstellern angeboten, unterstützen aber keinen integrierten Zugriff auf Daten und Funktionen. Damit stellen sie keine echte Alternative als Kopplungsmechanismus dar. Bei ihrem Einsatz müsste die eigentliche Integration von Daten und Funktionen in der Anwendung erfolgen, was wir auf jeden Fall verhindern wollen. Nichtsdestotrotz unterstützen sie derzeit als einzige Lösung schreibenden Zugriff und können für diesen Zweck wieder interessant werden.

Schreibende Zugriffe (Insert, Update, Delete) beziehen sich nämlich immer auf genau eine Tabelle. Daher ist es kein Nachteil mehr, dass man mit einer gespeicherten Prozedur nur auf eine Funktion zugreifen kann. Eine Schachtelung der Prozeduren, wie wir sie in Abschnitt 5.1.1.1 beschrieben haben, darf bei schreibenden Zugriffen nicht umgesetzt werden.

Benutzerdefinierte Tabellenfunktionen werden ebenfalls von den meisten DB-Herstellern angeboten, sind aber in der Form momentan nicht standardisiert. Ihre Implementierung ist wenig aufwendig. Außerdem können Daten und Funktionen integriert werden, da man Tabellenfunktionen wie Basistabellen in SELECT-Anweisungen einsetzen kann. Von großem Nachteil ist jedoch die fehlende Unterstützung des schreibenden Zugriffs. Da dieser auch zukünftig nicht zu erwarten ist, sind auch Tabellenfunktionen nicht die geeignete Variante zur Kopplung von FDBS und KIF.

Im Gegensatz zu den anderen Alternativen werden Wrapper kaum in Produkten unterstützt und sind somit der Exot der Kopplungsmechanismen. Ebenso ist der Aufwand für die eigene Implementierung am höchsten einzuschätzen. Dem steht jedoch eine deutliche Zugriffstransparenz und optimale Integrationsbreite gegenüber. Funktionen können in Form von fremden Tabellen genauso wie Basistabellen behandelt werden. Dadurch ist die Integration von Daten und Funktionen möglich. Durch die enge Kopplung über den Wrapper kann auch die globale Anfrageverarbeitung hinsichtlich der Anfrageoptimierung maßgeblich unterstützt werden. Da der schreibende Zugriff erst für zukünftige Versionen des SQL/MED-Standards geplant ist, können gespeicherte Prozeduren vorübergehend für Schreibfunktionen eingesetzt werden. Wir erwarten jedoch in naher Zukunft die Unterstützung des Schreibzugriffs durch die DB-Hersteller, da bereits in der Vergangenheit ähnliche Funktionalität angeboten wurde.

In den folgenden Abschnitten legen wir den Wrapper-Ansatz als Kopplungsmechanismus zugrunde und betrachten die verschiedenen Aspekte der Anfrageverarbeitung auf Basis von Wrappern.

5.1.2 Erforderliche Kernfunktionalität des Kopplungsmechanismus

Der Wrapper muss mehrere grundlegende Funktionen anbieten, um eine enge Kopplung von FDBS und KIF umzusetzen. Die beiden Hauptaufgaben betreffen das Zusammenspiel von FDBS, Wrapper und KIF bei der systemübergreifenden Anfrageverarbeitung und die Abbildung von Tabellen auf Funktionen. Bei der Abbildung unterscheiden wir zwischen struktureller und semantischer Abbildung.

5.1.2.1 Zusammenspiel in der Anfrageverarbeitung

Das Zusammenspiel aller beteiligten Komponenten erfordert den Verbindungsaufbau zwischen FDBS, Wrapper und KIF. Der Wrapper muss hierfür die SQL/MED-Schnittstelle zum FDBS hin und die KIF-spezifische Schnittstelle unterstützen. Über die entsprechenden Routinen initiiert das FDBS eine Verbindung zum Wrapper (vgl. Abschnitt 3.2.3.2). Nach Verbindungsaufbau verhandelt der Wrapper in der Anfrageplanungsphase mit dem FDBS, welcher Anteil des angeforderten Anfragefragments zusammen mit der KIF ausgeführt werden kann. Dies kann sich auf die KIF-Funktionalität zur

Ausführung von föderierten Funktionen beschränken oder durch zusätzlich unterstützte Operationen im Wrapper erweitert werden.

In der anschließenden Ausführungsphase ermittelt der Wrapper, welche Funktionsaufrufe notwendig sind. Er stellt die Verbindung zur KIF her, initiiert die Ausführung der föderierten Funktionen und greift die Ergebnisse ab. Abhängig von der Wrapper-Funktionalität werden diese Ergebnisse gegebenenfalls weiterbearbeitet und schließlich an das FDBS zurückgeliefert.

Um jedoch zu wissen, welche Funktionen der Wrapper aufrufen muss, ist eine Abbildung zwischen den referenzierten Tabellen im FDBS und den zugehörigen Funktionen notwendig. Diese Abbildung beschreiben wir im Folgenden hinsichtlich Struktur und Semantik.

5.1.2.2 Strukturelle Abbildung

Die strukturelle Abbildung zwischen Tabellen und Funktionen beschreibt die Darstellung einer Funktionssignatur in Form einer Tabelle, die in SQL-Anweisungen referenziert werden kann. Diese Abbildung muss die konzeptionellen Unterschiede zwischen FDBS und KIF überbrücken. Während das FDBS auf einem relationalen Datenmodell basiert, baut die KIF auf einem funktionalen Modell auf. Der Wrapper muss daher den Zugang zur KIF über Tabellen im FDBS ermöglichen. Diese Tabellen sind *abstrakte Tabellen*, die selbst keine Daten enthalten. Sie dienen als Referenz auf Daten in externen Systemen und werden im SQL/MED-Standard fremde Tabellen genannt (vgl. Abschnitt 3.2.3.1).

Eine denkbare kanonische Abbildung stellt jede Funktion als eine Zeile ihrer Ein- und Ausgabewerte dar. Diese Struktur kann wiederum als Tabelle beschrieben werden. Folglich wird unsere Beispielfunktion `GetBestand(IN LiefNr, IN KompNr, OUT Lager, OUT Order)` wie in Abschnitt 5.1.1.3 als eine abstrakte Tabelle mit den Spalten `LiefNr`, `KompNr`, `Lager` und `Order` dargestellt.

Setzen wir dies als eine allgemeingültige Abbildung voraus, wird im FDBS für jede zu integrierende (föderierte) Funktion eine abstrakte bzw. fremde Tabelle angelegt. Diese werden in SQL-Befehlen referenziert und mit Hilfe des Wrappers auf die zugehörigen Funktionen abgebildet.

Die Funktionalität der KIF unterstützt jedoch nur einen eingeschränkten Zugriff auf die fremden Tabellen. Die Ausführung einer Funktion entspricht nämlich genau einer SQL-Anweisung, in welcher die Ausgabeparameter projiziert und auf Basis der Eingabeparameterwerte selektiert werden. Somit entspricht ein Funktionsaufruf dem folgenden SELECT-Befehl:

```
SELECT Lager, Order
FROM GetBestand
WHERE LiefNr=wert1 AND KompNr=wert2
```

Daher ist neben der strukturellen Abbildung eine semantische Abbildung notwendig, die wir im nächsten Abschnitt erläutern.

5.1.2.3 Semantische Abbildung

Die semantische Abbildung zwischen Tabellen und Funktionen beschreibt die Darstellung der Funktionssemantik durch eine SQL-Anweisung. Diese Abbildung kann jedoch nicht automatisch von der Funktionssignatur abgeleitet werden, da sie nicht genug Informationen über die zugrunde liegende Semantik liefert.

Die meisten der zu integrierenden Funktionen werden von Legacy-Systemen bereitgestellt, die eine hierarchische Sicht auf ihre Daten unterstützen. Viele Funktionen benutzen die Eingabewerte für Vergleichsprädikate mit einer „=“-Operation. Dies gilt auch für unsere Beispielfunktion, welche die verfügbare Anzahl einer bestimmten Komponente eines bestimmten Lieferanten ermittelt: `SELECT Anzahl FROM GetBestand WHERE LiefNr=1 AND KompNr=13`. Die Funktionalität dieser Legacy-Systeme ist eher starr und daher ungeeignet zur Unterstützung von SQL-Befehlen. Denn mit SQL können auch Anfragen über Wertebereiche formuliert werden, wie z. B. `SELECT Anzahl FROM GetBestand WHERE LiefNr=1 AND KompNr<13`. Offensichtlich muss in solchen Fällen die Semantik des „<“-Operators durch mehrere kompensierende Funktionsaufrufe reproduziert werden.

Ein weiterer Aspekt ist die Verknüpfung der Prädikate. Bei den Funktionen liegt im Allgemeinen eine logische UND-Verknüpfung vor, während SQL auch ODER-Verknüpfungen erlaubt.

Folglich muss der Wrapper nicht nur die Funktionssignatur auf eine Tabelle abbilden können, sondern muss auch die Semantik der SQL-Befehle mit der Funktionssemantik abgleichen. Gegebenenfalls müssen hierzu für eine SQL-Anweisung mehrere Funktionsaufrufe mit unterschiedlichen Eingabewerten abgesetzt werden.

5.2 Optimierung

Dieses Kapitel geht auf die Anfrageverarbeitung der Integrationsarchitektur ein. Dabei untersuchen wir, wie die Verarbeitung über die beiden Komponenten hinweg unterstützt werden kann und welche Rolle der Wrapper als Kopplungsmechanismus übernimmt. Neben der Betrachtung der Funktionalität des Wrappers entwickeln wir ein Kostenmodell zur Abschätzung der Kosten von Ausführungsplänen, die Funktionsaufrufe enthalten.

5.2.1 Gesamtkonzept der Anfrageverarbeitung

Bevor wir in den nächsten Abschnitten die verschiedenen Optimierungsmöglichkeiten im Detail erläutern, geben wir zunächst einen Überblick über das Gesamtkonzept der Anfrageverarbeitung in unserer Integrationsarchitektur und den Verarbeitungsablauf.

Die Verarbeitung der globalen Anfragen erfolgt über mehrere Komponenten hinweg. Dazu gehören

- das FDBS;
- die Wrapper für die zu integrierenden Datenbanksysteme;
- die Datenbanksysteme selbst;

- der Wrapper für die KIF;
- die KIF und
- die Anwendungssysteme.

Alle beteiligten Komponenten übernehmen Teile der Anfrageverarbeitung und müssen sich untereinander abstimmen. Die verteilte Anfrageverarbeitung über heterogene DBS hinweg wurde bereits in mehreren Arbeiten betrachtet [RS97, JS+02]. Daher konzentrieren wir uns auf die Verarbeitung der Funktionsaufrufe. Somit erstreckt sich die zu betrachtende Anfrageverarbeitung über das FDBS, die KIF und den dazwischen liegenden Wrapper.

Abbildung 5.1 veranschaulicht den Ablauf einer globalen Anfrage. Sie wird über die SQL-Schnittstelle an das FDBS gerichtet, das zunächst die Anweisung lexikalisch und syntaktisch analysiert (Schritt 1). Bei der semantischen Analyse überprüft das FDBS anhand des Systemkatalogs die Existenz der referenzierten DB-Objekte und stellt fest, in welchem integrierten System diese Objekte vorliegen. Wurden Zugriffsrechte und Integrität kontrolliert, zerlegt das FDBS die globale Anfrage in Teilanfragen, die von den lokalen Systemen bearbeitet werden. Dabei spielt die vorhandene Funktionalität der lokalen Systeme eine wichtige Rolle, da sie bestimmt, welche Operationen zu den Systemen weitergereicht werden können (*push down*). Operationen, die nicht in den lokalen Systemen verarbeitet werden können, kompensiert das FDBS.

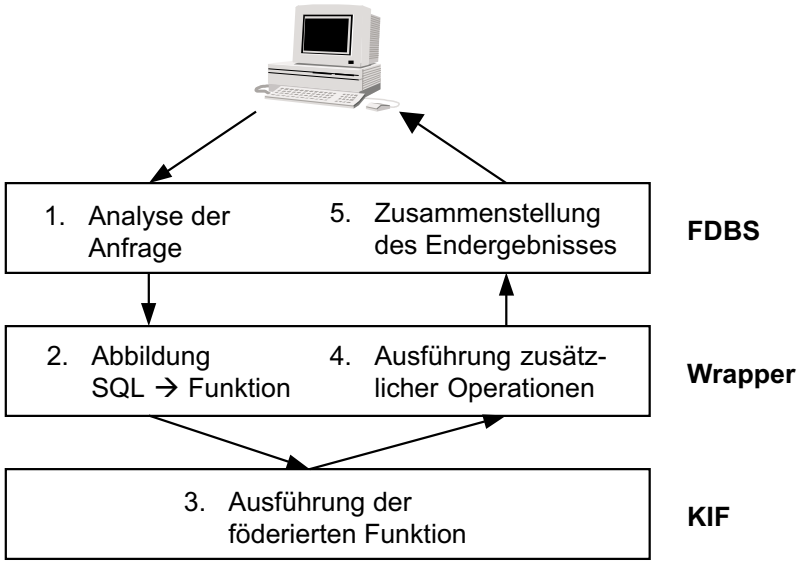

Abbildung 5.1: Die einzelnen Schritte der Anfrageverarbeitung in den Komponenten.

Erkennt das FDBS eine Referenz auf föderierte Funktionen, legt es mit dem Wrapper fest, welcher Teil der Anfrage von Wrapper und KIF übernommen werden kann. Anschließend bildet der Wrapper die SQL-Teilanfrage auf die entsprechenden Funktionsaufrufe ab (Schritt 2).

Die KIF führt die föderierte Funktion aus, indem sie die dazu notwendigen lokalen Funktionen aufruft (Schritt 3). Da die Abbildung von föderierten Funktionen auf lokale Funktionen fest vorgegeben ist und damit keine Optimierung stattfinden kann, gehen wir auf diesen Aspekt nicht weiter ein.

Nach der Funktionsausführung wird das Ergebnis an den Wrapper zurückgeliefert, der gegebenenfalls zusätzliche Operationen auf den Daten ausführt (Schritt 4).

Abschließend wird das Zwischenergebnis an das FDBS zurückgegeben, das eventuell weitere Operationen auf den Daten ausführt und das Endergebnis zusammenstellt.

Möglichkeiten zur Optimierung bestehen in der gewählten Position des Wrappers und dessen vorhandene Funktionalität. Welche Operationen im Wrapper unterstützt werden sollten und wie sich dies auswirkt, betrachten wir in den folgenden Abschnitten. Bietet der Wrapper zusätzliche Funktionalität an, so kommt diese jedoch nur zur Geltung, wenn der Wrapper nicht auf Seiten des FDBS liegt, sondern auf dem Rechnerknoten der lokalen Systeme installiert ist. Abbildung 5.2 verdeutlicht die Vorteile. Arbeitet der Wrapper auf der Seite des FDBS (links), so liefert der Wrapper zwar eine kleine Datenmenge an das FDBS, da jedoch beide auf einem Rechnerknoten liegen, wirkt sich dies nicht besonders aus. Den größeren Effekt hat die Übermittlung der großen Datenmenge zwischen KIF und Wrapper, die auf verschiedenen Rechnerknoten liegen. Da insbesondere der Wrapper die datenreduzierenden Operationen anbietet, sind die zu transportierenden Mengen groß.

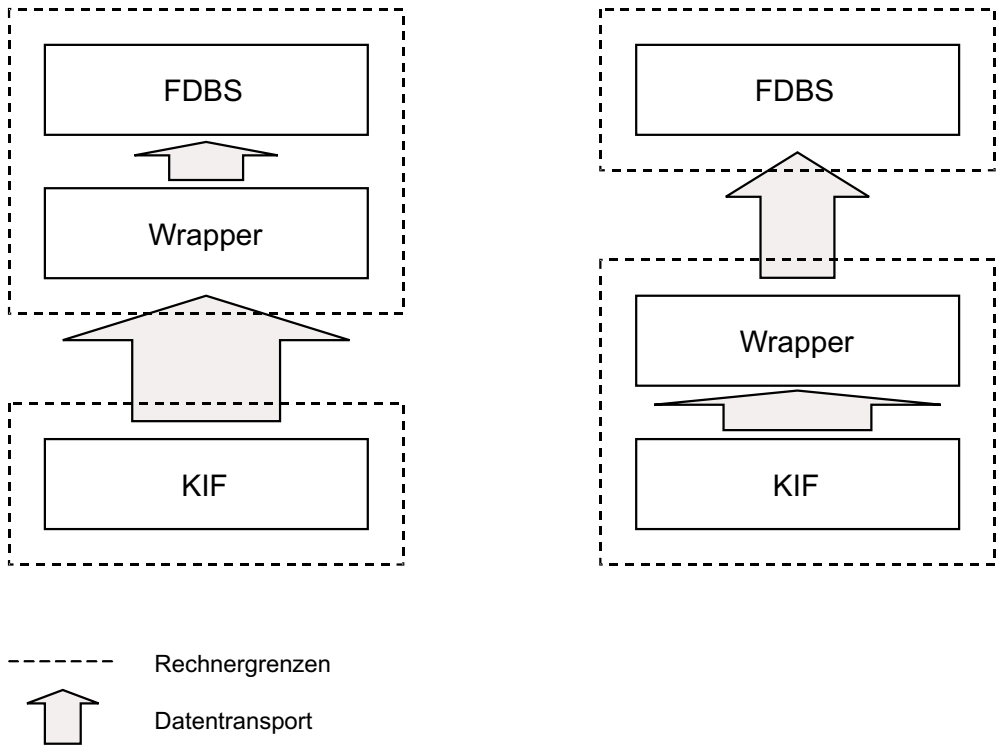

Abbildung 5.2: Mögliche Positionen des Wrappers: auf einem Rechner mit dem FDBS (links) oder der KIF (rechts).

Ganz anders sieht es aus, wenn der Wrapper auf einem Rechnerknoten mit der KIF sitzt (Abbildung 5.2 rechts). In diesem Fall werden die größeren Datenmengen in demselben Rechner übermittelt und fallen nicht stark ins Gewicht. Nachdem der Wrapper seine Operationen durchgeführt und die Datenmenge weiter reduziert hat, erfolgt der kostspieligere Transport über das Netzwerk zum FDBS. Somit sollte der Wrapper immer auf dem Rechnerknoten des zu integrierenden Systems installiert werden.

5.2.2 Klassifizierung der Wrapper-Mächtigkeit

Die im Wrapper unterstützte Funktionalität kann die Anfrageoptimierung maßgeblich beeinflussen. Je mehr der Wrapper an Operationen anbietet, die über die Funktionalität der föderierten Funktionen hinausgehen, desto weniger Daten müssen im Allgemeinen von Wrapper zu FDBS übertragen werden. So kann besonders an den Kommunikationskosten gespart werden.

Im Folgenden untersuchen wir, wie eine sinnvolle Erweiterung der Wrapper-Mächtigkeit aussehen kann. Wir unterteilen die zu unterstützenden Operationen in Kern-, Basis- und erweiterte Funktionalität [HH01, HBH01]. Während Kern- und Basisfunktionalität unseres Erachtens auf jeden Fall implementiert werden sollten, sind die Operationen der erweiterten Funktionalität als optional und hilfreich, aber nicht zwingend notwendig zu betrachten. Welche Operationen den einzelnen Klassen zugeteilt werden und wie sich deren Unterstützung auf die globale Anfrageverarbeitung auswirkt, beschreiben wir in den nächsten Abschnitten.

5.2.2.1 Kernfunktionalität

Unter Kernfunktionalität fassen wir all jene Aufgaben zusammen, die wir bereits in Abschnitt 5.1.2 erläutert haben. Sie sind zwingend notwendig, um eine Verbindung zwischen FDBS und KIF herzustellen und eine komponentenübergreifende Anfrageverarbeitung zu ermöglichen. Dazu gehört insbesondere die Entgegennahme einer Teilanfrage seitens des FDBS und deren Abbildung auf Funktionen.

5.2.2.2 Basisfunktionalität

Die Kernfunktionalität unterstützt nur jene SQL-Anweisung, die auf genau eine Funktion abgebildet werden kann, d. h. die Projektion der Ausgabeparameter und die für die Eingabeparameter notwendigen Selektionsprädikate. Alle anderen Operationen werden im Wrapper nicht unterstützt und müssen daher im FDBS ausgeführt werden.

Um eine Optimierung der Anfrage zu ermöglichen, sollte der Wrapper zumindest Selektion und Projektion generell unterstützen. Projektionen sollten zusätzlich Eingabeparameter und Selektionen Ausgabeparameter einschließen. Die Selektion kann aufgrund beliebiger Boolescher Kombinationen von Vergleichsprädikaten mit Konstanten erfolgen. Der Vergleich von zwei Attributen bzw. Parameterwerten wird jedoch nicht unterstützt.

Wie kann eine solche Erweiterung realisiert werden? Dies geschieht in erster Linie mit kompensierenden Funktionsaufrufen, d. h., der Wrapper muss mehrere Funktionen aufrufen, um die benötigten Daten zu erhalten. Anschließend können die Operationen durchgeführt und das Ergebnis an das FDBS zurückgeliefert werden.

Da Selektion und Projektion die Datenmenge im Allgemeinen erheblich verringern, ist ein Pushdown dieser Operationen zum Wrapper ein wichtiger Optimierungsschritt in der Anfrageverarbeitung.

5.2.2.3 Erweiterte Funktionalität

Die erweiterte Funktionalität schließlich beinhaltet Operationen, die nicht zwingend im Wrapper vorhanden sein müssen, aber gegebenenfalls eine weitergehende Optimierung der Anfragen ermöglichen. Zu diesen Operationen gehören Gruppierung und Aggregation, Teilanfragen bezüglich abstrakter Tabellen sowie Mengenvergleiche. Außerdem ist die Implementierung der Verbundoperation zweier abstrakter Tabellen innerhalb des Wrappers hilfreich, um die Kommunikation, verursacht durch Wrapper-Aufrufe seitens des FDBS, zu minimieren. Andererseits werden Verbunde im FDBS sehr effizient umgesetzt, so dass kritisch abgewogen werden sollte, ob eine Reimplementierung des Verbundes gerechtfertigt ist.

Die Unterstützung der genannten Operationen dient vor allem dazu, die zu übertragende Datenmenge und damit die Kommunikationskosten weiter zu reduzieren. Werden alle aufgeführten Operationen implementiert, ist die vom Wrapper angebotene Schnittstelle einer SQL-Schnittstelle sehr ähnlich und erleichtert die globale Anfrageverarbeitung im FDBS erheblich. Es wird nicht nur die Datenmenge beachtlich reduziert, es wird auch die Anzahl an Wrapper-Aufrufen stark verringert.

5.2.2.4 Weitere Aspekte

Es gibt weitere Operationen, von deren Implementierung im Wrapper wir abraten. Da der Wrapper nur Teilanfragen erhält, die abstrakte Tabellen referenzieren, besteht keine Notwendigkeit, Funktionalität zu implementieren, die über die Verarbeitung dieser Teilanfragen hinausgeht. Dazu gehören Operationen wie Vereinigung, Differenz und Durchschnitt. Separate Teilanfragen werden normalerweise durch unabhängige Anfragen repräsentiert, die anschließend durch das FDBS zusammengesetzt werden. Teilanfragen bezüglich Basistabellen im FDBS sollten nicht durch den Wrapper ausgeführt werden, da das FDBS diese Funktionen in der Regel besser handhaben kann.

Der Transfer der Ergebnismenge von Wrapper zu FDBS wird mittels einer abstrakten Tabelle vorgenommen. Dieser Transfer lässt sich entweder im Pipeline-Modus nach Verfügbarkeit von Ergebniszeilen oder im Blockmodus am Ende des Funktionsaufrufes bereitstellen. Außerdem können mit der Ergebnismenge hilfreiche Informationen zurückgeliefert werden, die für die globale Anfrageverarbeitung wichtig sind. Ein Beispiel hierfür wäre ein Vermerk über eine in der Ergebnismenge enthaltene Sortierreihenfolge. Solche Informationen helfen der Optimierung im FDBS.

5.2.3 Optimierungsansätze

Als Nächstes erläutern wir die Umsetzung der Operationen der Basis- und erweiterten Funktionalität. Dazu führen wir zunächst einige Definitionen ein und beschreiben anschließend die Abbildung von relationalen Operatoren auf die föderierten Funktionen innerhalb des Wrappers.

5.2.3.1 Grundlegende Definitionen und Voraussetzungen

Zur formalen Darstellung der Optimierungsansätze greifen wir auf die relationale Algebra zurück, die wir um zwei Operatoren erweitern.

Für eine Funktionssignatur $f(i_1, \ldots, i_m, o_1, \ldots, o_n)$ definieren wir zunächst $I := \{i_k | k = 1, \ldots, m\}$ und $O := \{o_l | l = 1, \ldots, n\}$ als disjunktive Mengen, die jeweils die Eingabe- und Ausgabeparameter enthalten[1]. Eine Funktion kann demnach auch verkürzt als $f(I, O)$ dargestellt werden.

In Abschnitt 5.1.2.2 haben wir die strukturelle Abbildung zwischen Funktionen und Tabellen bzw. Relationen beschrieben. Die kanonische Abbildung repräsentiert die Funktion $f(i_1, \ldots, i_m, o_1, \ldots, o_n)$ als eine abstrakte Tabelle $T_a(i_1, \ldots, i_m, o_1, \ldots, o_n)$. Da die Eingabe- und Ausgabeparameter auf die Spalten der Tabelle abgebildet werden, können wir die Tabelle auch verkürzt mit $T_a(I, O)$ beschreiben.

Abschnitt 5.1.2.3 hat gezeigt, dass bei der gewählten kanonischen Abbildung ein Funktionsaufruf einem SQL-Befehl mit Projektion auf die Ausgabeparameter und Selektion der Eingabeparameter entspricht. Folglich werden nur Spalten zurückgegeben, welche die Ausgabewerte enthalten. Somit können wir den Funktionsaufruf wie folgt mit algebraischen Operatoren darstellen:

$$f(I, O) \leftrightarrow \pi_O(\sigma_I(T_a))$$

Diese Funktionalität entspricht exakt der Funktionalität der KIF. Durch die Bereitstellung von föderierten Funktionen kann sie genau diese Operationen auf den abstrakten Tabellen ausführen. Alles, was darüber hinaus unterstützt werden soll, muss der Wrapper oder das FDBS anbieten. Wir führen zur einfacheren Darstellung einen neuen Operator φ für die KIF-Funktionalität ein:

$$\varphi(T_a) = \pi_O(\sigma_I(T_a))$$

Um die „SELECT *“-Funktionalität, d. h. die Projektion aller Attribute zu implementieren, müssen nach Erhalt der Ausgabewerte von der KIF die korrespondierenden Eingabewerte konkateniert werden. Für diese Operation führen wir den Operator κ ein:

$$\kappa_I(\varphi(T_a)) := I||\varphi(T_a) = I||\pi_O(\sigma_I(T_a)) = \pi_{I,O}(\sigma_I(T_a)) = \sigma_I(T_a)$$

Es ist unser Ziel, die föderierten Funktionen so anzubinden, dass die Funktionsaufrufe und die damit einhergehenden Einschränkungen vollkommen transparent für den Benutzer sind. Er soll in der Lage sein, die vollständige Mächtigkeit von SQL nutzen zu können – für Basis- und abstrakte Tabellen. Folglich muss jeder SQL-Befehl in einen oder mehrere Funktionsaufrufe umgesetzt werden. Um die notwendigen Funktionen bestimmen zu können, sind folgende Voraussetzungen zu berücksichtigen:

- Die Kardinalität $K(T_a, i)$ jedes Eingabeparameters i einer abstrakten Tabelle T_a muss endlich sein

[1] Funktionsparameter, die als Eingabe- und Ausgabeparameter (`INOUT`) definiert sind, werden in beiden Mengen aufgeführt.

- Die Elemente seines Wertebereichs $W(T_a, i)$ müssen aufzählbar sein.

Warum benötigen wir diese Informationen? Wir haben gelernt, dass eine Funktion in den meisten Fällen einen oder mehrere Eingabewerte verarbeitet. Bei der Abbildung von Tabellen auf Funktionen werden diese Werte über die Selektion in der WHERE-Klausel definiert. Doch was passiert, wenn diese Werte teilweise oder vollständig fehlen? Dann muss man die Funktion für jeden möglichen Wert aufrufen. Hierzu sollten es aber nicht unendlich viele Werte sein und man sollte sie kennen. Genau dies sind die geforderten Voraussetzungen.

Mit den vorgestellten Operatoren und Bedingungen untersuchen wir in den nächsten Abschnitten die verschiedenen Funktionalitätsklassen und beschreiben ihre mögliche Umsetzung.

5.2.3.2 Umsetzung der Basisfunktionalität

Es werden nun die Selektion und Projektion eingehend hinsichtlich ihrer Umsetzung auf Basis von Funktionsaufrufen diskutiert. Dabei muss jeweils zwischen Selektion bzw. Projektion auf Eingabe- und Ausgabeparameter unterschieden werden. Eingabewerte können aufgrund der in Abschnitt 5.2.3.1 dargestellten Voraussetzungen selbst ermittelt werden, während sich die Werte der Ausgabeparameter nur über Funktionsaufrufe feststellen lassen. Somit ist bei diesen beiden Fällen unterschiedlich vorzugehen.

Bei der Selektion muss weiterhin hinsichtlich der Semantik der Vergleichsprädikate der SQL-Anweisung und der Semantik der Funktion unterschieden werden. Ein einfaches Beispiel hierfür ist ein SQL-Befehl mit einem „$\leq$“-Prädikat, während die Funktion den Wert des Eingabeparameters mit einer „=“-Operation verarbeitet. In den Beispielen beziehen wir uns stets auf die Funktion `GetBestand()`, bei der die Eingabeparameter `LiefNr` und `KompNr` eine „=“-Semantik nachbilden. Wie zuvor erläutert, gehen wir bei unseren Beispielen davon aus, dass die Kardinalitäten $K(T_{GetBestand}, LiefNr)$ und $K(T_{GetBestand}, KompNr)$ bekannt und die Elemente des jeweiligen Wertebereichs $W(T_{GetBestand}, LiefNr)$ und $W(T_{GetBestand}, KompNr)$ aufzählbar sind (vgl. Tabelle 5.1). Andernfalls können kompensierende Funktionsaufrufe nicht ermittelt werden.

5.2.3.3 Selektion

In Abschnitt 5.1.2.3 haben wir bereits beschrieben, dass ein Funktionsaufruf einer SQL-Anweisung entspricht, in der Selektionen für alle Eingabeparameter (mit „=“) spezifiziert sind. Darüber hinaus sind weitere Aspekte der Selektion möglich, die wir folgendermaßen aufteilen:

1. Selektion auf Basis von Eingabeparametern.
2. Selektion auf Basis von Ausgabeparametern.
3. Unterschiedliche Semantik der Vergleichsprädikate.

In den ersten beiden Fällen gehen wir davon aus, dass die Semantik des Vergleichsprädikates in der SQL-Anweisung mit der Semantik der Funktion übereinstimmt. Für

alle drei Fälle zeigen wir auf, wie SQL-Anweisungen auf die verfügbaren Funktionen abgebildet werden, um die Flexibilität und Offenheit von SQL zu unterstützen. Die Abbildung stützt sich auf kompensierende Funktionsaufrufe.

Selektion auf Basis von Eingabeparametern

Spezifiziert man alle benötigten Eingabewerte in der WHERE-Klausel, reicht genau ein Funktionsaufruf, an dessen Ausgabewerte die korrespondierenden Eingabewerte konkateniert werden:

$$\sigma_I(T_a) = \kappa_I(\varphi(T_a))$$

Projiziert man darüber hinaus nur auf die Ausgabewerte, dann entspricht dies exakt einem Funktionsaufruf ohne weitere Verarbeitung der Ergebniswerte:

$$\pi_O(\sigma_I(T_a)) = \pi_O(\kappa_I(\varphi(T_a))) = \pi_O(\pi_{I,O}(\sigma_I(T_a))) = \pi_O(\sigma_I(T_a)) = \varphi(T_a)$$

Das folgende SQL-Beispiel verdeutlich dies:

Beispiel:

Die SQL-Anweisung

```
SELECT Lager, Order
FROM GetBestand
WHERE LiefNr=1 AND KompNr=13
```

wird auf genau einen Funktionsaufruf abgebildet:

```
GetBestand(1, 13, :Lager, :Order)
```

Fehlt jedoch der Wert eines Eingabeparameters, müssen kompensierende Funktionsaufrufe ermittelt werden. Für jedes Element der Wertemenge des fehlenden Eingabewertes muss eine Funktion hinzugefügt werden. Die restlichen bekannten Eingabewerte bleiben dieselben. Folglich wird die Anzahl der Aufrufe multipliziert mit der Kardinalität jedes unspezifizierten Eingabewertes. Unser Beispiel verdeutlicht diesen Aspekt.

Beispiel:

Der SQL-Befehl

```
SELECT Lager, Order
FROM GetBestand
WHERE LiefNr=1
```

spezifiziert nur den Eingabewert für den Parameter `LiefNr`, während der Parameter `KompNr` unspezifiziert bleibt. Die Wertemenge von `KompNr` sei beispielsweise $W(GetBestand, KompNr) = \{11, 12, 13\}$; somit sind drei kompensierende Funktionen nötig:

```
GetBestand(1, 11, :Lager, :Order)
GetBestand(1, 12, :Lager, :Order)
GetBestand(1, 13, :Lager, :Order)
```

Allgemein ergibt sich die folgende Anzahl an kompensierenden Funktionsaufrufen für Selektionen auf Basis von Eingabeparametern:

$$fc_{input\ selection} = \prod_{m \in S} 1 \times \prod_{n \in U} K(T_a, n) = \prod_{n \in U} K(T_a, n)$$

Dabei enthält die Menge S all jene Eingabeparameter, deren Wert spezifiziert ist und die Menge U beschreibt alle unspezifizierten Eingabeparameter.

Selektion auf Basis von Ausgabeparametern

Erfolgen die Selektionen ausschließlich auf Basis der Ausgabeparameter, müssen alle Daten der abstrakten Tabelle mit kompensierenden Funktionsaufrufen ausgelesen werden. Dies entspricht einem „Table Scan“. Da keiner der Eingabewerte vorgegeben ist, müssen alle Kombinationen der Eingabeparameter ermittelt und entsprechend viele Funktionen aufgerufen werden. Die Anzahl an Funktionsaufrufen entspricht dem Produkt der Kardinalitäten der Eingabeparameter.

$$fc_{table\ scan} = \prod_{n \in I} K(T_a, n)$$

Anschließend wird die Selektion auf dem Ergebnis im Wrapper ausgeführt.

Beispiel:

Die SQL-Anweisung

```
SELECT LiefNr, KompNr, Order
FROM GetBestand
WHERE Lager<3
```

ermittelt die bestellte Anzahl derjenigen Komponenten, deren aktueller Lagerbestand weniger als 3 Stück beträgt.

Da keine Selektion auf Basis von Eingabeparametern vorliegt, muss zunächst ein „Table Scan“ durch kompensierende Funktionsaufrufe nachgebildet werden, d. h., es werden Funktionsaufrufe mit allen Kombinationen der Eingabewerte ausgeführt. Anschließend wird im Wrapper das Zwischenergebnis aufgrund der Werte von `Lager` selektiert. Die Ergebnismenge sieht wie folgt aus (vgl. Tabelle 5.1):

Lieferant	Komponente	Bestellte Anzahl
2	11	10
2	13	15

Wird auch auf Eingabeparametern selektiert, reduziert sich die Anzahl an Funktionsaufrufen wie zuvor beschrieben.

Unterschiedliche Semantik der Vergleichsprädikate

Ähnliche Aspekte muss man betrachten, wenn der Vergleichsoperator eines Prädikates nicht demjenigen entspricht, den die Funktion implementiert. Eine Funktion berechnet im Allgemeinen das Prädikat ($n = Konstante$), während eine SQL-Anweisung auch andere Prädikate, z. B. ($n \leq Konstante$), spezifizieren kann. In solchen Fällen sind mehrere Funktionsaufrufe notwendig, um den Operator nachzubilden. Hier sind erneut die in Abschnitt 5.2.3.1 aufgeführten Restriktionen zu beachten, um die erforderlichen Funktionsaufrufe zu ermitteln. Neben dem Funktionsaufruf für ($n = Konstante$) sind weitere Aufrufe für jedes Element aus der Wertemenge von n auszuführen, das kleiner als die spezifizierte Konstante ist.

Beispiel:

Die SQL-Anweisung

```
SELECT Lager
FROM GetBestand
WHERE LiefNr=1 AND KompNr<=12
```

muss auf zwei Funktionsaufrufe abgebildet werden. Für `KompNr` sind dabei alle Elemente der zugehörigen Wertemenge zu wählen, die kleiner oder gleich 12 sind:

```
GetBestand(1, 11, :Lager, :Order)
GetBestand(1, 12, :Lager, :Order)
```

5.2.3.4 Projektion

Bei der Projektion muss ebenfalls zwischen der Anwendung auf Eingabe- oder Ausgabeparametern und deren Kombination unterschieden werden.

Projektion auf Basis von Eingabeparametern

Findet die Projektion auf Basis der Eingabewerte statt, kann das Ergebnis in bestimmten Fällen ohne Funktionsaufruf ermittelt werden. Dazu prüft man zunächst, ob die spezifizierten Eingabewerte in der Wertemenge der Parameter enthalten ist. Dazu vergleicht man die gegebenen Werte für die Eingabeparameter i mit den zugehörigen Wertemengen $W(T_a, i)$. Sobald einer der Eingabewerte nicht in der zugehörigen Wertemenge enthalten ist, ist auch die Ergebnismenge der SQL-Anweisung leer, da der Funktionsaufruf eine leere Ausgabemenge zurückliefern würde. Im anderen Fall erfolgen Funktionsaufrufe. Unser Beispiel veranschaulicht das Vorgehen.

Beispiel:

Es wird die folgende SQL-Anweisung angesetzt:

```
SELECT LiefNr
FROM GetBestand
WHERE KompNr=19
```

Der Vergleich mit der Wertemenge für `KompNr` ergibt, dass 19 nicht existiert. Folglich wird ein Funktionsaufruf mit diesem Eingabewert nur die leere Menge als Ergebnis liefern. Daher kann ohne Funktionsaufruf die leere Menge als Resultat zurückgeliefert werden.

Betrachten wir nun folgende Anweisung:

```
SELECT LiefNr
FROM GetBestand
WHERE KompNr=12
```

Da 12 in der Wertemenge enthalten ist, muss ein Funktionsaufruf erfolgen. Da der Eingabewert für `LiefNr` nicht vorgegeben ist, müssen die kompensierenden Funktionsaufrufe ermittelt werden. Somit ergeben sich folgende Funktionsaufrufe:

```
GetBestand(1, 12, :Lager, :Order)
GetBestand(2, 12, :Lager, :Order)
GetBestand(3, 12, :Lager, :Order)
```

Da der erste Funktionsaufruf eine leere Ergebnismenge zurückliefert, werden nur die Werte 2 und 3 als Lieferantennummern ausgegeben.

Projektion auf Basis von Ausgabeparametern

Erfolgt die Projektion auf Basis aller Ausgabeparameter, entspricht sie exakt der Ausgabe von Funktionen. Sind zudem alle Eingabewerte vorgegeben, ist genau ein Funktionsaufruf notwendig. Fehlen Eingabewerte, sind kompensierende Funktionsaufrufe zu ermitteln.

Sollen nicht alle Ausgabewerte projiziert werden, muss der Wrapper die zurückgelieferte Ergebnismenge der Funktion um die überflüssigen Ausgabeparameter kürzen.

Beispiel:

Die SQL-Anweisung

```
SELECT Lager, Order
FROM GetBestand
WHERE LiefNr = 2
```

projiziert exakt die Ausgabeparameter der Funktion. Somit sind lediglich kompensierende Funktionsaufrufe für den fehlenden Wert von `KompNr` aufzurufen:

```
GetBestand(2, 11, :Lager, :Order)
GetBestand(2, 12, :Lager, :Order)
GetBestand(2, 13, :Lager, :Order)
```

Da alle Funktionsaufrufe ein Wertepaar zurückliefern, sind diese drei Wertepaare das Ergebnis der SQL-Anweisung.

Projektion auf Basis von Eingabe- und Ausgabeparametern

Werden alle Eingabe- und Ausgabeparameter ausgegeben, entspricht dies der SELECT *-Anweisung. Sind alle Eingabewerte vorgegeben, werden diese mit den zurückgelieferten Ausgabewerten konkateniert. Fehlen Eingabewerte, so müssen wie in Abschnitt 5.2.3.3 Kompensationsmaßnahmen ausgeführt und anschließend die Eingabe- und Ausgabewerte im Wrapper konkateniert werden. Jene Parameter, die nicht projiziert werden, können weggelassen werden.

5.2.3.5 Gruppierung und Aggregation

Gruppierung und Aggregation stufen wir als erweiterte Funktionalität ein, die ein Wrapper nicht zwingend unterstützen muss. Doch reduzieren diese Operationen im Allgemeinen die Ergebnismenge erheblich, so dass sie die Anfrageoptimierung maßgeblich beeinflussen können.

Wie bisher unterscheiden wir auch hier zwischen der Anwendung der Operationen auf Eingabe- und Ausgabeparameter. Da die Betrachtungen zur Aggregation denen zur Projektion sehr ähneln, gehen wir nicht näher auf sie ein. Die Aggregation muss im Wrapper ausgeführt werden, da die Funktionen sie nicht unterstützen. Um die Aggregation jedoch ausführen zu können, müssen die zu aggregierenden Parameterwerte vorliegen, was einer Projektion auf diesen Parameter entspricht.

Beispiel:

Die SQL-Anweisung

```
SELECT SUM(Lager)
FROM GetBestand
WHERE LiefNr=1
```

wird zunächst auf die kompensierenden Funktionsaufrufe

```
GetBestand(1, 11, :Lager, :Order)
GetBestand(1, 12, :Lager, :Order)
GetBestand(1, 13, :Lager, :Order)
```

abgebildet. Anschließend werden die Ergebniswerte für `Lager` summiert.

Für die Gruppierung betrachten wir Fälle ohne WHERE-Klausel, da wir die Selektion bereits in Abschnitt 5.2.3.3 erläutert haben. Außerdem gehen wir davon aus, dass die Mengen der gruppierenden und aggregierenden Attribute disjunkt sind.

Gruppierung auf Basis von Eingabeparametern

Unabhängig davon, ob auf Basis aller oder einiger Eingabewerte gruppiert wird, müssen alle Daten mit kompensierenden Funktionsaufrufen ausgelesen werden, d. h., es wird ein „Table Scan“ durchgeführt. Da keine Selektionen definiert sind, ist ein solcher „Table Scan“ notwendig. Dies ist ebenso unabhängig davon, ob die Aggregation von Eingabe- oder Ausgabeparametern geschieht. Das folgende Beispiel veranschaulicht diesen Aspekt.

Beispiel:

Die SQL-Anweisung aggregiert und gruppiert auf Basis von Eingabeparametern. Es wird für jeden Lieferanten die Anzahl an gelieferten Komponententypen ermittelt:

```
SELECT LiefNr, COUNT(KompNr)
FROM GetBestand
GROUP BY LiefNr
```

Dazu werden alle kompensierenden Funktionsaufrufe ausgeführt, die sich aus dem kartesischen Produkt der Wertemengen von `LiefNr` und `KompNr` ergeben. Für all jene Funktionen, die eine leere Ergebnismenge zurückliefern, wird die Komponente von dem Lieferanten nicht geliefert. Alle anderen Werte werden pro Lieferant gezählt. Die resultierende Tabelle sieht wie folgt aus:

Lieferant	Anzahl Komponententypen
1	2
2	3
3	2

Bei der Aggregation von Ausgabeparametern gilt Ähnliches. Die SQL-Anweisung ermittelt die gelagerte Anzahl an Komponenten jeglichen Typs pro Lieferant:

```
SELECT LiefNr, SUM(Lager)
FROM GetBestand
GROUP BY LiefNr
```

Auch in diesem Fall müssen zunächst alle Werte ausgelesen und anschließend eine Summierung durchgeführt werden, um das Ergebnis zu erhalten:

Lieferant	Anzahl gelagerter Komponenten
1	15
2	5
3	13

Sobald wieder Selektionen stattfinden, reduziert sich die Zahl der kompensierenden Funktionsaufrufe, da nicht alle Daten ausgelesen werden müssen.

Gruppierung auf Basis von Ausgabeparametern

Wird auf Basis der Ausgabeparameter gruppiert, müssen ebenfalls alle Daten basierend auf allen Kombinationen der Eingabewerte ausgelesen werden. Um die verschiedenen Gruppen zu identifizieren, muss die Ergebnismenge im Wrapper nach dem Ausgabeparameter sortiert werden. Anschließend wird für jede Gruppe die Aggregation durchgeführt.

Beispiel:

Die SQL-Anweisung aggregiert auf Basis von Eingabeparametern und gruppiert anhand eines Ausgabeparameters. Die Anweisung liefert die unterschiedlichen Lagerbestände und wie viele Lieferanten diesen Bestand haben:

```
SELECT Lager, COUNT(LiefNr)
FROM GetBestand
GROUP BY Lager
```

Dazu werden alle kompensierenden Funktionsaufrufe ausgeführt, die sich aus dem kartesischen Produkt der Wertemengen von `LiefNr` und `KompNr` ergeben. Nachdem alle Rückgabewerte für `Lager` sortiert wurden, zählt man die gleichen Bestände und bekommt folgendes Ergebnis:

Lager	Anzahl Lieferanten
0	1
2	1
3	1
5	1
6	1
7	1
10	1

Wird die HAVING-Klausel hinzugefügt, so reduziert sich die Anzahl kompensierender Funktionen nur, wenn sich die HAVING-Klausel auf Eingabeparameter bezieht (vgl. Selektionen auf Eingabeparametern in Abschnitt 5.2.3.3). Basiert sie auf Ausgabeparametern, müssen nach wie vor alle Daten ausgelesen werden, da alle Ausgabewerte benötigt werden, um anschließend im Wrapper zu selektieren.

5.2.3.6 Teilanfragen und Mengenvergleiche

Die Ausführung von Teilanfragen innerhalb des Wrappers bezieht sich ausschließlich auf abstrakte Tabellen. Außerdem dürfen die Anfragen nur Operationen enthalten, die der Wrapper zusammen mit den Funktionen unterstützt. Nur wenn diese Anforderungen eingehalten werden, kann der Wrapper die Anfrage vollständig ausführen.

Wir unterscheiden bei den Teilanfragen zwischen korrelierten und unkorrelierten Teilanfragen. Im unkorrelierten Fall wird die Teilanfrage einmal ausgeführt. Dieser Fall bringt keine neuen Aspekte mit sich. Daher fokussieren wir auf die korrelierte Teilanfrage, die für jede Zeile der äußeren Tabelle ausgewertet werden muss. Da die meisten Teilanfragen durch Prädikate zum Mengenvergleich wie IN oder EXISTS eingeleitet werden, sollte der Wrapper auch solche Prädikate unterstützen. So kann die Zahl der Wrapper-Aufrufe und auch die zu transportierende Datenmenge erheblich reduziert werden. Das folgende Beispiel veranschaulicht diese Aspekte.

Beispiel:

In diesem Beispiel existiert neben den bereits bekannten abstrakten Tabellen eine Basistabelle im FDBS, die für bestimmte Lieferanten alternative Lieferanten nennt. Die folgende SQL-Anweisung ermittelt alternative Lieferanten für all jene Lieferanten, deren gelieferte Komponenten derzeit keinen Lagerbestand aufweisen und somit auf Lieferprobleme hinweisen.

```
SELECT Alternative
FROM GetLieferantenAlternative LA
WHERE 0 IN (
SELECT Lager
FROM GetBestand
WHERE LiefNr = LA.LiefNr)
```

Beim ersten Wrapper-Aufruf werden alle Lieferantennummern der Basistabelle sowie der Operator `IN` mitgegeben. Dieser führt die Teilanfrage für jeden dieser Werte mit den entsprechenden Funktionsaufrufen aus. Alle Lieferantennummern, für welche die Bedingung zutrifft, werden an das FDBS zurückgeliefert.

Nachdem wir gezeigt haben, welche Funktionalität der Wrapper implementieren sollte und wie sie umgesetzt werden kann, gehen wir im nächsten Abschnitt auf ein angepasstes Kostenmodell ein.

5.2.4 Kostenmodell

Abschließend betrachten wir Aspekte des Kostenmodells, die durch die Berücksichtigung bzw. Integration von Funktionen hinzukommen. Dazu passen wir zunächst die traditionelle Kostenberechnung an und zeigen für ausgewählte Operationen, wie sich die Kosten für eine Ausführung im FDBS oder im Wrapper unterscheiden.

5.2.4.1 Neue Kostenformel

Im Allgemeinen enthalten die Kosten einer Anfrage die Kosten für die Kommunikationszeit, Prozessorzeit, Platten-E/A und Hauptspeicher. Überträgt man diese Aufteilung auf die vorgestellte Integrationslösung, stellen die Kosten für die Plattenzugriffe und die Kommunikation aufgrund der Verteilung der Architekturkomponenten den Löwenanteil bei der Anfrageausführung dar. Die existierenden Kostenmodelle für die heterogene Anfrageverarbeitung sind hinsichtlich der Kommunikationszeit auch für unseren Fall geeignet.

Zur Minimierung der Kommunikationszeit zwischen FDBS und Datenquelle sollte die zu transportierende Datenmenge möglichst stark reduziert werden. Dies wird in erster Linie durch das Verschieben möglichst vieler Operationen (*push down*) zur Datenquelle erreicht. Sitzt der Wrapper auf Seiten der Datenquelle und stellt dieser zusätzliche Funktionalität bereit, können diese Operationen zum Wrapper verschoben werden.

Trotzdem muss die traditionelle Kostenberechnung für Tabellenzugriffe überarbeitet werden, wenn zukünftig die Kosten für Funktionsaufrufe berücksichtigt werden sollen.

Selinger et al. [SAC+79] führten die folgende Kostenabschätzung basierend auf Plattenzugriffen und Prozessorzeit für eine Tabelle T ein (vgl. Abschnitt 3.1.1.3):

$$Cost(T) = pagefetches(T) + W \times systemcalls(T)$$

Diese Kostenformel beschreibt die erwarteten E/A- und CPU-Kosten für einen Anfrageplan auf Basis von Instruktionen. Diese Kostenanteile lassen sich über einen Faktor W gewichten, um bei der Bestimmung des kostengünstigsten Anfrageplans die Art der Auslastung des Systems berücksichtigen zu können. W kann dabei als das effektive Verhältnis des Aufwandes für die DBS-interne Verarbeitung zur Ein-/Ausgabe (beispielsweise in System R der Aufruf des Zugriffssystems zu einem Seitenzugriff) gewählt werden.

Die aufgeführte Kostenformel bezieht sich jedoch auf die Anfrageverarbeitung eines zentralisierten DBS. In unserer Integrationsarchitektur ist die Anfrageverarbeitung aber über mehrere Komponenten verteilt: dem FDBS, dem Wrapper und der KIF. Für die Berechnung der Kosten über mehrere Komponenten hinweg gehen wir davon aus, dass die Systemaufrufe für eine Operation synchron nacheinander ausgeführt werden. Die Last in den einzelnen Knoten und im Netz wird nur statisch berücksichtigt. Somit ergibt sich folgende Kostenformel, welche die Kosten in den Knoten und die Kosten der Kommunikation addiert [HH03]:

$$Cost = LocalCost + CommCost$$

Wenn wir die Komponenten unserer Integrationsarchitektur einbringen, können wir die Formel weiter präzisieren:

$$\begin{aligned} Cost &= LocalCost_{FDBS} \\ &\quad + CommCost_{FDBS/Wrapper} \\ &\quad + LocalCost_{Wrapper} \\ &\quad + CommCost_{Wrapper/KIF} \\ &\quad + LocalCost_{KIF} \end{aligned}$$

Wir gehen davon aus, dass der Wrapper zusammen mit der KIF auf demselben Rechnerknoten liegt. Daher können wir $CommCost_{Wrapper/KIF}$ vernachlässigen. Als Kommunikationskosten setzt man die benötigte Zeit zur Übertragung einer Anzahl an Nachrichten einer gegebenen Länge bei einer bestimmten Bandbreite des Netzwerkes an und erhält somit:

$$CommCost = \#Messages/Bandwidth$$

Die Kosten im FDBS ergeben sich aus der Addition der Systemaufrufe und der E/A-Operationen, die synchron und asynchron anfallen können:

$$\begin{aligned} LocalCost_{FDBS} &= Speed * \#SystCalls \\ &\quad + w_1 * \#SynchI/O \\ &\quad + w_2 * \#AsynchI/O \end{aligned}$$

Dabei wird die Geschwindigkeit des Knotens gemessen oder rechnerspezifisch angesetzt. w_1 bzw. w_2 beschreiben den Transfer, wobei im synchronen Fall die Latenz hinzuzufügen ist. w_2 kann beispielsweise oft herangezogen werden, wenn ein Tabellen-Scan durch Prefetching unterstützt wird. Die Berechnung der lokalen Kosten setzt eine genaue Kenntnis des Operationsverhaltens voraus, um zu wissen, wann synchrone und asynchrone Operationen angesetzt werden müssen.

Für den Wrapper sind die Kosten ähnlich aufgebaut. Jedoch werden die Systemaufrufe den Hauptteil der Kosten verursachen, da der Wrapper E/A-Operationen nur zum Auslesen von Abbildungsinformationen benötigt.

Für die lokalen Kosten der KIF muss die Formel um die Anzahl der Aufrufe lokaler Funktionen erweitert werden:

$$\begin{aligned} LocalCost_{KIF} &= Speed * \#SystCalls \\ &\quad + w_1 * \#Synch I/O \\ &\quad + w_2 * \#Asynch I/O \\ &\quad + \#FuncCalls \end{aligned}$$

Wobei sich *FuncCalls* wiederum aus mehreren *LocalCost* und *CommCost* für die integrierten Systeme zusammensetzt.

Welche Größen in den aufgeführten Kostenformel sind relevant, wenn wir die Ausführung spezieller Operationen in FDBS und Wrapper vergleichen wollen? Um diese Frage zu beantworten, untersuchen wir die einzelnen Kostenformeln nach konstanten Größen, die wir vernachlässigen können. Betrachten wir zunächst die lokalen Kosten $LocalCost_{FDBS}$ für das FDBS. Die Geschwindigkeit des Rechnerknotens bleibt gleich, unabhängig von der Anzahl ausgeführter Operationen. Geht man darüber hinaus davon aus, dass beim Aufbau der Integrationsarchitektur identische Rechner eingesetzt wurden, dann kann der Faktor *Speed* vollständig vernachlässigt werden, da er überall gleich ist. Die E/A-Operationen können wir ebenfalls vernachlässigen, da die Daten aus der KIF und den integrierten DBS kommen und daher das FDBS nur wenige E/A-Operationen ausführt. Somit sind die Systemaufrufe die für uns relevante Größe für den Vergleich FDBS gegen Wrapper. Die Anzahl der Systemaufrufe ist abhängig von den ausgeführten Operationen im FDBS. Folglich sind hier die Verarbeitungsoperationen der Daten wie Selektion, Projektion usw. ausschlaggebend.

Ähnliches gilt für die lokalen Kosten im Wrapper. Auch hier können wir aus den oben aufgeführten Gründen die Geschwindigkeit des Rechnerknotens vernachlässigen. Ebenso sind die E/A-Operationen im Vergleich zu den Systemaufrufen gering und damit für uns nicht von Bedeutung für den angestrebten Vergleich. Folglich ist auch im Wrapper die Anzahl der Systemaufrufe und damit die Anzahl der ausgeführten Operationen relevant.

Die lokalen Kosten der KIF müssen wir nicht im Detail untersuchen, da diese für die Ausführung einer föderierten Funktion immer identisch sind. Stattdessen ist hier die Anzahl der Aufrufe derselben föderierten Funktion (mit unterschiedlichen Eingabewerten) von Bedeutung. Damit erhalten wir Vielfache von $LocalCost_{KIF}$. Folglich ist die Anzahl der Funktionsaufrufe von föderierten Funktionen die relevante Größe für uns.

Nachdem wir die Kommunikationskosten zwischen KIF und Wrapper aufgrund der Platzierung beider Komponenten auf einem Rechnerknoten als vernachlässigbar eingestuft haben, betrachten wir abschließend die Kommunikationskosten $CommCost_{FDBS/Wrapper}$ zwischen FDBS und Wrapper. Die Bandbreite des Netzwerkes bleibt dieselbe, unabhängig davon, ob Teile der Anfrageverarbeitung im FDBS oder im Wrapper ausgeführt werden. Somit ist die Anzahl der zu übertragenden Nachrichten ausschlaggebend. Diese wird wiederum von der zu transportierenden Datenmenge beeinflusst, so dass wir auf das zwischen FDBS und Wrapper zu bewegende Datenvolumen achten müssen.

Zusammengefasst kommen wir damit auf folgende Größen, die wir beim Vergleich von FDBS und Wrapper beachten müssen:

- Anzahl Operationen im FDBS
- Anzahl Operationen im Wrapper
- Anzahl Aufrufe der föderierten Funktion
- Größe der zu transportierenden Datenmenge zwischen FDBS und Wrapper

Sind die Auswertungskosten in FDBS und Wrapper mit denen der KIF vergleichbar, so ist ein Pushdown von Operationen zum Wrapper vorzuziehen, um die Kommunikationskosten gering zu halten. Lediglich bei komplexen und teuren Prädikaten, für die Wrapper und KIF ungeeignet sind, sollten die Operationen im FDBS durchgeführt werden.

Abschließend betrachten wir den Fall, dass das Netzwerk stark belastet ist. Es gilt nach wie vor, dass die Operationen möglichst von Wrapper und KIF ausgeführt werden sollten. Doch was ergibt sich für die komplexen Prädikate, die eigentlich im FDBS verarbeitet werden sollten? Hier kann keine pauschale Aussage gemacht werden. Eine Verlagerung solcher Operationen in den Wrapper macht nur dann Sinn, wenn die geringeren Ausführungskosten im FDBS die Übertragungskosten der größeren Datenmenge nicht ausgleichen können. Dies ist wiederum stark von den Daten und Prädikaten abhängig. Bewirkt das Prädikat eine starke Reduzierung der ursprünglichen Datenmenge, dann kann es sinnvoll sein, auch die komplexen Prädikate im Wrapper durchzuführen. Wird die Datenmenge hingegen nur geringfügig reduziert, dann könnte sich nach wie vor die Ausführung im FDBS als günstiger erweisen.

5.2.4.2 Kostenabschätzung der Wrapper-Funktionalität

Im Folgenden untersuchen wir, wie sich die Unterstützung zusätzlicher Funktionalität im Wrapper auf die Gesamtkosten auswirkt. Was kann an Operationen im FDBS und an zu transportierenden Datenvolumina eingespart werden, wenn die entsprechende Funktionalität im Wrapper bereitgestellt wird? Wir werden nicht die Kosten für die beiden Varianten explizit berechnen, sondern untersuchen, wie sich die folgenden Größen verändern: zu transportierende Datenmenge sowie Anzahl der Wrapper- und föderierten Funktionsaufrufe. Die Datenmenge und die Funktionsaufrufe sind laut unserer vorangegangenen Erläuterungen relevante Größen bei der Kostenbetrachtung. Geht man davon aus, dass zumindest einfache Operationen in FDBS und Wrapper ähnlich ausgeführt werden, dann spielt es für die Gesamtkosten keine Rolle, wo sie ausgeführt werden. Stattdessen achten wir auf die Anzahl der Wrapper-Aufrufe, denn diese verursachen zusätzliche Kosten.

Selektion

Die Selektion auf Basis der Eingabeparameter wird in der KIF ausgeführt. Folglich steht die Ausführung dieser Operation im FDBS nicht zur Wahl. Sind alle Eingabewerte spezifiziert, reicht ein Funktionsaufruf. Je mehr Werte fehlen, desto mehr Funktionsaufrufe werden generiert. Folglich variiert nur die Anzahl der Funktionsaufrufe und damit *FuncCall*. Dies geschieht unabhängig vom Ort der Ausführung, FDBS oder Wrapper.

Für eine Selektion auf Basis der Ausgabeparameter werden Unterschiede deutlich. Um diese Selektion durchführen zu können, müssen zunächst alle Ausgabewerte durch entsprechende Funktionsaufrufe ermittelt werden. Sind keine Eingabewerte vorgegeben, ist sogar ein „Table Scan“ notwendig. Wird die Selektion nicht im Wrapper unterstützt, muss die vollständige Ergebnismenge zur Selektion an das FDBS übertragen werden. Lässt sich hingegen die Selektion im Wrapper durchführen, kann die zu übertragende Datenmenge in der Regel erheblich reduziert werden. Aus Sicht des FDBS wirkt sich eine Selektion im Wrapper folglich hauptsächlich auf die Einsparung von CPU-Kosten für die aktuelle Selektion und für die nachfolgenden Evaluationskosten aus, da diese erheblich durch die Größe und Sortierordnung der zurückgelieferten Ergebnismenge bestimmt werden. Das folgende Beispiel verdeutlicht diesen Aspekt.

Beispiel:

Die folgende SQL-Anweisung ermittelt die Lieferanten all jener Komponenten, die einen Lagerbestand von mehr als 5 Stück aufweisen:

```
SELECT LiefNr
FROM GetBestand
WHERE Lager>5
```

Wird die Selektion im FDBS durchgeführt, ergeben sich folgende Kostenfaktoren:

1 Wrapper-Aufruf
7 Funktionsaufrufe (*FuncCall*) für alle Kombinationen der Eingabewerte
7 × 2 Werte ($CommCost_{FDBS/Wrapper}$) für 7 Ergebniszeilen à 2 Werten

Wird die Selektion im Wrapper ausgeführt, kommen wir auf:

1 Wrapper-Aufruf
7 Funktionsaufrufe (*FuncCall*) für alle Kombinationen der Eingabewerte
3 × 1 Werte ($CommCost_{FDBS/Wrapper}$) für 3 Ergebniszeilen à 1 Wert

Folglich kann die zu transportierende Datenmenge von 14 auf 3 Werte reduziert werden.

Betrachten wir die beiden Fälle allgemein, zeigt sich, dass die Ausführung der Selektion auf Basis der Ausgabewerte im Wrapper zusammen mit der KIF immer kostengünstiger oder höchstens gleich teuer ist. Ausschlaggebend ist in diesem Fall die zu übertragende Datenmenge zwischen Wrapper und FDBS, denn die Anzahl der Funktionsaufrufe bleibt gleich, da wir in jedem Fall alle Ausgabewerte auslesen müssen. Außerdem gehen wir

davon aus, dass die Ausführungsdauer der Selektion jeweils in FDBS und Wrapper vergleichbar ist.

Sei $Z(T_a)$ die Anzahl Zeilen der abstrakten Tabelle T_a. Wird die Selektion im FDBS durchgeführt, dann werden immer alle $Z(T_a)$ Zeilen vom Wrapper zum FDBS übertragen. Führt der Wrapper die Selektion aus, verringert sich die Zahl der Zeilen bei der KIF um den Selektivitätsfaktor SF (wobei $SF \leq 1$). Dieser Faktor ist abhängig vom Vergleichsprädikat und bewegt sich zwischen $SF_{=} = 1/K(T_a, i)$ für den „=“-Operator, $SF_{<,>} = 1/3$ für „<“- oder „>“-Operationen und $SF_{\neq} = 1$ für das „≠“-Prädikat. Somit gilt $SF \leq 1$ und $SF \times Z(T_a) \leq Z(T_a)$. Damit verursacht die Ausführung der Selektionsoperation im Wrapper die geringeren Kommunikationskosten.

Projektion

Der Kostenvergleich für die Projektion verhält sich ähnlich der Selektion. Auch in diesem Fall bleiben die Wrapper- und Funktionsaufrufe dieselben, lediglich die zu übertragende Datenmenge und damit die Kommunikationskosten sind unterschiedlich. Bei Ausführung der Projektion im Wrapper müssen weniger Daten zum FDBS transportiert werden. Folglich werden die Gesamtkosten für die Verarbeitung der Anweisung geringer.

Beispiel:

Die SQL-Anweisung

```
SELECT KompNr, Order
FROM GetBestand
```

liefert die bestellte Menge für jede Komponente. Wird die Projektion im FDBS durchgeführt, ergeben sich folgende Größen:

1 Wrapper-Aufruf
7 Funktionsaufrufe (*FuncCall*) für alle Kombinationen der Eingabewerte
7×4 Werte ($CommCost_{FDBS/Wrapper}$) für 7 Ergebniszeilen à 4 Werten

Werden die Operationen im Wrapper ausgeführt, ergibt sich:

1 Wrapper-Aufruf
7 Funktionsaufrufe (*FuncCall*) für alle Kombinationen der Eingabewerte
7×2 Werte ($CommCost_{FDBS/Wrapper}$) für 7 Ergebniszeilen à 2 Werten

Folglich können die Kommunikationskosten in diesem Beispiel auf die Hälfte reduziert werden.

Allgemein betrachtet ist die Ausführung der Projektion im Wrapper in jedem Fall kostengünstiger. Während die Anzahl der Wrapper- und Funktionsaufrufe in beiden Fällen identisch ist, unterscheiden sich auch in diesem Fall die Kommunikationskosten. Für die Ausführung im FDBS müssen alle $Z(T_a)$ Zeilen der abstrakten Tabelle vom Wrapper zum FDBS übertragen werden. Die Ausführung im Wrapper verringert die Zahl der zu transportierenden Zeilen um den Projektionsfaktor PF, der auf jeden Fall geringer als 1

ist, da sich die Länge der einzelnen Zeilen aufgrund der Projektion auf eine Untermenge der Tabellenspalten verringert. Somit gilt $PF < 1$ und $PF \times Z(T_a) < Z(T_a)$. Folglich fallen die Kommunikationskosten bei Ausführung der Projektion im Wrapper immer geringer aus.

Gruppierung und Aggregation

Die Ausführung von Gruppierungen und Aggregationen erfordert je nach optionaler Spezifikation von Selektionen mehrere bis zu $Z(T_a)$ kompensierende Funktionsaufrufe, um einen „Table Scan“ durchzuführen. Werden Gruppierung und Aggregation erst im FDBS durchgeführt, muss die gesamte Datenmenge zum FDBS transportiert werden. Dabei entstehen die gleichen Kosten wie für Selektionen oder Projektionen.

Werden diese Operationen bereits im Wrapper durchgeführt, reduziert sich die zu übertragende Datenmenge erheblich. Außerdem sind weniger Funktionsaufrufe notwendig, wenn überdies die HAVING-Klausel im Wrapper unterstützt wird. Da die HAVING-Klausel einer Selektion entspricht, müssen nicht alle Daten über kompensierende Funktionsaufrufe ausgelesen werden.

Das folgende Beispiel verdeutlicht unsere Erläuterungen.

Beispiel:

Der folgende SQL-Befehl ermittelt, von wie vielen Lieferanten eine Komponente geliefert wird:

```
SELECT COUNT(LiefNr), KompNr
FROM GetBestand
GROUP BY KompNr
```

Bei Durchführung der Gruppierung und Aggregation im FDBS kommen folgende Kostengrößen auf:

1 Wrapper-Aufruf
7 Funktionsaufrufe (*FuncCall*) für alle Kombinationen der Eingabewerte
7×2 Werte ($CommCost_{FDBS/Wrapper}$) für 7 Ergebniszeilen à 2 Werten

Werden die Operationen im Wrapper ausgeführt, ergeben sich folgende Größen:

1 Wrapper-Aufruf
7 Funktionsaufrufe (*FuncCall*) für alle Kombinationen der Eingabewerte
3×2 Werte ($CommCost_{FDBS/Wrapper}$) für 3 Ergebniszeilen à 2 Werten

Folglich können die Kommunikationskosten von 14 auf 6 reduziert werden.

Wir erweitern nun das Beispiel um eine HAVING-Klausel:

```
SELECT COUNT(LiefNr), KompNr
FROM GetBestand
GROUP BY KompNr
HAVING KompNr<=12
```

Dann ergibt sich:

1 Wrapper-Aufruf
\+ 4 Funktionsaufrufe (*FuncCall*) für alle Kombinationen der Eingabewerte
\+ 2×2 Werte ($CommCost_{FDBS/Wrapper}$) für 2 Ergebniszeilen à 2 Werten

In diesem Fall konnten weitere drei Funktionsaufrufe eingespart und die Kommunikationskosten nochmals um 2 Einheiten verringert werden.

Die allgemeine Betrachtung von Gruppierung und Aggregation zeigt, dass bei Ausführung im FDBS erneut alle $Z(T_a)$ Zeilen der abstrakten Tabelle T_a vom Wrapper zum FDBS übertragen werden müssen. Bei der Ausführung im Wrapper und in der KIF wird die Datenmenge verringert. Sei $G := \{GROUPING\text{-}Spalten\}$, dann gilt für das zu transportierende Datenvolumen: $min(\frac{1}{2}Z(T_a), \prod_{g \in G} W(T_a, g)) \leq \frac{1}{2}Z(T_a)$ und damit $\frac{1}{2}Z(T_a) < Z(T_a)$. Folglich ist die Ausführung im Wrapper immer kostengünstiger als im FDBS.

Wird darüber hinaus die HAVING-Klausel auf Basis von Eingabeparametern einbezogen, so sind die Kosten für das FDBS identisch zu denen in vorherigen Fall. Die Kosten für die Wrapper-Ausführung verändern sich jedoch, da sich auch die Anzahl der Funktionsaufrufe ändert, sofern die HAVING-Klausel auf Eingabeparametern basiert. Statt einem „Table Scan“ und damit dem Produkt *aller* Eingabewerte beinhaltet das Produkt nur noch jene Eingabewerte, die nicht zu den GROUPING-Werten gehören. Folglich ist $\prod_{i \notin G} K(T_a, i) \leq \prod_{i \in I} K(T_a, i)$ und damit sind weniger Funktionsaufrufe notwendig als für einen „Table Scan“.

Auch die Anzahl der zu übertragenden Ergebniszeilen verringert sich durch den Einsatz der HAVING-Klausel weiter, da der Faktor $\prod_{h \in H} \frac{1}{K(T_a,h)} \leq 1$ (mit $H := \{HAVING\text{-}Spalten\}$) ist und somit die im vorherigen Fall berechnete Datenmenge reduzieren kann.

Teilanfragen und Mengenvergleiche

Abschließend betrachten wir die Verarbeitung von Teilanfragen und Mengenvergleichen. Insbesondere der korrelierte Fall von Teilanfragen verursacht rege Interaktion zwischen FDBS und Wrapper bzw. KIF, da die Teilanfrage nicht in einem Schritt ausgeführt werden kann, sondern mehrmals aufgerufen werden muss. Folglich unterscheiden sich die Kosten zumindest in der Anzahl der Wrapper-Aufrufe.

Unterstützt der Wrapper darüber hinaus den Mengenvergleich, kann auch die zu übertragende Datenmenge reduziert werden.

Beispiel:

Der folgende SQL-Befehl ermittelt die alternativen Lieferanten all jener Zulieferer, für die keine Komponenten auf Lager sind:

```
SELECT Alternative
FROM GetLieferantenAlternative LA
WHERE 0 IN (SELECT Lager
FROM GetBestand
WHERE LiefNr=LA.LiefNr)
```

Wenn das FDBS alle Operationen durchführt, muss ein „Table Scan“ von `GetBestand` durchgeführt werden. Es ergeben sich folgende Kostengrößen:

1 Wrapper-Aufruf
7 Funktionsaufrufe (*FuncCall*) für alle Kombinationen der Eingabewerte
7×4 Werte ($CommCost_{FDBS/Wrapper}$) für 7 Ergebniszeilen à 4 Werten

Wird die Teilanfrage im Wrapper ausgeführt und der Mengenvergleich im FDBS, ergibt sich Folgendes:

4 Wrapper-Aufruf
7 Funktionsaufrufe (*FuncCall*) für alle Kombinationen der Eingabewerte
7×1 Wert ($CommCost_{FDBS/Wrapper}$) für 7 Ergebniszeilen à 1 Wert

In diesem Fall konnten die Kommunikationskosten auf ein Viertel reduziert werden, im Gegenzug hat sich aber die Zahl der Wrapper-Aufrufe vervierfacht.

Wird auch der Mengenvergleich im Wrapper unterstützt, kommen wir nun zu diesen Größen:

1 Wrapper-Aufruf
7 Funktionsaufrufe (*FuncCall*) für alle Kombinationen der Eingabewerte
1×2 Werte ($CommCost_{FDBS/Wrapper}$) für 3 Ergebniszeilen à 2 Werten

Offensichtlich konnten die Wrapper-Aufrufe erneut reduziert und die Kommunikationskosten verringert werden.

Auch in diesem Fall zeigt die allgemeine Betrachtung der Kosten, dass die Unterstützung zusätzlicher Funktionalität im Wrapper die globale Anfrageoptimierung maßgeblich beeinflussen kann. Untersuchen wir zunächst die Kosten für die Ausführung der Teilanfrage und des Mengenvergleichs im FDBS. In diesem Fall werden alle Daten (also alle $Z(T_a)$ Zeilen) der abstrakten Tabelle `GetBestand` an das FDBS geschickt, um dort die geforderten Operationen auszuführen.

Übernimmt der Wrapper die Ausführung der Teilanfrage, erhöht sich die Anzahl der Wrapper-Aufrufe deutlich, die nun der Kardinalität der Korrelationsspalte entspricht. Im Gegensatz dazu verringern sich die Funktionsaufrufe seitens der KIF. Es muss kein „Table Scan“ ausgeführt werden, da zumindest der Wert eines Eingabeparameters durch die Korrelationsspalte vorgegeben ist. Auch die zu übertragende Datenmenge verringert sich, weil die Anzahl der Ergebniszeilen der Kardinalität der Korrelationsspalte entspricht. Außerdem werden die Ergebniszeilen kürzer, da nicht alle Spalten zurückgeliefert werden müssen, sondern eine Projektion durchgeführt wird.

Es stellt sich hier die Frage, ob die geringere Anzahl an Funktionsaufrufen und die reduzierte Datenmenge die zusätzlichen Wrapper-Aufrufe kompensieren können. Dazu kann keine pauschale Antwort gegeben werden. Alle Teile der Kostenformel werden von mehreren Faktoren beeinflusst, wie die Geschwindigkeit des Netzes als auch die Implementierung des Wrappers. Hier muss im Einzelfall entschieden werden, welche Faktoren sich stärker auswirken.

Werden Teilanfrage als auch Mengenvergleich im Wrapper und somit auf der KIF-Seite ausgeführt, ist die kostengünstigste Ausführung möglich. Es wird nur ein Wrapper-Aufruf benötigt und die Kommunikationskosten können ebenfalls reduziert werden. Zum

Projektionsfaktor kommt nun noch ein Selektionsfaktor hinzu, da nur jene Zeilen an das FDBS geschickt werden müssen, auf die der Mengenvergleich zutrifft.

Die Untersuchungen der zusätzlichen Wrapper-Funktionalität haben verdeutlicht, dass Operationen, die auf Seiten von KIF und Wrapper ausgeführt werden, die Kosten erheblich reduzieren können. Folglich ist die Implementierung der Basisfunktionalität auf jeden Fall gerechtfertigt.

5.2.4.3 Alternative Vorgehensweisen

Nach Betrachtung aller Operationen wird deutlich, dass deren Verschiebung zum Wrapper die Kosten nicht nur durch geringere Datenmengen senkt, sondern auch die Anzahl der Wrapper- und Funktionsaufrufe erheblich reduzieren kann. Nichtsdestotrotz zeigt sich, dass das Ziel, die komplette Funktionalität von SQL durch kompensierende Funktionsaufrufe unterstützen zu wollen, einige Nachteile mit sich bringt. Dies liegt vor allem an der Zahl der benötigten kompensierenden Funktionsaufrufe, die regelrecht explodieren kann. Ein kleines Rechenbeispiel soll dies verdeutlichen. Angenommen eine Funktion hat drei Eingabeparameter mit einer Kardinalität von lediglich zehn für jeden dieser Parameter. Wenn eine Selektion vorliegt, die den Wert für nur einen dieser Parameter spezifiziert, sind bereits $10 \times 10 = 100$ kompensierende Funktionsaufrufe notwendig, um die fehlenden Werte auszugleichen. Noch gravierender ist das Ergebnis, wenn gar kein Wert spezifiziert wird, denn dann erhält man sogar $10 \times 10 \times 10 = 1000$ Funktionsaufrufe.

Daher macht die Betrachtung alternativer Einbindungsmöglichkeiten der föderierten Funktionen mittels abstrakter Tabellen Sinn, um die große Anzahl der Funktionsaufrufe zu verringern. Dies führt leider auch dazu, dass sich der Benutzer der Einschränkungen oder zumindest der besonderen Handhabung abstrakter Tabellen bewusst sein muss.

Eine Alternative könnte ein erweiterter Parser sein, der bei der Zergliederung der SQL-Anfrage feststellt, ob beispielsweise für alle Eingabeparameter Werte spezifiziert wurden. Ist dies nicht der Fall, so wird die Anfrage mit einer Fehlermeldung zurückgewiesen und der Benutzer auf die fehlenden Informationen aufmerksam gemacht. Auf diese Weise könnten alle kompensierenden Funktionsaufrufe aufgrund fehlender Eingabewerte vermieden werden. Ein anderer Ansatz verschiebt die Spezifikation der Eingabewerte von Vergleichsprädikaten zu benutzerdefinierten Funktionen in der WHERE-Klausel, die immer zusammen mit den referenzierten abstrakten Tabellen eingesetzt werden müssen. Mit Hilfe der benutzerdefinierten Funktionen können die benötigten Eingabewerte übergeben werden, so dass auch hier keine kompensierenden Funktionsaufrufe aufgrund fehlender Werte nötig sind. Ein Nachteil bei dieser Vorgehensweise ist jedoch, dass der Benutzer wissen muss, bei welchen Tabellen es sich um abstrakte Tabellen handelt und welche benutzerdefinierte Funktion er in diesem Zusammenhang einzusetzen hat. Die Entscheidung, welche Alternative anzustreben ist, hängt sicherlich von den Systemen und den Benutzern ab.

5.3 Transaktionsverwaltung

Im folgenden Abschnitt erläutern wir unser Konzept für die Transaktionsverwaltung in der Integrationsarchitektur. Hierzu sind im Speziellen Ansätze für den verteilten, heterogenen Fall von Bedeutung, da sich die Transaktionen nicht nur über Datenbanksysteme, sondern auch über Anwendungssysteme erstrecken sollen. Jedoch unterstützen diese Anwendungssysteme in den meisten Fällen nicht die aktive Teilnahme an verteilten Transaktionen, so dass neue Ansätze notwendig werden. Zur Erarbeitung möglicher Lösungen für die vorgestellte Integrationsarchitektur werden zunächst die zu integrierenden Systeme bezüglich ihrer Transaktionseigenschaften klassifiziert. Anschließend stellen wir einen Lösungsansatz vor, der auf der Arbeit von [Sch96a] aufsetzt und diese für die Integration von Anwendungssystemen erweitert. Dabei konzentrieren wir uns zunächst auf die KIF und ihre Transaktionsunterstützung und dehnen den Ansatz anschließend auf die Gesamtarchitektur aus. Abschließend betrachten wir die Transaktionsunterstützung seitens kommerzieller Produkte und beschreiben eine Umsetzung mit einem WfMS als KIF-Implementierung.

Wir wollen in den folgenden Abschnitten ein Transaktionsmodell erarbeiten, dass globale Serialisierbarkeit, globale Atomarität sowie globale Erkennung und Vermeidung von Deadlocks auch für Anwendungssysteme unterstützt[2]. Uns ist kein Ansatz bekannt, der eine Lösung für diese Anforderung bietet. Bestehende Arbeiten wie [Sch96a] zeigen, dass Umsetzungen in diesem Themenfeld sehr komplex und aufwendig sein können. Darüber hinaus müssen häufig spezielle Anforderungen an die integrierten Systeme eingehalten werden. Nichtsdestotrotz wollen wir untersuchen, welche Möglichkeiten es überhaupt gibt und mit welchen Einschränkungen wir zu rechnen haben. Ob letztendlich eine Implementierung Sinn macht, muss sicherlich von Fall zu Fall und anhand der bestehenden Anforderungen entschieden werden.

5.3.1 Vorbetrachtungen und Voraussetzungen

ACID-Eigenschaften von Transaktionen sind geeignet für flache Transaktionen. Wie bereits in Abschnitt 3.1.2.3 dargestellt, benötigt man erweiterte Konzepte wie z. B. geschachtelte Transaktionen zur Aufteilung von globalen Transaktionen in Subtransaktionen, die von den integrierten Systemen verarbeitet werden. Die unterschiedlichen Ansätze erweiterter Transaktionskonzepte gehen von unterschiedlichen Eigenschaften der lokalen Systeme aus. Es folgt daher zunächst eine Klassifikation der lokalen Systeme hinsichtlich ihrer Transaktionseigenschaften, um anschließend beschreiben zu können, welche Systeme zum Einsatz kommen.

- *Nicht-wiederherstellbare Systeme (non-recoverable systems)*:
  Diese Systemart ist nicht in der Lage, ihre Daten nach einem Fehler wieder in einen konsistenten Zustand zurückzuführen. Anwendungen auf Dateisystemen werden dieser Kategorie zugeteilt.

- *Wiederherstellbare Systeme (recoverable systems)*:
  Diese Systeme können bei einem Fehler ihre Daten wieder in den Originalzustand

[2] Obwohl vor allem Stimmen aus der Industrie die Meinung vertreten, dass Gewährleistung globaler Serialisierbarkeit nicht notwendig ist.

zurücksetzen, d. h., sie unterstützen Recovery. Folglich weisen sie eine grundlegende Transaktionsunterstützung auf und sind somit nach innen transaktional. Da sie ihre eigenen Transaktionsgrenzen haben, können Transaktionen innerhalb dieses Systems beliebig zurückgesetzt werden.

- *Transaktionale Systeme (transactional systems)*:
  Systeme dieser Art stellen eine Erweiterung der wiederherstellbaren Systeme dar, indem sie einen sichtbaren Prepare-to-Commit-Zustand aufweisen und somit externe Transaktionsgrenzen beachten können. Zu diesem Zweck wird in den meisten Fällen eine Form des Zwei-Phasen-Commit-Protokolls unterstützt.

- *Offene transaktionale Systeme (open transactional systems)*:
  Diese Systeme werden als offen bezeichnet, da sie Informationen über ihre Transaktionen nach außen geben. Dazu gehören beispielsweise Informationen zu den Sperren oder auch Deadlocks, die eine globale Transaktionsverwaltung unterstützen können.

Die zu integrierenden Anwendungssysteme gehören in den meisten Fällen zu den wiederherstellbaren Systemen, da sie eine Kopplung aus Datenbanksystem und zugehöriger Anwendung darstellen. Nicht-wiederherstellbare Systeme und Anwendungssysteme auf Basis von Dateisystemen werden nicht weiter betrachtet, da sich deren Einsatz für unternehmenskritische Daten und Anwendungen verbietet. Die Integration von Anwendungssystemen und ihrer Funktionen wurde ursprünglich notwendig, gerade weil der direkte Zugriff auf das darunter liegende DBS nicht möglich war. Daher liegt der Fokus in den weiteren Betrachtungen auf wiederherstellbare Systeme, d. h., man geht davon aus, dass die mittels der KIF zu integrierenden Systeme intern transaktional sind, jedoch keinen sichtbaren Prepare-to-Commit-Zustand haben und auch keine Informationen über ihre Transaktionen und deren Verarbeitung nach außen zur Verfügung stellen. Folglich verarbeiten diese Systeme ihre Transaktionen vollständig autonom und bieten keinerlei Unterstützung für eine globale Transaktionsverwaltung an.

Außerdem gehen wir davon aus, dass keine Veränderungen an den Systemen vorgenommen werden können. Eine durch Änderungen nötig werdende Rekompilierung der Anwendungen ist im Allgemeinen nicht tolerierbar. Ebenso sollte das Datenbankschema nach Möglichkeit unberührt bleiben. Sind trotzdem Änderungen des Schemas notwendig, so sollten bestehende Datenbankobjekte wie Tabellen, die von den Anwendungen referenziert werden, keinesfalls verändert werden. Ansonsten muss man auch die Anwendungen anpassen. Somit ist lediglich das Hinzufügen von Objekten möglich, die aber die Leistungsfähigkeit des Datenbanksystems nicht beeinflussen dürfen. Dies können beispielsweise Hilfstabellen mit Verwaltungsinformationen für eine heterogene Transaktionsverwaltung sein.

Bevor wir in den nächsten Abschnitten einen Lösungsansatz für die Transaktionsverarbeitung in der betrachteten Integrationsarchitektur vorstellen, verdeutlichen wir zunächst, welcher Teil der Integrationsarchitektur im Fokus stehen soll. Da bereits einige Lösungsvorschläge für FDBS vorliegen und in dieser Arbeit die Integration von Funktionen über eine spezielle Komponente die neuen Aspekte liefert, soll vor allem diese Komponente im Vordergrund stehen. Abbildung 5.3 zeigt, dass die Integrationsarchitektur in mehrere Schichten aufgeteilt ist. Die oberste, grau hinterlegte Schicht bildet das

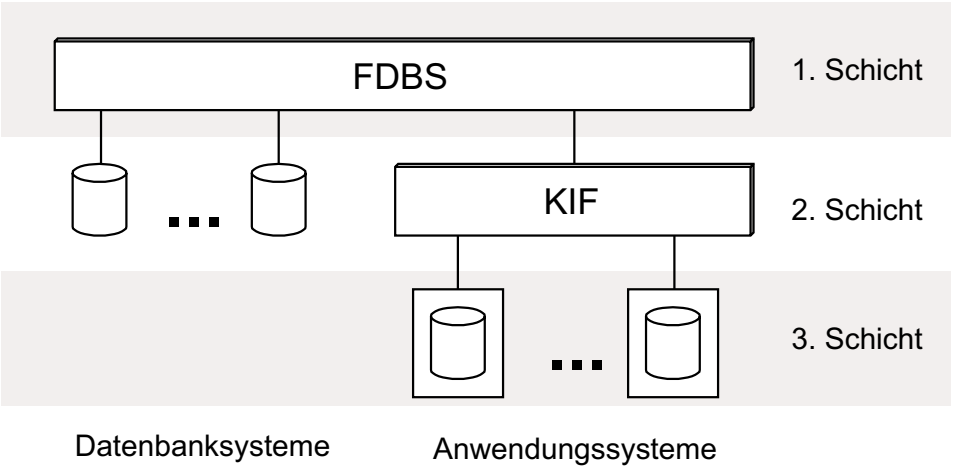

Abbildung 5.3: Die drei Schichten der betrachteten Integrationsarchitektur.

FDBS, das den Clients eine SQL-Schnittstelle anbietet. Die zweite Schicht besteht zum einen aus den Datenbanksystemen, die vom FDBS integriert werden und zum anderen aus der KIF, welche die Anwendungssysteme integriert. Da alle Systeme der zweiten Schicht zumindest wiederherstellbare, besser noch transaktionale Systeme sein sollten, muss auch die KIF eine Transaktionsverwaltung zur Verfügung stellen. Da die KIF aber selbst wiederum Systeme integriert, welche die dritte und unterste Schicht in der gesamten Architektur bilden, gilt es auch in der KIF eine globale Transaktionsverwaltung umzusetzen. Hier ergeben sich dieselben Probleme der heterogenen Transaktionsverwaltung wie für das FDBS. Im Folgenden liegt daher das Augenmerk auf der Realisierung einer globalen Transaktionsverwaltung in der Komponente zur Funktionsintegration.

Ein Transaktionsmodell für die KIF muss ebenfalls Lösungen für globale Serialisierbarkeit, globale Atomarität und globale Deadlock-Erkennung und -Vermeidung bereitstellen. Grund hierfür ist die Tatsache, dass die Anwendungssysteme auf Datenbanksystemen basieren und somit dieselben Problematiken wie bei einem FDBS aufkommen. Ausgehend davon, dass die Anwendungssysteme autonom und grundsätzlich nicht dafür vorbereitet sind, eine Integration zu unterstützen, wird in den folgenden Abschnitten ein Transaktionsmodell eingeführt, das auf den Ansätzen von [Sch96a] für FDBS aufsetzt und diese für die Integration von Funktionen erweitert. Dazu beschreiben wir zunächst den Lösungsansatz von Schaad und erläutern anschließend den erweiterten Ansatz.

5.3.2 Transaktionsverwaltung in heterogenen, föderierten Datenbanksystemen nach Schaad

Schaad hat in den neunziger Jahren einen Ansatz zur Unterstützung von Transaktionen in heterogenen, föderierten Datenbanksystemen entwickelt [Sch96a], der die Basis unserer Lösung darstellt. Grund hierfür ist die Tatsache, dass den zu integrierenden Anwendungssystemen wiederum Datenbanksysteme zugrunde liegen. Folglich werden auch hier heterogene Datenbanksysteme integriert. Die Arbeit von Schaad baut auf

einer zusätzlichen Transaktionsschicht auf, mit welcher Mehrebenen-Transaktionen implementiert werden.

In den folgenden Abschnitten werden wir zunächst den Ansatz von Schaad detailliert beschreiben. Anschließend zeigen wir auf, welche Probleme sich bei der Integration von Anwendungssystemen ergeben und somit Erweiterungen des Ansatzes notwendig werden.

5.3.2.1 Architektur

Schaad erweitert für sein Transaktionsmodell die zweischichtige Architektur traditioneller föderierter Datenbanksysteme um eine zusätzliche Schicht zwischen FDBS und den lokalen DBS (siehe Abbildung 5.4). Agenten für jedes DBS bilden diese Schicht und unterstützen die Verarbeitung globaler Transaktionen. Die globalen Transaktionen werden in Subtransaktionen aufgeteilt, die wiederum an die lokalen DBS weitergereicht werden. Während das FDBS für die Ausführung der globalen Transaktionen zuständig ist, sind die lokalen DBS für die Subtransaktionen verantwortlich. Somit liegen zwei Stufen der Transaktionsverwaltung vor (FDBS und lokale DBS) und das Konzept der Mehrebenen-Transaktionen kann angewendet werden. Neben den globalen Transaktionen werden auch lokale Transaktionen und deren indirekt verursachten Konflikte zwischen globalen Transaktionen berücksichtigt. Somit können existierende Anwendungen bei der Einführung eines FDBS weiter bestehen und müssen nicht geändert werden.

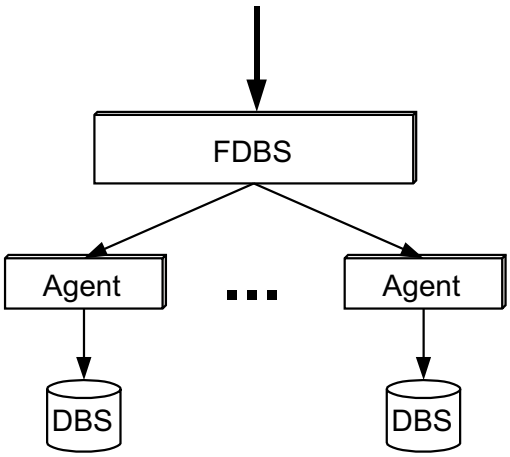

Abbildung 5.4: Dreischichtige FDBS-Architektur mit Agenten.

Bevor wir die einzelnen Komponenten der in [Sch96a] vorgestellten Architektur und deren Aufgaben näher betrachten, beschreiben wir die hierfür verwendeten Konzepte zur Unterstützung von globaler Serialisierbarkeit, globaler Atomarität und globaler Deadlock-Erkennung und -Vermeidung.

5.3.2.2 Globale Serialisierbarkeit

In einer Architektur mit Mehrebenen-Transaktionen, die von verschiedenen Transaktionsverwaltungen bearbeitet werden, kann globale Serialisierbarkeit durch eine Schicht-

zu-Schicht-Serialisierbarkeit garantiert werden [BBG89, BSW88, Wei91, WS92]. Mit dieser Methode lässt sich ein serialisierbarer Mehrebenen-Schedule in einen seriellen Schedule umwandeln und somit die Serialisierbarkeit überprüfen [DSW94, MRW+93, BBG89, WDSS93]. So kann einfach erklärt werden, warum in einem FDBS mehrere unterschiedliche Transaktionsverwaltungen auf einer Ebene zulässig sind und sogar die strenge Schichtung der Mehrebenen-Transaktionen teilweise aufgegeben werden kann. Diese Aspekte sind für eine Integrationsarchitektur von Bedeutung, da aufgrund der unterschiedlichen zu integrierenden Systeme mit heterogenen Transaktionsverwaltungen innerhalb einer Ebene zu rechnen ist. Ebenso kann die strenge Schichtung nicht garantiert werden, da sich die einzelnen Subtransaktionen in den verschiedenen Systemen über unterschiedlich viele Ebenen hinweg aufteilen.

Die Methode betrachtet die Transaktion als Baum von Operationsaufrufen, deren Teilbäume man durch Aufrufe der Operationen einer tieferen Ebene erhält. Die Wurzel stellt die Operation auf der höchsten Ebene dar, von wo aus alle anderen Operationen aufgerufen werden. Ein Teilbaum ist isoliert, wenn keine Überlappung mit anderen Teilbäumen auftritt. Dies trifft zu, wenn im Schedule eine serielle Ausführung aller von der Wurzel des Teilbaums abhängigen Operationen stattfindet. Mit Hilfe bestimmter Regeln kann ein Schedule in einen seriellen Schedule umgewandelt werden. Ist dies möglich, so ist die Serialisierbarkeit bewiesen.

Mit der ersten Regel *„Kommutativität von Operationen“* werden in der tiefsten Schicht zunächst die Teilbäume isoliert. Dies erfolgt über die Vertauschung von Operationen, die kommutierbar sind. Zwei Operationen o und o' kommutieren, wenn beide möglichen Ausführungsreihenfolgen $o < o'$ und $o' < o$ für sie selbst und für alle anderen nachfolgenden Operationen dasselbe Ergebnis liefern.

Anschließend kann man mit der zweiten Regel *„Reduktion von isolierten Bäumen“* die isolierten Teilbäume bis auf die Wurzel reduzieren, da nun die Serialisierungsreihenfolge der Teilbäume auf die Operationen der nächst höheren Ebene übernommen werden kann. Diese beiden Regeln werden so lange angewendet, bis entweder nur noch die Wurzel der globalen Transaktion übrig ist oder sich Teilbäume nicht mehr isolieren lassen und damit der Schedule nicht serialisierbar ist.

Abbildung 5.5 verdeutlicht die Anwendung der vorgestellten Regeln. In diesem Beispiel führen zwei Transaktionen T_1 und T_2 Buchungen durch, die je aus einer Auszahlung und einer Einzahlung bestehen. Diese Operationen auf Ebene L_1 werden wiederum von je zwei Änderungsoperationen auf Ebene L_0 implementiert. Wendet man zunächst Regel 1 auf die tiefste Ebene L_0 an, kann man die einzige Überlappung durch Vertauschen der Operationen entfernen, da sie auf unterschiedlichen Objekten arbeiten. Da nun alle Teilbäume isoliert sind, können sie mit Regel 2 bis auf die Wurzel reduziert werden. Jetzt wird erneut Regel 1 auf der Ebene L_1 angewendet. Zunächst können die beiden Einzahlungen unter der Annahme vertauscht werden, dass sie mit demselben Effekt in verschiedener Reihenfolge ausgeführt werden können. Anschließend tauscht man Auszahlung und Einzahlung, da sie auf verschiedenen Objekten arbeiten und somit kein Konflikt besteht. Die erneute Anwendung der zweiten Regel auf den nun isolierten Teilbäumen führt zu einem äquivalenten seriellen Schedule mit T_1 vor T_2.

Die dritte Regel *„Kompensation von Operationen“* wird für die Recovery benötigt. Hier geht man davon aus, dass die für die Recovery notwendigen Operationen, also die Kom-

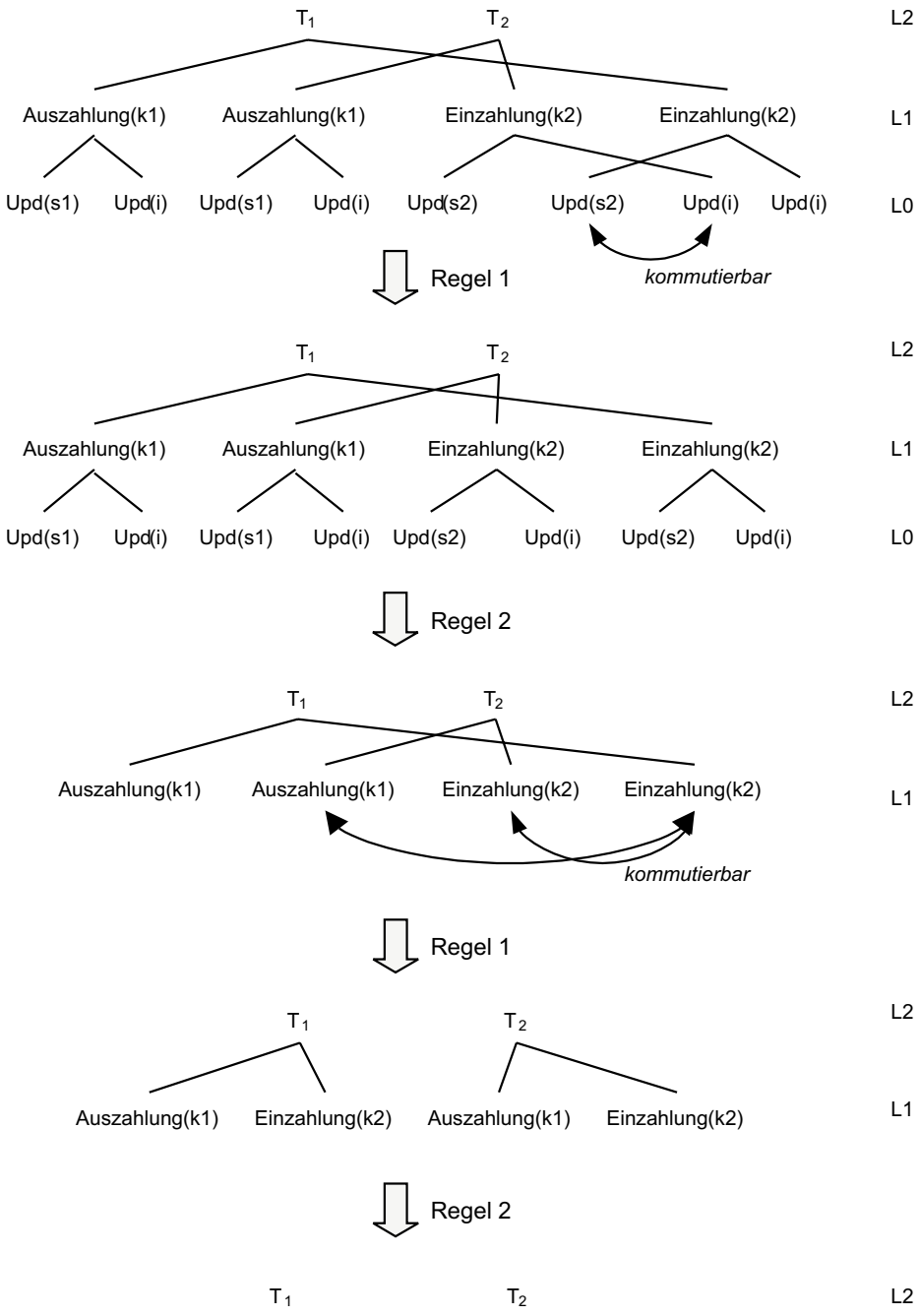

Abbildung 5.5: Schrittweise Konvertierung in einen seriellen Schedule [Sch96a].

pensationen, zur Transaktion gehören und auch dem Synchronisationsverfahren unterworfen werden. Abgebrochene Transaktionen einschließlich ihrer Recovery-Operationen dürfen die Serialisierbarkeit der anderen Transaktionen nicht beeinträchtigen. Daher müssen diese Transaktionen zur Überprüfung der Serialisierbarkeit wieder aus dem Schedule entfernt werden, was durch die dritte Regel geschieht. Wenn eine Operation und ihre Kompensationsoperation unmittelbar aufeinander folgen und beide Operationen

isoliert sind, heben sich die beiden in ihrer Wirkung auf und können aus dem Schedule entfernt werden.

Das Beispiel mit einer Kompensationsoperation in Abbildung 5.6 zeigt die Anwendung der drei vorgestellten Regeln. Der gezeigte Schedule kann bis auf die Wurzeln der beiden Transaktionen reduziert werden und damit ist erwiesen, dass dieser Schedule serialisierbar ist.

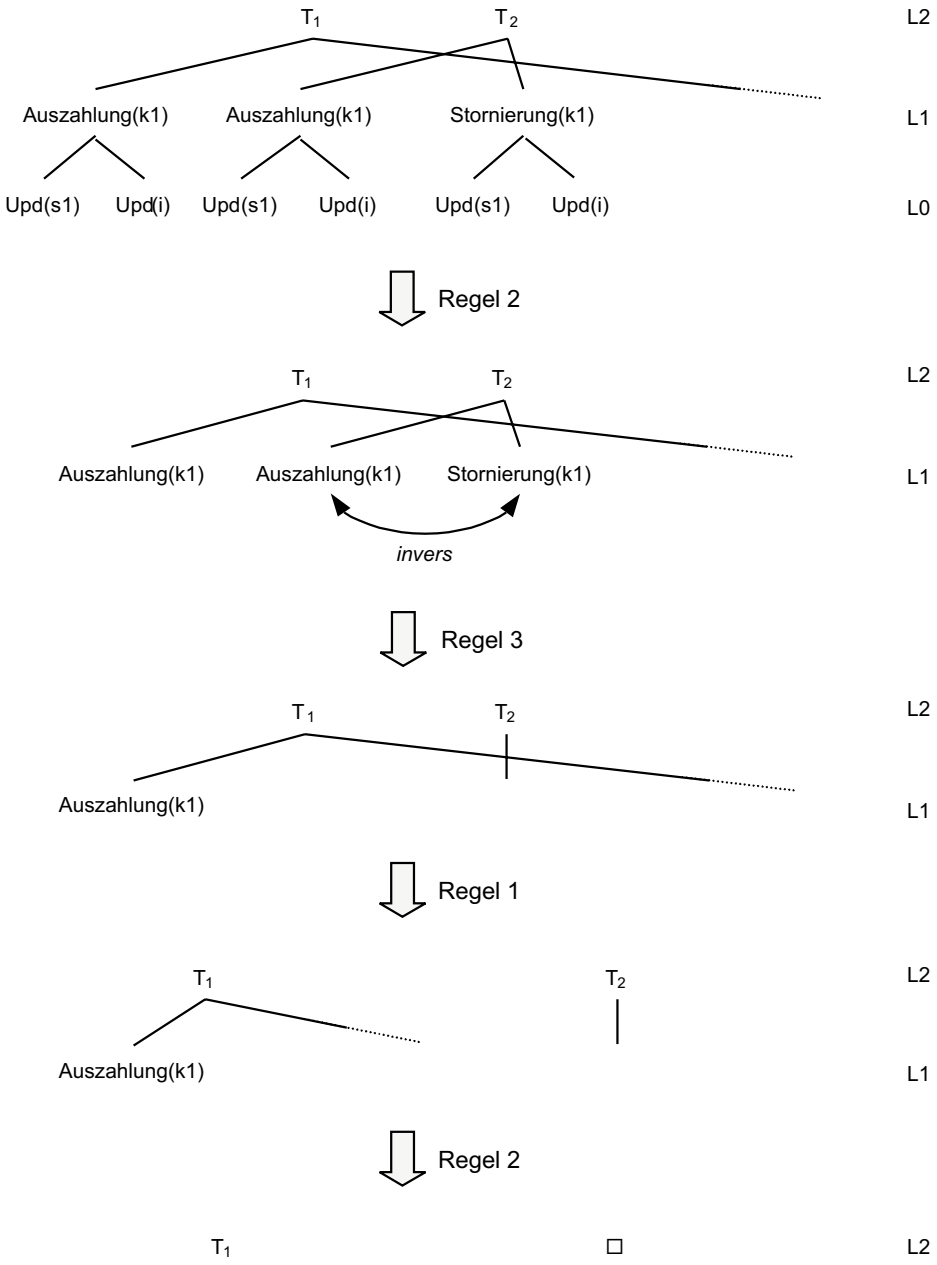

Abbildung 5.6: Behandlung einer abgebrochenen Mehrebenen-Transaktion [Sch96a].

Bis jetzt ist jedoch nur bekannt, wie die *globalen* Transaktionen serialisiert werden können. Zur Sicherstellung der globalen Serialisierbarkeit müssen jedoch nicht nur Kon-

flikte auf globaler Ebene erkannt werden. Ebenso müssen durch *lokale* Transaktionen verursachte indirekte Konflikte identifiziert werden. Das Mehrebenen-Transaktionsmodell stützt sich auf die semantischen Aspekte der Transaktionen und die mögliche Vertauschung von Operationen, sofern sie aus semantischer Sicht unabhängig sind. Nun kommt aber hinzu, dass lokale Transaktionen die globalen Operationen der Subtransaktionen derart beeinflussen können, dass sie nicht mehr unabhängig sind und deshalb nicht mehr vertauscht werden dürfen. Wird z. B. eine lokale Transaktion zwischen zwei globalen Subtransaktionen ausgeführt und schreibt oder liest sie Daten, die auch die Subtransaktionen bearbeiten, sind diese nicht mehr kommutierbar, da ihre Zwischenergebnisse bereits von der lokalen Transaktion aufgenommen wurden.

Unter der Annahme, die lokalen Systeme geben keine Informationen über die Transaktionen preis und auch eine Modifikation ist nicht möglich, soll nun ein Konzept eingeführt werden, das die Erkennung und implizite Vermeidung von denjenigen Konflikten vorsieht, welche die Serialisierbarkeit der Transaktionen verhindern würden. Dieses Konzept basiert auf der Idee, dass Konflikttests zwischen lokalen Transaktionen und Subtransaktionen durchgeführt werden [DSW94, WDSS93]. Hierzu wird ein spezielles Sperrverfahren eingesetzt. Dies ist möglich, da bei Mehrebenen-Transaktionen beliebige Synchronisationsverfahren auf den Ebenen genutzt werden können.

Das Sperrverfahren arbeitet mit zurückgehaltenen Sperren, d. h., die Sperren einer Subtransaktion werden an ihrem Ende nicht freigegeben (wie es das offen geschachtelte Transaktionsmodell vorsieht), sondern zurückgehalten. Damit ändert sich die Bedeutung der Sperren. Denn diese zurückgehaltenen Sperren werden nicht von allen Transaktionen beachtet, sondern es wird zwischen normalen und zurückgehaltenen Sperren unterschieden. Normale Sperren gelten für alle Transaktionen, während zurückgehaltene Sperren von globalen Subtransaktionen nur von lokalen Transaktionen beachtet werden müssen. Möchte also eine lokale Transaktion auf ein mit zurückgehaltener Sperre belegtes Objekt zugreifen, muss diese die Sperre berücksichtigen und bei einem Konflikt auf die Freigabe der zurückgehaltenen Sperren warten. Eine globale Subtransaktion hingegen kann die zurückgehaltene Sperre ignorieren und selbst eine normale Sperre anfordern. Dies gilt auch für Subtransaktionen anderer globaler Transaktionen, da die globalen Transaktionen auf globaler Ebene serialisiert werden und somit direkte Konflikte zwischen den Operationen der globalen Transaktionen vermieden werden. Alle zurückgehaltenen Sperren werden am Ende der globalen Transaktion wieder freigegeben.

Mit dieser Vorgehensweise wird verhindert, dass eine unverträgliche lokale Transaktion zwischen zwei globalen Transaktionen ausgeführt wird, die sich zeitlich überschneiden. Trotzdem kann es immer noch zu indirekten Abhängigkeiten zwischen globalen Transaktionen kommen, die von lokalen Transaktionen verursacht werden. In [Sch96a] wird jedoch gezeigt, dass diese Abhängigkeiten nicht zu Zyklen im globalen Abhängigkeitsgraphen führen können.

Es kann aber passieren, dass die lokalen Transaktionen ausgehungert werden. Dies geschieht genau dann, wenn vor Ende einer globalen Transaktion neue Subtransaktionen anderer globaler Transaktionen auf Objekte zugreifen, die von lokalen Transaktionen angefordert wurden. Dieses Problem lässt sich beheben, indem das Alter der lokalen und globalen Transaktionen berücksichtigt wird. Subtransaktionen, die mehr als eine zuvor festgesetzte Zeitspanne jünger sind als wartende lokale Transaktionen, müssen

nun ebenfalls auf die Freigabe der zurückgehaltenen Sperren warten. Nach Freigabe der Sperren erfolgt das Scheduling der Transaktionen anhand ihres Alters.

Wie kann nun dieser Ansatz auf Basis existierender lokaler Datenbanksysteme umgesetzt werden? Insbesondere unter der Voraussetzung, dass keine Änderungen wie beispielsweise erweiterte Schnittstellen und Schemata an den Datenbanken möglich sind. Obwohl die lokalen DBS eine Transaktionsverwaltung besitzen und ACID-Eigenschaften aufweisen, ist zur Realisierung der Konflikterkennung eine Erweiterung notwendig, die durch die Einführung einer zusätzlichen Transaktionsschicht über dem lokalen DBS erfolgt. Diese Schicht wird durch einen DBS-spezifischen Agenten realisiert, der folgende Anforderungen erfüllen muss:

1. Die Subtransaktionen globaler Transaktionen müssen erkannt werden. Dazu können zusätzliche Befehle wie „begin subtransaction“ und „end subtransaction“ eingeführt werden.

2. Subtransaktionen müssen an ihrem Ende ihre Sperren bis zum Ende der globalen Transaktion zurückhalten.

3. Das Ende einer globalen Transaktion muss erkennbar sein.

Da der Agent zwischen das lokale DBS und die existierende lokale Anwendung geschoben wird, muss er mindestens den Schnittstellenumfang des lokalen DBS enthalten und auch die Sicht auf die Daten muss dieselbe bleiben. Nur dann ist gewährleistet, dass die bestehende Anwendung nicht angepasst werden muss. Die Schnittstelle kann aber um Operationen zur Unterstützung der globalen Transaktionsverwaltung erweitert werden. Wie in Abbildung 5.7 dargestellt, laufen die globalen Subtransaktionen (GST) und die lokalen Transaktionen (LT) über den Agenten. Da der Agent eine vollständige Transaktionsverwaltung beinhaltet und somit eine weitere Ebene im Mehrebenen-Transaktionsmodell bildet, müssen die lokalen Transaktionen und Subtransaktionen nicht exakt Transaktionen in den lokalen DBS entsprechen. Stattdessen können neue Subtransaktionen gebildet werden, indem z. B. jeder Operationsaufruf einer Agenten-Transaktion eine eigene DBS-Transaktion (T1 - Tn) darstellt. Daraus ergeben sich mehrere Vorteile: durch die kürzeren Subtransaktionen sind die Sperrdauern in den lokalen DBS ebenfalls kürzer, Pseudokonflikte werden vermieden, Deadlocks über verschiedene Ebenen werden verhindert und aufgrund des sofortigen Commit der Subtransaktionen ist kein atomares Commit zwischen Agent und DBS-Transaktion notwendig. Als Nachteil erweist sich, dass mehr DBS-Transaktionen und dadurch auch mehr Transaktionsverwaltungsaufwand erzeugt wird und auch ein einfaches Rollback einer Agententransaktion nicht mehr möglich ist. Der Kommunikationsaufwand zwischen Agent und DBS muss deswegen aber nicht notwendigerweise höher sein, sondern kann aufgrund der entfallenen Begin- und Commit-Operationen geringer werden.

Trotz der Bildung von zusätzlichen Subtransaktionen ist die korrekte Ausführung von lokalen und globalen Transaktionen gewährleistet, da die Agenten mit ihrer Transaktionsverwaltung eine weitere Transaktionsebene bilden. Über diese Ebene laufen die globalen als auch die lokalen Transaktionen und somit sind alle Transaktionen mindestens zweischichtig. Direkt auf den DBS sind keine lokalen Transaktionen mehr zugelassen, so dass keine weiteren indirekten Konflikte entstehen können. Die Konflikterkennung in den

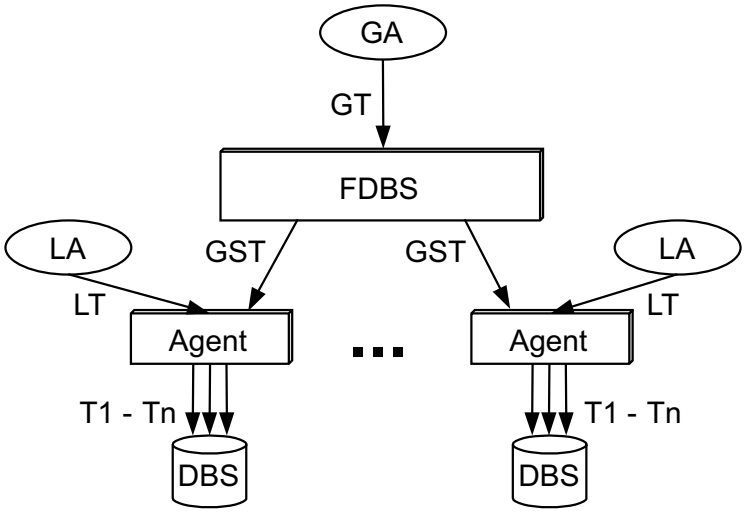

Abbildung 5.7: Alle Transaktionen laufen über den Agenten.

Agenten kann unterschiedlich genau sein, es muss aber jeder Konflikt erkannt werden. Die Korrektheit ist auch dann noch garantiert, wenn als Folge eines zu groben Konflikttests nicht vorhandene Konflikte angezeigt werden. Dies hat lediglich einen Einfluss auf die Effizienz des Systems, da zusätzliche Sperren einen zusätzlichen Vergleichsaufwand bedeuten und zusätzliche Transaktionsabbrüche und Wiederholungen nach sich ziehen.

5.3.2.3 Globale Atomarität

Aufgrund der offenen Ausführung von Mehrebenen-Transaktionen ist ein einfaches Rollback der bereits abgeschlossenen Subtransaktionen nicht mehr möglich. Stattdessen müssen sie abgebrochen und ihre Effekte mit Hilfe von Kompensationen zurückgesetzt werden. Um jedoch die Rücksetzbarkeit gewährleisten zu können, darf wie im zentralisierten Fall keine unverträgliche Operation ausgeführt werden, bevor die entsprechende Transaktion abgeschlossen ist. Das kann im FDBS für die globalen Transaktionen sichergestellt werden, die Einflüsse der lokalen Transaktionen werden jedoch nicht erkannt. Diese müssen von den Agenten koordiniert werden, die vor allem die Einflüsse der lokalen Transaktionen auf die Kompensierbarkeit der globalen Subtransaktionen kontrollieren müssen. Das folgende Beispiel verdeutlicht die Problematik.

Beispiel:

Eine globale Subtransaktion GST schreibt einem Konto einen Betrag von 100 Euro gut. Die zugehörige Kompensationstransaktion GST_K zieht diesen Betrag wieder ab. Sei beispielsweise der ursprüngliche Kontostand 1000 Euro, dann liegt er nach Ausführung GST bei 1100 Euro und nach Ausführung von GST_K wieder bei 1000 Euro. Damit ist wieder der ursprüngliche Zustand hergestellt.

Dies kann aber nur gewährleistet werden, wenn keine unverträgliche Operation dazwischen ausgeführt wird. Eine lokale Transaktion LK führt eine Zinsverrechnung auf dem aktuellen Kontostand durch. Schiebt sich LK zwischen GST und GST_K und schreibt dem Konto nach Ausführung von GST drei Prozent Zinsen gut und wird erst anschließend GST_K durchgeführt, dann erhält man folgenden Kontostand: (1000 + 100) * 1,03 - 100 = 1033 (Euro). Dies entspricht nicht dem ursprünglichen Guthaben von 1000 Euro.

Würde LT erst nach GST_K ausgeführt, dann ergäbe sich ein Kontostand von (1000 + 100 - 100) * 1,03 = 1030 (Euro).

Die zurückgehaltenen Sperren spielen in diesem Fall eine wichtige Rolle. Durch Anwendung der zurückgehaltenen Sperren werden alle lokalen Transaktionen, die einen Konflikt verursachen, bis an das Ende der globalen Transaktion verzögert. Dies ist eine wichtige Voraussetzung für die Kompensierbarkeit der globalen Subtransaktionen, da deren Effekte nur dann zurückgesetzt werden können, wenn keine unverträgliche Operation durchgeführt wurde.

Um unverträgliche Transaktionen auch nach einem Fehlerfall zurückzuhalten, müssen zurückgehaltene Sperren Transaktionsabstürze überleben. Denn selbst wenn es in einem lokalen System einen Transaktionsabsturz gibt, kann die globale Transaktion fortgesetzt werden. Da die Subtransaktion bereits erfolgreich abgeschlossen ist und ihre zurückgehaltenen Sperren unverträgliche Transaktionen zurückhalten, müssen bei Durchführung der Recovery zunächst wieder die zurückgehaltenen Sperren angefordert werden, bevor neue Transaktionen gestartet werden dürfen. Dazu müssen entsprechende Log-Einträge geschrieben werden, bevor die Sperren umgewandelt werden.

Im FDBS wird ein 2PC-Protokoll realisiert, bei welchem in der ersten Phase eine implizite Abstimmung zwischen dem Koordinator und den Partizipanten erfolgt. Die Änderungen einer Subtransaktion werden jedoch an ihrem Ende sofort sichtbar gemacht und nicht erst während der Commit-Phase. Nach Durchführung aller Operationen wandelt eine Subtransaktion alle ihren Sperren in zurückgehaltene Sperren um und geht in den Prepare-Status über. Dieser Status wird dem Koordinator mittels einer Commit-Bestätigung implizit mitgeteilt. Erst wenn alle Subtransaktionen einer globalen Transaktion abgeschlossen sind, kann der Koordinator das globale Commit entscheiden und festhalten. Nun können auch alle zurückgehaltenen Sperren aufgehoben und damit die Subtransaktionen endgültig abgeschlossen werden.

Im FDBS wird ein Fehlerfall wie gewohnt behandelt. Eine Ausnahme stellen die Subtransaktionen dar, die bei Transaktionsabbruch nicht durch ein einfaches Rollback, sondern durch Kompensationen zurückgesetzt werden. Für die globalen Transaktionen im FDBS ergeben sich dann die in Abbildung 5.8 dargestellten Transaktionszustände.

Eine globale Transaktion wird mit einem Begin-of-Transaction (BOT) gestartet und endet nach einer Sequenz von Subtransaktionen mit einem Commit (COT) oder Abort (AOT). Eine Subtransaktion einer gobalen Transaktion, kurz globale Subtransaktion, bezieht sich auf ein lokales DBS und wird durch das FDBS initiiert. Sie beginnt mit einem Begin-of-Subtransaction (BOS) und endet nach einer Folge von Operationen (op)

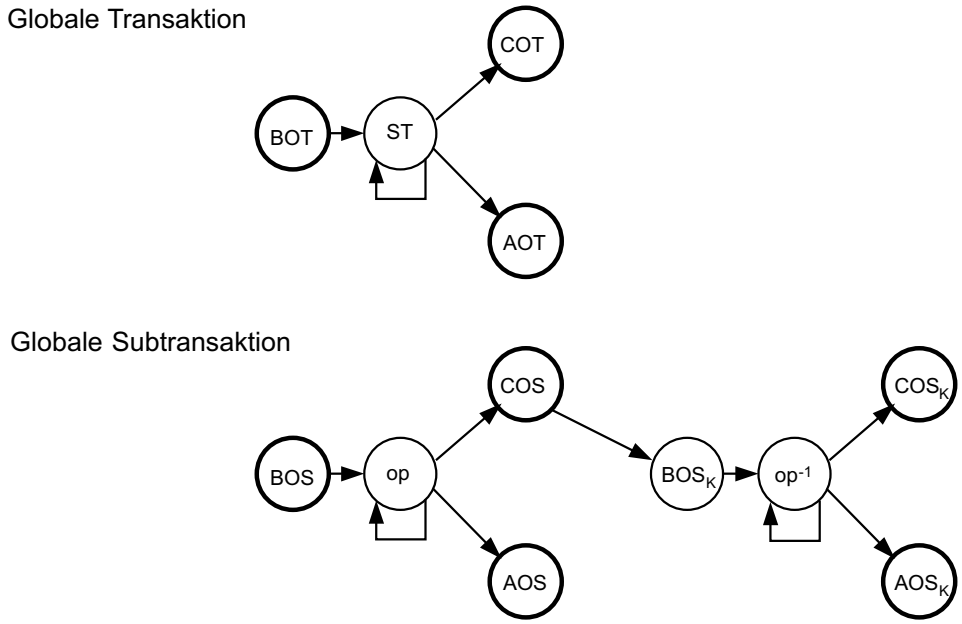

Abbildung 5.8: Zustände einer globalen Transaktion und einer globalen Subtransaktion.

mit einem Commit (COS) oder Abort (AOS). Muss eine Subtransaktion kompensiert werden, dann wird eine Subtransaktion zur Kompensation mit $\mathrm{BOS_K}$ gestartet, die eine Sequenz inverser Operationen ausführt und anschließend mit $\mathrm{COS_K}$ abschließt. Damit erreicht auch die Subtransaktion den Abort-Zustand. Bricht jedoch die kompensierende Subtransaktion ab ($\mathrm{AOS_K}$), so kann sie wiederholt werden, bis sie erfolgreich durchgeführt wurde. Ist ein Commit auch nach mehreren Versuchen nicht möglich, so konnte die Subtransaktion nicht kompensiert werden und das FDBS befindet sich nicht mehr in einem konsistenten Zustand.

Auch lokale Transaktionen werden mindestens zweischichtig ausgeführt, da sie durch den Agenten und somit durch eine weitere Transaktionsverwaltung (neben der im DBS) geschleust werden. Da es sich bei Mehrebenen-Transaktionen um ein offen geschachteltes Modell handelt, werden alle Transaktionen (lokale Transaktionen als auch Subtransaktionen) in den lokalen DBS immer sofort mit Commit abgeschlossen, während im Agenten die Operationen im Log gehalten werden müssen. Dies ist notwendig, um im Fehlerfall die inversen Operationen der Kompensation bestimmen zu können. Das wiederum impliziert, dass für jede Operation im lokalen DBS eine inverse Operation vorhanden sein muss. Existiert eine solche Operation nicht, dann kann nicht mit frühzeitigen Commit-Operationen gearbeitet werden, sondern es muss ein 2PC-Protokoll eingesetzt werden. Auf semantisch hohen Ebenen wie Anwendungen ist es teilweise sehr schwer, passende Inversen zu den ausgeführten Operationen zu bestimmen. Einfacher lässt sich dies auf niedrigeren Ebenen umsetzen, in denen eine simplere Semantik auf Basis einfacher Lese- und Schreiboperationen vorliegt.

Es ist sogar möglich, die inversen Operationen automatisch zu bestimmen. In [Sch96a] wird gezeigt, wie für SQL-Befehle die jeweils inversen SQL-Anweisungen ermittelt werden können. Dazu muss der Agent das Schema des DBS kennen (indem er es beispielswei-

se aus den Katalogtabellen ausliest) und z. B. für jede Insert-Operation die eindeutige Inverse bestimmen. Bei der Ausführung der Insert-Operation muss nun der Insert-Befehl, der Tabellenname und der eingefügte Wert in das Log-Protokoll geschrieben werden. Alternativ dazu könnte auch als inverse Operation das Delete, der Tabellenname sowie ein eindeutiger Schlüssel protokolliert werden. Diese Informationen werden mit einer Transaktions-ID gespeichert. Bei anderen Operationen müssen erst die zu ändernden Daten gelesen und deren Werte ebenfalls in das Log geschrieben werden, um später eine inverse Operation durchführen zu können. Dies trifft beispielsweise bei Delete- oder Update-Operationen zu.

Muss eine Crash-Recovery durchgeführt werden, dürfen beim Neustart des Systems neue Transaktionen erst zugelassen werden, wenn alle Sperren wiederhergestellt wurden. Ist dies nicht der Fall, wird eventuell nicht erkannt, dass neue Transaktionen in Konflikt mit laufenden Transaktionen stehen und somit nicht serialisierbare Abläufe entstehen. Dieser Fall ist vergleichbar zu Transaktionsabstürzen. Auch hier wurden die Subtransaktionen bereits erfolgreich abgeschlossen und zurückgehaltene Sperren angefordert. Folglich werden beim Neustart des Systems die Subtransaktionen wiederhergestellt. Anschließend müssen die zurückgehaltenen Sperren wieder angefordert werden, bevor neue Transaktionen bearbeitet werden können.

5.3.2.4 Globale Deadlock-Erkennung und -Vermeidung

Zur Erkennung von Deadlocks sind zwei Wege denkbar. Zum einen ist es durch die Kommunikation zwischen Agent und FDBS möglich, Informationen über die Wartebeziehungen in den lokalen Systemen abzurufen. Die zusätzliche Transaktionsschicht auf Basis der Agenten unterstützt mehr Offenheit und kann die benötigten Informationen bereitstellen, um im FDBS einen globalen Wartegraphen aufzubauen. Es werden auch keine Wartebeziehungen übersehen, da alle Transaktionen, d. h. globale Subtransaktionen als auch lokale Transaktionen, über den Agenten laufen. Durch den Austausch zwischen Agent und FDBS kann daher eine globale Deadlock-Erkennung und -Vermeidung implementiert werden.

Zum anderen kann es erwünscht sein, die Implementierung der Agenten so einfach wie möglich zu halten und eine Kommunikation zwischen Agent und FDBS weitestgehend einzuschränken. In diesem Fall ist der Einsatz von Timeout-Verfahren denkbar (vgl. Abschnitt 3.1.2.1). Es ist durch seine einfache Realisierbarkeit vor allem für heterogene Systeme geeignet. Problematisch ist aber die richtige Wahl des Timeout-Wertes, um Deadlocks nicht unnötig lange bestehen zu lassen oder Transaktionen unnötigerweise abzubrechen.

5.3.2.5 Aufbau und Aufgaben des Agenten

Der Agent als zusätzliche Transaktionsschicht ist in erster Linie dafür verantwortlich, Konflikte zwischen globalen Subtransaktionen und lokalen Transaktionen zu entdecken. Zusammen mit der Implementierung von zurückgehaltenen Sperren wird globale Serialisierbarkeit ermöglicht.

Das FDBS ist für den Agenten eine Anwendung. Die Schnittstelle des Agenten bietet aber einige spezielle Funktionen für das FDBS an, um zusätzliche Informationen zu

den im Agenten laufenden Transaktionen zu liefern. Dies unterstützt beispielsweise eine effizientere globale Deadlock-Erkennung.

Besonders wichtig ist jedoch die Unterscheidung zwischen globalen Subtransaktionen und lokalen Transaktionen im Agenten, um gegebenenfalls zurückgehaltene Sperren aufzusetzen. Dazu bietet die erweiterte Agenten-Schnittstelle neben dem üblichen „begin transaction“ für lokale Transaktionen ein „begin subtransaction“ für die Subtransaktionen. Nun weiß der Agent, ob eine Subtransaktion vorliegt und demzufolge eine zurückgehaltene Sperre angefordert werden muss.

Der Agent setzt sich aus mehreren Hauptbestandteilen zusammen: Anfrageanalysator, Schemaverwaltung, Transaktionsverwaltung mit Sperrverwaltung und Recovery. Die Aufgaben der einzelnen Bestandteile gestalten sich derart, dass der Anfrageanalysator jene Teile der abgesetzten Anfragen extrahiert, die für die Konflikttests relevant sind. Über die Schemaverwaltung werden die zur Analyse notwendigen Informationen herangezogen. Die Sperrverwaltung koordiniert die angeforderten Sperren und verzögert Operationen, wenn es zu einem Konflikt kommt. Die Recovery-Verwaltung hält alle relevanten Informationen in den Log-Dateien fest.

Wir gehen an dieser Stelle jedoch noch nicht auf die technischen Details der einzelnen Komponenten ein, da wir den Aufbau und die Aufgaben von FDBS und Agenten für den Einsatz in unserer Integrationsarchitektur mit der KIF erweitern. Dazu zeigen wir im nächsten Abschnitt zunächst die Probleme des Schaad-Ansatzes auf.

5.3.2.6 Bewertung hinsichtlich Unterstützung von Anwendungssystemen

Wendet man das vorgestellte Transaktionsmodell auf unsere Gesamtarchitektur und insbesondere auf die KIF und die zu integrierenden Anwendungssysteme an, kommen folgende Probleme auf:

1. Ein FDBS stellt eine Transaktionsverwaltung bereit, wie wir sie von konventionellen DBS gewohnt sind. Insbesondere werden ACID-Transaktionen unterstützt. Da dies bei der KIF nicht vorausgesetzt werden kann, ist eine Erweiterung um eine Transaktionskomponente notwendig.

2. Die Agenten können nicht zwischen KIF und Anwendungssystem positioniert werden, da sie hier die lokalen Transaktionen weder registrieren noch beeinflussen können. Diese laufen nämlich eine Ebene tiefer zwischen Anwendung und zugehörigem DBS ab. Stattdessen müssen die Agenten zwischen Anwendung und DBS geschoben werden.

3. Außerdem geht der Ansatz davon aus, dass das FDBS die Subtransaktionen direkt an den Agenten weitergibt. Dies ist bei der Integration von Anwendungssystemen jedoch nicht der Fall, da die Anfragen der KIF über Funktionsaufrufe der Anwendung umgesetzt werden und nicht direkt an den Agenten geschickt werden. Folglich ist eine Unterscheidung zwischen Subtransaktionen und lokalen Transaktionen nicht mehr möglich. Stattdessen erscheinen dem Agenten alle Transaktionen als lokale Transaktionen, da alle Transaktionen über die Anwendung laufen – auch die globalen Subtransaktionen. Hier muss zur Unterscheidung der Transaktionsarten anders vorgegangen werden.

Diese Probleme fordern eine Weiterentwicklung des in [Sch96a] vorgestellten Transaktionsmodell. Unser erweitertes Modell beschreiben wir im nächsten Abschnitt.

5.3.3 Erweitertes Transaktionsmodell

Das in Abschnitt 5.3.2 vorgestellte Transaktionsmodell kann nicht direkt für den Einsatz mit der KIF und den Anwendungssystemen übernommen werden. Es ist aber grundsätzlich als Transaktionsmodell für unsere Funktionsintegrationskomponente geeignet, da ebenfalls lokale DBS integriert und globale Serialisierbarkeit und Atomarität unterstützt werden sollen. Die Hauptunterschiede liegen in der Integration von Anwendungssystemen und einer Integrations-Middleware, die nicht auf einem DBS basiert.

Im Folgenden beschreiben wir die Erweiterungen an der Architektur, der KIF und den Agenten.

5.3.3.1 Erweiterte Architektur

Wir erweitern die Architektur in Abbildung 5.4 wie in Abbildung 5.9 dargestellt. Wie bereits zu Beginn dieses Kapitels erwähnt, konzentrieren wir uns zunächst auf die Funktionsintegrationskomponente, bevor wir anschließend die Integrationsarchitektur in ihrer Ganzheit bzgl. Transaktionsunterstützung betrachten.

Die KIF und die zu integrierenden Anwendungssysteme stellen ebenfalls eine Art FDBS dar. Der Unterschied liegt hier allerdings vor allem in der Funktionalität der KIF, die nicht mit bekannter DBS-Funktionalität zu vergleichen ist. Statt Ad-hoc-Anfragen zu unterstützen und eine Anfrageverarbeitung bereitzustellen, sind feste Abläufe der Anfragen, nämlich die Funktionsabbildungen abzuarbeiten. Diese Abläufe sind vordefiniert und unter einem bestimmten Namen abrufbar. Folglich ist beim Aufruf einer föderierten Funktion genau bekannt ist, was in der KIF verarbeitet werden muss. Dazu liegen Metainformationen vor, wie z. B. welches System und welche Funktionen und somit Operationen ausgeführt werden, welche Parameter mit den übergebenen Werten gefüttert werden usw. Mit Hilfe dieser Informationen und den Eingabewerten beim Aufruf der föderierten Funktionen kann die KIF während der Laufzeit die Verträglichkeit der Transaktionen prüfen. Folglich muss die KIF eine Transaktionsverwaltung unterstützen, deren Funktionalität ähnlich der eines DBS ist. Da die KIF auf Basis von kommerziellen Produkten umgesetzt werden soll, kann die Transaktionsunterstützung nicht ohne weiteres direkt in die KIF integriert werden. Daher stellen wir sie als separate Komponente neben die KIF.

Außerdem verändern wir die Position der Agenten. Sollen Konflikte zwischen globalen Subtransaktionen und lokalen Transaktionen erkannt werden, müssen die Agenten tiefer angesetzt werden. Diese liegen nun zwischen Anwendung und DBS der Anwendungssysteme. Würden wir sie zwischen KIF und Anwendungssystemen belassen, könnten sie die lokalen Transaktionen nicht registrieren und abfangen. Leider ist diese Position für den Agenten nicht bei jedem Anwendungssystem umsetzbar. Das Transaktionsmodell ist primär auf relationale DBS ausgelegt, so dass es bei anderen DBS-Typen wie Netzwerkdatenbanken nicht direkt angewendet werden kann. Auch Systeme, die einen Großteil der Logik (z. B. referentielle Integrität) direkt in der Anwendung implementieren, können Probleme bereiten. Wir müssen daher den Kreis der Anwendungssysteme

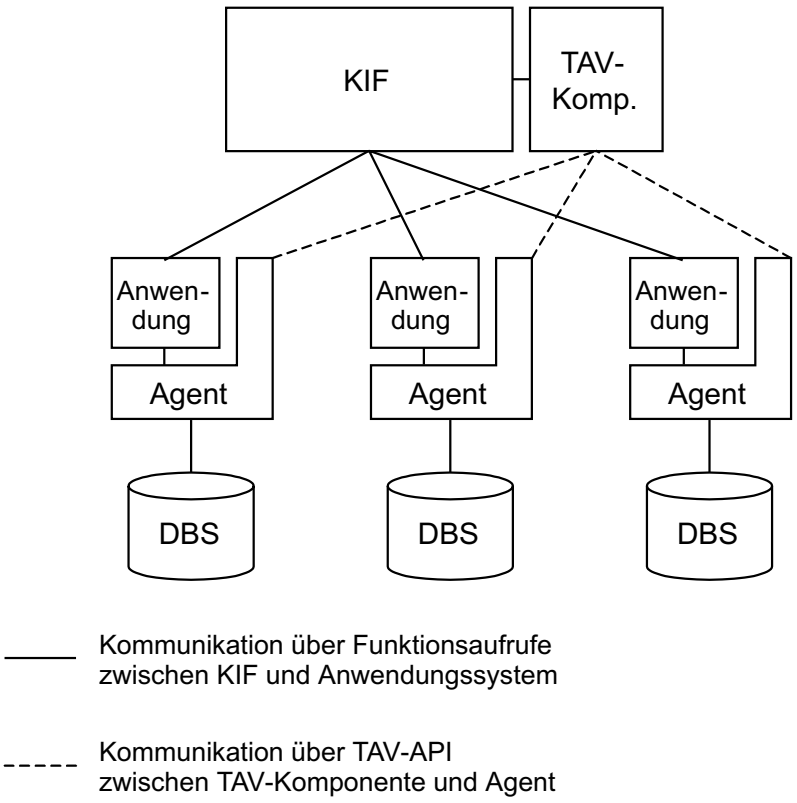

Abbildung 5.9: Architektur des erweiterten Transaktionsmodells auf Basis von Agenten.

weiter einschränken, für die wir eine Lösung anbieten können. Daraus resultierende Einschränkungen und alternative Vorgehensweisen beschreiben wir in Abschnitt 5.3.5.

Der Agent muss nun zwischen Subtransaktionen und lokalen Transaktionen unterscheiden können. Jedoch gestaltet sich diese Aufgabe schwierig, da beide Transaktionsvarianten von der Anwendung kommen. Diese teilt dem Agenten jedoch nicht mit, um welche Variante es sich handelt. Die Anwendung selbst weiß es nämlich auch nicht. Da die Anwendungen nicht verändert werden dürfen, ist es zudem zwingend notwendig, dass die Agenten die gleiche Schnittstelle anbieten wie das DBS. Somit muss für jeden DBS-Typ ein Agent bereitgestellt werden.

In den folgenden Abschnitten gehen wir näher auf die Aufgaben der KIF und des Agenten in unserem erweiterten Transaktionsmodell ein.

5.3.3.2 Aufgaben der KIF

Die KIF stellt im Gesamtbild der Integrationsarchitektur ein zu integrierendes Quellsystem dar und sollte daher zumindest die Eigenschaften eines wiederherstellbaren Systems aufweisen. Folglich muss der Prozess der Funktionsintegration selbst transaktional sein. Somit sollte die KIF eine Transaktionsverwaltung bereitstellen, welche die ACID-Eigenschaften der KIF-Transaktionen gewährleistet. Die KIF-Transaktionsverwaltung muss demnach ein Synchronisationsverfahren als auch Recovery-Mechanismen bereitstellen.

Zur Gewährleistung der Serialisierbarkeit wird das strikte 2PL-Verfahren als Synchronisationsverfahren eingesetzt, um kaskadierende Rücksetzungen zu vermeiden. Eine interessante Frage ist die Wahl der Sperrobjekte. Grundsätzlich kann bei der KIF zwischen den lokalen Systemen oder Funktionsparametern bzw. den korrespondierenden DB-Objekten gewählt werden. Stellen die lokalen Systeme die Sperrobjekte dar, so ist aufgrund der Größe der Objekte der Verwaltungsaufwand zwar relativ niedrig, es kommt jedoch zu sehr vielen Konflikten. Daher ist diese Granularität nicht zu empfehlen. Entscheidet man sich für die Funktionsparameter, so stellen Datenbankobjekte implizit die Sperrobjekte dar. Dabei muss darauf geachtet werden, welchem System die Parameter zugeordnet sind. Es kann der Fall auftreten, in welchem die Parameternamen identisch sind und somit ein Konflikt erkannt würde, die Parameter sich jedoch auf unterschiedliche Systeme beziehen und somit eigentlich gar kein Konflikt vorliegt.

Werden Funktionsparameter als Sperrobjekte angenommen, dann handelt es sich um logische Sperren. Dabei muss auf folgende Problematik geachtet werden. Da den Funktionen eines Systems dasselbe Datenbankschema zugrunde liegt, kann man davon ausgehen, dass sich identische Parameternamen in unterschiedlichen Funktionen auf dieselben Datenbankobjekte beziehen. Ist dies nicht der Fall, d. h., identische Namen weisen auf unterschiedliche Objekte, dann werden Konflikte ermittelt, die auf DB-Ebene nicht vorliegen. Dieses Verhalten hat keinen Einfluss auf die korrekte Ausführung der Transaktionen, sondern mindert lediglich den Durchsatz des Systems. Der umgekehrte Fall, d. h., dieselben Objekte werden über unterschiedliche Parameternamen referenziert, kann jedoch zu nicht erkannten Konflikten führen und damit zu inkonsistenten Daten. Da die Funktionen aber zu einem System gehören, liegt im Allgemeinen eine homogene Abbildung der Parameternamen auf die DB-Objekte vor. Im Zweifelsfall sollte dies bei der Erstellung der Funktionsabbildung genau untersucht werden, um in der KIF entsprechende Informationen zu hinterlegen. Diese Metadaten können dann darauf hinweisen, dass bestimmte Parameternamen dieselben Objekte referenzieren.

Zur Unterstützung von Recovery schreibt die KIF ein Log, das festhält, welche globalen Transaktionen in Bearbeitung waren und insbesondere welche Operationen bzw. Funktionen bereits abgeschlossen wurden oder gerade in Bearbeitung waren. Da die Agenten lokal dieselben Informationen für lokale Transaktionen als auch Subtransaktionen halten müssen, können bei Ausfall und Wiederanlaufen der KIF Zwischenstände der Subtransaktionen bei den Agenten erfragt werden.

Die Atomarität der globalen Funktionen wird nicht allein mit einem 2PC-Protokoll unterstützt, da die lokalen Systeme zumeist keine sichtbaren Prepare-to-Commit-Zustände bereitstellen. Daher muss im Zusammenspiel mit den Agenten ein Verfahren auf Basis von Kompensationen unterstützt werden, welche aber direkt von den Agenten implementiert werden.

5.3.3.3 Aufgaben des Agenten

Der Agent als zusätzliche Transaktionsschicht ist in erster Linie dafür verantwortlich, Konflikte zwischen globalen Subtransaktionen und lokalen Transaktionen zu entdecken und zurückgehaltene Sperren zu implementieren, um somit globale Serialisierbarkeit zu ermöglichen. Abbildung 5.10 zeigt die Hauptbestandteile und Schnittstellen des Agenten und der lokalen Anwendung. Wie in Abschnitt 5.3.2.5 beschrieben, besteht der

Agent aus Anfrageanalysator, Schemaverwaltung, Transaktionsverwaltung mit Sperrverwaltung und Recovery. Zum DBS hin bedient der Agent die DBS-spezifische SQL-Schnittstelle und muss dieselbe der Anwendung bereitstellen. Die Agentenschnittstelle ist um Operationen zur Unterstützung der globalen Transaktionsverwaltung erweitert (TAV-API), über welche die Transaktionsverarbeitung zwischen KIF und Agent abgestimmt wird. Die Anwendung stellt eine proprietäre Schnittstelle bereit, die von der KIF genutzt wird.

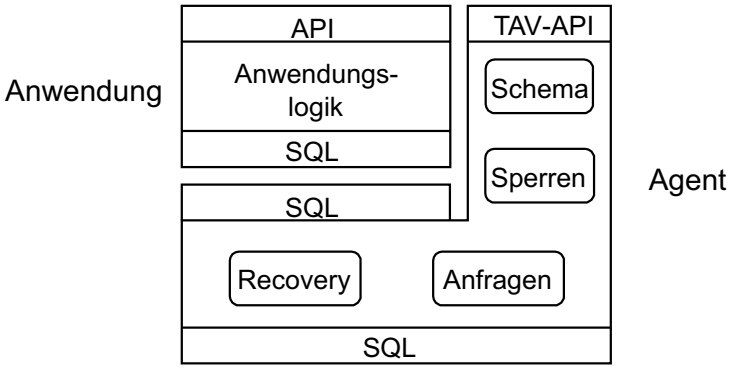

Abbildung 5.10: Die Komponenten und Schnittstellen des Agenten und der lokalen Anwendung.

Bevor wir die einzelnen Komponenten näher betrachten, werfen wir einen Blick auf die Konflikttests. Bei den unterstützten Sperren handelt es sich um logische bzw. Prädikatsperren. Physische Sperren auf Seiten oder Indizes sind nicht möglich, da der Agent hierzu keine Informationen hat. Er kann lediglich SQL-Anweisungen beobachten und mit den spezifizierten Prädikaten ermitteln, ob es zu Konflikten kommen kann. In [Sch96a] wird gezeigt, wie mit Hilfe von Obermengen komplexe Prädikate vereinfacht und Fälle vermieden werden, die sonst nicht entscheidbar wären. Bei Insert- und Update-Anweisungen können auch Präzisionssperren eingesetzt werden, um die komplexen Prädikatsvergleiche zu vereinfachen. Bei diesem Sperrverfahren werden nur die gelesenen Daten durch Prädikate gesperrt. Für zu aktualisierende Tupel werden hingegen Schreibsperren erteilt, wenn sie die Prädikate für gelesene Daten nicht erfüllen.

Außerdem ist die Kommutierbarkeit von SQL-Operationen zur Unterstützung der Mehrebenen-Transaktionen wichtig. Um die Verwaltung nicht komplex werden zu lassen, sollen lediglich Leseoperationen, d. h. SELECT-Anweisungen, als kommutierbar definiert sein. Sobald zwei SQL-Anweisungen auf ein gemeinsames Tupel zugreifen wollen und mindestens eine der beiden Anweisungen einem Insert, Update oder Delete entspricht, liegt ein Konflikt vor. Beispielsweise ist ein Prädikat ($10 > pno$ *and* $pno < 20$) verträglich mit den Prädikaten ($pno = 5$) oder ($name =$ '*chair*' *and* $pno > 100$), kommt aber in Konflikt mit den Prädikaten ($pno = 15$) oder ($name =$ '*chair*' *or* $pno > 100$) oder auch ($name =$ '*chair*'). In [Sch96a] werden unterschiedliche Verfahren zur Bestimmung von Konflikten aufgezeigt, die einen exakten Test machen oder mit Approximationen arbeiten.

Anfrageanalysator

Der Anfrageanalysator analysiert die SQL-Anweisungen und ermittelt mögliche Konflikte aufgrund der Befehlsart (Select, Insert, Update oder Delete), der Relationennamen sowie der Prädikate. Die Extraktion der benötigten Informationen kann spätestens zur Laufzeit der Anweisungen erfolgen. Bei Anwendungssystemen werden keine Ad-hoc-Anfragen mit beliebigem Aufbau abgeschickt, sondern durch die Funktionsschnittstelle sind für jede Funktion die korrespondierenden SQL-Befehle bekannt. Daher können diese Informationen teilweise schon vorab ermittelt und als Metadaten abgelegt werden. Beispielsweise sind die Operatoren der Prädikate als auch die Attribute vorher bekannt. Ähnliches gilt für die Tabellen, auf die zugegriffen wird. Somit konzentriert sich die Analyse vor allem auf die Werte, die mit den Prädikaten übergeben werden.

Weiterhin ist die Behandlung von gespeicherten Prozeduren, Sichten und Trigger nicht einfach, da deren Definition und damit die auszuführenden SQL-Befehle in der Datenbank hinterlegt sind. Folglich sieht der Agent diese SQL-Befehle normalerweise gar nicht, sondern nur den Aufruf der Prozeduren oder Sichten. Die Aktivierung von Triggern kann der Agent überhaupt nicht registrieren. Daher müssen ihm Informationen zu diesen Mechanismen vorliegen, damit der Anfrageanalysator beim Aufruf von Prozeduren oder Sichten diese ersetzen kann und die resultierenden SQL-Befehle als Grundlage für die Konfliktests heranzieht. Bei Triggern müssen zum einen die SQL-Befehle bekannt sein, die bei Aktivierung ausgeführt werden. Zum anderen muss bei der Analyse von SQL-Befehlen allgemein geprüft werden, ob durch deren Ausführung Trigger angestoßen werden. Ist dies der Fall, müssen auch für deren SQL-Anweisungen Sperren angefordert werden.

Da sich die implementierten SQL-Dialekte der verschiedenen Datenbankhersteller unterscheiden, muss für jeden DB-Typ ein angepasster Anfrageanalysator implementiert werden. Da sich die Unterschiede in Grenzen halten, ist das Erstellen von weiteren Anfrageanalysatoren einfacher, wenn bereits ein Analysator für ein relationales DBS existiert.

Schemaverwaltung

Die Schemaverwaltung enthält all jene Metadaten, die für die Analyse der SQL-Befehle sowie deren Umformungen und Vereinfachungen notwendig sind. Die Definitionen von gespeicherten Prozeduren, Sichten und Triggern gehören ebenso dazu wie Informationen aus den Katalogen der Datenbanksysteme. Mit den Metadaten prüft man, zu welchen Relationen die in der SELECT-Klausel aufgeführten Attribute gehören. Die in [Sch96a] vorgestellte Schemaverwaltung muss für die Funktionsintegration jedoch erweitert werden. Grund hierfür ist die notwendige Auseinanderhaltung von globalen Subtransaktionen und lokalen Transaktionen. Wie in Abschnitt 5.3.2.5 beschrieben, werden die Befehle „begin subtransaction“ und „end subtransaction“ eingeführt, um im Agenten zwischen lokalen Transaktionen und Subtransaktionen unterscheiden zu können. Diese Erweiterung hilft aber nur dann, wenn die vom FDBS bzw. der KIF abgesetzten Befehle direkt an den Agenten geschickt werden. Denn auf diese Weise kann der Agent die beiden Transaktionstypen unterscheiden.

Bei der Integration von Anwendungssystemen liegen andere Rahmenbedingungen vor. Da die Anfragen der KIF ebenfalls über die Anwendungen laufen, indem ihre Funktionen

aufgerufen werden, kommen alle Befehle von der Anwendung. Folglich kann der Agent zunächst nicht unterscheiden, ob es sich tatsächlich um eine lokale Transaktion oder vielmehr um eine globale Subtransaktion handelt. Um hier trotzdem eine Unterscheidung zu ermöglichen, müssen weitere Informationen in der Schemaverwaltung abgelegt werden.

Damit der Agent die SQL-Befehle der globalen Subtransaktionen erkennt, muss er wissen, welche SQL-Anweisungen durch eine Funktion angestoßen werden. Folglich muss der Schemaverwaltung zu allen über das Anwendungssystem zur Integration genutzten Funktionen die korrespondierende Sequenz an SQL-Anweisungen bekannt sein. Wird nun eine föderierte Funktion aufgerufen und damit eine globale Transaktion in der KIF initiiert, meldet die KIF den Beginn dieser Transaktion und deren Transaktions-ID zuerst an die Agenten der betroffenen Anwendungssysteme. Anschließend wird jedem Agenten jeweils direkt vor dem Aufruf einer der Funktionen seines Systems der Funktionsname als auch die zu übergebenden Parameterwerte mitgeteilt. Erst dann erfolgt der eigentliche Funktionsaufruf über das Anwendungssystem. Aufgrund der Informationen in der Schemaverwaltung zu dieser Funktion weiß der Agent, auf welche Sequenz von SQL-Befehlen mit welchen Werten er achten muss. Kommt bei der Bearbeitung der Transaktionen eine Sequenz vor, die genau diese SQL-Anweisungen mit den passenden Werten enthält, dann kann der Agent diese Transaktion als globale Subtransaktion kennzeichnen.

Es stellt sich die Frage, ob sich die Anweisungen der Subtransaktionen tatsächlich identifizieren lassen. Dies ist möglich, weil die Befehle der Sequenz nicht einzeln zusammengesucht werden müssen. Da diese SQL-Befehle innerhalb einer Funktion nach außen hin bereitgestellt werden, ist davon auszugehen, dass sie in einer Transaktion zusammengefasst sind. Folglich sucht der Agent nach einer Transaktion, die genau diese SQL-Befehle enthält. Nun könnte man annehmen, dass diese Funktion mehrfach aufgerufen wurde und somit nicht eindeutig feststellbar ist, welche zu genau dieser Subtransaktion gehört. Für diesen Fall helfen die Parameterwerte weiter, die der Funktion mitgegeben wurden. Wird eine Funktion mehrmals mit genau denselben Parameterwerten aufgerufen, bricht der Agent diese Transaktionen ab. Nach Ausführung aller Funktionen der globalen Transaktionen meldet die KIF das Ende der Transaktion mit der Transaktions-ID.

Die Unterscheidung zwischen den einzelnen Funktionsaufrufen und damit den Transaktionen lässt sich vereinfachen, wenn eine Erweiterung der Anwendungssysteme möglich ist. Kann man dem Agenten eine Art Transaktions-ID für die Transaktionen der lokalen Anwendung (also nicht die Transaktions-ID der globalen Transaktion) mitteilen, dann ist die Identifizierung der einzelnen Transaktionen einfacher und sicherer. Eine mögliche Umsetzung könnte wie folgt aussehen. Alle bereitgestellten Funktionen des Anwendungssystems werden um einen Eingabeparameter erweitert, über den ein Zeitstempel weitergegeben wird. Anschließend werden alle SQL-Befehle im lokalen DBS, die zu einer Funktion des Anwendungssystems gehören, um ein Prädikat in der WHERE-Klausel erweitert, das diesen Zeitstempel beinhaltet. Wenn die SQL-Anweisungen an den Agenten geschickt werden, kann dieser anhand des Prädikats mit dem Zeitstempel erkennen, welche SQL-Befehle zu einer Funktion gehören. Da der Agent diesen Zeitstempel auch von der KIF gemeldet bekommt, kann er auch zwischen globalen Subtransaktionen und lokalen Transaktionen unterscheiden. Bevor der Agent die SQL-Anweisungen an das lokale DBS weiterschickt, entfernt er das Zeitstempel-Prädikat. Auf diese Weise muss

zumindest das DB-Schema nicht verändert werden. Eine solche Änderung des lokalen Anwendungssystems ist sicher nur mit selbst implementierten Systemen umzusetzen. Bei Standard-Software ist dieser Ansatz eher nicht realisierbar.

Transaktions- und Sperrverwaltung

Die Sperrverwaltung implementiert das in Abschnitt 5.3.2.5 beschriebene strikte 2PL-Verfahren mit zurückgehaltenen Sperren. Die Sperren werden aufgrund der Ergebnisse des Anfrageanalysators vergeben. Die Sperrverwaltung prüft für alle Sperranforderungen, ob ein Konflikt mit bestehenden Sperren auftritt. Dies ist der Fall, wenn eine zu überprüfende Operation auf dieselbe Relation mit überlappenden Wertebereichen der Prädikate einer bereits laufenden Operation zugreifen möchte. Anschließend wird mit Hilfe von Signaturen für die relevanten Teile des Prädikats je Relation ein Vergleich gemacht, um mögliche Konfliktprädikate zu identifizieren. Verbleibende Kandidaten werden durch Vergleich der Wertebereiche je Attribut auf Überlappung getestet. Wird ein Konflikt entdeckt, wird die Transaktion in eine Warteschlange gehängt, welche die Sperre angefordert hat. Dabei können Deadlocks auftreten, die mittels eines Deadlock-Erkennungsalgorithmus vermieden werden.

In der Sperrverwaltung wird zwischen lokalen Transaktionen und globalen Subtransaktionen unterschieden. Wie die Unterscheidung erfolgt, wurde bei der Schemaverwaltung beschrieben. Gibt eine Subtransaktion an ihrem Ende Sperren frei, so werden diese von der Sperrverwaltung in zurückgehaltene Sperren umgewandelt, bis die komplette globale Transaktion abgeschlossen ist. Da die zurückgehaltenen Sperren nur für lokale Transaktionen gelten, kann nach Beendigung einer Subtransaktion geprüft werden, ob anstehende Subtransaktionen, die auf eine der Sperren gewartet haben, nun gestartet werden können. Stehen mehrere Subtransaktionen zur Wahl, muss hier erneut ein Konflikttest ausgeführt werden. Liegen keine Konflikte vor und können mehrere Transaktionen gestartet werden, dann bekommt die ältere Transaktion die Sperre zuerst. Subtransaktionen werden aber nur gestartet, wenn dadurch keine lokalen Transaktionen verhungern. Dies kann mit Hilfe von vordefinierten Zeitlimits entschieden werden. Abbildung 5.11 zeigt die möglichen Zustände einer globalen Subtransaktion. Sie verdeutlicht, dass nach dem Commit einer Subtransaktion deren Sperren in zurückgehaltene Sperren konvertiert werden und erst nach Commit der globalen Transaktionen endgültig freigegeben werden. In [Sch96a] werden außerdem Algorithmen zum Anfordern, Konvertieren und Freigeben von Sperren vorgestellt, auf die wir in dieser Arbeit jedoch nicht weiter eingehen.

Recovery-Verwaltung

Die Recovery-Komponente hat mehrere Aufgaben. Sie übernimmt die Verwaltung und Sicherung der Log-Einträge, das Erzeugen von inversen Operationen, das Durchführen der Recovery bei Systemabsturz und die Rücksetzung oder Kompensation einer Transaktion bei Transaktionsabbruch.

Vor der Ausführung eines SQL-Befehls wird ein Log-Eintrag in einen stabilen Speicher geschrieben, der erst am Ende einer Transaktion wieder gelöscht wird. Jedoch kommt die Erzeugung von Log-Einträgen für inverse Operationen neu hinzu. Außerdem ist überlegenswert, die Recovery-Verwaltung mit Hilfe des lokalen DBS zu realisieren. Dies

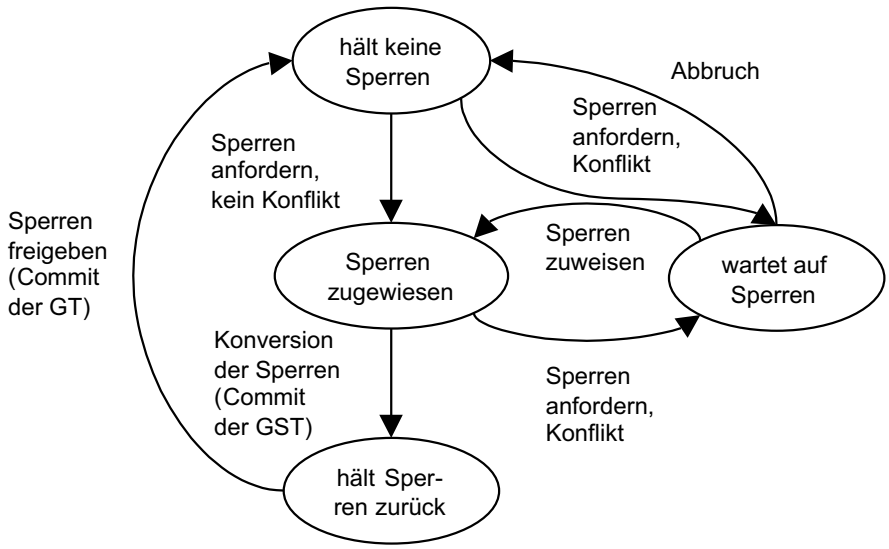

Abbildung 5.11: Zustände einer globalen Subtransaktion (GST) [Sch96a].

ist jedoch nur möglich, wenn der Datenbankeigner eine Erweiterung seines Datenbankschemas erlaubt. Eine Alternative wäre eine zusätzlich angelegte Datenbank in den existierenden DBS, um keine Modifikationen in den bestehenden Datenbanken durchführen zu müssen. Die Log-Einträge können dann auch mittels SQL ausgewertet werden, indem beispielsweise nach unvollständigen Transaktionen gesucht wird.

In Abschnitt 5.3.2 wurde erläutert, dass zur Ausführung von Kompensationen entsprechende inverse Operationen zu den ausgeführten Operationen benötigt werden, um diese wieder zurückzusetzen. Um jedoch z. B. ein Delete wieder zu kompensieren, muss zunächst das entsprechende Tupel gelesen werden, um die zu löschenden Daten in das Log zu schreiben. Solche zusätzlichen Anweisungen als auch die inversen Operationen selbst können in der Schemaverwaltung vordefiniert sein. Dieses Vorgehen hat jedoch den Nachteil, dass die Leistungsfähigkeit des Systems durch diese zusätzlichen, unter Umständen nicht wenigen Operationen beeinträchtigt werden kann. Vor allem die Datenübertragung zwischen Agent und DBS kann teuer werden. Daher wird in [Sch96a] vorgeschlagen, die Log-Einträge zu inversen Operationen durch Trigger umzusetzen. Im Detail bedeutet das, dass zu jeder Relation Trigger definiert werden, die bei einem Insert, Update oder Delete automatisch den entsprechenden Log-Eintrag schreiben. Auf diese Weise muss der Agent nicht involviert werden und zusätzlicher Kommunikationsaufwand entfällt. Außerdem erfolgt der Log-Eintrag innerhalb derselben Transaktion wie die Operation, wodurch Atomarität gewährleistet ist. In [Sch96a] sind Beispiele zu Insert- und Delete-Triggern nachzulesen.

5.3.3.4 Zusammenspiel von KIF und Agent

Der Agent ist als offenes System mit Schnittstelle konzipiert, so dass eine Kommunikation zwischen KIF und Agent möglich ist. Die KIF könnte beispielsweise die im Agenten vorkommenden aktuellen Wartebeziehungen abfragen, um eine globale Deadlock-Erkennung zu realisieren. Um die Kommunikation relativ einfach umzusetzen, kann ein

erweitertes SQL eingesetzt werden, das Funktionen wie 'show waits' zum Anzeigen der Wartebeziehungen bereitstellt. Diese zusätzlichen Funktionen müssen aber vom Agenten ausgeführt werden und können nicht an das DBS weitergereicht werden, da sie dort nicht zur Verfügung stehen.

Die Transaktionsverwaltung der KIF muss ein Commit-Protokoll unterstützen, um die Atomarität der globalen Transaktionen zu gewährleisten. Wie in Abschnitt 5.3.1 beschrieben, geht man jedoch nicht von einem sichtbaren Prepare-to-Commit-Zustand der integrierten Systeme aus. Stattdessen werden alle Änderungen am Ende einer Subtransaktion permanent gemacht und bei einem Abort der globalen Transaktion mittels Kompensationen wieder zurückgesetzt. In [Sch96a] können die notwendigen Algorithmen zur Durchführung der Commits für die KIF und den Agenten sowie genauere Erläuterungen nachgelesen werden.

5.3.4 Erweiterung des Transaktionsmodells auf die Gesamtintegrationsarchitektur

Nachdem in den bisherigen Betrachtungen die KIF mit den integrierten Anwendungssystemen im Vordergrund stand, soll nun der Blick auf die Gesamtarchitektur und deren Unterstützung von globalen Transaktionen gerichtet werden. Da die KIF selbst wieder ein zu integrierendes System darstellt, steht sie auf der gleichen Stufe wie die übrigen durch das FDBS zu integrierenden DBS.

Aufgrund der Transaktionsunterstützung in der KIF, die wir in den vorangegangenen Abschnitten erläutert haben, ist sie laut der vorgestellten Klassifikation zumindest ein wiederherstellbares System. Sind alle durch das FDBS integrierte DBS sowie die KIF mindestens wiederherstellbare Systeme, kann auch auf dieser Stufe das soeben vorgestellte Modell auf Basis von Mehrebenen-Transaktionen mit Agenten implementiert werden. Folglich werden auch Agenten für diejenigen DBS benötigt, die direkt vom FDBS integriert werden. Dies sollte aber keinen großen Aufwand verursachen, da man in den meisten Fällen dieselben DBS-Typen vorfinden wird wie bei den Anwendungssystemen. Bleibt lediglich ein Agent für die KIF hinzuzufügen. Das FDBS muss dieselbe Funktionalität zur Unterstützung globaler Funktionen anbieten, wie es für die KIF beschrieben wurde. Folglich implementiert es ein striktes 2PL-Verfahren und zusammen mit den in den Agenten verwirklichten zurückgehaltenen Sperren wird die globale Serialisierbarkeit gewährleistet.

Da die KIF nun selbst globale Subtransaktionen verarbeitet, untersuchen wir die Aufgaben des KIF-Agenten näher. Im Vergleich zu den DB-Agenten sind zwei Aspekte anders gelagert:

1. Es sind keine lokalen Transaktionen auf KIF-Ebene vorhanden, da die KIF Teil der Integrations-Middleware ist und daher nur globale Subtransaktionen verarbeitet.

2. Kompensationen auf SQL-Ebene können nicht umgesetzt werden. Stattdessen müssen die Effekte der Funktionsaufrufe der KIF durch kompensierende Funktionsaufrufe rückgängig gemacht werden.

Der erste Aspekt vereinfacht die Agentenfunktionalität, da aufgrund fehlender lokaler Transaktionen die zurückgehaltenen Sperren überflüssig werden. Folglich muss der KIF-Agent keine zurückgehaltenen Sperren implementieren.

Der zweite Aspekt erschwert die Unterstützung von Kompensationen. Solange die Agenten auf SQL-Ebene arbeiten, können kompensierende SQL-Anweisungen generiert werden. Dies ermöglicht vor allem die flexible SQL-Schnittstelle. Wie sieht es aber mit fest vordefinierten Funktionen einer Anwendungssystemschnittstelle aus? Das Anwendungssystem muss für jede bereitgestellte Funktion eine Kompensationsfunktion anbieten, damit frühzeitig veröffentlichte Effekte bei einem globalen Abort wieder rückgängig gemacht werden können. Diese Voraussetzung ist aber im Allgemeinen nicht zu erfüllen. Die lokalen Systeme bieten ihre Funktionen nicht paarweise mit Kompensationsfunktionen an. Daher benötigen wir für die KIF eine andere Vorgehensweise.

Wie in den Abschnitten zuvor erläutert, werden in der KIF die Kompensationen der lokalen Funktionen auf unterster Ebene – der SQL-Ebene im lokalen DBS – durchgeführt. Auf dieser Ebene werden die Transaktionen vorzeitig abgeschlossen und mit zurückgehaltenen Sperren unverträgliche Transaktionen verzögert. Sind in einer globalen SQL-Anweisung auf FDBS-Ebene mehrere SQL- und KIF-Aufrufe enthalten, dann muss auch für einen KIF-Aufruf eine Kompensation bereitgestellt werden. Modellieren wir dies jedoch in einer separaten KIF-Transaktion, so können wir nicht mehr auf die zurückgehaltenen Sperren in den lokalen DB zurückgreifen, da diese bereits beim ersten erfolgreichen KIF-Aufruf wieder aufgegeben werden. Daher sollten die durch KIF-Aufrufe initiierten Subtransaktionen nicht vorzeitig abgeschlossen werden, sondern bis zum Ende der globalen Transaktion auf FDBS-Ebene gehalten werden. Dies können wir folgendermaßen umsetzen. Nach der erfolgreichen Ausführung aller lokalen Funktionen einer föderierten Funktion wird diese nicht als erfolgreich abgeschlossen. Stattdessen meldet sie dem KIF-Agenten ein Ready-to-Commit und wartet anschließend auf die Rückmeldung des KIF-Agenten. In der Zwischenzeit bestehen nach wie vor alle zurückgehaltenen Sperren in den lokalen DBS. Meldet das FDBS ein Commit, meldet dies der KIF-Agent an die KIF und diese kann nun die Ausführung der föderierten Funktion erfolgreich abschließen. In diesem Moment werden auch alle zurückgehaltenen Sperren wieder freigegeben. Meldet das FDBS hingegen ein Abort, dann meldet dies ebenfalls der KIF-Agent an die KIF, die wiederum alle lokalen Transaktionen zurücksetzen kann. Dies ist auch unter Wahrung der globalen Serialisierbarkeit möglich, da noch alle zurückgehaltenen Sperren bestehen und somit die Kompensationsoperationen durchgeführt werden können, ohne dass inzwischen unverträgliche Transaktionen ausgeführt wurden.

Der Ablauf einer KIF-Subtransaktion gestaltet sich dann folgendermaßen. Auf FDBS-Ebene wird eine globale Transaktion angestoßen, die mehrere SQL-Aufrufe und auch mehrere KIF-Aufrufe enthalten kann. Diese Aufrufe initiieren Subtransaktionen in den direkt integrierten DBS und der KIF. In den DBS laufen die Subtransaktionen ab, wie dies der Ansatz von Schaad vorsieht. Die KIF verfügt über eine Transaktionsverwaltung, welche die Verarbeitung der föderierten Funktionen serialisiert. Die Transaktionen der föderierten Funktionen werden wiederum in Subtransaktionen aufgeteilt, die von den integrierten Anwendungssystemen durchgeführt werden. Während die Subtransaktionen in den Anwendungssystemen ihre Ergebnisse vor dem globalen Commit gegenüber der KIF sichtbar machen, melden die Subtransaktionen in der KIF lediglich ein Prepare-

to-Commit an das FDBS. Erst wenn das FDBS das globale Commit ausgibt, schließen auch die KIF-Subtransaktionen erfolgreich ab und die Agenten auf der untersten Ebene können die zurückgehaltenen Sperren freigeben. Abbildung 5.12 zeigt die Gesamtarchitektur des Integrationsansatzes.

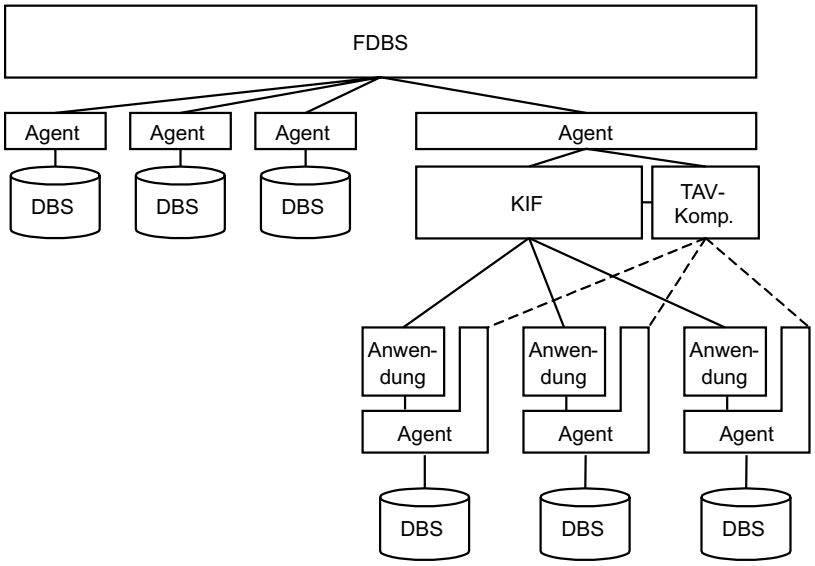

Abbildung 5.12: Gesamtarchitektur des erweiterten Transaktionsmodells.

Abschließend bewerten wir unser Transaktionsmodell anhand der Konsistenzebenen von SQL:1999 (vgl. 3.1.2.1). Die einzelnen Stufen werden nach den in Kauf genommenen Anomalien Lost Updates, Dirty Read, Non-Repeatbale Read und Phantome definiert. Die Konsistenzstufe SERIALIZABLE erlaubt keine der Anomalien.

Das vorgestellte erweiterte Transaktionsmodell unterstützt die Stufe SERIALIZABLE, was wir in zwei Schritten erläutern. Betrachten wir zunächst die KIF und die zu integrierenden Anwendungssysteme. Die KIF implementiert für ihre Transaktionen ein striktes 2PL-Verfahren, das außer dem Phantom-Problem keine Anomalien zulässt. Phantome treten bei physischen Sperren auf, werden aber bei logischen Sperren vermieden. Da unser Transaktionsmodell notwendigerweise auf logischen Sperren aufbaut, können auch Phantome ausgeschlossen werden. Allerdings kommt es aufgrund der logischen Sperren zu Einschränkungen bezüglich der Sperrgranulate.

Auch in den Anwendungssystemen wird ein SERIALIZABLE hinsichtlich dem Zusammenspiel von globalen Subtransaktionen und lokalen Transaktionen erreicht. Dafür sorgen vor allem die zurückgehaltenen Sperren. Zwar werden die Ergebnisse der Subtransaktionen früh sichtbar gemacht, doch nicht gegenüber lokalen Transaktionen. Muss ein Abort durchgeführt und dadurch eine Subtransaktion kompensiert werden, ist aufgrund der zurückgehaltenen Sperren keine lokale Transaktion dazwischen verarbeitet worden, die nun ein Dirty Read durchgeführt hat.

Eine Ebene höher unterstützt das FDBS ebenso ein striktes 2PL-Verfahren zusammen mit den zurückgehaltenen Sperren der Agenten. Somit folgt die gleiche Argumentati-

on wie bei der KIF und unsere Integrationsarchitektur unterstützt die Konsistenzstufe SERIALIZABLE.

5.3.5 Transaktionsunterstützung in Produkten

Wie gut unterstützen kommerziell verfügbare Produkte das vorgestellte Transaktionsmodell? Dieser Frage gehen wir in diesem Abschnitt nach. Dabei betrachten wir zunächst, welche Funktionalitäten unterstützt werden und wie sich diese auf die Verarbeitung von Transaktionen und mögliche Einschränkungen auswirken.

Momentan stellen keine Produkte Mechanismen zur Unterstützung globaler Serialisierbarkeit bereit. Zwar werden von den führenden Datenbankherstellern Lösungen zur Datenintegration angeboten, doch ermöglichen diese Produkte in erster Linie den Zugriff von einem Datenbank-Server auf mehrere eventuell auch heterogene DBS. Obwohl sich die Forschung schon seit einiger Zeit mit der Datenintegration auseinandersetzt, sind die kommerziellen Produkte eher als junge Entwicklungen einzustufen, die zunächst die Zugriffstransparenz unterstützen wollen. Die zahlreichen Arbeiten auf dem Gebiet der globalen Transaktionen und insbesondere der globalen Serialisierbarkeit zeigen, dass es sich um ein sehr komplexes Problem handelt. Häufig arbeitet man mit bestimmten Einschränkungen oder Anforderungen an die lokalen Systeme, um überhaupt eine Lösung anbieten zu können. Es scheint daher eher unwahrscheinlich, dass in naher Zukunft mit einer allgemeinen Unterstützung von globaler Serialisierbarkeit in Produkten zu rechnen ist.

Etwas besser sieht es mit globaler Atomarität aus. Hier arbeiten die Hersteller vor allem mit dem 2PC-Protokoll, um sicherzustellen, dass eine Transaktion über mehrere Systeme hinweg vollständig oder gar nicht ausgeführt wird. Dies ist aber nur möglich, wenn die beteiligten Systeme ebenso das 2PC-Protokoll unterstützen, also ein sichtbares Prepare-to-Commit anbieten. Liegen größtenteils nicht-transaktionale Systeme vor, können schreibende Zugriffe immer nur auf einzelne Systeme ausgeführt werden. Logisch zusammengehörende Operationen, die normalerweise in einer Transaktion durchgeführt werden sollten, können nicht mehr in einer Transaktion verarbeitet werden, sondern müssen in mehrere zerlegt werden. Folglich muss der Entwickler die Atomarität einer solchen Transaktionsfolge in der Anwendung implementieren.

Abschließend betrachten wir die globale Deadlock-Erkennung und -Vermeidung. Auch hier ist derzeit keine Unterstützung von den Herstellern bekannt, da auf dieser Ebene derzeit keine Kommunikation zwischen den Systemen möglich ist. Es kann lediglich über SQL-Anweisungen gearbeitet werden. Um jedoch Informationen zu Wartebeziehungen untereinander austauschen zu können, benötigt man eine erweiterte Schnittstelle. Da eine solche Schnittstelle nicht standardisiert ist, kann auch in naher Zukunft nicht mit besserer Unterstützung zur globalen Deadlock-Vermeidung gerechnet werden.

Wie müssen demnach die Anweisungen bzw. Operationen aussehen, mit denen die Konsistenz der Daten trotzdem gewährleistet werden kann? Grundsätzlich sind zwei Vorgehensweisen denkbar. Entweder gibt es in einer Transaktion immer nur genau eine schreibende Operation auf einem System. In diesem Fall kommen die Probleme der globalen Serialisierbarkeit und globalen Atomarität gar nicht auf, da die Transaktionen nicht systemübergreifend sind. Für die globale Deadlock-Erkennung bleibt nur ein

auf Timeout basierendes Verfahren. Es bleibt jedoch die Frage, wie es mit Operationen aussieht, die logisch zusammengehören und auch nur alle zusammen oder gar nicht durchgeführt werden sollen, damit die Datenkonsistenz gesichert ist. Hier muss der Entwickler eine Lösung mit Hilfe von Anwendungslogik entwickeln, um ein Zurücksetzen der bereits erfolgreich abgeschlossenen Transaktionen zu ermöglichen.

Der zweite Weg sieht vor, dass zwar systemübergreifende Schreiboperationen erlaubt sind, nun aber die Parallelität der Ausführung leidet, da diese Transaktionen sequentiell abgearbeitet werden müssen, um keine Inkonsistenzen zu verursachen. Die Deadlocks stellen aber nach wie vor ein Problem dar, das erneut mit Timeouts anzugehen ist.

5.3.6 KIF-Implementierung mit einem Workflow-System

Abschließend beschreiben wir die Umsetzung des vorgestellten Transaktionsmodells mit einem Workflow-System, das die Rolle der KIF und damit die Integration der Anwendungssysteme übernimmt. Es soll erneut die Integration der Anwendungssysteme im Vordergrund stehen und nicht die komplette Integrationsarchitektur.

In der Vergangenheit gab es bereits einige Überlegungen zu transaktionalen Workflows (*transactional workflows*, [SR93]). Nichtsdestotrotz ist bis heute das Workflow-System nicht dazu gedacht, Transaktionen zu unterstützen, wie sie im Bereich von Datenbanken bekannt sind. Die meisten kommerziellen WfMS sehen keine Unterstützung von Serialisierbarkeit vor. Ihr Fokus richtet sich auf die Atomarität der Workflows und der damit verbundenen Recovery-Mechanismen. Um globale Serialisierbarkeit der globalen Transaktionen zu gewährleisten, muss das WfMS um eine Transaktionsverwaltung erweitert werden. Da die Transaktionsverwaltung nicht direkt in die WfMS-Produkte eingebaut werden kann, müssen die Produkte erweitert werden. Dies geschieht über eine externe Komponente, deren Funktionalität über eine Aktivität des Workflows abgerufen werden kann (siehe Abbildung 5.13).

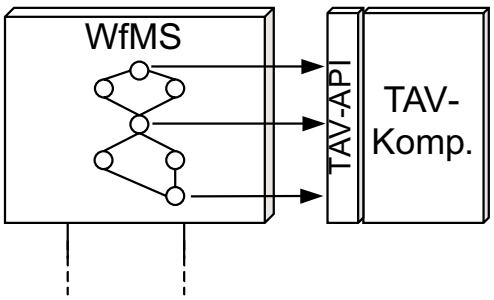

Abbildung 5.13: Zusammenspiel von WfMS und Transaktionsverwaltung (TAV-Komp.).

Wie erfolgt die Einbindung dieser Komponente? Hierzu müssen die Workflows zunächst um eine Aktivität erweitert werden, die zu Beginn des Workflows die Transaktionsverwaltung aufruft und mitteilt, welcher Workflow-Prozess anläuft. Die Transaktionskomponente hat zu allen definierten Workflow-Prozessen Informationen hinterlegt und weiß daher, welche Sperren angefordert werden müssen. Da es sich aber um logische Sperren

handelt, müssen in einer weiteren Aktivität die Werte der Parameter übergeben werden. Somit können zumindest für die erste Subtransaktion die Sperren angefordert und erteilt werden. Da in einem Workflow-Prozess Abhängigkeiten zwischen den einzelnen Aktivitäten vorkommen können und somit die weiteren Parameterwerte erst durch die Ergebnisse zuvor ausgeführter Aktivitäten ermittelt werden, können die entsprechenden Sperren erst im Laufe des Workflows angefordert werden. Da hierzu wieder eine Kommunikation mit der Transaktionsverwaltung notwendig ist, muss vor jeder Aktivität des ursprünglichen Workflows eine weitere Hilfsaktivität eingebaut werden, über welche die Parameterwerte für die anzufordernden Sperren übergeben werden. Da die Sperren sukzessive angefordert und vergeben werden, kann es vorkommen, dass eine globale Transaktion bzw. ein Workflow-Prozess warten muss, bis die benötigten Sperren genehmigt werden. Um den Workflow-Prozess entsprechend aufzuhalten, sind zwei Arten der Kommunikation zwischen WfMS und Transaktionsverwaltung denkbar. Entweder wird das Ergebnis der Sperranforderung erst an die Aktivität zurückgesandt, wenn die Sperre tatsächlich vergeben wurde (auch nach Wartezeiten). Oder es wird im Workflow eine Schleife implementiert, die bei der Transaktionskomponente immer wieder abfragen lässt, ob die Sperre nun genehmigt ist, wenn sie zuvor abgelehnt wurde. Sind alle Funktionen durch den Workflow-Prozess aufgerufen worden, dann muss am Ende des Prozesses eine letzte Aktivität eingefügt werden, die nun der Transaktionskomponente das Commit der Transaktion mitteilt.

Für die Agenten bei den Anwendungssystemen ergeben sich keine Änderungen zu den Beschreibungen in Abschnitt 5.3.2. Diese arbeiten mit der Transaktionsverwaltung des WfMS zusammen, um die globale Serialisierbarkeit mit zurückgehaltenen Sperren zu gewährleisten.

Abschließend weisen wir darauf hin, dass das Einfügen der notwendigen Aktivitäten zur Transaktionsunterstützung automatisiert erfolgen sollte. Dies ist in der Form möglich, dass das Ergebnis der Transformationen der Abbildungssprache in die entsprechende Workflow-Beschreibungssprache noch einmal mittels einer XSLT-Definition in einen erweiterten Transaktions-Workflow-Prozess abgebildet wird. Dazu werden automatisch Aktivitäten an den Anfang und das Ende des Workflows gesetzt und für jede Aktivität, die einen Funktionsaufruf in einem Anwendungssystem aktiviert, eine weitere Aktivität eingefügt. Für Aktivitäten, die Hilfsfunktionen zur Bearbeitung der Daten heranziehen, werden keine zusätzlichen Transaktionsaktivitäten benötigt.

Das folgende Beispiel verdeutlicht die Umsetzung mit dem Workflow-System (siehe Abbildung 5.14). In diesem Beispiel betrachten wir den Aufruf einer föderierten Funktion als eine globale Transaktion innerhalb der KIF. Die föderierte Funktion `SetQualitaet-Zuverlaessigkeit(IN LiefNr, IN Faktor)` ändert die Werte für Qualität und Zuverlässigkeit des mit `LiefNr` identifizierten Lieferanten. Der neue Wert wird mit dem mitgelieferten `Faktor` berechnet. Der Workflow-Prozess links in Abbildung 5.14 zeigt die zugehörige Abbildung auf die lokalen Funktionen. Zunächst werden die aktuellen Werte für Qualität und Zuverlässigkeit mit den Funktionen `GetQualitaet(IN LiefNr, OUT Qual)` und `GetZuverlaessigkeit(IN LiefNr, OUT Zuverl)` aus den jeweiligen Systemen ausgelesen. Anschließend werden die zurückgelieferten Werte mit dem angegebenen Faktor multipliziert (nicht explizit als Aktivität dargestellt). Die neuen Werte werden mit den Funktionen `SetQualitaet(IN LiefNr, IN Qual)` und `SetZuverlaessigkeit-`

`(IN LiefNr, IN Zuverl)` in die Systeme zurückgeschrieben, die anschließend einen Return Code (RC) zurückmelden.

Der erweiterte Workflow-Prozess ist rechts in Abbildung 5.14 dargestellt. Der Workflow-Prozess beginnt nun mit der Aktivität 1, die der KIF-Transaktionskomponente den Beginn dieser Transaktion meldet. Die Transaktionsverwaltung nimmt diese in ihre Synchronisationsverfahren mit auf. Als Nächstes läuft der Prozess weiter und kommt zu den Verwaltungsaktivitäten 2 und 3. Diese melden der Transaktionsverwaltung, welche lokale Funktion ausgeführt werden soll und welche Sperren (Read oder Write) auf welchen Objekten in welchem System benötigt werden. Hinzukommen die für die logischen Sperren benötigten Werte der Objekte. Die Transaktionskomponente teilt dem Agenten mit, welche Funktionen mit welchen Parameterwerten aufgerufen werden. Der Agent weiß nun, auf welche zugehörige Sequenz an SQL-Anweisungen er achten und als globale Subtransaktion deklarieren muss.

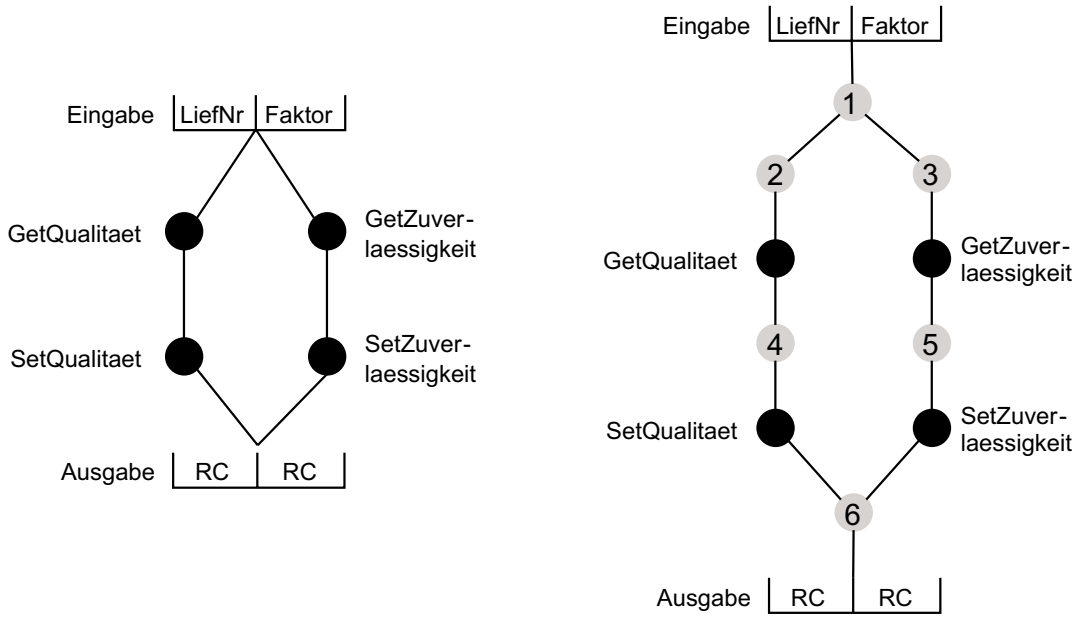

Abbildung 5.14: Der Workflow-Prozess für die Abbildung der föderierten Funktion `SetQualitaetZuverlaessigkeit` (links) und die Erweiterungen für die Transaktionsunterstützung (rechts).

Sind alle angeforderten Sperren verfügbar, können die Aktivitäten für die Funktionsaufrufe gestartet werden. Erkennt der Agent die entsprechenden Anweisungen, werden nach erfolgreicher Durchführung zurückgehaltene Sperren für diese Transaktionen erteilt und der KIF-Transaktionsverwaltung ein Prepare-to-Commit gemeldet. Der Workflow schreitet zu den beiden Verwaltungsaktivitäten 4 und 5 fort. Diese melden wie die Aktivitäten 2 und 3, welche Funktionen im nächsten Schritt ausgeführt werden und welche Sperren dazu benötigt werden. Diese Informationen gehen wiederum an den Agenten und die TAV-Komponente lässt den Prozess weiterlaufen, wenn die angeforderten Sperren zugeteilt wurden. Es können nun die Aktivitäten für die Funktionen `SetQualitaet` und `SetZuverlaessigkeit` gestartet werden. Der Agent fängt wieder die zugehörigen

SQL-Anweisungen ab und teilt nach erfolgreicher Durchführung die zurückgehaltenen Sperren zu.

Nachdem der ursprüngliche Workflow-Prozess durchlaufen ist, wird eine letzte Aktivität eingefügt, die der Transaktionsverwaltung das Ende der Transaktion oder des Prozesses meldet. Sobald die TAV-Komponente ein Commit vom FDBS angeordnet bekommt, gibt sie das Commit an die Agenten weiter. Diese können nun ihre zurückgehaltenen Sperren aufheben und bestätigen das Commit an die TAV-Komponente, die dies wiederum an die letzte Aktivität zurückmeldet. Damit sind der Workflow und die föderierte Funktion erfolgreich verarbeitet. Meldet das FDBS hingegen ein Abort, geht diese Information ebenfalls an die Agenten. Diese initiieren nun die Kompensationsoperationen für die SQL-Anweisungen. Nach erfolgreicher Kompensation bestätigen die Agenten den Abort an die TAV-Komponente, die dies an die letzte Aktivität meldet; der Workflow-Prozess ist dann beendet.

5.4 Ausführungskomponente und Alternativen

In den vergangenen Abschnitten haben wir mit dem Anbindungsmechanismus, die Anfrageverarbeitung und die Transaktionsverwaltung die wichtigsten Aspekte des Ausführungsmodells untersucht. Diese Betrachtungen basierten jedoch auf einem abstrakten Modell unserer Komponente zur Integration von Funktionen (KIF). Wir wollen nun konkretisieren, mit welcher Technologie wir das Ausführungsmodell und insbesondere die KIF implementieren. Wie bereits zu Beginn dieses Kapitels erläutert, bildet das FDBS eine Hauptkomponente unserer Integrationsarchitektur. Bei der Auswahl des FDBS stand unter anderem der Einsatz von bestehenden Technologien und die Einhaltung von Standards im Vordergrund. Dies wollen wir auch für die KIF erreichen. Hier sollen ebenfalls bestehende Technologien und Standards zum Einsatz kommen.

In den nächsten beiden Abschnitten stellen wir die von uns gewählte Technologie für die KIF vor. Wir haben uns für ein Workflow-Managementsystem (WfMS) entschieden, da es unseres Erachtens sehr gut zu unserem Beschreibungsmodell passt. Kapitel 5.4.1 erläutert die Gründe für unsere Wahl.

Die gewählte Technologie muss nicht immer die beste Lösung darstellen. Die Integrationsszenarien können sehr unterschiedlich sein, ebenso wie die zu integrierenden Systeme. Daher sind mehrere Implementierungen vorstellbar, die wir in Abschnitt 5.4.2 vorstellen und bewerten.

5.4.1 Ausführungskomponente zur Integration von Funktionen

Für die Implementierung eines Prototypen unserer Integrationsarchitektur haben wir uns für ein Workflow-System als KIF-Umsetzung entschieden. Nach eingehender Untersuchung mehrerer Alternativen waren die folgenden Gründe ausschlaggebend für unsere Entscheidung.

- Der Einsatz von FDBS und WfMS erlaubt uns, mit der jeweiligen Komponente gezielt auf Daten- bzw. Funktionsintegration einzugehen. Da das WfMS die Funktionsintegration vollständig übernimmt, isoliert es das FDBS von der Komplexität

der föderierten Funktionen. Die Kopplung der beiden Komponenten ermöglicht die Kombination von Daten- und Funktionsintegration, ohne dass das FDBS in seiner Funktionalität erweitert werden muss.

- Das Workflow-System ist eine seit mehreren Jahren bekannte Technologie. Inzwischen sind mehrere Produkte auf dem Markt, die eine gewisse Reife erreicht haben und sinnvoll eingesetzt werden können. Darüber hinaus liegen von der Workflow Management Coalition mehrere Standards vor (vgl. Abschnitt 3.3.4). Auch wenn diese Standards momentan nicht die wünschenswerte Akzeptanz bei den Herstellern finden, so ist hier zumindest die Basis für zukünftige einheitliche Implementierungen von Workflow-Systemen geschaffen.

- Das Schema eines Workflow-Prozesses entspricht dem Konzept unseres Abhängigkeitsgraphen. Beide Ansätze arbeiten gerichtete Graphen ab, wobei ein Workflow-Prozess einer föderierten Funktion entspricht. Die Aktivitäten des Workflows lassen sich auf die lokalen Funktionen unserer Integrationslösung abbilden, der Workflow-Datenfluss auf die Parameterabhängigkeiten und der Workflow-Kontrollfluss auf die Funktionsausführungsreihenfolge. Somit können wir die Beschreibung einer föderierten Funktion direkt mit einen Workflow-Prozess implementieren.

- Ein WfMS ermöglicht die Umsetzung sehr komplexer Abbildungsszenarien. Die Unterstützung von bedingten Kontrollflüssen und Kontrollstrukturen, wie z. B. Schleifen für den Aktivitätsablauf, erlaubt die Modellierung von komplexen Prozessen mit vielen Einzelschritten. Folglich können fast beliebige Funktionsabbildungen definiert werden, ohne das FDBS in irgendeiner Form modifizieren zu müssen.

- Ein WfMS stellt Schnittstellen zu Anwendungssystemen und damit zu Funktionen bereit. Auf diese Weise ergänzt es das FDBS, das hauptsächlich Datenschnittstellen und insbesondere SQL unterstützt. Der Einsatz eines WfMS ermöglicht somit eine verteilte Programmierung über heterogene Anwendungen hinweg.

- Die Buildtime-Komponente eines WfMS unterstützt die Wartbarkeit der implementierten föderierten Funktionen. Änderungen können leicht eingebracht werden, da sie zunächst in der Buildtime-Komponente definiert und anschließend in die Runtime-Komponente übersetzt werden. Die von den meisten Produkten unterstützte Visualisierung des Workflow-Prozesses als Graphen erleichtern die Definition der Abbildung.

- Derzeit werden verstärkt Aspekte der Transaktionsverwaltung im Kontext von Workflow-Systemen diskutiert. Unter dem Begriff der *Transactional Workflows* werden zunehmend Konzepte zur Unterstützung transaktionaler Eigenschaften von Workflows erarbeitet. Diese Konzepte finden momentan ihren Weg in bestehende Produkte. Sobald sich sinnvolle Lösungen etabliert haben, können wir globale Transaktionen über FDBS und WfMS besser unterstützen.

Die Abbildung von föderierten Funktionen auf Workflows ist direkt und einfach möglich. Workflow-Systeme sind jedoch ein sehr mächtiges Konzept mit entsprechendem Verwaltungsaufwand. Da für die Integration von Funktionen viele dieser Funktionalitäten

nicht benötigt werden, sind andere Flow-Systeme interessant, die eher leichtgewichtig sind. Die Industrie hat diesen Bedarf bereits erkannt und arbeitet an neuen Systemen. Diese sind nicht für die Interaktion mit Benutzern bestimmt, sondern unterstützen so genannte *Produktions-Workflows*. Produktions-Workflows beschreiben die voll automatisierte Ausführung von Systemen ohne jegliche Benutzerinteraktion. Diese Art der Flow-Systeme sind für unsere Architektur die angestrebte Ausführungskomponente.

5.4.2 Realisierungsalternativen

Neben dem Workflow-System gibt es weitere Alternativen für die Umsetzung des Ausführungsmodells. Je nach Komplexität der Funktionsabbildung und den zu integrierenden Systemen können unterschiedliche Lösungen sinnvoll sein. Grundsätzlich ist immer eine Lösung denkbar, die vollständig selbst implementiert wird. Obwohl wir auf verfügbaren Produkten aufbauen wollen, kann es sehr einfache Szenarien geben, in denen der Einsatz von Produkten zu überdimensioniert ist. So kann beispielsweise bei rein lesenden Zugriffen auf kleine Programme mit einfachen Abbildungen ein Batch-Skript vollkommen ausreichend sein, sofern kein Mehrbenutzerbetrieb unterstützt werden muss und eine wiederholte Ausführung bei Abbrüchen dasselbe Ergebnis liefert. Es ist unter Umständen auch sinnvoll, einen eigenen Server in C++ zu schreiben, der sehr spezifische Anforderungen seitens der Schnittstellen bedient.

Nichtsdestotrotz ziehen wir eine Lösung auf Basis von Produkten und Standards vor und betrachten in den folgenden Abschnitten weitere Technologien zur Implementierung unseres Beschreibungsmodells. Dazu gehören Datenbank-Middleware mit Wrappern, Message Broker, J2EE-Applikations-Server, Web Services und CORBA [HH02b].

5.4.2.1 DB-Middleware-Systeme mit Wrappern

Da wir bereits ein FDBS als Teil unserer Integrationsarchitektur einsetzen, ist eine reine DB-Middleware-Lösung denkbar. Statt einer weiteren Komponente zur Funktionsintegration werden die lokalen Funktionen direkt an das FDBS angebunden. Diese Anbindung kann in unterschiedlichen Formen stattfinden: mittels gespeicherter Prozeduren, benutzerdefinierter Tabellenfunktionen oder Wrapper (vgl. auch Abschnitt 5.1). Die Wrapper-Variante ist hierbei die mächtigste Anbindungsvariante, die wir in den weiteren Erläuterungen verfolgen.

Für die Implementierung der Daten- und Funktionsintegration werden zunächst alle zu integrierenden Systeme über systemspezifische Wrapper an das FDBS angebunden. Dabei werden Funktionen in Tabellen überführt. Anschließend erfolgt die Implementierung der Abbildungslogik von föderierten Funktionen auf lokale Funktionen mit den Mitteln des FDBS. Folglich müssen sie mit SQL-Anweisungen abgedeckt werden können. Genau hier können Probleme auftreten, da SQL eine deklarative Anfragsprache ist und keine prozedurale Abarbeitung vorsieht. Daher muss man für komplexe Abbildungen auf herstellerspezifische Funktionalität aufbauen oder auf benutzerdefinierte Funktionen, geschrieben in Programmiersprachen wie Java, zurückgreifen. Der Aufruf einer föderierten Funktion kann über den Aufruf einer gespeicherten Prozedur oder SQL-Anweisungen erfolgen.

Dieser Ansatz ist von Vorteil, wenn große Datenmengen verarbeitet werden, da die Verarbeitung im Datenbanksystem erfolgt. Außerdem basiert er mit SQL und den SQL/MED-Wrappern vollständig auf Standards. Leider lässt die Unterstützung von Wrappern in den aktuellen Produkten zu wünschen übrig. Daher muss man momentan auf alternative Lösungen ausweichen. Diesen Aspekt untersuchen wir in Kapitel 6 näher und stellen mögliche Alternativen vor. Mit Einschränkungen ist bei Abbildungen mit komplexer Logik zu rechnen, da SQL keine prozedurale Sprache ist.

5.4.2.2 Message Broker

Message Broker [Sch96b] basieren auf nachrichtenorientierter Middleware (*message-oriented middleware*, MOM) – besser bekannt als Message-Queuing-Systeme –, die eine asynchrone Kommunikation über Nachrichten zwischen Systemen ermöglicht. Diese Systeme können unterschiedlichsten Typs sein, wie z. B. DBMS, Legacy-, TP- oder PDM-Systeme. Der Message Broker stellt eine Art Vermittler dar, welcher den Transfer der Nachrichten zwischen den beteiligten Systemen verwaltet. Dabei übernimmt er mehrere Rollen: Er sorgt für die richtige Verteilung der Nachrichten (*message routing*) und nimmt notwendige Transformationen derselben vor. Außerdem unterstützt er die Definition eines Kontrollflusses der Nachrichten auf Basis eines Regelwerks. Mit Hilfe dieses Regelwerks kann die Verarbeitung und Verteilung der Nachrichten über Regeln definiert werden, die z. B. Bedingungsüberprüfungen, mathematischen Funktionen oder auch Datentypkonvertierungen enthalten.

Soll nun die Abbildungslogik einer Funktionsintegration mittels eines Message Broker implementiert werden, so muss diese Logik mit Hilfe des vom Message Broker unterstützten Regelwerks erfolgen. Handelt es sich um ein mächtiges Regelwerk, so können hinter den Werten eines dedizierten Feldes der Nachricht unterschiedliche Kontrollflüsse hinterlegt werden, um Nachrichten für die einzelnen Anwendungssysteme zu generieren. Unterstützt das Regelwerk nur einfache Flüsse, so kann die Information darüber, welche Nachrichten generiert werden müssen, beispielsweise in einer Datenbank abgelegt werden, die als Teil des Kontrollflusses abgefragt wird. Die über Nachrichten zurückgemeldeten Ergebnisse werden vom Message Broker zusammengeführt (Transformationen) und dem aufrufenden System zur Verfügung gestellt.

Im Vergleich zu dem vorgestellten Ansatz auf Basis eines Workflow-Systems ist festzustellen, dass die Abbildungsmächtigkeit eines Message Broker bei weitem nicht an die des Workflow-Systems heranreicht. Werden die Abbildungen komplexer, so können sie nicht mehr ausreichend durch das Regelwerk des Message Broker unterstützt werden. Der Einsatz eines Message Broker ist in solchen Fällen zu empfehlen, in welchen vielmehr die Verbindung zwischen mehreren Systemen als deren Integration im Vordergrund steht. In unserem Fall muss man jedoch von komplexen Abbildungen ausgehen, so dass der Einsatz eines Message Broker nicht ausreichend ist.

5.4.2.3 J2EE-Applikations-Server

Ein Integrationsansatz auf Basis von J2EE [Sun02] stützt sich auf einen Standard und verringert die Abhängigkeit von einem bestimmten Hersteller. Die J2EE-Architektur verkörpert einen Mehr-Schichten-Ansatz, bei welchem zwischen Präsentation, Geschäftslogik und Datenhaltung unterschieden wird. Für die Funktionsintegration ist die Schicht

mit der Geschäftslogik relevant, da hier die Abbildung der föderierten Funktionen auf die lokalen Funktionen mit Hilfe von EJBs (*Enterprise JavaBeans*) implementiert wird, die von den Anwendungssystemen in der Datenhaltungsschicht bereitgestellt werden. Im Gegensatz zu den bisher vorgestellten Ansätzen muss die Abbildungslogik vollständig selbst implementiert werden, d. h., man kann nicht auf bestehende Engines zurückgreifen, wie dies bei Workflow-Systemen, DB-Middleware oder Message Broker der Fall ist.

Die Umsetzung einer Funktionsintegration auf Basis eines J2EE-Applikations-Servers stellt sich grob umrissen folgendermaßen dar. Zunächst wird der Zugriff auf die Daten in Form von Entity Beans ermöglicht. Dieser Zugriff kann entweder direkt auf die Datenbank erfolgen oder über existierende APIs und Wrapper stattfinden. Die Abbildungslogik selbst, d. h. die Festlegung, welche Systeme in welcher Reihenfolge aufgerufen werden und welche Abhängigkeiten zwischen ihnen bestehen, muss mit Session Beans implementiert werden. Auf diese Weise wird jede föderierte Funktion durch eine Session Bean ausgeführt. Zur Standardisierung der Zugriffschnittstelle auf die zu integrierenden Systeme wurde die *J2EE Connector Architecture* definiert [Sun00]. Diese Architektur soll die Integration vereinfachen, indem vor allem der Zugriff auf die Systeme standardisiert ist und somit das Hinzufügen von neuen Systemen erleichtert wird. Dazu gehören u. a. Funktionalitäten wie Verbindungsverwaltung, Transaktionsverwaltung und Sicherheitsverwaltung.

Bei diesem Ansatz ist vor allem positiv zu bewerten, dass er sich auf Standards stützt. Des Weiteren sollte es möglich sein, alle denkbaren Integrations- und Abbildungsszenarien umzusetzen, da man sie direkt mit Java implementiert und somit die ganze Mächtigkeit dieser Programmiersprache einsetzen kann. Nichtsdestotrotz muss man aber die Logik von Hand implementieren, was zu altbekannten Problemen führen kann, wie mangelnde Dokumentation und die erschwerte Wartung und Erweiterung der bestehenden Integration durch Wegnahme oder Hinzufügen von Systemen. Der Einsatz eines J2EE-Ansatzes scheint nur dann sinnvoll, wenn die umzusetzende Logik sich nicht mit anderen Systemen realisieren lässt. Dabei sollte der Programmiercode, soweit möglich, mit Hilfe der beschriebenen Abbildungssprache generiert werden, um zumindest eine ausreichende Dokumentation sicherzustellen.

5.4.2.4 Web Services

Ein weiterer Ansatz, der vor allem in den letzten beiden Jahren große Beachtung gefunden hat, sind Web Services [W3C02]. Mit Hilfe von Web Services möchte man die Integration von Systemen nicht nur innerhalb einer Abteilung oder einer Organisation, sondern über Unternehmensgrenzen hinweg ermöglichen. Das Web erlaubt es auf einfachem Weg, dass Systeme ihre Funktionalität, oder auch Services, zur Verfügung stellen. Web Services basieren auf offenen Standards, wie z. B. SOAP [BEK+00], und nehmen Anfragen anderer Systeme entgegen. Diese Anfragen werden über eine leichtgewichtige und herstellerunabhängige Kommunikationstechnologie verschickt.

Übertragen auf die angestrebte Funktionsintegration bedeutet dies, dass die Anwendungssysteme ihre Funktionen als Web Services anbieten. Diese werden mit Hilfe der Sprache WSDL (*Web Services Description Language*, [CCMW01]) beschrieben und in einer Registry, dem UDDI (*Universal Description, Discovery, and Integration*, [BCE+02]), abgelegt. Bei der Umsetzung der Abbildungslogik können die Web Services jedoch nicht

weiterhelfen. Diese Logik muss in einer separaten Komponente implementiert werden und die föderierten Funktionen werden als weitere Web Services angeboten. Der Aufruf einer föderierten Funktion gestaltet sich dann folgendermaßen. Eine Applikation möchte eine föderierte Funktion aufrufen und sucht diese zunächst in der UDDI Registry. Hat sie die gesuchte Funktion gefunden, so ruft sie in der Integrationskomponente den entsprechenden Service auf. Dieser Service wiederum bedient sich all jener von den Anwendungssystemen bereitgestellten Web Services, die er für die Ausführung der föderierten Funktion benötigt. Das Endergebnis wird an die anfragende Komponente zurückgegeben.

Der große Vorteil dieses Ansatzes ist sicherlich darin zu sehen, dass eine Verbindung über Unternehmensgrenzen hinweg möglich ist und die bisherigen Probleme wie beispielsweise Firewalls umgangen werden. Web Services ermöglichen aber in erster Linie eine gemeinsame Sprache zwischen den Systemen, aber keineswegs eine Integration, wie sie in dieser Arbeit beschrieben wurde. Denn auch hier ist wieder eine spezielle Komponente für die Abbildungslogik zu finden, sei es selbst implementiert oder durch Einsatz anderer bereits besprochener Technologien wie Workflow-Systeme und Message Broker.

5.4.2.5 Object Request Broker

Object Request Broker (ORB) sind eine Middleware-Technologie, welche die Kommunikation zwischen verteilten Objekten oder Komponenten verwaltet und unterstützt. ORBs ermöglichen die nahtlose Interoperabilität zwischen den verteilten Objekten und Komponenten, ohne sich über die Details der Kommunikation Gedanken machen zu müssen. Dazu unterstützen sie die Transparenz hinsichtlich Rechnerknoten, Programmiersprache, Protokoll und Betriebssystem. Die Kommunikation zwischen den Objekten und Komponenten basiert in den meisten Fällen auf synchronen Schnittstellen. Die drei bekanntesten ORB-Standards sind OMG CORBA ORB [OMG03], Java RMI and RMI-IIOP [Sun03] und Microsoft COM/DCOM/COM+ [MS03].

Möchte man die Funktionsintegration mit einem ORB umsetzen, so findet man sich in einer ähnlichen Situation wie bei den Web Services wieder. Auch in diesem Fall werden die lokalen Funktionen zunächst in der Sprache IDL (*Interface Definition Language*) beschrieben und in einem Schnittstellen-Repository, dem *Interface Repository*, hinterlegt. Da der ORB hauptsächlich die Kommunikation zwischen Objekten und Komponenten unterstützt, muss die Implementierung der Abbildungslogik separat erfolgen. Dies geschieht beispielsweise in einer weiteren Komponente. Die föderierten Funktionen werden anschließend wie die lokalen Funktionen in IDL-Notation im Interface Repository veröffentlicht. Beim Aufruf einer föderierten Funktion bedient sich diese Komponente der lokalen Funktionen zur Ausführung der föderierten Funktion. Die Zwischenergebnisse werden an die aufrufende Komponente zurückgeliefert und von ihr zum Endergebnis zusammengesetzt.

Der Hauptfokus und damit der Vorteil von ORBs liegt hauptsächlich in der Überwindung von Heterogenitäten bei der Kommunikation zwischen verteilten Objekten und Komponenten. Eine Integration in unserem Sinne ist jedoch nicht vorgesehen. Folglich ist keine Unterstützung der Abbildungslogik vorgesehen. Stattdessen muss diese selbst implementiert werden. Muss man eine komplexe Logik umsetzen, so ist von dem Einsatz von ORBs eher abzuraten. Sind die Abbildungen aber einfach und liegt die Problematik

vielmehr in sehr heterogenen Systemen mit unterschiedlichen Protokollen, so können ORBs die Integration stark vereinfachen.

5.4.2.6 Abschließende Bemerkungen

Die aufgeführten Lösungen lassen sich in zwei Kategorien aufteilen: Interoperabilitäts- und Integrationstechnologien. Die Interoperabilitätslösungen sind horizontal ausgerichtet und schaffen eine gemeinsame Sprache und damit eine Verbindung zwischen den Systemen. Integrationslösungen hingegen arbeiten vertikal und setzen Einzelsysteme zusammen, um eine neue Schicht aufzubauen. Für unseren Integrationsansatz benötigen wir hauptsächlich die Integrationstechnologien und das sind Workflow-Systeme, Datenbank-Middleware und Message Broker.

Auch Web Services können zukünftig eine interessante Variante der Implementierung der Funktionsintegration sein. Bisher gibt es keine Unterstützung der Abbildungslogik, d. h. mann muss sie selbst implementieren. Jedoch arbeiten mehrere Hersteller an der *Business Process Execution Language for Web Services* (BPEL4WS, [BPE03]). Mit BPEL4WS können Geschäftsprozesse beschrieben werden, die anschließend auf Basis von Web Services ausgeführt werden. Wenn die Hersteller die Verarbeitung von BPEL4WS in ihre Produkte integrieren, können föderierte Funktionen mit wenig Programmieraufwand ausgeführt werden.

Wir weisen darauf hin, dass die aufgeführten Lösungsansätze in erster Linie die Integration von Anwendungssystemen unterstützen, nicht aber die Integration von Datenbanksystemen auf einer reinen Datenebene. Folglich ist eine Verarbeitung der zu integrierenden Daten in der Form, wie man es von Datenbanksystemen kennt, nicht möglich. Da aber genau dies zu den gestellten Anforderungen gehört, ist der Einsatz von Technologien allein zur Funktionsintegration nicht ausreichend. Der Zugriff auf Daten als auch Funktionen kann zwar unterstützt werden, aber die effiziente Weiterverarbeitung der Daten ist nicht sichergestellt. Daher scheint der Einsatz eines Datenbanksystems – in unserem Fall in Form eines föderierten Datenbanksystems – unvermeidlich. Letztendlich sollen Daten integriert und verarbeitet werden, auch wenn diese nicht direkt, sondern nur über Funktionen abrufbar sind.

5.5 Zusammenfassung

In diesem Kapitel haben wir das Ausführungsmodell betrachtet, das als Ergänzung zum Beschreibungsmodell den zweiten Teil unseres Lösungsansatzes zur Daten- und Funktionsintegration darstellt. Wir haben eine Architektur entwickelt, die aus zwei Hauptkomponenten besteht, dem FDBS und der KIF (Komponente zur Integration von Funktionen). Das FDBS realisiert die Integration von Daten und die KIF unterstützt die Integration von Funktionen zur Erstellung von föderierten Funktionen. Die Kopplung dieser beiden Komponenten ermöglicht den integrierten Zugriff auf Daten und (föderierte) Funktionen über eine SQL-Schnittstelle. Bei der Untersuchung dieser Integrationsarchitektur haben wir den Mechanismus zur Kopplung der Komponenten betrachtet, Aspekte der Anfrageoptimierung beleuchtet sowie Möglichkeiten und Grenzen der Transaktionsverwaltung aufgezeigt. Abschließend haben wir mögliche technische Implementierungen der Ausführungskomponente aufgeführt und bewertet.

Die Betrachtungen hinsichtlich denkbarer Kopplungsmechanismen von FDBS und KIF haben drei Alternativen hervorgebracht: den Einsatz einer Prozedur, einer benutzerdefinierten Tabellenfunktion oder eines Wrappers. Wir haben uns für den Wrapper entschieden, da er die mächtigste Variante darstellt und der Zugriff auf Funktionen völlig transparent gegenüber dem Benutzer ist. Somit sind wir bei den anschließenden Betrachtungen von einer Wrapper-Kopplung ausgegangen.

FDBS und KIF können nur dann sinnvoll eingesetzt werden, wenn eine übergreifende Anfrageverarbeitung bereitgestellt werden kann. Sie muss in der Lage sein, zwei völlig unterschiedliche Anfragemodelle zusammenzuführen und akzeptable Antwortzeiten zu gewährleisten. Die Funktionalität des Wrapper kann die Optimierung maßgeblich beeinflussen. Wir haben daher untersucht, welche Mindestanforderungen an den Wrapper gestellt werden und welche zusätzliche Funktionalität wünschenswert ist. Wie sich zusätzliche Operationen im Wrapper auf die gesamte Anfrageverarbeitung auswirken, haben wir an Beispielen und anhand eines angepassten Kostenmodells aufgezeigt. Das Ergebnis unserer Untersuchungen ist, dass die Unterstützung zusätzlicher Funktionalität im Wrapper die Anfrageoptimierung verbessert.

Anschließend haben wir ein Transaktionsmodell für die Integrationsarchitektur beschrieben. Das Mehrebenen-Transaktionsmodell sowie der in [Sch96a] beschriebene Ansatz für eine Transaktionsverwaltung in FDBS bilden die Basis für die vorgestellte Lösung. Zunächst auf die KIF fokussierend, haben wir unser Modell anschließend auf die gesamte Architektur ausgedehnt. Zur Verwirklichung ist die Einführung von Agenten notwendig, die zwischen den Anwendungen und dem darunter liegenden Datenbanksystem platziert werden. Sie erkennen indirekte Konflikte zwischen globalen Transaktionen und vermeiden diese durch die Implementierung zurückgehaltener Sperren. Anschließend haben wir untersucht, wie Transaktionen von kommerziellen Produkten unterstützt werden und welche Einschränkungen sich dadurch für die Bearbeitung von schreibenden Zugriffen ergeben. Die Beschreibung der Umsetzung in einem kommerziellen Workflow-System hat die Betrachtungen der Transaktionsunterstützung abgeschlossen.

Abschließend haben wir untersucht, wie die KIF implementiert werden kann. Unser Ziel ist hier gewesen, auf bestehende Technologien und Standards aufzubauen. Als mögliche Kandidaten wurden Datenbank-Middleware, Message Broker, J2EE-Applikations-Server, Web Services, Object Request Broker und Workflow-Systeme untersucht und bewertet. Wir haben uns für den Einsatz eines Workflow-Systems entschieden, da es unseren Anforderungen am nächsten kommt.

Kapitel 6

Integrationsarchitekturen und deren Bewertung

In Kapitel 5 haben wir den Aufbau unserer Integrationsarchitektur auf Basis von FDBS und WfMS erläutert, die durch einen Wrapper gekoppelt sind. Momentan gibt es jedoch keine kommerziell verfügbaren Produkte, die einen SQL/MED-Wrapper unterstützen. Daher stellen wir in diesem Kapitel eine Architektur vor, die den Wrapper durch benutzerdefinierte Tabellenfunktionen ersetzt. Außerdem betrachten wir weitere Architekturalternativen, die vollständig auf das WfMS verzichten und stattdessen auf objektrelationaler Datenbanktechnologie basieren.

Im Folgenden betrachten wir Lösungen mit und ohne WfMS und untersuchen die unterstützte Abbildungsmächtigkeit und die Performanz der von uns implementierten Prototypen. Sie ermöglichen eine Bewertung der von uns angestrebten Architektur auf Basis eines WfMS.

Anhand des folgenden Beispiels beschreiben wir die unterschiedlichen Architekturen. Es entspricht dem Beispiel in der Einleitung. Wir führen es an dieser Stelle aber noch einmal explizit auf, da es die Grundlage unserer folgenden Betrachtungen der unterschiedlichen Architekturen ist.

Beispiel:

Im Einkauf einer Firma soll ein Mitarbeiter entscheiden, ob eine weitere Komponente eines bereits bekannten Lieferanten bestellt werden soll. Ein Einkaufssystem unterstützt ihn mit der Funktion `KaufEntscheid()`. Diese Funktion schlägt eine Entscheidung vor, die auf einer berechneten Bewertung des Lieferanten auf Basis von Qualitäts- und Zuverlässigkeitsnoten zur gegebenen Komponentennummer beruht. Der Mitarbeiter hat aber nur den Komponentennamen und die Lieferantennummer. Komponentennummer und die Noten sind zunächst nicht bekannt. Er kann aber weitere Systeme befragen, um die benötigten Information zum Aufruf der Funktion `KaufEntscheid()` abzurufen.

Abbildung 6.1 zeigt die einzelnen Schritte, die der Mitarbeiter durchlaufen muss. Jeder Schritt entspricht dabei einem einzelnen Funktionsaufruf. Die Noten bekommt er mit den Funktionen `GetQualitaet()` und `GetZuverlaessigkeit()`

aus dem Lagerhaltungs- und Einkaufssystem. Anschließend übernimmt er diese Ergebnisse als Eingabewerte für die Funktion `GetBewertung()`, um die Bewertung zu errechnen. Das Ergebnis dieser Funktion liefert den ersten Eingabewert für die Funktion `KaufEntscheid()`. Um die Komponentennummer zum bekannten Komponentennamen zu erhalten, führt er die Funktion `GetKompNr()` des Produktdatenmanagementsystems aus und erhält somit den zweiten Eingabewert. Nun kann der Mitarbeiter die Funktion `KaufEntscheid()` aufrufen und das System errechnet einen Entscheidungsvorschlag.

Diese fünf Einzelschritte bzw. Funktionen sollen für den Endbenutzer durch einen einzigen Schritt bzw. eine einzige Funktion ersetzt werden. Dazu stellt man ihm eine föderierte Funktion `KaufLiefKomp()` zur Verfügung, welche die anderen Funktionen verbirgt.

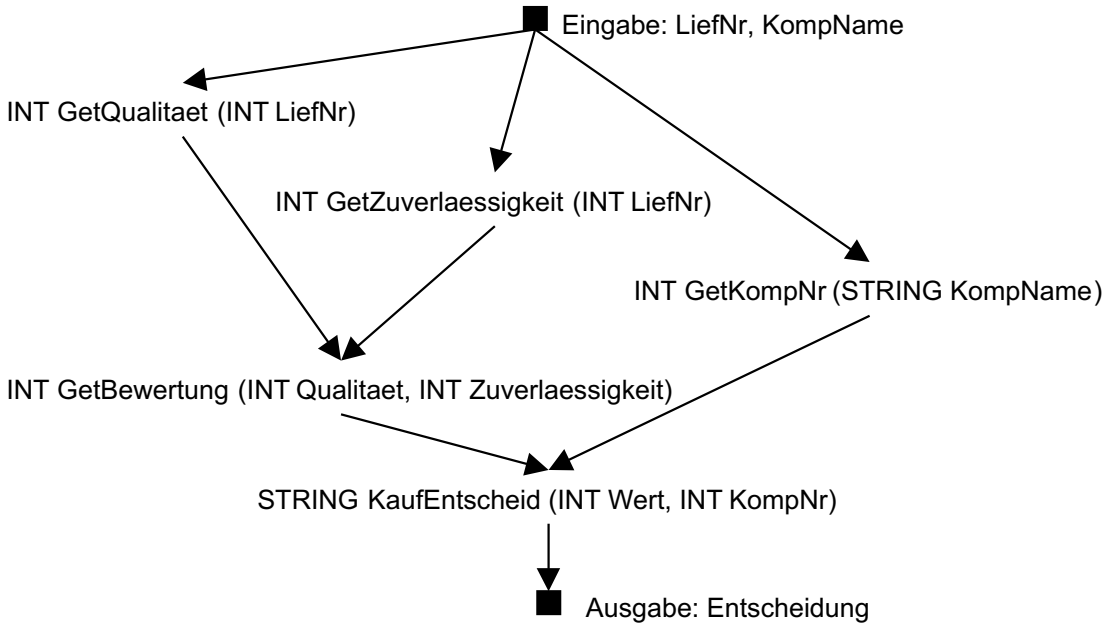

Abbildung 6.1: Die einzelnen Funktionsaufrufe und deren Abhängigkeiten untereinander.

Dieses Beispiel soll verdeutlichen, dass bisher keine integrierte Verarbeitung von Funktionen möglich ist und ihre Nutzung „manuell“ erfolgt, d. h., sie müssen explizit durch den Benutzer in der richtigen Reihenfolge aufgerufen werden. Dabei muss er Daten zwischen verschiedenen Systemen hin und her kopieren. Dies ist nicht nur umständlich, sondern auch fehleranfällig. Beispielsweise können beim Kopieren und Einfügen falsche Werte im Zwischenspeicher sein. Daher sollten Funktionsaufrufe, die immer wieder in derselben Reihenfolge stattfinden, von einem System unterstützt werden. Zu diesem Zweck stellt das System dem Benutzer eine föderierte Funktion zur Verfügung.

In den folgenden Abschnitten untersuchen wir, wie die Funktionsintegration mit verschiedenen Architekturlösungen umgesetzt werden kann. Wir beschrieben zunächst die

möglichen Architekturvarianten und erläutern anschließend anhand unterschiedlicher föderierter Funktionen deren Abbildungsmächtigkeit.

Da auch die Performanz der Lösungen von Bedeutung ist, werden mehrere föderierte Funktionen mit ausgewählten Ansätzen implementiert und deren Ausführungszeiten gemessen und verglichen.

6.1 Architekturalternativen

Neben der mangelnden Unterstützung standardisierter Wrapper-Technologie seitens der DB-Hersteller ist auch der Einsatz heutiger Workflow-Systeme ein Grund, warum wir alternative Architekturen für unseren Integrationsansatz untersuchen. Gängige Workflow-Systeme als Teil unserer Architektur mögen überdimensioniert erscheinen, da sie größtenteils als *people-driven* Workflow-Systeme, also auf Benutzerinteraktion ausgelegte Systeme sind. Folglich wird eine Vielzahl an Funktionalitäten zur Verfügung gestellt, die für die Integrationsarchitektur nicht benötigt wird. Berechtigterweise kommen daher Zweifel hinsichtlich der Performanz einer Architektur auf Basis eines WfMS auf. Daher stellen wir in den folgenden Abschnitten Architekturalternativen vor, die auf die Wrapper-Technologie verzichten und einige von ihnen auch das WfMS durch objektrelationale Datenbanktechnologie ersetzen. Diese Ansätze stützen sich in erster Linie auf benutzerdefinierte Funktionen, welche die Logik der Funktionsintegration übernehmen. Auch bei diesen Lösungsansätzen gilt die Einschränkung, dass lediglich lesender Zugriff unterstützt wird.

6.1.1 Workflow-Architektur

Nachdem sich eine Integrationsarchitektur auf Basis von Wrapper-Technologie derzeit nicht mit kommerziellen Produkten umsetzen lässt, soll der Wrapper durch benutzerdefinierte Tabellenfunktionen ersetzt werden (siehe Abbildung 6.2). Hierbei wird jede föderierte Funktion, die durch einen Workflow-Prozess implementiert wird, mittels einer UDTF an das FDBS angebunden. Die UDTF bzw. föderierte Funktion kann in der FROM-Klausel wie eine Tabelle referenziert werden. Eingabeparameterwerte werden beim Funktionsaufruf direkt übergeben.

In unserem Beispiel wird die Logik der föderierten Funktion `KaufLiefKomp()` auf einen Workflow-Prozess abgebildet. Dieser Prozess übernimmt die Aufrufe der lokalen Funktionen und liefert den Ausgabewert der föderierten Funktion als Ergebnis. Die Anbindung des Workflow-Prozesses erfolgt über eine Tabellenfunktion mit demselben Namen. Der Benutzer kann anschließend mit einer einfachen SELECT-Anweisung die föderierte Funktion mit den Eingabewerten ausführen.

Zuvor muss jedoch die entsprechende UDTF mit dem CREATE FUNCTION-Befehl angelegt werden. Dessen Syntax ist wie folgt in DB2 festgelegt (siehe auch Abschnitt 3.2.2):

```
CREATE FUNCTION funktionname (datentyp, datentyp, ...)
RETURNS TABLE (spaltenname1 datentyp, spaltenname2 datentyp, ...)
EXTERNAL
LANGUAGE programmiersprache
PARAMETER STYLE DB2GENERAL
```

```
NO SQL
DISALLOW PARALLEL
```

Mit diesem Befehl wird die Signatur der UDTF mit Funktionsname, Datentypen der Eingabeparameter sowie Namen und Datentypen der Ausgabewerte angegeben. Außerdem legt man fest, in welcher Programmiersprache die Funktion implementiert ist[1]. Für unser Beispiel sieht der Befehl wie folgt aus:

```
CREATE FUNCTION KaufLiefKomp (LiefNr INT, KompName VARCHAR)
RETURNS TABLE (Entscheidung VARCHAR)
EXTERNAL
LANGUAGE JAVA
PARAMETER STYLE DB2GENERAL
NO SQL
DISALLOW PARALLEL
```

Die Syntax für den Aufruf von UDTFs ist wie folgt definiert (siehe auch Abschnitt 3.2.2):

```
SELECT *
FROM TABLE (funktionname(parameter1, parameter2)) AS fn
WHERE fn.spaltenname1 = wert
```

Mit „SELECT *“ liest man alle Ausgabewerte der UDTF aus, die im CREATE-Befehl unter der RETURNS TABLE-Klausel angegeben wurden. In unserem Beispiel ist dies der Parameter `Entscheidung`. Die Referenzierung der UDTFs in der FROM-Klausel wird mit dem Schlüsselwort `TABLE` eingeleitet. Es folgen der Funktionsname mit den Eingabewerten und ein Korrelationsname. Dieser muss immer angegeben werden. Es können mehrere durch Kommata getrennte UDTFs aufgeführt werden. In der WHERE-Klausel können Spalten der UDTFs in Prädikaten eingesetzt werden.

In unserem Beispiel muss der Benutzer demnach folgende SELECT-Anweisung formulieren, um die föderierte Funktion `KaufLiefKomp()` auszuführen:

```
SELECT *
FROM TABLE KaufLiefKomp(LiefNr, KompName) AS KLK
```

Die UDTF-Anbindung kann die Wrapper-Lösung jedoch nicht vollständig ersetzen, da auf Basis von benutzerdefinierten Tabellenfunktionen nur lesender Zugriff implementiert werden kann. Folglich sind keine Schreibzugriffe möglich ist, d. h. UDTFs können nicht in INSERT-, UPDATE- und DELETE-Anweisungen referenziert werden. Ein weiterer Nachteil zeigt sich in der Verarbeitung der Anfrage, da UDTFs in der Anfrageoptimierung nicht in dem Maße eingebunden werden können wie Wrapper. Folglich muss man mit längeren Bearbeitungszeiten rechnen.

[1] Detaillierte Erläuterungen finden sich in Abschnitt 3.2.2.

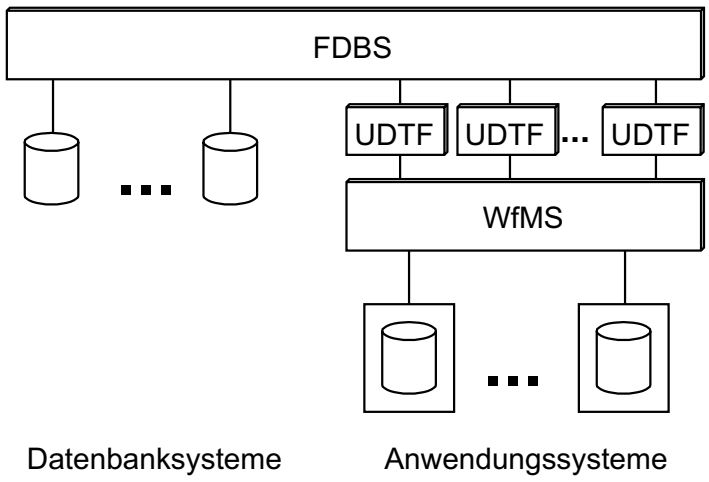

Abbildung 6.2: Kopplung von FDBS und WfMS mit benutzerdefinierten Tabellenfunktionen (UDTFs).

6.1.2 Einfache UDTF-Architektur

In der einfachsten Variante ohne WfMS wird jede lokale, zu integrierende Funktion mittels einer UDTF an das FDBS angebunden. Diese benutzerdefinierten Funktionen nennen wir im Folgenden A-UDTFs (kurz für *access user-defined table functions*, vgl. Abbildung 6.3). Diese Funktionen werden in einer Programmiersprache wie z. B. Java implementiert. Sie rufen die lokalen Funktionen mit entsprechenden Eingabewerten auf und liefern das Ergebnis in Tabellenform an das FDBS zurück. Die so angebundenen Funktionen können nun in der FROM-Klausel wie Tabellen referenziert werden.

Die Integrationslogik muss bei dieser Vorgehensweise aber in der Anwendung implementiert werden, da vom FDBS keine föderierten Funktionen direkt zum Aufruf bereitgestellt werden. Folglich sieht der Benutzer die über UDTFs angebundenen und referenzierbaren lokalen Funktionen und kann diese nun selbst zusammenführen. Dies kann in zwei Formen erfolgen. Entweder man beschreibt in der Anwendung die erforderliche Verknüpfung der Tabellenfunktionen in SELECT-Anweisungen. Oder man setzt die Logik vollständig in der Anwendung um. Dies ist beispielsweise möglich, indem SELECT-Anweisungen mit einzelnen Funktionsaufrufen abgesetzt werden und die Reihenfolge der Funktionsaufrufe und das Verarbeiten der Ergebnisse mittels Programmiersprachenkonstrukte erfolgt. Sollen auch die Abhängigkeiten von lokalen Funktionen in der SELECT-Anweisung implementiert werden, dann werden diese durch die Verfügbarkeit der Eingabeparameter realisiert.

In unserem Beispiel sind die Funktionen `GetBewertung()` und `Kaufentscheid()` von den Funktionen `GetQualitaet()` und `GetZuverlaessigkeit()` bzw. `GetKompNr()` abhängig. Somit müssen diese Funktionen zuerst ausgeführt werden. Die folgende SQL-Anweisung bildet die Logik der föderierten Funktion `KaufLiefKomp()` ab, wie sie die Anwendung in der einfachen UDTF-Architektur enthalten müsste:

```
SELECT KE.Entscheidung
FROM TABLE (GetQualitaet(LiefNr)) AS GQ,
```

```
TABLE (GetZuverlaessigkeit(LiefNr)) AS GZ,
TABLE (GetBewertung(GQ.Qual, GZ.Zuverl)) AS GB,
TABLE (GetKompNr(KompName)) AS GKN,
TABLE (KaufEntscheid(GB.Wert, GKN.Nr)) AS KE
```

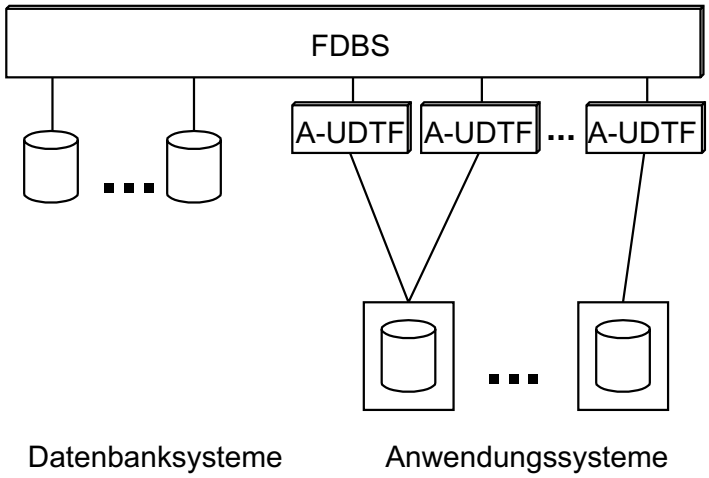

Abbildung 6.3: Einfache UDTF-Architektur ohne WfMS.

Offensichtlich ist diese Architektur nicht zufriedenstellend, da die Integrationslogik in den SQL-Anweisungen im Anwendungscode versteckt ist. Falls die Entwickler das Integrationsszenario ändern müssen, weil neue Datenquellen hinzugefügt oder bestehende entfernt werden sollen, müssen sie die bestehende Logik verstehen. Dazu muss Programmiercode untersucht und verstanden werden, der in den meisten Fällen von anderen Entwicklern implementiert wurde. Normalerweise ist zudem nur wenig oder gar keine Dokumentation vorhanden, so dass Anpassungen sehr zeitaufwendig sind. Aus diesem Grund verfolgen wir diesen Ansatz nicht weiter.

6.1.3 Erweiterte SQL-UDTF-Architektur

Bei diesem Ansatz wird der größte Nachteil der einfachen UDTF-Architektur vermieden, indem die Integrationslogik zurück in den Server, d. h. in das FDBS verlegt wird. Dazu baut man eine weitere Schicht bestehend aus benutzerdefinierten Funktionen auf, die I-UDTFs (kurz für *integration user-defined table function*) genannt werden (siehe Abbildung 6.4). Wie bei der einfachen UDTF-Architektur werden die lokalen Funktionen mittels A-UDTFs angebunden. In einem weiteren Schritt werden die A-UDTFs nun von I-UDTFs referenziert, um die Integrationslogik umzusetzen. Die I-UDTFs enthalten im Gegensatz zu den A-UDTFs nur SQL-Anweisungen. Die Implementierung der föderierten Funktion `KaufLiefKomp()` aus unserem Beispiel sieht dann folgendermaßen aus:

```
CREATE FUNCTION KaufLiefKomp (LiefNr INT, KompName VARCHAR)
RETURNS TABLE (Entscheidung VARCHAR)
LANGUAGE SQL RETURN
SELECT KE.Entscheidung
```

```
FROM TABLE (GetQualitaet(LiefNr)) AS GQ,
     TABLE (GetZuverlaessigkeit(LiefNr)) AS GZ,
     TABLE (GetBewertung(GQ.Qual, GZ.Zuverl)) AS GB,
     TABLE (GetKompNr(KompName)) AS GKN,
     TABLE (KaufEntscheid(GB.Wert, GKN.Nr)) AS KE
```

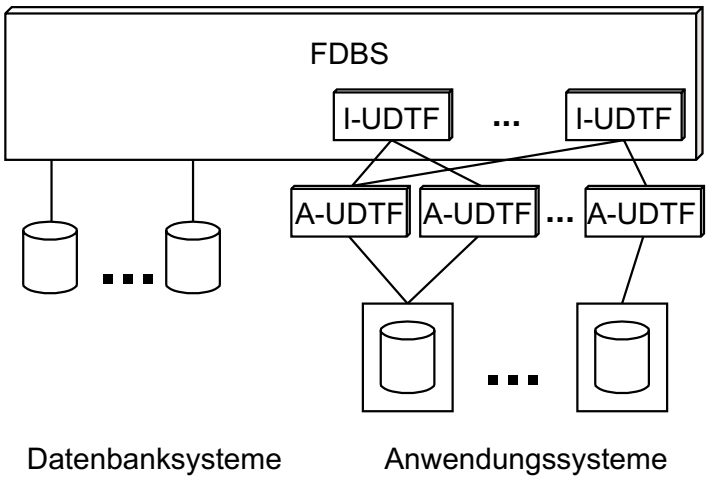

Abbildung 6.4: Erweiterte UDTF-Architektur mit SQL-Integrations-UDTFs.

Um die Integrationslogik in den Server zu verlegen, müssen wir also lediglich die SQL-Anweisung, die zuvor, in der Anwendung erstellt wurde, in eine benutzerdefinierte Funktion packen. Diese wird anschließend mit einer einfachen SELECT-Anweisung aufgerufen:

```
SELECT KLK.Entscheidung
FROM TABLE (KaufLiefKomp(LiefNr, KompName)) AS KLK
```

Leider besteht in V7.1 von DB2 die Einschränkung, dass der Rumpf einer SQL-Funktion nur eine SQL-Anweisung enthalten kann. Dies führt wiederum zu Einschränkungen hinsichtlich der Abbildungsmächtigkeit der I-UDTFs. Sofern die Abbildung jedoch mit einer Anweisung ausgedrückt werden kann, ist diese Vorgehensweise sehr übersichtlich.

Die erweiterte SQL-UDTF-Architektur ist in der Lage, Anwendungsentwicklern föderierte Funktionen zur Verfügung zu stellen, die in SELECT-Anweisungen referenziert werden und somit mit Referenzen zu anderen föderierten Funktionen als auch Tabellen kombiniert werden können. Da die föderierten Funktionen auf Basis von SQL auf der Server-Seite implementiert werden, ist die Wartung dieser Funktionen einfacher als bei dem einfachen UDTF-Ansatz. Nichtsdestotrotz kann die Abbildungsmächtigkeit durch die Beschränkung auf eine SQL-Anweisung in der I-UDTF eingeschränkt sein. Überdies ist SQL eine deklarative Sprache und unterstützt demnach keine prozeduralen Strukturen, die aber für die Integrationslogik nötig sein können.

6.1.4 Erweiterte Java-UDTF-Architektur

Um die Einschränkungen der erweiterten SQL-UDTF-Architektur zu umgehen, wird nun die erweiterte Java-UDTF-Architektur vorgestellt. Dieser Ansatz bindet zunächst ebenfalls alle lokalen Funktionen über A-UDTFs an. Im Gegensatz zum SQL-UDTF-Ansatz auf Basis von reinen SQL-Anweisungen werden die I-UDTFs in diesem Fall in Java implementiert (siehe Abbildung 6.5). Auf diese Weise umgeht man die Einschränkungen der SQL-I-UDTFs, d. h., es können beliebig viele SQL-Anweisungen in den I-UDTFs enthalten sein. Zudem kann man mittels Java prozedurale Strukturen einsetzen und somit auch komplexere Abbildungen umsetzen. Der Aufruf der föderierten Funktion entspricht dem Aufruf in der erweiterten UDTF-Architektur.

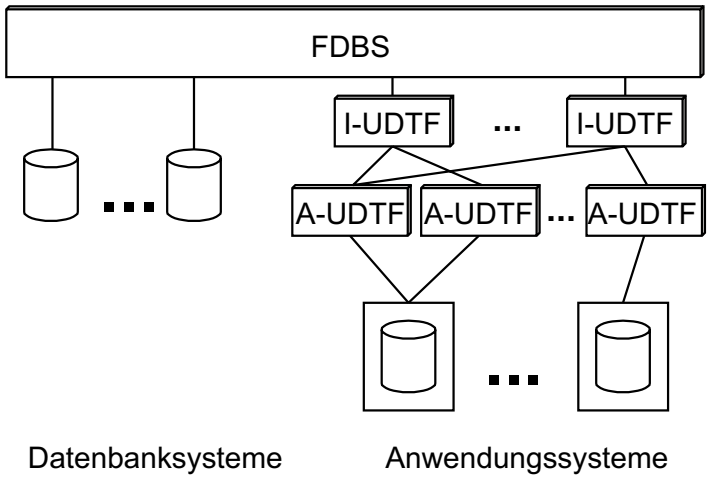

Abbildung 6.5: Erweiterte UDTF-Architektur mit Java-Integrations-UDTFs.

Die erweiterte Java-UDTF-Architektur scheint daher den mächtigsten Ansatz zur Implementierung von föderierten Funktionen in einem FDBS darzustellen. Nichtsdestotrotz wird die Wartung der Abbildungslogik erneut komplizierter, da die Integrationslogik teilweise im Programmiercode der Java-I-UDTFs verborgen ist. Die Umsetzung mit SQL-I-UDTFs scheint daher einfacher.

Die aktuelle Entwicklung der DBS bekannter Hersteller zeigt überdies, dass zukünftig die Einschränkung auf eine SQL-Anweisung nicht bestehen bleiben wird. Zudem werden prozedurale Kontrollstrukturen unterstützt, wie man sie aus PSM Stored Procedures (gespeicherte Prozeduren) kennt. Sollte dies nicht der Fall sein, so können statt der SQL-I-UDTFs PSM Stored Procedures implementiert werden. Hierbei handelt man sich aber wiederum den Nachteil ein, dass man eine gespeicherte Prozedur nicht in einer SELECT-Anweisung referenzieren und dementsprechend nicht mit Aufrufen von weiteren föderierten Funktionen und Tabellen kombinieren kann.

6.2 Unterstützte Abbildungsmächtigkeit der Alternativen

Nach der Vorstellung der Architekturalternativen untersuchen wir die beiden interessantesten Alternativen auf ihre Abbildungsmächtigkeit hin. Dies ist zum einen die erweiterte SQL-UDTF-Architektur, im Folgenden kurz UDTF-Architektur genannt, und zum anderen die Workflow-Architektur. Wir gehen der Frage nach, ob mit der UDTF-Architektur grundsätzlich dieselbe Komplexität an Abbildungen unterstützt werden kann. Als Grundlage für diesen Vergleich wird die in Kapitel 2 vorgestellte Klassifikation sowie ein Beispielszenario herangezogen.

Zusammengefasst kann man die Klassifikation in folgende fünf Fälle aufteilen:

- Der **triviale Fall** mit einer (1:1)-Abbildung, in welcher sich die Funktions- und Parameternamen unterscheiden können.
- Der **einfache Fall**, der den trivialen Fall um sich unterscheidende Funktionssignaturen erweitert.
- Der **unabhängige Fall** mit einer (1:n)-Abbildung, in welcher die lokalen Funktionen unabhängig sind und damit parallel ausgeführt werden können.
- Der **abhängige Fall** mit linearen, (1:n)-, (n:1)- und iterativen Abhängigkeiten zwischen den lokalen Funktionen.
- Der **allgemeine Fall** mit einer (n:m)-Abbildung, in welcher alle aufgeführten Fälle enthalten sein können.

In den folgenden Abschnitten werden alle Fälle beispielhaft auf Basis der beiden Architekturen umgesetzt. Anschließend bewerten wir die beiden Architekturen.

6.2.1 Trivialer Fall

Beim trivialen Fall liegen eine lokale und eine föderierte Funktion vor. Die beiden Funktionen weisen dabei eine identische Funktionalität und Signatur vor. Lediglich die Funktionsnamen oder die Parameternamen variieren. Das folgende Beispiel stellt einen trivialen Fall dar (siehe Abbildung 6.6).

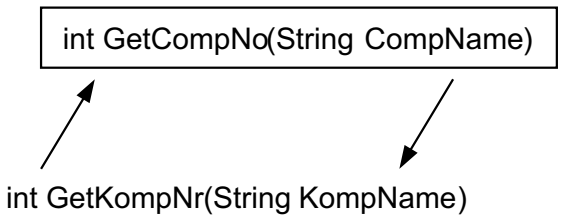

Abbildung 6.6: Parameterabhängigkeiten der föderierten Funktion `GetCompNo()` im trivialen Fall.

Umsetzung mit Workflow-Architektur

Bei der Umsetzung mit der Workflow-Architektur erhält man einen Workflow-Prozess mit genau einer Aktivität. Unterschiedliche Funktionsnamen haben keine weitreichenden Auswirkungen. Die Abbildung erfolgt durch die unterschiedliche Benennung der föderierten Funktion, die wiederum die lokale Funktion aufruft und somit deren Namen verbirgt. Im Beispiel heißt der die föderierte Funktion repräsentierende Workflow-Prozess `GetCompNo`, während der Name der Aktivität `GetKompNr` lautet. Unterschiedliche Parameternamen stellen ebenfalls keine große Hürde dar. Hier erfolgt ebenso eine Umbenennung durch die Signatur der föderierten Funktion. Diese Abbildung ist auch insofern nicht problematisch, da Parameter über ihre Position in der Signatur sowie ihren Datentyp bestimmt werden und somit der Name selber keine tragende Rolle hat.

Nachdem der Prozess über eine gleichlautende UDTF an das FDBS angebunden wurde, kann man mit folgender SQL-Anweisung die föderierte Funktion aufrufen:

```
SELECT *
FROM TABLE (GetCompNo(CompName)) AS GCN
```

Umsetzung mit UDTF-Architektur

Bei der Umsetzung mit der zweistufigen UDTF-Architektur setzt man eine I-UDTF `GetCompNo()` ein, welche genau eine A-UDTF `GetKompNr()` aufruft. Dabei werden beim Anlegen der I-UDTF alle Namensänderungen durchgeführt, indem die deutschen Funktionsnamen und Parameternamen übersetzt werden.

```
CREATE FUNCTION GetCompNo (CompName VARCHAR)
RETURNS TABLE (Number INT)
LANGUAGE SQL RETURN
SELECT *
FROM TABLE (GetKompNr(GetCompNo.CompName)) AS GKN
```

Wie man dem CREATE-Befehl entnehmen kann, wird über die föderierte Funktion `GetCompNo()` die lokale Funktion `GetKompNr()` aufgerufen. Der Eingabeparameter heißt nun `CompName` und dessen Wert wird als Eingabe an die lokale Funktion übergeben. Der Rückgabewert ist vom Typ `integer`.
Der Zugriff auf die föderierte Funktion erfolgt über eine SELECT-Anweisung:

```
SELECT *
FROM TABLE (GetCompNo(CompName)) AS GCN
```

Fazit

Der triviale Fall kann mit beiden Architekturen problemlos umgesetzt werden.

6.2.2 Einfacher Fall

Der einfache Fall erweitert den trivialen Fall in der Form, dass die Datentypen der Parameter als auch die Funktionssignatur und somit die Anzahl der Parameter unterschiedlich sein können. Das folgende Beispiel 1 (siehe Abbildung 6.7) verdeutlicht

zunächst den Fall mit unterschiedlichen Datentypen. An einem weiteren Beispiel 2 wird der Fall mit einer unterschiedlichen Anzahl an Parametern erläutert.

Beispiel 1:

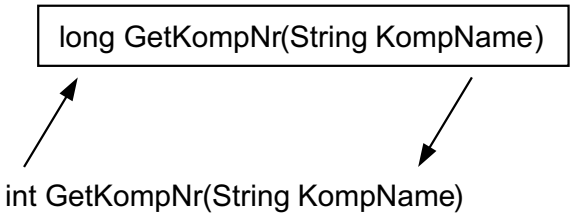

Abbildung 6.7: Parameterabhängigkeiten der föderierten Funktion `GetKompNr()` im einfachen Fall.

In diesem Beispiel erfolgt erneut eine Abbildung auf die lokale Funktion `GetKompNr()`. Hierbei soll der Rückgabewert den Datentyp `long` statt `integer` haben.

Umsetzung mit Workflow-Architektur

Im Fall von unterschiedlichen Datentypen bei Eingabe- oder Ausgabeparametern sind bei der Workflow-Architektur zwei Vorgehensweisen möglich. Zum einen kann bereits innerhalb der A-UDTF eine Datentypkonvertierung (Type Cast) durchgeführt werden. Dies ist jedoch nur sinnvoll, wenn diese lokale Funktion immer mit einem `long`-Datentyp benötigt wird. Zum anderen kann für die Datentypkonvertierung eine der Hilfsfunktionen herangezogen werden, die in einer Standardbibliothek für das Workflow-System zur Verfügung gestellt werden. Auf diesem Wege kann die Hilfsfunktion in beliebig vielen Funktionsabbildungen eingesetzt werden. Diese Lösung ist allerdings etwas langsamer, da die Hilfsfunktion eine eigene Aktivität im Workflow-Prozess darstellt, die den Prozess somit auch komplexer macht. Abbildung 6.8 zeigt den Workflow-Prozess mit Einbeziehung einer Hilfsfunktion `int2long()`.

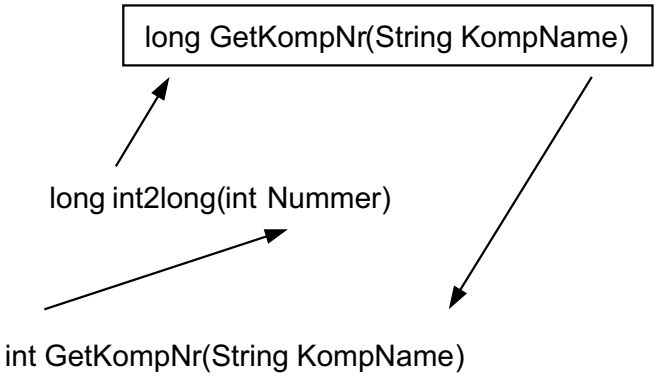

Abbildung 6.8: Parameterabhängigkeiten der föderierten Funktion `GetKompNr()` mit der Hilfsfunktion `int2long()`.

Umsetzung mit UDTF-Architektur

In der UDTF-Architektur kann man auf die in SQL verfügbaren Cast-Funktionen zurückgreifen, so dass die Konvertierung innerhalb der CREATE-Anweisung veranlasst wird.

```
CREATE FUNCTION GetKompNr (KompName VARCHAR)
RETURNS TABLE (KompNr BIGINT)
LANGUAGE SQL RETURN
SELECT BIGINT(GKN.KompNr)
FROM TABLE (GetKompNr(GetKompNr.KompName)) AS GKN
```

In der SELECT-Klausel findet die Konvertierung von `integer` nach `long` über die Cast-Funktion `BIGINT` statt. Dieser Wert wiederum wird als Rückgabewert der föderierten Funktion zurückgereicht.

Die föderierte Funktion kann nun in einer SELECT-Anweisung aufgerufen werden.

Beispiel 2:
In unserem zweiten Beispiel liefert die lokale Funktion `GetAnsprechpartner()` zu einer gegebenen Lieferantennummer `LiefNr` eine Zeichenkette mit dem Vor- und Nachnamen des Ansprechpartners des Lieferanten. Die föderierte Funktion `GetAnsprechpartner_2()` soll nun den Vor- und Nachnamen getrennt in Form von zwei Zeichenketten zurückliefern (siehe Abbildung 6.9).

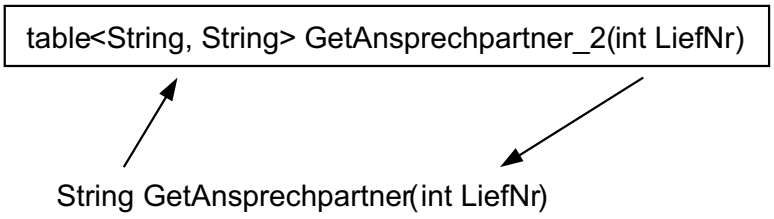

Abbildung 6.9: Parameterabhängigkeiten der föderierten Funktion `GetAnsprechpartner_2()` im einfachen Fall.

Umsetzung mit Workflow-Architektur

Auch in diesem Fall kann auf eine Hilfsfunktion zurückgegriffen werden, die den vollständigen Namen in zwei Zeichenketten mit jeweils dem Vor- und dem Nachnamen aufteilt (siehe Abbildung 6.10).

Umsetzung mit UDTF-Architektur

Grundsätzlich kann bei der UDTF-Architektur sehr häufig auf bestehende Funktionen aufgebaut werden. Beispielsweise unterstützt die DB2-Funktion `SUBSTR()` die Aufteilung einer Zeichenkette. Da hierzu jedoch die genaue Position der Teilung angegeben werden muss, die Vornamen jedoch unterschiedliche Längen aufweisen, stellt sie keine Lösung des Problems dar. Um dennoch das aufgeführte Beispiel umsetzen zu können, wird

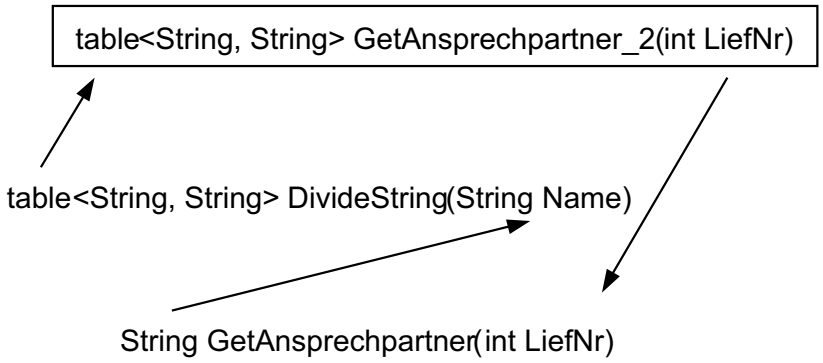

Abbildung 6.10: Parameterabhängigkeiten der föderierten Funktion `GetAnsprechpartner_2()` mit Hilfsfunktion `DivideString()` im einfachen Fall.

eine UDTF als Hilfsfunktion `DivideString()` geschrieben und eingesetzt, wie dies auch bei der Workflow-Architektur erfolgt ist. Dabei ist die Hilfsfunktion keine Workflow-Aktivität, sondern eine Java-UDTF, welche die komplette Zeichenkette als Eingabe entgegennimmt, sie aufteilt und die beiden resultierenden Zeichenketten als einzeilige Tabelle zurückgibt.

Die beiden Funktionen `GetAnsprechpartner()` und `DivideString()` werden für die Abbildung in einer SELECT-Anweisung in einem kartesischen Produkt referenziert. Dabei ist die Reihenfolge der referenzierten UDTFs wichtig. Die zuerst auszuführende A-UDTF (hier `GetAnsprechpartner`) muss an erster Stelle stehen, da die Hilfsfunktion `DivideString()` das Ergebnis von `GetAnsprechpartner` als Eingabe benötigt. Der CREATE-Befehl der föderierten Funktion sieht demnach wie folgt aus:

```
CREATE FUNCTION GetAnsprechpartner_2 (LiefNr INT)
RETURNS TABLE (Vorname VARCHAR(25), Nachname VARCHAR(25))
LANGUAGE SQL RETURN
SELECT DS.Vorname, DS.Nachname
FROM TABLE (GetAnsprechpartner(GetAnsprechPartner_2.LiefNr)) AS GA,
     TABLE (DivideString(GA.Name)) AS DS
```

Es sei nochmals darauf hingewiesen, dass die Unterstützung einer solch prozeduralen Verarbeitung nicht von SQL vorgesehen ist, sondern in diesem Fall von dem eingesetzten Datenbanksystem unterstützt wird. Man kann daher nicht allgemein davon ausgehen, dass diese Implementierung auch von anderen Datenbanksystemen unterstützt wird.

Fazit

Beide Beispiele des einfachen Falles lassen sich mit den Architekturen umsetzen. Jedoch müssen in beiden Fällen Hilfsfunktionen hinzugezogen werden.

6.2.3 Unabhängiger Fall

Der unabhängige Fall erweitert den einfachen Fall um die Einführung mehrerer voneinander unabhängiger lokaler Funktionen. Aufgrund der Unabhängigkeit ist deren Ausführungsreihenfolge irrelevant, d. h., die lokalen Funktionen können parallel ausgeführt

werden. Abbildung 6.11 zeigt ein Beispiel, in welchem zu einer gegebenen Lieferantennummer die Qualität der Produkte sowie die Zuverlässigkeit des Lieferanten abgerufen werden kann. Um diese Funktionalität abzudecken, werden zwei Systeme einbezogen: die Qualität erhält man aus dem Lagersystem und die Zuverlässigkeit aus dem Einkaufssystem. Die Ergebnisse dieser Aufrufe sind unabhängig, d. h., das Ergebnis der lokalen Funktion `GetQualität()` hat keine Auswirkungen auf das Ergebnis der lokalen Funktion `GetZuverlaessigkeit()` und umgekehrt.

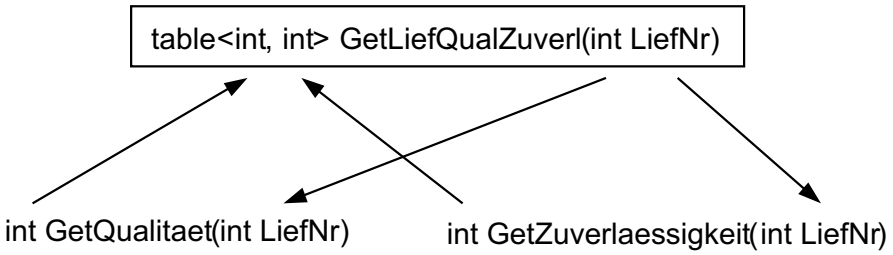

Abbildung 6.11: Parameterabhängigkeiten der föderierten Funktion `GetLiefQualZuverl()` im unabhängigen Fall.

Umsetzung mit Workflow-Architektur

Unabhängige Funktionen werden in der Workflow-Architektur als parallel ablaufende Aktivitäten umgesetzt, wobei jede Aktivität eine lokale Funktion repräsentiert. In dem Beispiel werden `GetQualitaet` und `GetZuverlaessigkeit()` jeweils in einer eigenen Aktivität aufgerufen. Die Ergebnistabelle kann anschließend entweder in einer weiteren Aktivität mit einer Hilfsfunktion oder direkt in der UDTF zusammengesetzt werden, die FDBS und WfMS koppelt. Abbildung 6.12 zeigt, wie die Hilfsfunktion `MergeInt()` die Ergebnisse der beiden lokalen Funktionen zusammenfasst.

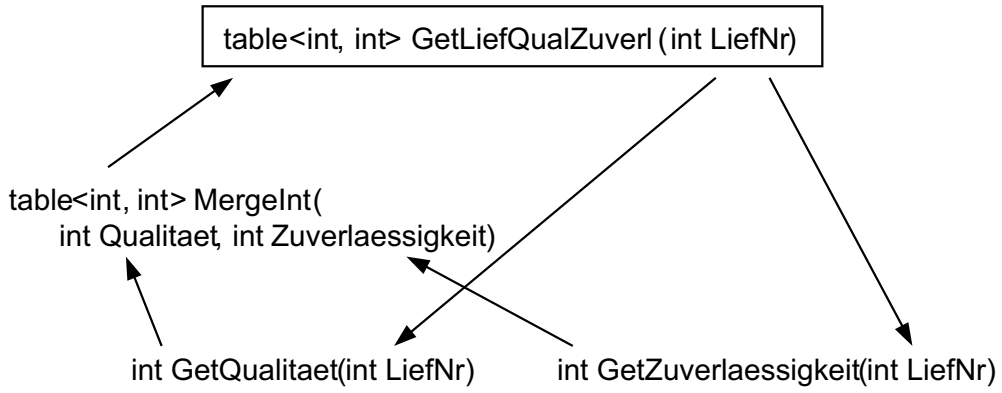

Abbildung 6.12: Parameterabhängigkeiten der föderierten Funktion `GetLiefQualZuverl()` mit der Hilfsfunktion `MergeInt()` im unabhängigen Fall.

Umsetzung mit UDTF-Architektur

Im Falle einer Umsetzung auf die UDTF-Architektur bietet sich die Verwendung des kartesischen Produkts mehrerer angebundener Funktionen in der SQL-Anfrage der I-UDTF an. Die Ergebnisse können dann, falls nötig, mittels Selektion gefiltert werden. Hierbei können die Vergleichsprädikate mit booleschen Operatoren verknüpft werden:

```
SELECT *
FROM TABLE (funktion_1(a)) AS f1, TABLE (funktion_2(b)) AS f2
WHERE f1.x=wert_1 OR f2.y=wert_2
```

Da die beiden Funktionen des aktuellen Beispiels jeweils nur einen Wert zurückliefern, bekommt man eine zweispaltige Tabelle mit nur einer Zeile. Hier ist deshalb keine Selektion notwendig. Das Anlegen der föderierten Funktion sieht folgendermaßen aus:

```
CREATE FUNCTION GetLiefQualZuverl (LiefNr INT)
RETURNS TABLE (Qualitaet INT, Zuverlaessigkeit INT)
LANGUAGE SQL RETURN
SELECT GQ.Qualitaet, GZ.Zuverlaessigkeit
FROM TABLE (GetQualitaet(GetLiefQualZuverl.LiefNr)) AS GQ,
     TABLE (GetZuverlaessigkeit(GetLiefQualZuverl.LiefNr)) AS GZ
```

Fazit

Auch der unabhängige Fall kann mit beiden Architekturen realisiert werden.

6.2.4 Abhängiger Fall

Der abhängige Fall erweitert den unabhängigen Fall in der Form, dass nun Abhängigkeiten zwischen den lokalen Funktionen auftreten können. Dabei sind zwei Funktionen voneinander abhängig, wenn die Eingabeparameter der einen Funktion von den Ausgabeparametern der anderen Funktion abhängig sind, also das Ergebnis der einen Funktion das der anderen Funktion beeinflusst. Der unabhängige Fall wird in vier Typen oder Unterfälle aufgeteilt: den linearen, (1:n)-, (n:1)- und iterativen Fall.

6.2.4.1 Linearer abhängiger Fall

Ein abhängiger Fall ist linear, wenn eine Funktion von genau einer anderen Funktion abhängt. Ein paralleler Ablauf der Quellfunktionen ist daher nicht möglich. Im Beispiel für diesen Fall ermittelt die föderierte Funktion `GetLiefQual()` für einen gegebenen Lieferantennamen die Qualität seiner Produkte. Die lokale Funktion `GetQualitaet()` erfordert dazu jedoch die Lieferantennummer, die zuvor mit der lokalen Funktion `GetLiefNr()` abgefragt werden kann (vgl. Abbildung 6.13).

Umsetzung mit Workflow-Architektur

Der lineare abhängige Fall führt zu sequentiell ablaufenden Aktivitäten innerhalb des Workflows. Es wird zuerst die Funktion `GetLiefNr()` innerhalb einer Aktivität aufgerufen. Die so erhaltene Lieferantennummer wird vom Workflow-System an die nächste Aktivität weitergereicht. Danach erfolgt der Aufruf von `GetQualitaet()` innerhalb dieser Aktivität.

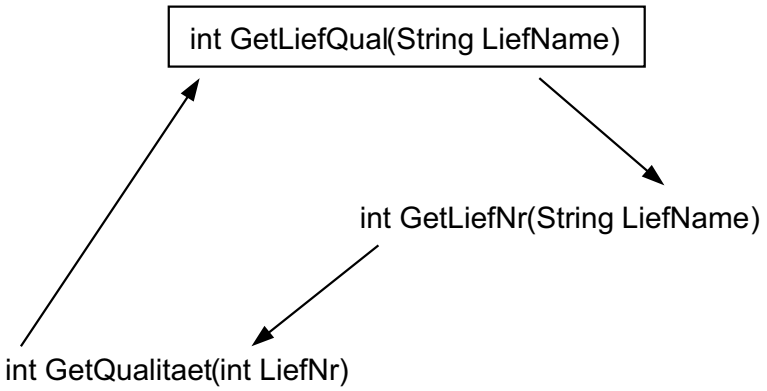

Abbildung 6.13: Parameterabhängigkeiten der föderierten Funktion `GetLiefQual()` im linearen abhängigen Fall.

Umsetzung mit UDTF-Architektur

Ähnlich wie beim unabhängigen Fall nutzt man hier erneut das kartesische Produkt. Die I-UDTF verbindet mittels des kartesischen Produkts die beiden A-Funktionen `GetLiefNr()` und `GetQualitaet()`. Hierbei muss man darauf achten, dass die Funktion `GetLiefNr()` vor der Funktion `GetQualitaet()` ausgeführt wird, damit die Lieferantennummer beim Aufruf von `GetQualitaet()` bereits zur Verfügung steht. Dies erreicht man durch die entsprechende Auflistung der UDTFs in der FROM-Klausel, d. h. `GetLiefNr()` vor `GetQualitaet()`. In DB2 werden die Funktionen in der Reihenfolge ausgeführt, in der sie referenziert werden. Somit ergibt sich folgender CREATE FUNCTION-Befehl:

```
CREATE FUNCTION GetLiefQual (LiefName VARCHAR(40))
RETURNS TABLE (Qualitaet INT)
LANGUAGE SQL RETURN
SELECT GQ.Qualitaet
FROM TABLE (GetLiefNr(GetLiefQual.LiefName)) AS GL,
     TABLE (GetQualitaet(GL.LiefNr)) AS GQ
```

Wir weisen darauf hin, dass SQL eine deklarative Sprache ist und somit solche prozeduralen Konstrukte an sich nicht unterstützt. In dem aufgeführten Fall erlaubt das eingesetzte Datenbanksystem eine solche Handhabung. Folglich kann nicht pauschal von der Unterstützung einer solchen Anweisung in relationalen Datenbanksystemen ausgegangen werden.

Fazit

Für die Workflow-Architektur stellt der lineare abhängige Fall keinerlei Problem dar. Bei der UDTF-Architektur ist mit den eingesetzten Produkten eine Umsetzung möglich. Jedoch ist dies nicht allgemein gültig, da relationale Datenbanksysteme an sich keine prozeduralen Konstrukte unterstützen.

6.2.4.2 (1:n)-abhängiger Fall

Der (1:n)-Typ des abhängigen Falles liegt vor, wenn eine lokale Funktion von mehreren lokalen Funktionen abhängig ist. Diese lokale Funktion kann erst aufgerufen werden, wenn alle Funktionen, von denen sie abhängig ist, ihre Arbeit beendet haben.

Das Beispiel zu diesem Fall sieht eine föderierte Funktion vor, die bei gegebenen Lieferanten- und Komponentennamen die Anzahl der gelagerten Komponenten zurückgibt. Die Anzahl erhält man mit der Funktion `GetAnzahl()` des Lagersystems. Diese benötigt jedoch nicht die Namen von Lieferant und Komponente, sondern die entsprechenden Nummern. Die Komponentennummer liefert das CAD-System über die Funktion `GetKompNr`. Die Lieferantennummer erhält man im Einkaufssystem durch den Aufruf der Funktion `GetLiefNr()`. Somit ist die Funktion `GetAnzahl()` von den anderen beiden Funktionen abhängig und kann erst aufgerufen werden, wenn die Ergebnisse der anderen beiden vorliegen (vgl. Abbildung 6.14).

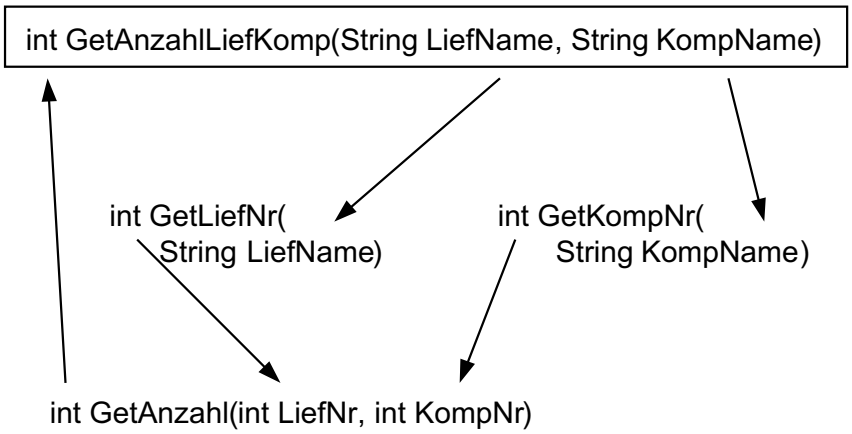

Abbildung 6.14: Parameterabhängigkeiten der föderierten Funktion `GetAnzahlLiefKomp()` im (1:n)-abhängigen Fall.

Umsetzung mit Workflow-Architektur

Der (1:n)-abhängige Fall wird im Workflow-Prozess so abgebildet, dass der Kontrollfluss von mehreren Aktivitäten zu einer einzigen verweist. Alle Vorgängeraktivitäten dieser einen Aktivität müssen abgeschlossen sein, bevor sie gestartet werden kann. Somit enthält der Prozess drei Aktivitäten, wobei die ersten beiden parallel ablaufen können und die Funktionen `GetLiefNr()` und `GetKompNr()` aufrufen. Die dritte Aktivität, welche die Funktion `GetAnzahl()` aufruft, wird erst gestartet, nachdem die ersten beiden erfolgreich durchgeführt wurden.

Umsetzung mit UDTF-Architektur

Vergleichbar zum linearen abhängigen Fall kann auch bei diesem Fall mit dem kartesischen Produkt über zwei A-UDTFs gearbeitet werden. Dabei führt man die A-UDTFs für die drei lokalen Funktionen wie folgt in der I-UDTF zusammen:

```
CREATE FUNCTION GetAnzahlLiefKomp (LiefName VARCHAR(40), KompName VARCHAR(60))
RETURNS TABLE (Anzahl INT)
LANGUAGE SQL RETURN
SELECT GA.Anzahl
FROM TABLE (GetLiefNr(GetAnzahlLiefKomp.LiefName)) AS GL,
     TABLE (GetKompNr(GetAnzahlLiefKomp.KompName)) AS GK,
     TABLE (GetAnzahl(GL.LiefNr, GK.KompNr)) AS GA
```

Auch hier gilt wegen der prozeduralen Abarbeitung, dass die Funktion `GetAnzahl()` als letztes referenziert wird. Die Reihenfolge der anderen beiden Funktionen ist dagegen nicht von Bedeutung, da sie parallel ablaufen können.

Fazit

Die Realisierung des (1:n)-Typs des abhängigen Falles ist mit beiden Architekturen möglich. Aber auch in diesem Fall gelten die Aussagen aus dem linearen abhängigen Fall zu prozeduralen Konstrukten in SQL.

6.2.4.3 (n:1)-abhängiger Fall

Der (n:1)-Typ des abhängigen Falls ist die Umkehrung des vorangegangenen Falls, wobei nun mehrere lokale Funktionen von dem Ergebnis einer lokalen Funktion abhängig sind. Nachdem diese Funktion beendet ist, können alle von ihr abhängigen Funktionen parallel ablaufen.

Für diesen Fall wird das Beispiel aus Abschnitt 6.2.3 *(unabhängiger Fall)* erweitert, indem nun der Lieferantenname als Eingabe zur Verfügung steht und dieser zunächst mit der Funktion `GetLiefNr()` des Einkaufssystems in die entsprechende Lieferantennummer gewandelt werden muss (vgl. Abbildung 6.15). Anschließend können die lokalen Funktionen `GetQualitaet()` und `GetZuverlaessigkeit()` aufgerufen werden.

Umsetzung mit Workflow-Architektur

Der (n:1)-Typ des abhängigen Falls hat Aufgabelungen innerhalb des Workflow-Prozesses zur Folge. Auf eine Aktivität folgen mindestens zwei weitere Aktivitäten, die parallel ablaufen können. Das Beispiel wird auf drei Aktivitäten innerhalb des Workflows abgebildet. Die erste ruft die Funktion `GetLiefNr()` auf. Die Aktivitäten, welche `GetQualitaet()` und `GetZuverlaessigkeit()` aufrufen, können anschließend von dem Workflow-System parallel ausgeführt werden. Die Zusammenführung der Ergebnistabelle erfolgt in der UDTF oder wird durch eine Hilfsfunktion im Workflow-System umgesetzt.

Umsetzung mit UDTF-Architektur

Analog zum linearen und (1:n)-Typ des abhängigen Falles wird erneut das kartesische Produkt in einer I-UDTF eingesetzt:

```
CREATE FUNCTION GetLiefQualZuverl_2 (LiefName VARCHAR(40))
RETURNS TABLE (Qualitaet INT, Zuverlaessigkeit INT)
LANGUAGE SQL RETURN
```

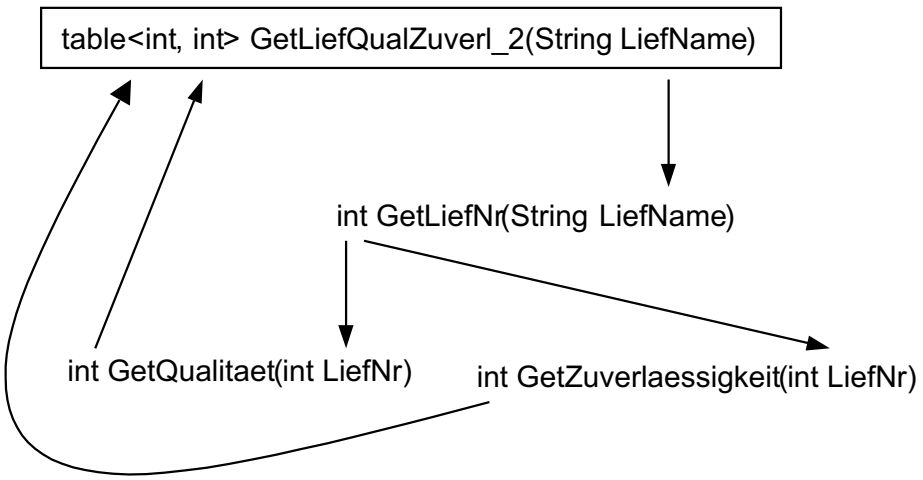

Abbildung 6.15: Parameterabhängigkeiten der föderierten Funktion `GetLiefQualZuverl_2()` im (n:1)-abhängigen Fall.

```
SELECT GQ.Qualitaet, GZ.Zuverlaessigkeit
FROM TABLE (GetLiefNr(GetLiefQualZuverl_2.LiefName)) AS GL,
     TABLE (GetQualitaet(GL.LiefNr)) AS GQ,
     TABLE (GetZuverlaessigkeit(GL.LiefNr)) AS GZ
```

Bei der Produktbildung ist darauf zu achten, dass die A-UDTF `GetLiefNr()` vor den A-UDTFs `GetZuverlaessigkeit()` und `GetQualitaet()` aufgerufen wird, da die beiden letzteren deren Ausgabeparameter benötigen.

Fazit

Wie im (1:n)-Typ des abhängigen Falles ist die Umsetzung mit beiden Architekturen möglich. Es gelten ebenfalls die Aussagen aus dem linearen abhängigen Fall zu prozeduralen Konstrukten in SQL.

6.2.4.4 Iterativer abhängiger Fall

Bei diesem Typ existiert eine iterative Abhängigkeit von einer Funktion f_1 zu einer weiteren Funktion f_2. Iterativ bedeutet, dass mehrere Instanzen von f_1 abhängig von f_2 sind. Dies tritt z. B. auf, wenn f_2 als Ausgabeparameter eine Liste hat und jedes einzelne Listenelement von jeweils einer Instanz von f_1 bearbeitet wird. Im Abhängigkeitsgraphen werden Zyklen durch einen Doppelpfeil symbolisiert.

In unserem Beispiel soll zu einer gegebenen Lieferantennummer der Name der gelieferten Komponente zurückgegeben werden. Dazu benötigt man die lokale Funktion `GetKompNr()` des Lagersystems, die eine Liste mit Nummern der gelieferten Komponenten zurückliefert. Für jedes Element dieser Liste wird anschließend die Funktion

`GetKompName()` des CAD-Systems aufgerufen und die Ergebnisse werden in einer Liste zusammengefasst[2] (vgl. Abbildung 6.16).

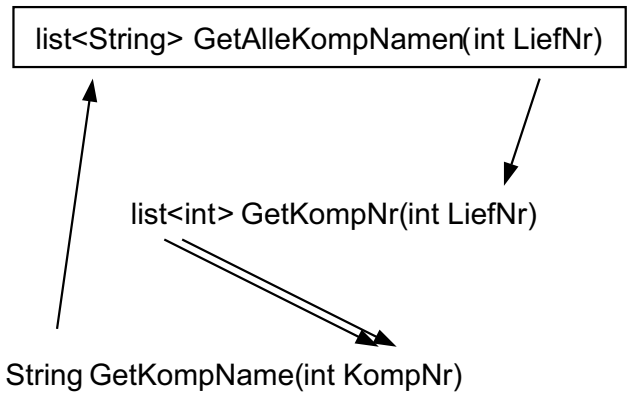

Abbildung 6.16: Parameterabhängigkeiten der föderierten Funktion `GetAlleKompNamen()` im iterativen abhängigen Fall.

Umsetzung mit Workflow-Architektur

Die Umsetzung des iterativen abhängigen Falles in der Workflow-Architektur setzt die Unterstützung von Schleifen voraus. Das für diese Arbeit eingesetzte Workflow-System bietet diese Funktionalität an. Das Beispiel wird auf zwei Aktivitäten abgebildet, wobei die erste Aktivität, welche die Funktion `GetKompNr()` ausführt, zuerst aufgerufen wird. Sie übergibt die Ergebnisliste mit den Komponentennummern und eine Zählvariable an das Workflow-System, das diese an die zweite Aktivität weiterreicht. Die zweite Aktivität wird von dem Workflow-System in einer Schleife für jedes Listenelement aufgerufen und die Zählvariable entsprechend hochgezählt. Die Schleife wird von dem Workflow-System beendet, wenn die Zählvariable den Wert der Listenlänge hat.

Umsetzung mit UDTF-Architektur

SQL:1999 kennt keine prozeduralen Konstrukte und somit auch keine Schleifen. Ein Schleifenkonstrukt wird lediglich für gespeicherte Prozeduren [ISO02] angeboten, die in SQL geschrieben werden. Da diese nicht eingesetzt werden, ist die Zuhilfenahme einer Schleife nicht möglich. Die Simulation einer Schleife durch eine SQL-Rekursionsabfrage mit dem Schlüsselwort WITH weist Einschränkungen auf, die in diesem Fall nicht weiterhelfen. Dabei soll auf die SQL-Rekursion an dieser Stelle nicht näher eingegangen werden, da sie detaillierte SQL-Kenntnisse voraussetzt. Eine Umsetzung des Beispiels auf die UDTF-Architektur ist also nicht möglich.

Zur Lösung des Problem kann lediglich die erweiterte Java-UDTF-Architektur verwendet werden. Da die I-UDTFs in Java geschrieben sind, können auch Schleifen implementiert werden. Die I-UDTF enthält in diesem Fall die folgende SQL-Anfrage, welche eine einspaltige Tabelle mit den Komponentennummern liefert:

[2] Listen werden in den Architekturen mit UDTFs durch einspaltige Tabellen dargestellt

```
SELECT GKN.KompNr
FROM TABLE (GetKompNr(LiefNr)) AS GKN
```

Für jede zurückgelieferte Komponentennummer wird anschließend die folgende Anweisung in einer Schleife ausgeführt:

```
SELECT GKA.KompName
FROM TABLE (GetKompName(KompNr[i])) AS GKA
```

Fazit

Der iterative abhängige Fall kann mit der Workflow-Architektur realisiert werden, sofern das Workflow-System Schleifen unterstützt. Mit der UDTF-Architektur ist sie jedoch nur unter Verwendung von Java-I-UDTFs statt der SQL-I-UDTFs umzusetzen.

6.2.5 Allgemeiner Fall

Der allgemeine Fall setzt sich aus den bisher aufgeführten Fällen zusammen und erweitert sie in der Weise, dass nun mehrere föderierte Funktionen möglich sind. Da zwischen den föderierten Funktionen keine Abhängigkeiten bestehen, kommen keine neuen Aspekte hinzu, so dass dieser Fall nicht weiter betrachtet werden muss.

6.2.6 Zusammenfassung

In diesem Abschnitt wurde gezeigt, dass die erweiterte SQL-UDTF-Architektur viele der untersuchten Fälle unterstützen kann und somit den Anfang einer leichtgewichteten Workflow-Technologie darstellt. Jedoch bietet das WfMS weit mehr Funktionalität beispielsweise in Form von Bedingungen, die nicht mit SQL ausgedrückt werden können. Die Beispiele zeigen überdies, dass es viel einfacher ist, die Abbildungen mit Hilfe eines Workflows zu spezifizieren und zu implementieren. Zudem werden die SQL-Anweisungen mit zunehmender Abbildungskomplexität, d. h. mit wachsender Anzahl an lokalen, zu integrierenden Funktionen, ebenfalls komplexer und unübersichtlich. Außerdem ist zu beachten, dass derzeit lediglich ein lesender Zugriff auf Basis von UDTFs unterstützt werden kann. Nichtsdestotrotz kann der UDTF-Ansatz sehr nützlich für einfache Anwendungen mit weniger komplexen Abbildungen sein, die nicht die volle Bandbreite eines WfMS benötigen [HH02a].

6.3 Empirische Untersuchungen

Nachdem in Kapitel 6.2 die Abildungsmächtigkeit der beiden Architekturalternativen untersucht wurde, betrachten wir nun deren Leistungsverhalten. Von besonderem Interesse ist hier das Abschneiden der Workflow-Architektur. Wir erhoffen uns dabei eine Antwort auf die Frage, welchen Verwaltungsaufwand und damit welche Leistungseinbußen man für den Einsatz einer vermeintlich überdimensionierten Komponente in Kauf nehmen muss. Sind die Antwortzeiten akzeptabel und wie groß ist der Unterschied zu den Zeiten der UDTF-Architektur?

In den folgenden Abschnitten stellen wir zunächst die implementierten Prototypen und notwendige Erweiterungen an den Architekturen vor. Anschließend werden die Messergebnisse bewertet, welche für die implementierten Beispiele aus Kapitel 6.2 ermittelt wurden. Da die Hersteller inzwischen auch „leichtgewichtige“ Varianten von Workflows anbieten, wurde auch dazu ein Prototyp angefertigt. So können wir unterschiedliche WfMS-Implementierungen miteinander vergleichen und auch der UDTF-Architektur gegenüberstellen.

6.3.1 Implementierung der Prototypen

Für die Implementierung der Beispiele wurden zunächst Anwendungssysteme mit Hilfe von Java-Programmen simuliert, welche die lokalen Funktionen zur Verfügung stellen und bei deren Aufruf auf eine DB2 zugreifen. Beide Prototypen wurden mit IBM's DB2 Universal Database Version 7.1 (kurz DB2) implementiert. Bei der Workflow-Architektur wurde IBM's MQ Series Workflow Version 3.2.2 (kurz MQ Workflow) eingesetzt.

In beiden Fällen waren Erweiterungen der Architektur notwendig, da DB2 aus Sicherheitsgründen nicht erlaubt, dass ein Datenbankanwendungsprozess E/A-Operationen mit anderen Systemen als dem DBS selbst vornimmt. Der Grund hierfür ist, dass der Prozess Zugang zum kompletten Adressraum des DBS hat und er somit gezielt oder unbeabsichtigt Daten des DBS unkontrolliert modifizieren kann. Daher wurde in beiden Prototypen ein Controller zur Entkopplung von Prozessen eingeführt. Die Funktionsweise des Controllers wird bei der Beschreibung der Prototypen genauer erläutert.

Wir weisen darauf hin, dass bei den implementierten Beispielen nur die Umsetzung von föderierten Funktionen im Vordergrund stand und nicht die Kombination von Daten- und Funktionsintegration. Dies ist dadurch begründet, dass zum einen die Funktionsintegration die neue und unbekannte Vorgehensweise darstellt, während die Datenintegration zu großen Teilen erforscht ist. Zum anderen soll der Overhead des WfMS gemessen werden, der bei der Funktionsintegration auftritt, so dass die Datenintegration hier nicht von Bedeutung ist.

6.3.1.1 Implementierung der Workflow-Architektur

Wie in Abbildung 6.17 dargestellt, basiert die Implementierung auf dem Zusammenspiel von DB2 und MQ Workflow. DB2 übernimmt die Integration von Datenbanken. (Teil-) Anfragen, die föderierte Funktionen referenzieren, leitet DB2 an MQ Workflow weiter. Da die derzeit auf dem Markt verfügbaren Produkte SQL/MED-Wrapper nicht unterstützen, wurden UDTFs zur Kopplung der beiden Systeme implementiert. Aus den oben erläuterten Sicherheitsgründen müssen jedoch der UDTF-Prozess und der Prozess zum Aufruf des WfMS entkoppelt werden. Dazu wird ein Workflow-Controller eingeführt, welcher einen eigenen Prozess benötigt und dem ein Java-Programm zugrunde liegt. Er verbindet die UDTFs, die ihn über RMI kontaktieren, mit dem WfMS. Wird der Controller gestartet, baut er eine Verbindung zur Workflow-Laufzeitumgebung auf, führt die Passwortabfrage zum WfMS durch und registriert sich bei der RMI Registry, die ebenfalls zu starten ist. Danach wartet der Controller, bis eine seiner Funktionen über RMI aufgerufen wird. Dabei ist pro UDTF eine Funktion implementiert, die für den

Start genau eines Workflow-Prozesses zuständig ist. Bevor die Funktion den Workflow starten kann, muss sie die über RMI erhaltenen Eingabeparameter in einen Workflow-Container packen und diesen dem WfMS übergeben. Sollte der Controller sich als Nadelöhr erweisen, so können mehrere davon eingesetzt und somit die Last auf mehrere Systeme verteilt werden. Die Kommunikation zwischen den UDTF-Prozessen und dem Prozess des Workflow-Controllers muss nicht zwingend über RMI erfolgen. Alternative Lösungen sind beispielsweise mit CORBA, Sockets oder Dateien denkbar.

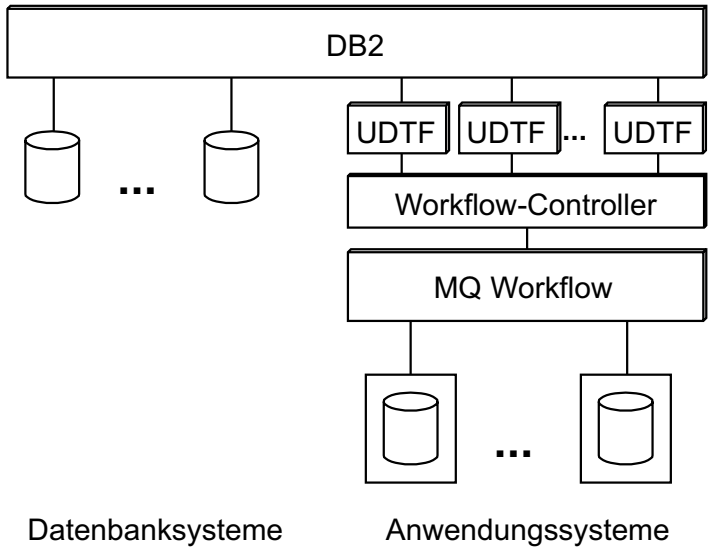

Abbildung 6.17: Die Workflow-Architektur mit Controller-Erweiterung.

Der Einsatz des Workflow-Controller hat einen weiteren positiven Nebeneffekt, der zur Erhöhung der Performanz des Systems beiträgt. Ohne den Controller müsste bei jedem Aufruf einer UDTF das Laufzeitsystem des WfMS kontaktiert und vorbereitet werden. Durch den Controller wird dies nur einmal beim Start durchgeführt und anschließend die bestehende Verbindung gehalten. Die Aufgaben der UDTFs ändern sich im Vergleich zu der in Kapitel 5 vorgestellten Architektur insofern, dass sie nun nicht mehr direkt das WfMS kontaktieren, sondern ihre zugehörige Funktion im Workflow-Controller über RMI aufrufen. Der Aufruf enthält die Eingabeparameter, die von dem DBMS übergeben werden. Ist der RMI-Aufruf abgesetzt, wartet die UDTF auf die Rückgabeparameter und erstellt aus diesen die Ergebnistabelle für die Anfrageverarbeitung von DB2.

Betrachtet man als Nächstes MQ Workflow, so benötigt man zwischen dem Workflow-System und den Anwendungssystemen ein Stück Software, hier Adapter genannt, die den Funktionsaufruf aus den Workflow-Aktivitäten heraus unterstützt. Dieser Aufruf kann nicht direkt erfolgen, da sich die zu übergebenden Parameter in Workflow-Containern befinden und das Anwendungssystem normalerweise nichts mit den Containern anzufangen weiß. Folglich ist die Aufgabe der Adapter, die Workflow-Container der Aktivitäten zu entpacken und die Funktion mit den so gewonnenen Eingabeparametern aufzurufen. Wurde die Funktion erfolgreich durchgeführt, verpacken die Adapter das Ergebnis der Funktionsaufrufe erneut in einen Workflow-Container und übergeben diesen an das WfMS.

6.3.1.2 Implementierung der UDTF-Architektur

Bei der UDTF-Architektur übernimmt DB2 dieselbe Rolle wie bei der Workflow-Architektur. Die Verbindung zu föderierten Funktionen erfolgt ebenfalls über UDTFs (siehe Abbildung 6.18). Wie in Abschnitt 6.1.3 beschrieben, wird jedoch das WfMS durch eine zweistufige Kombination von Tabellenfunktionen ersetzt. Die erste Stufe bilden die UDTFs, welche die Anbindung jeder einzelnen lokalen Funktion realisieren und in SQL-Anweisungen referenziert werden können. Die föderierten Funktionen werden auf Basis von SQL-I-UDTFs umgesetzt, die wiederum auf die Java-A-UDTFs zugreifen. Zur Funktionsintegration wird somit per SELECT-Befehl auf eine föderierte Funktion in Form einer I-UDTF zugegriffen, die selbst A-UDTFs aufruft. Diese greifen wiederum auf die lokalen Funktionen zu.

Genau an diesem Punkt treten erneut die Sicherheitseinschänkungen seitens DB2 auf den Plan. Auch hier muss eine Entkopplung des UDTF-Prozesses und der Prozesse der Anwendungssysteme stattfinden. Daher wird erneut ein Controller eingeführt, welcher die Entkopplung der UDTF-Prozesse von den Anwendungssystemprozessen vornimmt. Der Controller wird über RMI aufgerufen und startet für die JDBC-Zugriffe neue Prozesse.

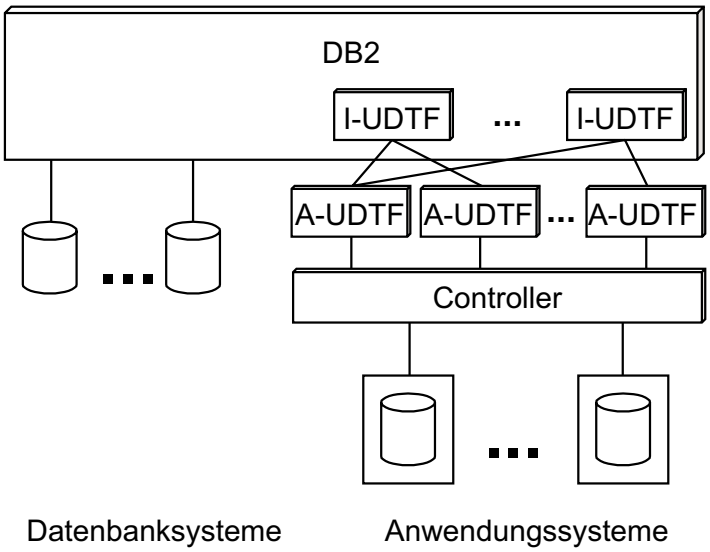

Abbildung 6.18: Die UDTF-Architektur mit Controller-Erweiterung.

Wir möchten nochmals betonen, dass diese Entkopplung nur notwendig wurde, weil die Anwendungssysteme in unserer Implementierung als JDBC-Programme simuliert wurden, die auf dieselbe Datenbankinstanz zugreifen. In der Realität werden die zu integrierenden Anwendungssysteme auf anderen Rechnern mit anderen Datenbanken arbeiten, so dass die beschriebenen Sicherheitseinschränkungen seitens DB2 nicht mehr auftreten und folglich der Controller nicht benötigt wird.

Durch den Einsatz des Controllers und RMI haben die beiden implementierten Architekturen einen ähnlichen Aufbau. Dies erleichtert den folgenden Vergleich der Ergebnisse der Leistungsmessungen.

6.3.2 Ergebnisse der Leistungsmessungen

In den folgenden Abschnitten analysieren wir die Ergebnisse der Performanztests der beiden Architekturen. Die in Abschnitt 6.2 beschriebenen föderierten Funktionen wurden auf Basis der beiden Architekturen implementiert und deren Ausführungszeiten gemessen. Die Ausführungszeiten der föderierten Funktionen hängen stark davon ab, welche Funktionen bereits zuvor ausgeführt wurden. Der Inhalt des Hauptspeichers und der Caches spielt eine maßgebliche Rolle. Um dies zu berücksichtigen, wurden pro föderierter Funktion drei verschiedene Messungen durchgeführt:

- Ausführung der föderierten Funktion nach Systemstart.
- Ausführung der föderierten Funktion nach anderen bereits ausgeführten Funktionen. Diese Funktionen durften aber keine gemeinsamen Aktivitäten aufweisen.
- Wiederholte Ausführung derselben föderierten Funktion.

Wir stellen die Ergebnisse gegenüber und erläutern unsere Beobachtungen.

6.3.2.1 Leistungsverhalten der Workflow-Architektur

Die Messungen auf Basis der Workflow-Architektur zeigen, dass die Funktionsausführungen nach Neustart ungefähr doppelt so lang dauern wie die anderen beiden gemessenen Fälle (vgl. Abbildung 6.19). Die gemessenen Werte liegen zwischen 5 und 24 Sekunden. Da ein Neustart bei normalem Systembetrieb eher selten durchgeführt wird, sind die Messwerte der anderen beiden Fälle eher als realistisch anzusehen. Wir beziehen daher unsere weiteren Beobachtungen auf gemessene Werte ohne Neustart.

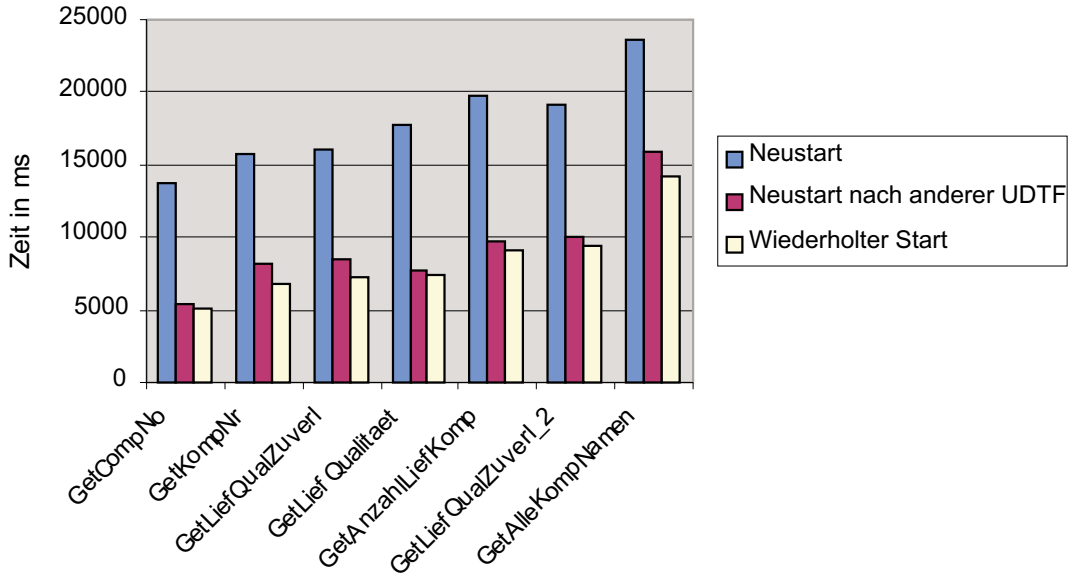

Abbildung 6.19: Messergebnisse für die föderierten Funktionen in der Workflow-Architektur.

Mit Hilfe der föderierten Funktion `GetAlleKompNamen()` haben wir das Verhalten des Systems bei steigender Anzahl von Aktivitäten gemessen. Die Skalierung der Zahl an auszuführenden Aktivitäten eines Workflows wurde über die Ergebnismenge der Funktion `GetKompNr()` gesteuert. Da hier eine iterative Abhängigkeit zur Funktion `GetKompName()` besteht, legt die Kardinalität des Ergebnisses von `GetKompNr()` die Anzahl wiederholter Aufrufe von `GetKompName()` fest.

Die Messungen zeigen, dass die Messkurve über der Anzahl an Aktivitäten einen nahezu linearen Verlauf nimmt[3] (siehe Abbildung 6.20). Dieses Verhalten kann bei sehr großen Datenmengen und damit einer wachsenden Anzahl an Aktivitätsausführungen zu inakzeptablen Antwortzeiten führen. Workflows ohne iterativen Fall zeigen hier keine Probleme.

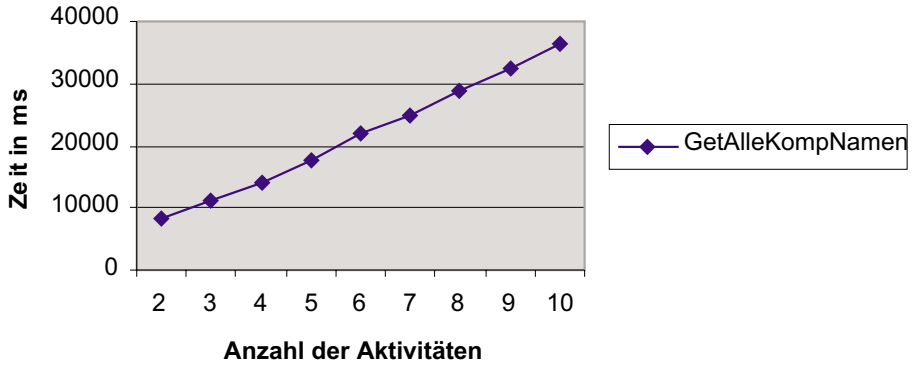

Abbildung 6.20: Zeitliches Verhalten bei steigender Anzahl an Aktivitäten in der Workflow-Architektur.

Die Messungen zeigen überdies, dass der unabhängige Fall schneller ausgeführt werden kann als der abhängige Fall, da Aktivitäten parallel ablaufen können. Die Antwortzeit für die föderierte Funktion `GetLiefQualZuverl()` ist kürzer als die für `GetLiefQualitaet()` (Abbildung 6.19), obwohl beide genau zwei Aktivitäten enthalten, deren alleinige Ausführungszeiten nahezu identisch sind (vgl. Anhang D).

Der Workflow-Controller nimmt 14,6 Sekunden für den erstmaligen Start und 11,6 Sekunden für den wiederholten Start in Anspruch. Diese Zeit verwendet er fast ausschließlich zum Kontaktieren der Laufzeitumgebung und der Anmeldung mit Benutzerkennung und Passwort. Läuft der Workflow-Controller bei der Ausführung einer föderierten Funktion, können folglich elf Sekunden eingespart werden.

Ein weiterer interessanter Aspekt ist die Aufteilung der benötigten Ausführungszeit in einzelne Phasen. Abbildung 6.21 verdeutlicht, wie viel Zeit die einzelnen Schritte der föderierten Funktion `GetAnzahlLiefKomp()` benötigen. Diese Funktion besteht aus drei Aktivitäten, wovon zwei parallel ablaufen. Die Ausführung der Aktivitäten nimmt mit 51% den größten Anteil an der Gesamtzeit in Anspruch. Weitere 23% werden benötigt,

[3] Hier weisen wir darauf hin, dass es sich um eine variierende Anzahl ein und derselben Aktivität bzw. Funktion handelt und nicht unterschiedlicher Aktivitäten und Funktionen.

um vor dem WfMS-Einsatz das Aufrufen und Ausführen der Tabellenfunktionen vorzubereiten. Dabei fällt auf, dass das Starten der Workflow-Prozessinstanz sowie der Java-Laufzeitumgebung für das Java-API des WfMS nit 10% relativ viel Zeit benötigt. Dieser relative Anteil wird aber mit steigender Anzahl integrierter Funktionen kleiner, weil seine Dauer konstant bleibt – unabhängig von der Anzahl ausgeführter Aktivitäten.

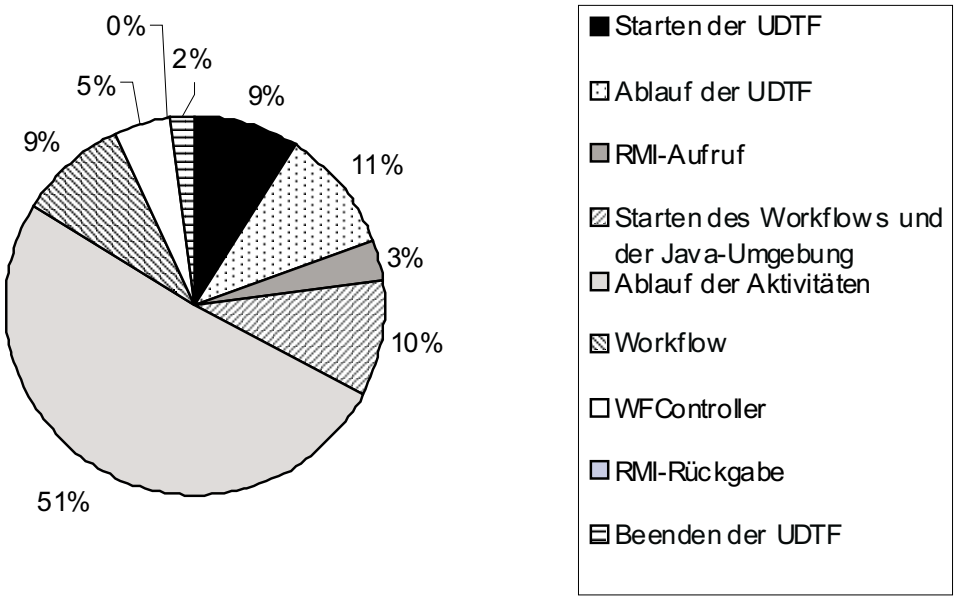

Abbildung 6.21: Zeitaufteilung für die einzelnen Aufgaben in der Workflow-Architektur für die föderierte Funktion `GetAnzahlLiefKomp()`.

Die genauen Messergebnisse können dem Anhang D entnommen werden.

6.3.2.2 Leistungsverhalten der UDTF-Architektur

Die den iterativen abhängigen Fall repräsentierende föderierte Funktion `GetAlleKompNamen()` konnte nicht mit der UDTF-Architektur implementiert werden, da DB2 v7.1 die Umsetzung des iterativen Aufrufes nicht unterstützt (vgl. Abschnitt 6.2). Alle anderen föderierten Funktionen wurden auf Basis der UDTF-Architektur implementiert und den gleichen drei Messungen wie bei der Workflow-Architektur unterzogen (nach Systemneustart, nach anderen Funktionen, als wiederholter Aufruf derselben Funktion). Bei den unterschiedlichen Messungen konnten ähnliche Effekte wie bei der Workflow-Architektur beobachtet werden (siehe Abbildung 6.22). Die Ausführung nach einem Systemneustart benötigt ungefähr doppelt so viel Zeit wie die anderen beiden Varianten.

Die gemessenen Zeiten für die föderierten Funktionen liegen zwischen zwei und sieben Sekunden. Folglich sind die Funktionen in akzeptabler Zeit ausführbar. Die Auswirkungen des iterativen Falles konnten leider nicht ermittelt werden.

Wie bei der Workflow-Architektur wurde auch bei der UDTF-Architektur die föderierte Funktion `GetAnzahlLiefKomp()` in Phasen aufgeteilt, um die benötigte Zeit für die ein-

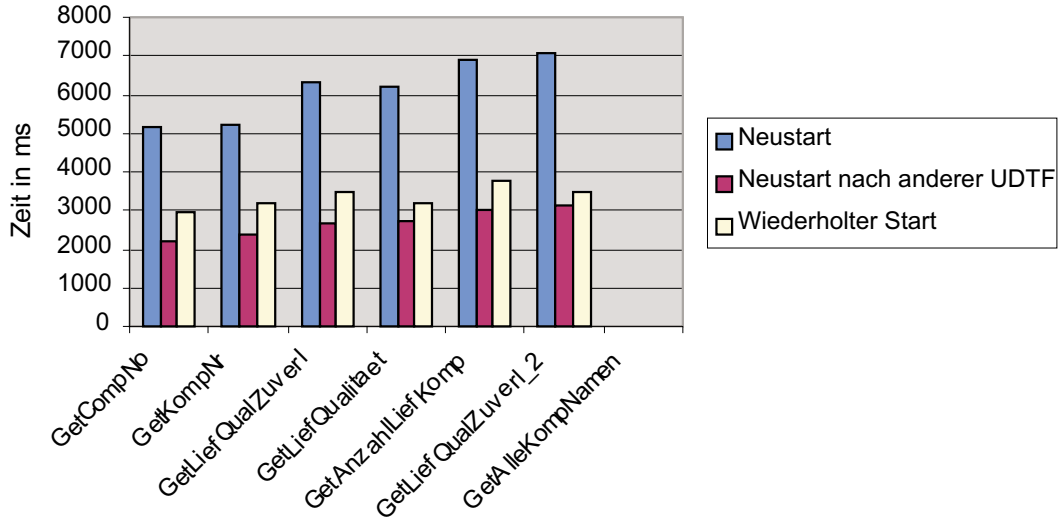

Abbildung 6.22: Messergebnisse für die föderierten Funktionen in der UDTF-Architektur.

zelnen Aufgaben zu messen (vgl. Abbildung 6.23). Im Vergleich zur Workflow-Architektur ergeben sich hier andere Verhältnisse. Das Starten und Beenden der Integrationstabellenfunktion beansprucht 20% der Gesamtzeit. Dies entspricht prozentual ungefähr dem Workflow-Ansatz mit 22% für Starten (9%), Ausführen (11%) und Beenden (2%) der Tabellenfunktion. Die Ausführungsdauer der drei Zugriffstabellenfunktionen beläuft sich auf 49% der Gesamtlaufzeit, während die eigentliche Ausführung der lokalen Funktionen lediglich 6% der gesamten Ausführungszeit benötigt.

Auffallend bei dieser Architektur ist, dass die Controller insgesamt 25% der Zeit beanspruchen. Dieses Verhalten hat zwei Gründe. Erstens dauern die Funktionen bei der Workflow-Architektur wesentlich länger als bei der UDTF-Architektur. Deshalb fällt ein RMI-Aufruf, der bei beiden Architekturen absolut gesehen ungefähr gleich lang braucht, relativ gesehen viel stärker ins Gewicht. Zweitens findet bei der Workflow-Architektur nur ein RMI-Aufruf statt, während die föderierte Funktion bei der UDTF-Architektur für alle drei A-UDTFs einen RMI-Aufruf benötigt. An dieser Stelle sind somit Optimierungspotentiale vorhanden, wenn der Controller durch eine effizientere Anbindungsmaßnahme ersetzt werden kann.

Ebenso auffällig sind die extremen Zeitunterschiede zwischen den beiden Architekturen beim Ablauf der Aktivitäten. Dieser Unterschied erklärt sich hauptsächlich durch den Start der Java-Programme. Bei der UDTF-Architektur werden die Aktivitäten innerhalb des bereits laufenden Controllers durchgeführt. Bei der Workflow-Architektur ist hingegen für jede Aktivität der Start eines neuen Java-Prozesses inklusive dem Laden der Java Virtual Machine notwendig. Darüber hinaus müssen für jede Aktivität die Container mit den Parameterwerten bearbeitet werden, was zusätzlich Zeit in Anspruch nimmt.

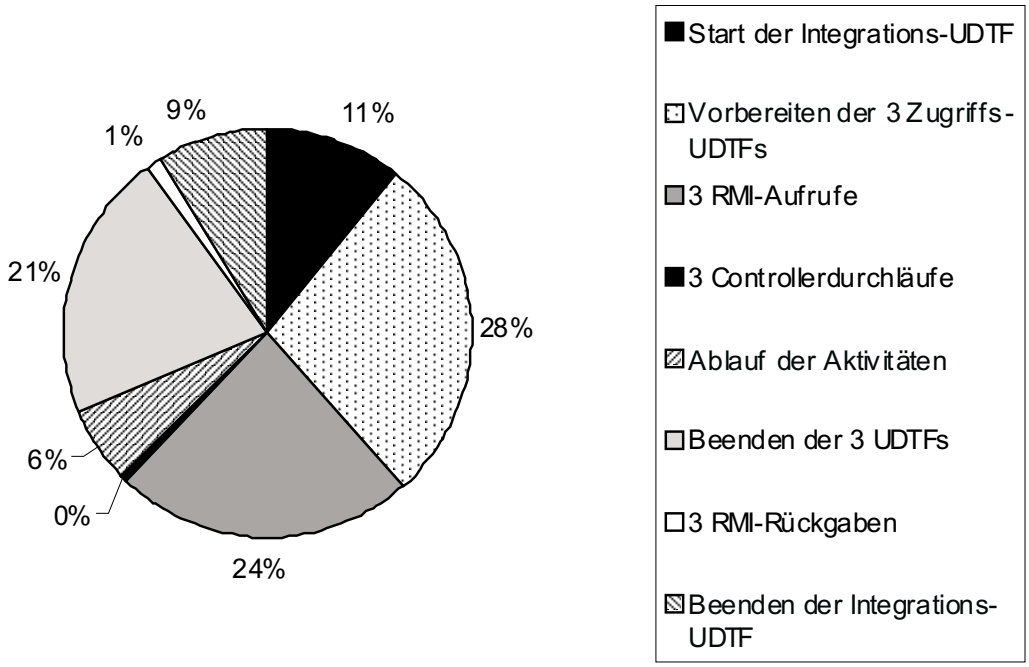

Abbildung 6.23: Zeitaufteilung für die einzelnen Aufgaben in der UDTF-Architektur für die föderierte Funktion `GetAnzahlLiefKomp()`.

6.3.2.3 Leistungsvergleich der beiden Architekturen

Abbildung 6.24 zeigt die durchschnittliche Gesamtdauer für die Ausführung der implementierten föderierten Funktionen mit der Workflow- und der UDTF-Architektur. Wir können nun die absoluten Werte vergleichen. Die Messergebnisse der beiden Architekturen zeigen, dass die Workflow-Architektur zwei- bis viermal langsamer ist als die UDTF-Architektur. Dies ist ein akzeptables Ergebnis angesichts einer zusätzlichen und sehr mächtigen Software-Komponente in der Workflow-Architektur. Des Weiteren sehen wir, dass bei ansteigender Anzahl lokaler Funktionen die Ausführungszeiten des Workflow-Ansatzes schneller steigen.

Vergleicht man die Aufteilung der Ausführungszeiten, fallen folgende Aspekte auf. Die Java-Umgebung für das Java-API und das Starten der Workflow-Prozessinstanz nehmen mit 10% sehr viel Zeit in Anspruch. Dieser relative Anteil wird jedoch kleiner, wenn die Anzahl der zu integrierenden Funktionen steigt, da seine Dauer konstant bleibt – unabhängig von den ausgeführten Funktionen. Bei der UDTF-Architektur belegt der Controller 25% der Gesamtdauer. Folglich sind hier Optimierungspotentiale vorhanden, sofern der Controller durch eine effizientere Anbindungsmaßnahme ersetzt werden kann.

Eine weitere interessante Beobachtung ist das Verhalten bei paralleler und sequentieller Verarbeitung der integrierten Funktionen. Offensichtlich kann die UDTF-Architektur eine parallele Verarbeitung nicht effizient umsetzen. Entgegen der Erwartungen ist die parallele Ausführung bei der UDTF-Architektur in manchen der gemessenen Fälle sogar langsamer als die sequentielle Ausführung.

Zusammenfassend können wir feststellen, dass ein Workflow-System nicht das Ausmaß an Mehraufwand mit sich bringt wie befürchtet. Die resultierenden Ausführungszeiten bleiben in einem akzeptablen Rahmen: sie sind nur zwei- bis viermal langsamer als bei einer vergleichbaren Lösung ohne WfMS, obwohl man auf den ersten Blick aufgrund des Einsatzes zweier Server sicherlich einen größeren Unterschied erwarten würde. Wir weisen jedoch darauf hin, dass die Performanz nicht das wichtigste Kriterium für unseren Ansatz ist. Essentielle Merkmale sind für uns die Unterstützung von komplexen Abbildungen, einfache Handhabung und vor allem die Tatsache, dass die Entwickler als auch das FDBS hinsichtlich verteilter Programmierung über heterogene Schnittstellen hinweg entlastet werden [HH02a].

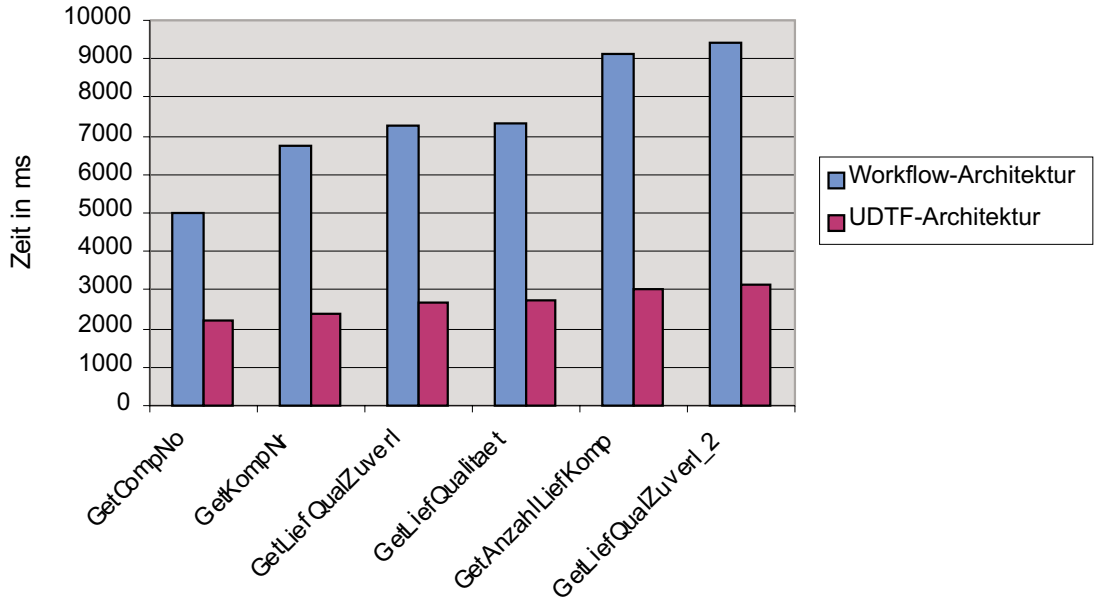

Abbildung 6.24: Die durchschnittliche Gesamtdauer für die Ausführung der einzelnen föderierten Funktionen mit der Workflow- und UDTF-Architektur.

6.4 Workflow-Varianten

In den Kapiteln zuvor haben wir gezeigt, dass man die Funktionsintegration auf Basis eines Workflow-Systems sinnvoll umsetzen kann und sich die Performanz in einem akzeptablen Bereich bewegt. Nichtsdestotrotz sind derzeitige Workflow-Systeme zu schwerfällig für den von uns angestrebten Anwendungsbereich. Ihr Gebrauch erscheint überdimensioniert, da sie für einen breiter angelegten Einsatz gedacht sind, bei dem Benutzerinteraktion eine wichtige Rolle spielt. Wir benötigen hingegen ein leichtgewichtiges Workflow-System, das für vollständig automatisierte Prozesse ausgelegt ist und jegliche darüber hinaus gehende Funktionalität weglässt. Eine solche Light-Version könnte die Performanz des Systems positiv beeinflussen.

Auch die bekannten Software-Hersteller haben diesen Bedarf an abgespeckten Workflow-Systemen erkannt und 2002 hat IBM die Unterstützung leichtgewichtiger Workflows in

ihren Applikations-Server eingebaut. IBM nennt diese Art von Workflows *Microflows* und beschreibt den Unterschied zwischen Workflows und Microflows wie folgt:

> *„Ein Microflow ist ein leichtgewichtiger Workflow und unterscheidet sich von diesem maßgeblich durch eine kurze, nicht unterbrechbare und zustandslose Abarbeitung.“* [IBM02]

Microflows sind somit ein guter Kandidat für die von uns gewünschten Workflow-Systeme. Daher wurden einige der Beispiele aus Abschnitt 6.2 auf Basis von Microflows implementiert und die Performanz gemessen [HH03].

In diesem Abschnitt beschreiben wir zunächst Microflows und ihre Eigenschaften, erläutern die Ergebnisse der Zeitmessungen und nehmen eine vergleichende Bewertung vor.

6.4.1 Microflows

Die Entwicklungsumgebung von IBM's Applikations-Server WebSphere unterstützt die von uns eingesetzten Microflows. Ihre Umsetzung basiert auf Web Services, die ein Komponentenmodell für das Internet darstellen. Web Services bauen auf offenen, einfachen plattform- und sprachneutralen Standards auf und erfordern keine neuen Technologien. Funktionalität von Komponenten werden in diesem Modell in Form von *Services* angeboten, die von anderen Komponenten aufgerufen werden können.

Im Gegensatz zu Workflows kennen Microflows weder den Aspekt der Organisation noch der Infrastruktur. Somit ist ein Service nicht mehr als eine aufrufbare Software-Komponente. Ein Microflow modelliert nur die Prozesslogik, die existierende Services zu einem neuen Service zusammenfügt. Wir erläutern im Folgenden Web Services nicht näher, sondern verweisen den interessierten Leser auf [W3C02].

Microflows sehen keine Benutzerinteraktion vor, sondern der definierte Prozess läuft automatisiert ab. Daher weisen sie i. Allg. eine kurze Zeitdauer auf. Des Weiteren kennen Microflows die Zustände der Unterbrechung oder des völligen Abbruchs nicht. Ihre Abarbeitung erfolgt zustandslos, so dass keine Aussage über den Gesamtfortschritt eines Microflows gemacht werden kann. Da die Information zum aktuellen Status des Microflows nicht in einem persistenten Speicher abgelegt werden, kann auch keine transaktionale Ausführung gewährleistet werden. Stattdessen wird der erforderliche Verwaltungsaufwand zugunsten einer effizienteren Ausführung eingespart.

Wie kann man nun die Integration von Funktionen mit Microflows implementieren? Da die Microflows auf Web Services basieren, müssen zunächst alle zu integrierenden Funktionen der lokalen Systeme als Services angeboten werden. Folglich ist für die Microflows nicht mehr ersichtlich, wie die eigentlichen Schnittstellen der lokalen Systeme aussehen. Microflows rufen Aktivitäten somit nur als Services auf.

Vergleichbar zu klassischen Workflows wird im nächsten Schritt der Prozessfluss bestehend aus Kontroll- und Datenfluss modelliert. Die einzelnen Elemente eines Microflows entsprechen ungefähr denen eines Workflows und enthalten Ein- und Ausgabeknoten, Aktivitäten, Schleifen, Datenzuordnungen und Datenkontext.

Nachdem der Microflow in der Buildtime-Komponente der WebSphere-Entwicklungsumgebung definiert wurde, erfolgt anschließend die Generierung einer *SessionBean* im Applikations-Server, die den Microflow implementiert.

6.4.2 Microflow-Integrationsarchitektur

Wie bei den Architekturen zuvor wird das Microflow-System über eine Java-UDTF an das FDBS (DB2) angebunden. Diese Java-UDTF baut mittels RMI eine Verbindung zum Microflow-Controller auf, der aber einen anderen Zweck erfüllt als der Workflow-Controller. Der Microflow-Controller wird benötigt, um von der Java-Version von DB2 (Version 1.1.8) auf die Version der auf SOAP-Kommunikation basierenden Web Services (Version 1.3) zu kommen. Folglich stellt der Miroflow-Controller eine zusätzliche Schicht zur Kommunikation zwischen verschiedenen Java-Versionen dar.

Der Microflow-Controller kommuniziert mit dem Applikations-Server von IBM auf Basis von SOAP. Ein entsprechender Request startet den Microflow in der SessionBean. Nach der Ausführung des Microflows wird das Ergebnis über den Microflow-Controller wieder an die Java-UDTF zurückgeliefert.

Die zu integrierenden lokalen Funktionen werden als JavaBeans realisiert, die den Datenbankzugriff durchführen. Folglich laufen die Funktionen innerhalb des Applikations-Servers ab anstatt als eigenständige Applikationen. Dabei ist von Vorteil, dass sie in einer bereits laufenden Java Virtual Machine gestartet werden und somit auch schneller ausgeführt werden. In einem realen Szenario mit Legacy-Systemen, die integriert werden sollen, wird dies jedoch nicht möglich sein.

6.4.3 Leistungsmessungen

Damit wir die beiden Flow-Architekturen vergleichen können, wurden drei Fälle aus verschiedenen Heterogenitätsklassen implementiert. Mit ihnen soll die Leistungsfähigkeit verglichen und die grundsätzlich gleiche Mächtigkeit der beiden Ansätze demonstriert werden. Darüber hinaus ist der Verwaltungsaufwand beider Architekturen interessant. Diesen messen wir mit Hilfe eines Testfalls, bei dem die Aktivitäten eine minimale Laufzeit aufweisen.

Die implementierten Beispiele repräsentieren den einfachen Fall (föderierte Funktion `GetKompNr()`), den abhängigen (1:n)-Fall (`GetAnzahlLiefKomp()`) sowie den abhängigen iterativen Fall (`GetAlleKompNamen()`). Die Modellierung der zugehörigen Microflow-Prozesse ist vergleichbar zu den Workflow-Prozessen. Wir gehen daher nicht näher auf sie ein, sondern verweisen auf [Ste02].

Die durchgeführten Messungen beziehen sich ausschließlich auf die Zeit. Der Speicherverbrauch oder die Anzahl von E/A-Operationen wurden nicht berücksichtigt. Die lokalen, zu integrierenden Funktionen werden wie in den Messungen zuvor durch Java-Programme verkörpert, die auf eine Datenbank zugreifen. Die abgefragten Daten werden nur von den Java-Programmen gelesen, aber nicht ausgegeben, da die Ausgabezeit auf unterschiedlichen Geräten stark variieren kann.

Wie bei den Messungen zuvor laufen auch in diesem Fall alle Software-Komponenten auf einem Rechner. Zwar entspricht dies keinem realen Szenario, allerdings sind durch

den Ausschluss der Netzwerkbandbreite als limitierender Faktor die Messwerte zur Abschätzung der Leistungsobergrenze auch für den verteilten Fall geeignet.

Die implementierten Beispiele zeigen, dass die Gesamtausführungszeiten der Microflow-Lösung um den Faktor neun bis 17 kürzer sind als die des Workflow-Systems (vgl. Abbildung 6.25). Die Untersuchung der Messergebnisse ergibt, dass das WfMS besonders beim Start des Workflows und bei der Verbindung zur Datenbank durch die lokalen Funktionen Zeit verliert. In beiden Fällen muss die Java Virtual Machine (JVM) gestartet werden, was zusätzlich Zeit kostet. Dies schlägt sich besonders bei den Aktivitäten nieder, da für jede von ihnen eine eigenständige JVM gestartet wird, so dass der Datenbanktreiber immer wieder neu geladen wird. Ganz anders sieht es bei den Microflows aus. Da diese innerhalb des Applikations-Servers laufen, können sie auf eine bereits gestartete JVM zurückgreifen. Das Gleiche gilt für die Aktivitäten, die als JavaBeans implementiert wurden und somit ebenfalls keine JVM starten müssen. Außerdem konnten wir beobachten, dass die Ausführungszeit mit steigender Anzahl an unterschiedlichen Aktivitäten bei beiden Flow-Varianten linear anwächst.

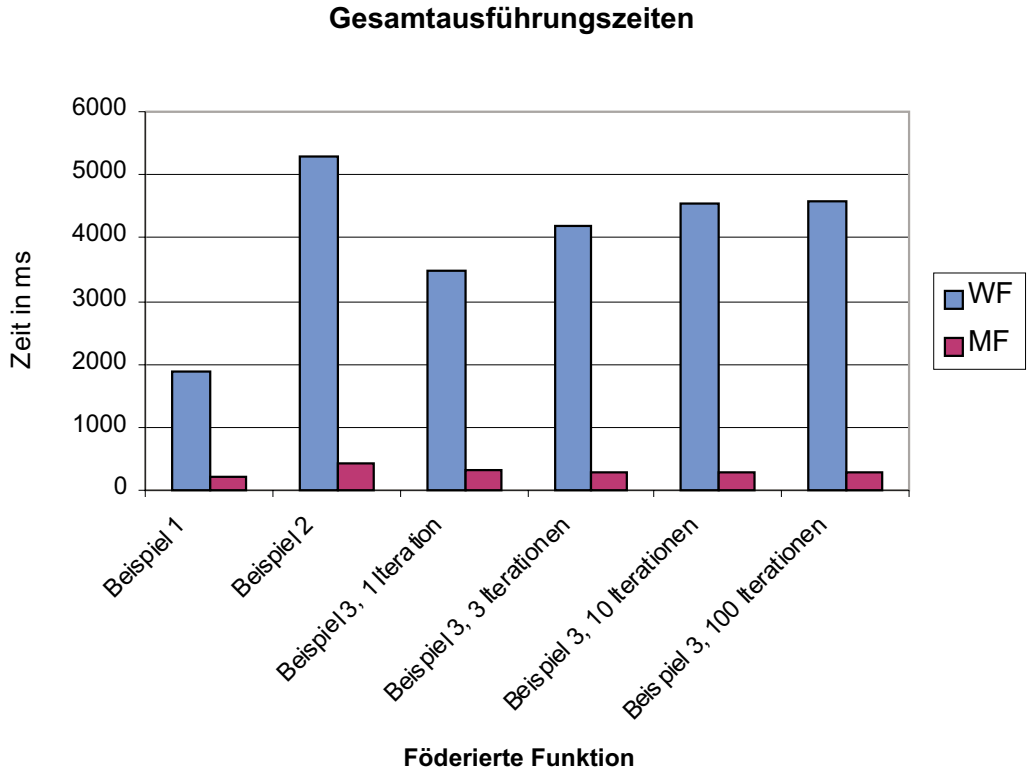

Abbildung 6.25: Gesamtausführungszeiten der Work- und Microflows im Vergleich.

Etwas anders gestaltet sich das Ergebnis, wenn die Anzahl der Ausführungen derselben Aktivität ansteigt. In diesem Fall ist beim WfMS nach wie vor ein lineares Wachstum zu beobachten, während bei den Microflows die Ausführungszeit pro Aktivität abnimmt. Dies führen wir erneut auf das einmalige Starten des Java-Interpreters zurück. Da der Datenbanktreiber nur einmal für den gesamten Testlauf geladen wird, wirkt sich dessen

Zeit bei einer höheren Wiederholungszahl geringer aus, so dass es zu einem leichten Abfallen der Ausführungszeit kommt.

Um den reinen Verwaltungsaufwand der Flow-Systeme vergleichen zu können, wurde eine Testanwendung realisiert, die einen Flow ohne Aktivität aufruft. Dieser Flow erhält eine Zeichenfolge variabler Länge als Eingabe, die unverändert zur Ausgabe übertragen wird. Mit diesem Prozess kann durch Aufrufe mit unterschiedlich langen Zeichenfolgen zusätzlich die Abhängigkeit der Verarbeitungszeit vom Datenvolumen ermittelt werden. Die Umsetzung ist jedoch nicht äquivalent möglich, da das WfMS keine Flows ohne Aktivitäten akzeptiert. Daher führen wir beim WfMS eine Pseudo-Aktivität ein, die möglichst wenig Rechenzeit in Anspruch nimmt. Die Testanwendung ruft die Flow-Systeme direkt auf. Folglich liegen nur noch zwei Schichten vor: die Anwendung und das Flow-System als WfMS bzw. Applikations-Server mit Microflows.

Damit wir die Abhängigkeit der Ausführungszeit vom Datenvolumen messen können, verwenden wir eine Zeichenfolge unterschiedlicher Größe als Parameter. Abbildung 6.26 zeigt, dass das WfMS (inkl. Pseudo-Aktivität) in diesem Fall um den Faktor sieben langsamer ist als die Microflow-Lösung. Außerdem ist erst ab einem Datenvolumen von mehr als 1000 Zeichen ein Einfluss auf die Gesamtausführungszeit zu erkennen. Diese steigt für das WfMS wesentlich schneller. Durch die relativ hohe Grundzeit eines WfMS-Aufrufes ist es ungünstig, viele Aufrufe mit kleinen Datenmengen durchzuführen. Stattdessen sollten möglichst wenig Workflow-Aufrufe mit großen Datenmengen angestrebt werden.

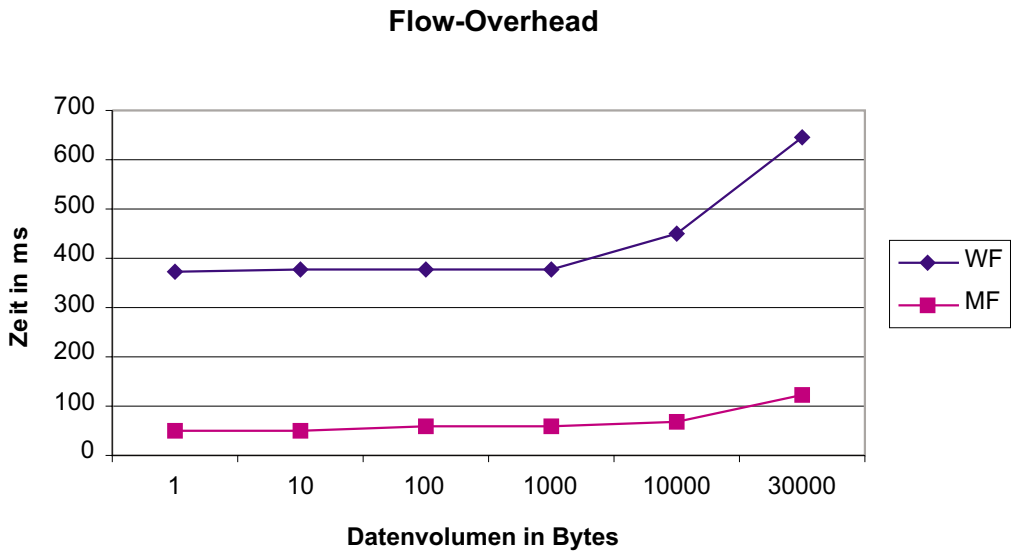

Abbildung 6.26: Messwerte des reinen Verwaltungsaufwands für unterschiedliche Datenmengen.

Fassen wir schließlich die Ergebnisse zusammen. Für die durchgeführten Messungen der Funktionsbeispiele weist die Microflow-Implementierung Gesamtausführungszeiten auf, die um den Faktor neun bis 17 kürzer sind als die der Workflow-Lösung (vgl. Abbildung 6.25). Die Microflows weisen bei fast allen Messpunkten die kürzeren Zeiten auf.

Dieser beachtliche Unterschied in der Ausführungszeit ist auf die unterschiedliche Funktionalität der beiden Flow-Varianten zurückzuführen. Während das WfMS den Prozesszustand und den Datenfluss zwischen den Aktivitäten in einem persistenten Speicher sichert, spart sich die Microflow-Umsetzung diesen Aufwand und damit auch Zeit.

Darüber hinaus hat Java als eingesetzte Programmiersprache erheblichen Einfluss. Während das WfMS immer wieder die JVM neu starten muss, laufen die Beans der Microflows in einer bereits gestarteten JVM und sparen so erneut deutlich Zeit.

Übertragen wir diese Ergebnisse auf unsere übergreifende Anfrageverarbeitung, so ist offensichtlich, dass die Microflow-Variante eine schnellere Ausführung der föderierten Funktionen und damit der ganzen Anfrageverarbeitung unterstützt. Dem steht jedoch fehlende Transaktionsunterstützung gegenüber, so dass eine verteilte Anfrageverarbeitung über Transaktionsgrenzen hinweg nicht möglich ist.

Bei der Wahl der Flow-Variante muss man daher entscheiden, welche Anforderungen die Flow-Komponente unterstützen soll. Erlaubt man nur lesenden Zugriff auf die integrierten Systeme und sollte dieser möglichst schnell sein, so kann der Einsatz der Microflow-Lösung durchaus Sinn machen. Möchte man hingegen auch schreibende Zugriffe und systemübergreifende Transaktionen unterstützen, so muss man derzeit auf die klassischen Workflow-Systeme zurückgreifen.

Stellt man die Ergebnisse der Flow-Alternativen der UDTF-Architektur gegenüber, so erweist sich die Microflow-Variante als die schnellste Implementierung. Die Ausführungszeiten der Microflows sind um den Faktor neun bis 17 kürzer als die der Workflow-Implementierung. Auch der Vergleich mit der UDTF-Architektur zeigt um den Faktor zwei bis acht kürzere Antwortzeiten. Die Ergebnisse zwischen Microflow und UDTF liegen deutlich näher beieinander. Dies ist vor allem vor dem Hintergrund interessant, dass die UDTF-Lösung (wie die Workflow-Lösung) eine Transaktionsunterstützung bereitstellt. Somit können wir folgende Einsatzszenarien ableiten:

- Sollen sehr komplexe Funktionsabbildungen möglich sein und spielt auch die Transaktionsverwaltung eine wichtige Rolle, so ist die Workflow-Architektur die beste Wahl.

- Ist die Antwortzeit wichtig und muss auch Transaktionsunterstützung gewährleistet werden, sollte man sich für die UDTF-Architektur entscheiden.

- Hat das Leistungsverhalten höchste Priorität und ist die Transaktionssicherheit aufgrund rein lesender und wiederholbarer Operationen weniger wichtig, dann können Microflows eingesetzt werden.

Da sich vor allem die Microflows momentan in einer sehr frühen Entwicklungsphase befinden und auch die beiden anderen Alternativen von den Herstellern enorm vorangetrieben und weiterentwickelt werden, gehen wir davon aus, dass sich die Lösungen hinsichtlich ihrer Funktionalität immer mehr annähern. So wird man zukünftig die Auswahl eher anhand der bestehenden Systemumgebung und den zu implementierenden Aufgaben (daten- oder funktionsorientiert) treffen.

6.5 Zusammenfassung

Unsere angestrebte Integrationsarchitektur, die ein FDBS mit einem WfMS mit Hilfe eines SQL/MED-Wrapper koppelt, ist derzeit nicht direkt umsetzbar. Dies liegt an fehlenden Produkten, die SQL/MED-Wrapper als Kopplungsmechanismus unterstützen. Daher haben wir den Einsatz von benutzerdefinierten Tabellenfunktionen untersucht. Mit Hilfe der Tabellenfunktionen können die auf Basis des WfMS realisierten föderierten Funktionen an das FDBS angebunden werden. Sie können wie Basistabellen innerhalb des FDBS in SQL-Anweisungen referenziert werden und liefern ihr Ergebnis in Tabellenform zurück. Jedoch unterstützen Tabellenfunktionen und damit diese Variante der Integrationsarchitektur nur lesenden Zugriff.

Da das WfMS als eine überdimensionierte Lösung für die Funktionsintegration eingeschätzt werden kann, haben wir weitere alternative Architekturen vorgestellt und untersucht. Diese Architekturen verzichten auf ein WfMS und setzen die Funktionsintegration mit Tabellenfunktionen um, welche direkt auf die lokalen, zu integrierenden Funktionen zugreifen.

Um die Abbildungsmächtigkeit der verschiedenen Ansätze zu untersuchen, haben wir zwei Prototypen implementiert: die Architektur auf Basis des WfMS und eine rein auf Tabellenfunktionen basierende Architektur. Der Vergleich der beiden Ansätze hat gezeigt, dass die Workflow-Lösung in der Lage ist, alle denkbaren Integrationsszenarien umzusetzen. Die Lösung auf Basis der Tabellenfunktionen hingegen hat Einschränkungen aufgezeigt.

Anschließend wurden die in Kapitel 2.2 beschriebenen Fälle von Heterogenitätsformen auf beiden Ansätzen implementiert und ihr Leistungsverhalten verglichen. Die Ausführung von föderierten Funktionen mit steigender Abbildungskomplexität hat gezeigt, dass die UDTF-Lösung um den Faktor zwei bis vier schneller ist als die Workflow-Architektur.

Da das eingesetzte WfMS eine weit größere Funktionsbandbreite aufweist als für die Funktionsintegration notwendig, haben wir nach leichtgewichtigeren Flow-Alternativen gesucht. Eine Möglichkeit sind so genannte Microflows, die zustandslos sind und keine Benutzerinteraktion unterstützen. Auf Basis dieser Microflows wurde ein weiterer Prototyp aufgesetzt. Die Leistungsmessungen beider Flow-Varianten haben gezeigt, dass die Ausführungszeiten für die Microflows um den Faktor neun bis 17 kürzer sind als die der Workflow-Lösung. Dies ist vor allem durch die fehlende persistente Speicherung des Prozesszustandes und den Datenfluss zu erklären. Überdies hat Java als eingesetzte Programmiersprache erheblichen Einfluss auf das Leistungsverhalten.

Betrachtet man den Gesamtaufwand für die Implementierung einer Funktionsintegration, so stellt das WfMS einen Ansatz dar, der einfach zu implementieren und zu nutzen ist. Überdies ist der Ansatz nicht so langsam, wie man vermuten könnte. Insbesondere die Microflows weisen ein gutes Leistungsverhalten auf, auch im Vergleich zum reinen UDTF-Ansatz. Unseres Erachtens stellt damit eine Integrationsarchitektur auf Basis eines Flow-Systems (sei es nun Work- oder Microflow) einen sinnvollen Ansatz dar. Vergleicht man alle drei Varianten, so ist die Mircoflow-Lösung die schnellste, kann im Gegenzug aber keine Transaktionssicherheit gewährleisten. Offensichtlich sind die untersuchten Architekturen für unterschiedliche Einsatzszenarien geeignet. Man kann jedoch

anhand der Anforderungen hinsichtlich Leistungsverhalten, Abbildungsmächtigkeit und Transaktionsunterstützung die Wahl der passenden Architektur ableiten.

Kapitel 7

Zusammenfassung

Der transparente und integrierte Zugriff auf alle relevanten Daten eines Unternehmens macht die Erweiterung der Datenintegration notwendig, da der Zugriff auf Datenbanken immer häufiger nur über die Schnittstellen der zugehörigen Anwendung möglich ist. Folglich muss eine Integrationslösung neben Daten auch Funktionen integrieren können. In der vorliegenden Arbeit stellten wir einen Integrationsansatz vor, der diese erweiterte Integrationsform unterstützt.

Unseren Lösungsansatz teilten wir auf in ein Beschreibungs- und ein Ausführungsmodell. Auf diese Weise kann die technische Umsetzung der Integration von deren Definition unabhängig gehalten und die Ausführungskomponente jeweils passend für ein bestimmtes Integrationsszenario gewählt werden. Für das Beschreibungsmodell entwickelten wir eine Beschreibungssprache für die Abbildung von föderierten Funktionen auf lokale Funktionen. Hierzu untersuchten wir zunächst alle auftretenden Heterogenitätsformen und entwickelten eine Abbildungssprache, welche die Überwindung dieser Heterogenitäten beschreiben kann. Eine der wichtigsten Anforderungen an die Abbildungssprache war, dass sie leicht nachzuvollziehen und „schlank“ sein sollte. Dies führte zu dem grundlegenden Konzept, die Abbildung mit Hilfe von Abhängigkeiten zwischen den lokalen und föderierten Funktionen zu beschreiben. Die Sprache wurde auf Basis von XML definiert, so dass sie zum einen auf einem Standard aufbaut und zum anderen verwandte Standards und vor allem die zugehörigen Werkzeuge genutzt werden können.

Das Ausführungsmodell ist ein weiterer wesentlicher Beitrag der Arbeit. Wir stellten unsere Integrationsarchitektur vor, die den transparenten Zugriff auf Daten und Funktionen ermöglicht. Diese Architektur ist eine Kombination aus einem FDBS und einem Workflow-System, die durch SQL/MED-Wrapper gekoppelt werden. Das WfMS unterstützt die Integration von Funktionen und stellt die resultierenden föderierten Funktionen dem FDBS zur Verfügung. Das FDBS wiederum integriert Daten und kombiniert diese mit den föderierten Funktionen. Wir untersuchten dabei verschiedene Kopplungsmechanismen für FDBS und WfMS und erläuterten, wie eine Abbildung von SQL-Anfragen auf Funktionsaufrufe und umgekehrt aussehen kann.

Zwei Aspekte der Architektur untersuchten wir im Detail: die Anfrageverarbeitung und Anfrageoptimierung sowie die Transaktionsverwaltung. Bei der Anfrageverarbeitung war die Einbindung der Funktionsaufrufe in die globalen SQL-Anfragen von besonderem Interesse. Um einen transparenten Zugriff auf Funktionen zu ermöglichen, muss die

Mächtigkeit von SQL auf Funktionsaufrufe abgebildet werden. Hierzu führten wir kompensierende Funktionsaufrufe ein, die Einschränkungen bei den SQL-Anfragen vermeiden sollen. Darüber hinaus untersuchten wir, wie zusätzliche Funktionalität im Wrapper die übergreifende Anfrageverarbeitung optimieren kann. Indem die Verarbeitung von bestimmtem Operationen in den Wrapper verlegt wird, können die Anfragen insbesondere durch geringere Mengen an zu transportierenden Daten schneller verarbeitet werden. Dieses Ergebnis verdeutlichten wir an einem entsprechenden Kostenmodell.

Schließlich entwickelten wir ein erweitertes Transaktionsmodell, das auch Systeme einbinden soll, die ursprünglich nicht dazu konzipiert wurden, in verteilten Transaktionen teilzunehmen. Dies sind vor allem die zu integrierenden Anwendungssysteme, die in den meisten Fällen keine externen Transaktionsgrenzen beachten. Wir erweiterten das Transaktionsmodell von Schaad in der Form, dass auch Anwendungssysteme eingebunden werden können. Darüber hinaus erläuterten wir, welche Fälle mit diesem Modell abgedeckt werden können und wo nach wie vor Einschränkungen bestehen.

Abschließend stellten wir alternative Architekturen vor, da der Einsatz eines WfMS überdimensioniert erscheinen kann. Hierbei wurden alternative Implementierungen der Komponente zur Funktionsintegration betrachtet und deren Abbildungsmächtigkeit bei föderierten Funktionen verglichen. Wir implementierten drei Prototypen auf Basis von Workflows, Microflows und benutzerdefinierten Tabellenfunktionen und führten Leistungsmessungen durch, deren Ergebnisse wir diskutierten und bewerteten. Diese Untersuchungen zeigten, dass die Architektur mit einem WfMS zwar eine langsamere Verarbeitung der globalen Anfragen aufweist, die Antwortzeiten an sich jedoch akzeptabel sind. Die beiden anderen Implementierungen zeigten zwar ein besseres Leistungsverhalten, wiesen aber auch Einschränkungen in der unterstützten Funktionalität auf. Anhand dieser Betrachtungen konnten wir beschreiben, welche Lösung welchen Anforderungen am besten entspricht.

7.1 Grenzen der Arbeit

Einige Aspekte wurden in der vorliegenden Arbeit nicht weiter ausgeführt. Diese Punkte lagen entweder nicht im Fokus unserer Arbeit oder es mangelte an Implementierungen von Technologien. So konnten wir die Prototypen unserer Integrationsarchitektur nicht mit dem SQL/MED-Wrapper implementieren, da es bisher keine Unterstützung seitens der Datenbankhersteller gab. Wir konnten daher insbesondere unsere Ideen für die Anfrageverarbeitung nicht anhand prototypischer Implementierungen verifizieren. Stattdessen mussten wir auf die Anbindung auf Basis von benutzerdefinierten Tabellenfunktionen ausweichen.

Das Problem mangelnder Herstellerunterstützung ändert sich inzwischen. IBM hat mit dem Information Integrator eine Integrationslösung für Mitte 2003 angekündigt, welche die Wrapper-Technologie bereitstellt [IBM03a]. Neben vorgefertigten Wrappern für die bekanntesten DBMS wie Oracle oder SQL Server wird eine Entwicklungsumgebung mit ausgeliefert, mit welcher man eigene Wrapper implementieren kann.

Bei der Anbindung der Funktionen entschieden wir uns für die vollständige Transparenz, d. h., die Funktionen sind nicht an der globalen bzw. föderierten Schnittstelle zu sehen. Unsere Betrachtungen zeigten jedoch, dass die Abbildung von SQL-Anfragen

auf Funktionsaufrufe zu einer sehr großen Zahl an kompensierenden Funktionsaufrufen führen kann. Sind die globalen SQL-Anfragen sehr komplex und weichen sie stark von der Funktionalität der angebundenen Funktionen ab, erwarten wir unzureichende Antwortzeiten. Wir haben zwei mögliche Alternativen zur Einbindung der Funktionen aufgeführt, aber nicht weiter verfolgt: ein erweiterter Parser, der die bereitgestellten Eingabewerte prüft sowie die Übergabe der notwendigen Eingabewerte durch benutzerdefinierte Funktionen in der Where-Klausel. Beide Varianten vermeiden kompensierende Funktionsaufrufe und halten somit die Anzahl der Funktionsaufrufe gering. Eine detaillierte Untersuchung dieser und anderer Alternativen könnte zu erheblich besseren Antwortzeiten führen.

Des Weiteren zeigt unser Vorschlag für eine heterogene Transaktionsverwaltung Einschränkungen hinsichtlich der zu integrierenden Anwendungssysteme. Die Platzierung des Agenten zwischen lokalem DBS und lokaler Anwendung ist unter Umständen nicht bei jedem Anwendungssystem möglich. Unsere Untersuchungen zeigten, dass es momentan keinen Lösungsansatz gibt, der für globale Transaktionen unserer Integrationsszenarien ohne Einschränkungen globale Atomarität, globale Serialisierbarkeit und globale Deadlock-Erkennung und -Vermeidung gewährleisten kann. Stimmen aus der Industrie sind der Meinung, dass die globale Serialisierbarkeit nicht wirklich notwendig ist. Die Untersuchung der Frage, welche Systeme eingebunden werden können, wenn wir auf die Serialisierbarkeit verzichten, könnte zu einem effektiveren Ansatz der Transaktionsverwaltung führen.

Schließlich entschieden wir uns für ein WfMS als die Ausführungskomponente der Funktionsintegration. Es zeigte sich, dass heutige WfMS für unsere Aufgabenstellung überdimensioniert sind, da wir nur vollständig automatisierte Workflows ohne Benutzerinteraktion benötigen. Schlankere Ansätze wie die Microflows haben andere Nachteile wie z. B. die fehlende persistente Speicherung der Workflow-Zustände. Wir sehen den Trend der Hersteller zu leichtgewichtigen Workflow-Systemen, wie wir sie benötigen. Daher sollte der Markt weiter beobachtet werden; neue WfMS-Implementierungen und -Produkte sollten getestet werden, da wir mit einer weiteren Verbesserung des Leistungsverhaltens der Integrationsarchitektur rechnen.

7.2 Ausblick

Das Thema Integration hat in den letzten Jahren vor allem mit den Technologien Enterprise Application Integration, J2EE-Applikations-Servern und Web Services erneut stark an Fahrt gewonnen. Nach den Arbeiten zum Thema Datenintegration Anfang der neunziger Jahre konzentriert man sich nun auf die Integration von Anwendungen und deren Schnittstellen [RSS+01]. Die Datenintegration ist dabei ein Teilaspekt. Die neuen Integrationsformen streben eher eine horizontale Integration und damit eine Verbindung der Systeme als eine vertikale Integration mit föderierten Schemata an. Zukünftig werden diese Technologien mit Datenintegrationsansätzen verschmelzen, damit jegliche Art von Quellsystemen miteinander verknüpft werden können. Beispielsweise könnte man ein FDBS und einen Applikations-Server für diesen Zweck kombinieren. Beide Technologien bieten eine Art Adapter zur Anbindung von Systemen an – SQL/MED-Wrapper bzw. J2EE-Konnektoren. Es sind mehrere Arten der Kombination der beiden Techno-

logien denkbar, die auch zu unterschiedlichen globalen Schnittstellen führen können. Es kommen Fragen auf wie „Ist der Applikations-Server oder das FDBS das führende System?“ oder „Können die Wrapper und Konnektoren zusammengeführt werden?“. In [Her01] werden denkbare Kombinationen vorgestellt, wie sich die beiden Technologien verschmelzen lassen.

Ein weiterer wichtiger Aspekt betrifft XML und entsprechende Schnittstellen. Vor allem Web Services entwickeln sich zu *der* Schnittstelle und Sprache zwischen Systemen. Unseres Erachtens werden Integrationslösungen zukünftig auf jeden Fall ihre globale Schnittstelle auch als Web Service anbieten müssen, um selbst wiederum für andere Systeme zugreifbar zu sein. Darüber hinaus könnte XQuery als globale Anfragesprache an Bedeutung gewinnen. Erste Prototypen hierzu gibt es bereits von den führenden Herstellern, wie z. B. Xperanto von IBM [CKS+00].

Weitere Trends, die vor allem die Hersteller betreffen, sind Sicherheitsaspekte und Werkzeuge. Werden die Systeme zunehmend vernetzt, spielen Zugriffsrechte eine immer größere Rolle. Die Benutzer sollen nur jene Informationen sehen, die sie sehen dürfen. Dies ist vor allem wichtig bei der Anwendung von Data Mining. Hier können mit Hilfe der entsprechenden Algorithmen möglicherweise Informationen ermittelt werden, die eigentlich nicht freigegeben sind.

Die hohen Investitionen der Industrie in die Umsetzung von Integrationslösungen sollten zukünftig durch bessere Unterstützung durch Werkzeuge geschützt werden. Mit Hilfe der Werkzeuge können komplexe Lösungen besser und vor allem einfacher gewartet werden und verringern damit die anfallenden Kosten bei notwendigen Anpassungen oder Erweiterungen. Heutzutage sind die Unternehmen noch stark auf eigene Implementierungen angewiesen und laufen damit Gefahr, dass mit dem Weggang von Mitarbeitern auch wichtiges Wissen über ihre Integrationsumsetzungen verloren geht.

Allgemein ist für die Thematik der Integration von Daten bzw. Anwendungssystemen noch lange kein Ende in Sicht. Angestrebte Kosteneinsparungen und das Erlangen von Wettbewerbsvorteilen zwingen die Unternehmen zu optimierten Prozessen in allen Bereichen – Entwicklung, Produktion, Vertrieb, Finanzen und Controlling. Optimierte Prozesse bedürfen vor allem der Integration der beteiligten Systeme, unabhängig welcher Art diese Systeme sind. Somit sehen wir uns einer wachsenden Zahl an verschiedenartigen Systemen gegenüber, die es zu integrieren gilt.

Anhang

Anhang A

Abbildungssprache FIX

Die DTD der Abbildungsprache FIX.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!--
     FIX DTD V1.0
     SYSTEM "http://dcx.com/fix/fix.dtd"
-->

<!-- builtin entities -->

<!ENTITY lt "&#38;#60;">
<!ENTITY gt "&#62;">
<!ENTITY amp "&#38;#38;">
<!ENTITY apos "&#39;">
<!ENTITY quot "&#34;">

<!-- Entities zur Beschreibung von Attributtypen -->

<!-- XML Schema Datentypen -->

<!ENTITY % string "CDATA">
<!ENTITY % boolean "(true | 1 | false | 0)">
<!ENTITY % float "CDATA">
<!ENTITY % uri "CDATA">
<!ENTITY % language "NMTOKEN">
<!ENTITY % Name "NMTOKEN">
<!ENTITY % QName "NMTOKEN">
<!ENTITY % NCName "NMTOKEN">
<!ENTITY % integer "CDATA">
<!ENTITY % non-negative-integer "CDATA">
<!ENTITY % positive-integer "CDATA">

<!-- weitere Datentypen -->

<!-- MIME Typ (RFC 2045) -->

<!ENTITY % contenttype "CDATA">
```

```
<!-- Semantic -->

<!ENTITY % semantic "NMTOKEN">

<!-- Coords (HTML 4.01) -->

<!ENTITY % coords "CDATA">


<!-- Entities zur Beschreibung oft benötigter Attribute -->

<!ENTITY % id
         "id ID #IMPLIED">

<!ENTITY % simplelink
         'xlink:type (simple) #FIXED "simple"
         xlink:href %uri; #REQUIRED'>


<!-- Entities zur Beschreibung bestimmter Typklassen -->

<!-- % base -->

<!ENTITY % base "(float | int | char | bool | octet | any | object | valuebase)">

<!-- % template -->

<!ENTITY % template "(sequence | string | fixed)">

<!-- % simple -->

<!ENTITY % simple "(%base; | %template; | type)">

<!-- % constr -->

<!ENTITY % constr "(struct | union | enum)">

<!-- % datatype -->

<!ENTITY % datatype "(%simple; | %constr;)">

<!-- % param -->

<!ENTITY % param "(%base; | string | type)">

<!-- % type_dcl -->

<!ENTITY % type_dcl "(typedef | struct | union | enum | native)">

<!-- % definition -->

<!ENTITY % definition "(%type_dcl; | const | except | interface |
                        module | valuetype)">
```

```
<!-- % export -->

<!ENTITY % export "(%type_dcl; | const | except | attr | op)">


<!-- Entities zur Beschreibung oft benötigten Contents -->

<!ENTITY % doc "(title, desc*)">

<!ENTITY % typedef "(%doc;, %datatype;, array*)">


<!-- Allgemeine Elemente -->

<!-- title -->

<!ELEMENT title (#PCDATA)>
<!ATTLIST title
          xlink:type (title) #FIXED "title">

<!-- desc -->

<!ELEMENT desc (#PCDATA | a)*>
<!ATTLIST desc
          xml:lang %language; #IMPLIED>

<!-- a -->

<!ELEMENT a (#PCDATA)>
<!ATTLIST a
          %simplelink;>


<!-- Systembeschreibung -->

<!-- system - Root Element -->

<!ELEMENT system (%doc;, address?, (dcom | iiop | local | oncrpc | rmi),
                  (%definition;)*)>
<!ATTLIST system
          xmlns       %uri; #FIXED "http://dcx.com/fix"
          xmlns:fix   %uri; #FIXED "http://dcx.com/fix"
          xmlns:xlink %uri; #FIXED "http://www.w3.org/1999/xlink">

<!-- address -->

<!ELEMENT address (#PCDATA)>

<!-- dcom -->

<!ELEMENT dcom EMPTY>
<!ATTLIST dcom
          uuid    %string; #REQUIRED
          version %float;  #REQUIRED>
```

```
<!-- iiop -->

<!ELEMENT iiop EMPTY>
<!ATTLIST iiop
          port %positive-integer; #REQUIRED>

<!-- local -->

<!ELEMENT local EMPTY>
<!ATTLIST local
          filename %string; #REQUIRED>

<!-- oncrpc -->

<!ELEMENT oncrpc EMPTY>
<!ATTLIST oncrpc
          program  %string;               #REQUIRED
          version  %non-negative-integer; #REQUIRED
          transport (tcp | udp)           "udp">

<!-- rmi -->

<!ELEMENT rmi EMPTY>
<!ATTLIST rmi
          uri %uri; #REQUIRED>


<!-- Schnittstellendefinition -->
<!-- Definition neuer Datentypen -->

<!-- typedef -->

<!ELEMENT typedef (%typedef;)>
<!ATTLIST typedef
          %id;>

<!-- array -->

<!ELEMENT array EMPTY>
<!ATTLIST array
          count %positive-integer; #REQUIRED>

<!-- type -->

<!ELEMENT type EMPTY>
<!ATTLIST type
          %simplelink;>



<!-- Einfache Datentypen -->

<!-- float -->

<!ELEMENT float EMPTY>
<!ATTLIST float
```

```
          type (float | double | long_double) "double">

<!-- int -->

<!ELEMENT int EMPTY>
<!ATTLIST int
          type      (short | long | long_long) "long"
          unsigned  %boolean;                  "false">

<!-- char -->

<!ELEMENT char EMPTY>
<!ATTLIST char
          wide %boolean; "false">

<!-- bool -->

<!ELEMENT bool EMPTY>

<!-- octet -->

<!ELEMENT octet EMPTY>

<!-- any -->

<!ELEMENT any EMPTY>

<!-- object -->

<!ELEMENT object EMPTY>

<!-- valuebase -->

<!ELEMENT valuebase EMPTY>


<!-- Vorlagentypen -->

<!-- sequence -->

<!ELEMENT sequence (%simple;)>
<!ATTLIST sequence
          count %positive-integer; #IMPLIED>

<!-- string -->

<!ELEMENT string EMPTY>
<!ATTLIST string
          count %positive-integer; #IMPLIED
          wide  %boolean;          "false">

<!-- fixed -->

<!ELEMENT fixed EMPTY>
<!ATTLIST fixed
          digits %positive-integer; #IMPLIED
```

```
          scale  %positive-integer; #IMPLIED>


<!-- Strukturierte Datentypen -->

<!-- struct -->

<!ELEMENT struct (%doc;, member+)>
<!ATTLIST struct
          %id;>

<!-- member -->

<!ELEMENT member (%typedef;)>
<!ATTLIST member
          %id;>

<!-- union -->

<!ELEMENT union (%doc;, (int | char | bool | enum | type), case+, default?)>
<!ATTLIST union
          %id;>

<!-- case -->

<!ELEMENT case (%typedef;, expr)>
<!ATTLIST case
          %id;>

<!-- default -->

<!ELEMENT default (%typedef;)>
<!ATTLIST default
          %id;>

<!-- enum -->

<!ELEMENT enum (%doc;, enumerator+)>
<!ATTLIST enum
          %id;>

<!-- enumerator -->

<!ELEMENT enumerator (%doc;)>
<!ATTLIST enumerator
          %id;>


<!-- native -->

<!ELEMENT native (%doc;)>
<!ATTLIST native
          %id;>
```

```
<!-- Konstanten -->

<!-- const -->

<!ELEMENT const (%doc;, (int | char | bool | float | string | fixed |
                 type | octet), expr?)>
<!ATTLIST const
          %id;>

<!-- expr -->

<!ELEMENT expr ANY>
<!ATTLIST expr
          type      %contenttype;        #REQUIRED
          xml:space (default | preserve) "default">


<!-- Operationen und Attribute -->

<!-- op -->

<!ELEMENT op (%doc;, param*, return?, context*, effect*, graph*)>
<!ATTLIST op
          %id;
          oneway   %boolean;  "false"
          semantic %semantic; "undefined">

<!-- param -->

<!ELEMENT param (%doc;, %param;)>
<!ATTLIST param
          %id;
          type (in | out | inout) "in">

<!-- return -->

<!ELEMENT return (desc*, %param;)>
<!ATTLIST return
          %id;>

<!-- except -->

<!ELEMENT except (%doc;, member*)>
<!ATTLIST except
          %id;>

<!-- context -->

<!ELEMENT context (#PCDATA)>

<!-- effect -->

<!ELEMENT effect (%doc;)>
<!ATTLIST effect
          %id;
          type (exec | read | write) "exec">
```

```
<!-- attr -->

<!ELEMENT attr (%doc;, %param;, effect*, graph*)>
<!ATTLIST attr
          %id;
          readonly %boolean; "false">


<!-- Module und Interfaces -->

<!-- module -->

<!ELEMENT module (%doc;, (%definition;)+)>

<!-- interface -->

<!ELEMENT interface (%doc;, inherits*, (%export;)*)>
<!ATTLIST interface
          %id;
          abstract %boolean; "false">

<!-- inherits -->

<!ELEMENT inherits EMPTY>
<!ATTLIST inherits
          %simplelink;>


<!-- Value Types -->

<!-- valuetype -->

<!ELEMENT valuetype (%doc;, inherits*, supports*,
                     ((%export; | statemember | factory)* | boxed))>
<!ATTLIST valuetype
          %id;
          custom      %boolean; "false"
          abstract    %boolean; "false"
          truncatable %boolean; "false">

<!-- supports -->

<!ELEMENT supports EMPTY>
<!ATTLIST supports
          %simplelink;>

<!-- boxed -->

<!ELEMENT boxed (%datatype;)>

<!-- statemember -->

<!ELEMENT statemember (%typedef;, graph*)>
<!ATTLIST statemember
          %id;
```

```
          type (public | private) "public">

<!-- factory -->

<!ELEMENT factory (%doc;, param*, graph*)>
<!ATTLIST factory
          %id;
          semantic %semantic; "undefined">


<!-- Abhängigkeitsbeschreibung -->

<!-- mapping - Root Element -->

<!ELEMENT mapping (graph)*>

<!-- graph -->

<!ELEMENT graph (desc*, ((node | todo | dep)+ | ext))>
<!ATTLIST graph
          %id;
          xlink:type (extended) #FIXED        "extended"
          xlink:role (exec | read | write |
                      xlink:external-linkset) "exec">

<!-- node -->

<!ELEMENT node (%doc;)>
<!ATTLIST node
          xlink:type  (locator)   #FIXED "locator"
          xlink:href   %uri;      #REQUIRED
          xlink:label  %Name;     #REQUIRED
          instance     %Name;     "fix:default"
          semantic     %semantic; "undefined"
          compensation %boolean;  "true">

<!-- todo -->

<!ELEMENT todo (%doc;)>
<!ATTLIST todo
          xlink:type (resource) #FIXED "resource"
          xlink:role %Name;    #REQUIRED>

<!-- dep -->

<!ELEMENT dep (%doc;)>
<!ATTLIST dep
          xlink:type    (arc) #FIXED "arc"
          xlink:from    %Name; #REQUIRED
          xlink:to      %Name; #REQUIRED
          xlink:arcrole (implementation | compensation | exception) "implementation"
          invocation    (single | cyclic) "single"
          completion    (required | sufficient | optional) "sufficient"
          priority      %integer; "0">
```

```
<!-- ext -->

<!ELEMENT ext EMPTY>
<!ATTLIST ext
          xlink:type (locator) #FIXED "locator"
          xlink:href %uri;    #REQUIRED>
```

Anhang B

XSLT Stylesheet

Das XSLT-Stylesheet zur Konvertierung der System- und Schnittstellenbeschreibung von FIX in die IDL-Synatx.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE xsl:stylesheet [

   <!ENTITY  base  "fix:float | fix:int | fix:char | fix:bool |
     fix:octet | fix:any | fix:object |
     fix:valuebase">

   <!ENTITY  template  "fix:sequence | fix:string | fix:fixed">

   <!ENTITY simple  "&base; | &template; | fix:type">

   <!ENTITY  constr  "fix:struct | fix:union | fix:enum">

   <!ENTITY  datatype  "&simple; | &constr;">

   <!ENTITY  type_dcl  "fix:typedef | fix:struct | fix:union | fix:enum | fix:native">

   <!ENTITY  param  "&base; | fix:string | fix:type">

   <!ENTITY  definition  "&type_dcl; | fix:const | fix:except |
    fix:interface | fix:module | fix:valuetype">

   <!ENTITY  export  "&type_dcl; | fix:const | fix:except | fix:attr | fix:op">

   <!ENTITY cr "<xsl:text>&#xA;</xsl:text>">
]>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:fix="http://dcx.com/fix"
 xmlns:xlink="http://www.w3.org/1999/xlink" version="1.0">

   <xsl:output method="text" encoding="iso-8859-1"
    media-type="application/x-idl"/>
   <xsl:strip-space elements="*"/>
```

```
<!-- mode="crossref" -->

<xsl:template match="*" mode="crossref">
   <xsl:apply-templates select="fix:title"/>
</xsl:template>

<!-- system -->

<xsl:template match="fix:system">
   <xsl:text>/*</xsl:text>&cr;
   <xsl:text> System: </xsl:text>
   <xsl:apply-templates select="fix:title"/>&cr;
   <xsl:text> Address: </xsl:text>
   <xsl:apply-templates select="fix:address"/>&cr;
   <xsl:text> Type: </xsl:text>
   <xsl:apply-templates
    select="fix:dcom|fix:iiop|fix:local|fix:oncrpc|fix:rmi"/>&cr;
   &cr;
   <xsl:text> THIS IS A GENERATED FILE. DO NOT EDIT!</xsl:text>&cr;
   <xsl:text> */</xsl:text>&cr;
   &cr;
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:for-each select="&definition;">
    <xsl:apply-templates select="."/><xsl:text>;</xsl:text>&cr;
   </xsl:for-each>
   &cr;
</xsl:template>

<!-- title|address -->

<xsl:template match="fix:title|fix:address">
   <xsl:value-of select="normalize-space(.)"/>
</xsl:template>

<!-- desc -->

<xsl:template match="fix:desc">
   &cr;
   <xsl:text>/* </xsl:text>
   <xsl:apply-templates/>&cr;
   <xsl:text> */</xsl:text>&cr;
</xsl:template>

<xsl:template match="fix:a">
   <xsl:apply-templates/>
   <xsl:text> (</xsl:text>
   <xsl:value-of select="@xlink:href"/>
   <xsl:text>)</xsl:text>
</xsl:template>

<!-- iiop -->

<xsl:template match="fix:iiop">
   <xsl:text>Corba</xsl:text>
```

```
</xsl:template>

<!-- dcom -->

<xsl:template match="fix:dcom">
   <xsl:text>DCOM</xsl:text>
</xsl:template>

<!-- rmi -->

<xsl:template match="fix:rmi">
   <xsl:text>RMI</xsl:text>
</xsl:template>

<!-- oncrpc -->

<xsl:template match="fix:oncrpc">
   <xsl:text>ONC RPC (Program </xsl:text>
   <xsl:value-of select="@program"/>
   <xsl:text>)</xsl:text>
</xsl:template>

<!-- local -->

<xsl:template match="fix:local">
   <xsl:text>Local (Filename </xsl:text>
   <xsl:value-of select="@filename"/>
   <xsl:text>)</xsl:text>
</xsl:template>

<!-- typedef -->

<xsl:template match="fix:typedef">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:text>typedef </xsl:text>
   <xsl:apply-templates select="&datatype;"/>
   <xsl:text> </xsl:text>
   <xsl:apply-templates select="fix:title"/>
   <xsl:apply-templates select="fix:array"/>
</xsl:template>

<!-- array -->

<xsl:template match="fix:array">
   <xsl:text>[</xsl:text>
   <xsl:value-of select="@count"/>
   <xsl:text>]</xsl:text>
</xsl:template>

<!-- const -->

<xsl:template match="fix:const">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:if test="not(fix:expr[@type='text/x-idl'
```

```
            or @type='text/plain'
            or @type='application/fix+xml'])">/* </xsl:if>
   <xsl:text>const </xsl:text>
   <xsl:apply-templates select="&datatype;"/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="fix:title"/>
   <xsl:text> = </xsl:text>
   <xsl:choose>
      <xsl:when test="fix:expr[@type='text/x-idl'
           or @type='text/plain'
        or @type='application/fix+xml']">
         <xsl:apply-templates select="fix:expr"/>
      </xsl:when>
      <xsl:otherwise>
         <xsl:text>VALUE NOT SPECIFIED OR UNPARSABLE */</xsl:text>
         <xsl:message terminate="no">
          Constant value not specified or unparsable!</xsl:message>
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>

<!-- expr -->

<xsl:template match="fix:expr">
   <xsl:choose>
      <xsl:when test="@type='application/fix+xml'">
         <xsl:apply-templates select="fix:type"/>
      </xsl:when>
      <xsl:otherwise>
         <xsl:if test="@type='text/plain'">"</xsl:if>
         <xsl:choose>
            <xsl:when test="@xml:space='preserve'">
               <xsl:value-of select="."/>
            </xsl:when>
            <xsl:otherwise>
               <xsl:value-of select="normalize-space(.)"/>
            </xsl:otherwise>
         </xsl:choose>
         <xsl:if test="@type='text/plain'">"</xsl:if>
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>

<!-- except -->

<xsl:template match="fix:except">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:text>exception </xsl:text>
   <xsl:apply-templates select="fix:title"/>
   <xsl:text> </xsl:text>&cr;
   <xsl:for-each select="fix:member">
      <xsl:apply-templates select="."/><xsl:text>;</xsl:text>&cr;
   </xsl:for-each>
   <xsl:text></xsl:text>
</xsl:template>
```

```
<!-- module -->

<xsl:template match="fix:module">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:text>module </xsl:text>
   <xsl:value-of select="fix:title"/>
   <xsl:text> </xsl:text>&cr;
   <xsl:for-each select="&definition;">
      <xsl:apply-templates select="."/><xsl:text>;</xsl:text>&cr;
   </xsl:for-each>
   &cr;
   <xsl:text></xsl:text>
</xsl:template>

<!-- interface -->

<xsl:template match="fix:interface">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:if test="@abstract='true'">
      <xsl:text>abstract </xsl:text>
   </xsl:if>
   <xsl:text>interface </xsl:text>
   <xsl:value-of select="fix:title"/>
   <xsl:text> </xsl:text>
   <xsl:if test="fix:inherits">
      <xsl:text>: </xsl:text>
      <xsl:for-each select="fix:inherits">
         <xsl:apply-templates select="."/>
         <xsl:if test="following-sibling::fix:inherits">, </xsl:if>
      </xsl:for-each>
   </xsl:if>
   <xsl:if test="&export;">
      <xsl:text> </xsl:text>&cr;
      <xsl:for-each select="&export;">
         <xsl:apply-templates select="."/><xsl:text>;</xsl:text>&cr;
      </xsl:for-each>
      &cr;
      <xsl:text></xsl:text>
   </xsl:if>
</xsl:template>

<!-- inherits|supports|type|raises -->

<xsl:template match="fix:inherits|fix:supports|fix:type|fix:raises">
   <xsl:variable name="link" select="@xlink:href"/>
   <xsl:apply-templates select="//*[@id=substring($link,2)]" mode="crossref"/>
</xsl:template>

<!-- valuetype -->

<xsl:template match="fix:valuetype">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
```

```
   <xsl:choose>
      <xsl:when test="@abstract='true'">abstract </xsl:when>
      <xsl:when test="@custom='true'">custom </xsl:when>
   </xsl:choose>
   <xsl:text>valuetype </xsl:text>
   <xsl:value-of select="fix:title"/>
   <xsl:text> </xsl:text>
   <xsl:if test="fix:inherits">
      <xsl:if test="@truncatable='true'">truncatable </xsl:if>
      <xsl:text>: </xsl:text>
      <xsl:for-each select="fix:inherits">
         <xsl:apply-templates select="."/>
         <xsl:if test="following-sibling::fix:inherits">, </xsl:if>
      </xsl:for-each>
   </xsl:if>
   <xsl:if test="fix:supports">
      <xsl:text> supports </xsl:text>
      <xsl:for-each select="fix:supports">
         <xsl:apply-templates select="."/>
         <xsl:if test="following-sibling::fix:supports">, </xsl:if>
      </xsl:for-each>
   </xsl:if>
   <xsl:choose>
      <xsl:when test="&export;|fix:statemember|fix:factory">
         <xsl:text> </xsl:text>&cr;
         <xsl:for-each select="&export;|fix:statemember|fix:factory">
            <xsl:apply-templates select="."/>
            <xsl:text>;</xsl:text>&cr;
         </xsl:for-each>
         &cr;
         <xsl:text></xsl:text>
      </xsl:when>
      <xsl:when test="fix:boxed">
         <xsl:apply-templates select="fix:boxed"/>
      </xsl:when>
   </xsl:choose>
</xsl:template>

<!-- boxed -->

<xsl:template match="fix:boxed">
   <xsl:apply-templates/>
</xsl:template>

<!-- statember -->

<xsl:template match="fix:statemember">
   <xsl:apply-templates select="fix:desc"/>
   <xsl:apply-templates select="@type"/>
   <xsl:text> </xsl:text>
   <xsl:apply-templates select="&datatype;"/>
   <xsl:text> </xsl:text>
   <xsl:apply-templates select="fix:title"/>
   <xsl:apply-templates select="fix:array"/>
</xsl:template>
```

```
<!-- factory -->

<xsl:template match="fix:factory">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:text>factory </xsl:text>
   <xsl:apply-templates select="fix:title"/>
   <xsl:text>(</xsl:text>
   <xsl:if test="fix:param">
      <xsl:for-each select="fix:param">
         <xsl:apply-templates select="."/>
         <xsl:if test="following-sibling::fix:param">, </xsl:if>
      </xsl:for-each>
   </xsl:if>
   <xsl:text>)</xsl:text>
</xsl:template>

<!-- float -->

<xsl:template match="fix:float">
   <xsl:choose>
      <xsl:when test="@type='float'">float</xsl:when>
      <xsl:when test="@type='long_double'">long double</xsl:when>
      <xsl:otherwise>double</xsl:otherwise>
   </xsl:choose>
</xsl:template>

<!-- int -->

<xsl:template match="fix:int">
   <xsl:if test="@unsigned='true'">unsigned </xsl:if>
   <xsl:choose>
      <xsl:when test="@type='short'">short</xsl:when>
      <xsl:when test="@type='long_long'">long long</xsl:when>
      <xsl:otherwise>long</xsl:otherwise>
   </xsl:choose>
</xsl:template>

<!-- char -->

<xsl:template match="fix:char">
   <xsl:if test="@wide='true'">w</xsl:if>
   <xsl:text>char</xsl:text>
</xsl:template>

<!-- bool -->

<xsl:template match="fix:bool">
   <xsl:text>boolean</xsl:text>
</xsl:template>

<!-- octet -->

<xsl:template match="fix:octet">
   <xsl:text>octet</xsl:text>
</xsl:template>
```

```
<!-- any -->

<xsl:template match="fix:any">
   <xsl:text>any</xsl:text>
</xsl:template>

<!-- object -->

<xsl:template match="fix:object">
   <xsl:text>Object</xsl:text>
</xsl:template>

<!-- valuebase -->

<xsl:template match="fix:valuebase">
   <xsl:text>ValueBase</xsl:text>
</xsl:template>

<!-- sequence -->

<xsl:template match="fix:sequence">
   <xsl:text>sequence&lt;</xsl:text>
   <xsl:apply-templates/>
   <xsl:if test="@count">
      <xsl:text>, </xsl:text>
      <xsl:value-of select="@count"/>
   </xsl:if>
   <xsl:text>&gt;</xsl:text>
</xsl:template>

<!-- string -->

<xsl:template match="fix:string">
   <xsl:if test="@wide='true'">w</xsl:if>
   <xsl:text>string</xsl:text>
   <xsl:if test="@count">
      <xsl:text>&lt;</xsl:text>
      <xsl:value-of select="@count"/>
      <xsl:text>&gt;</xsl:text>
   </xsl:if>
</xsl:template>

<!-- fixed -->

<xsl:template match="fix:fixed">
   <xsl:text>fixed&lt;</xsl:text>
   <xsl:value-of select="@digits"/>
   <xsl:text>, </xsl:text>
   <xsl:value-of select="@scale"/>
   <xsl:text>&gt;</xsl:text>
</xsl:template>

<xsl:template match="fix:const/fix:fixed">
   <xsl:text>fixed</xsl:text>
</xsl:template>
```

```
<!-- struct -->

<xsl:template match="fix:struct">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:text>struct </xsl:text>
   <xsl:apply-templates select="fix:title"/>
   <xsl:text> </xsl:text>&cr;
   <xsl:for-each select="fix:member">
      <xsl:apply-templates select="."/><xsl:text>;</xsl:text>&cr;
   </xsl:for-each>
   <xsl:text></xsl:text>
</xsl:template>

<!-- member -->

<xsl:template match="fix:member">
   <xsl:apply-templates select="fix:desc"/>
   <xsl:apply-templates select="&datatype;"/>
   <xsl:text> </xsl:text>
   <xsl:apply-templates select="fix:title"/>
   <xsl:apply-templates select="fix:array"/>
</xsl:template>

<xsl:template match="fix:union">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:text>union </xsl:text>
   <xsl:apply-templates select="fix:title"/>
   <xsl:text> switch (</xsl:text>
   <xsl:apply-templates select="fix:int|fix:char|fix:bool|fix:enum|fix:type"/>
   <xsl:text>) </xsl:text>&cr;
   <xsl:for-each select="fix:case|fix:default">
      <xsl:apply-templates select="."/><xsl:text>;</xsl:text>&cr;
   </xsl:for-each>
   <xsl:text></xsl:text>
</xsl:template>

<!-- case -->

<xsl:template match="fix:case">
   <xsl:apply-templates select="fix:desc"/>
   <xsl:text>case </xsl:text>
   <xsl:apply-templates select="fix:expr"/>
   <xsl:text> : </xsl:text>
   <xsl:apply-templates select="&datatype;"/>
   <xsl:text> </xsl:text>
   <xsl:apply-templates select="fix:title"/>
</xsl:template>

<!-- default -->

<xsl:template match="fix:default">
   <xsl:apply-templates select="fix:desc"/>
   <xsl:text>default : </xsl:text>
```

```
   <xsl:apply-templates select="&datatype;"/>
   <xsl:text> </xsl:text>
   <xsl:apply-templates select="fix:title"/>
</xsl:template>

<!-- enum -->

<xsl:template match="fix:enum">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:text>enum </xsl:text>
   <xsl:apply-templates select="fix:title"/>
   <xsl:text> </xsl:text>&cr;
   <xsl:for-each select="fix:enumerator">
      <xsl:apply-templates select="."/>
      <xsl:if test="following-sibling::fix:enumerator">,</xsl:if>&cr;
   </xsl:for-each>
   <xsl:text></xsl:text>
</xsl:template>

<!-- enumerator -->

<xsl:template match="fix:enumerator">
   <xsl:apply-templates select="fix:desc"/>
   <xsl:apply-templates select="fix:title"/>
</xsl:template>

<!-- native -->

<xsl:template match="fix:native">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:text>native </xsl:text>
   <xsl:apply-templates select="fix:title"/>
</xsl:template>

<!-- attr -->

<xsl:template match="fix:attr">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:if test="@readonly='true'">readonly </xsl:if>
   <xsl:text>attribute </xsl:text>
   <xsl:apply-templates select="&param;"/>
   <xsl:text> </xsl:text>
   <xsl:apply-templates select="fix:title"/>
</xsl:template>

<!-- op -->

<xsl:template match="fix:op">
   <xsl:apply-templates select="fix:desc"/>
   &cr;
   <xsl:if test="@oneway='true'">oneway</xsl:if>
   <xsl:choose>
      <xsl:when test="fix:return">
```

```
         <xsl:apply-templates select="fix:return"/>
      </xsl:when>
      <xsl:otherwise>void</xsl:otherwise>
   </xsl:choose>
   <xsl:text> </xsl:text>
   <xsl:apply-templates select="fix:title"/>
   <xsl:text>(</xsl:text>
   <xsl:if test="fix:param">
      <xsl:for-each select="fix:param">
         <xsl:apply-templates select="."/>
         <xsl:if test="following-sibling::fix:param">, </xsl:if>
      </xsl:for-each>
   </xsl:if>
   <xsl:text>)</xsl:text>
   <xsl:if test="fix:raises">
      <xsl:text> raises (</xsl:text>
      <xsl:for-each select="fix:raises">
         <xsl:apply-templates select="."/>
         <xsl:if test="following-sibling::fix:raises">, </xsl:if>
      </xsl:for-each>
      <xsl:text>)</xsl:text>
   </xsl:if>
   <xsl:if test="fix:context">
      <xsl:text> context (</xsl:text>
      <xsl:for-each select="fix:context">
         <xsl:apply-templates select="."/>
         <xsl:if test="following-sibling::fix:context">, </xsl:if>
      </xsl:for-each>
      <xsl:text>)</xsl:text>
   </xsl:if>
</xsl:template>

<!-- param -->

<xsl:template match="fix:param">
   <xsl:choose>
      <xsl:when test="@type='out'">out </xsl:when>
      <xsl:when test="@type='inout'">inout </xsl:when>
      <xsl:otherwise>in </xsl:otherwise>
   </xsl:choose>
   <xsl:apply-templates select="&param;"/>
   <xsl:text> </xsl:text>
   <xsl:apply-templates select="fix:title"/>
</xsl:template>

<!-- return -->

<xsl:template match="fix:return">
   <xsl:apply-templates select="&param;"/>
</xsl:template>

<!-- context -->

<xsl:template match="fix:context">
   <xsl:text>"</xsl:text>
   <xsl:value-of select="normalize-space(.)"/>
```

```
      <xsl:text>"</xsl:text>
   </xsl:template>

</xsl:stylesheet>
```

Anhang C

Standardbibliothek

Die Funktionen der Standardbibliothek.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE system SYSTEM "http://dcx.com/fix.fix.dtd">

<system xmlns="http://dcx.com/fix" xmlns:fix="http://dcx.com/fix"
 xmlns:xlink="http://www.w3.org/1999/xlink">

   <title>stdlib</title>
   <local filename="stdlib.o"/>

   <interface id="stdlib">
      <title>stdlib</title>

      <typedef id="anyseq">
         <title>anyseq</title>
         <sequence>
            <any/>
         </sequence>
      </typedef>

      <except id="EFALSE">
         <title>EFALSE</title>
      </except>

      <!-- eval -->

      <op id="eval" semantic="condition">
         <title>eval</title>
         <param id="eval.format">
            <title>format</title>
            <string/>
         </param>
         <param id="eval.value" type="inout">
            <title>value</title>
            <any/>
         </param>
         <raises xlink:href="#EFALSE"/>
```

```
</op>

<!-- evalseq -->

<op id="evalseq" semantic="condition">
   <title>evalseq</title>
   <param id="evalseq.format">
      <title>format</title>
      <string/>
   </param>
   <param id="evalseq.values" type="inout">
      <title>values</title>
      <type xlink:href="#anyseq"/>
   </param>
   <raises xlink:href="#EFALSE"/>
</op>

<!-- istrue -->

<op id="istrue" semantic="condition">
   <title>istrue</title>
   <param id="istrue.value" type="inout">
      <title>value</title>
      <bool/>
   </param>
   <raises xlink:href="#EFALSE"/>
</op>

<!-- isfalse -->

<op id="isfalse" semantic="condition">
   <title>isfalse</title>
   <param id="isfalse.value" type="inout">
      <title>value</title>
      <bool/>
   </param>
   <raises xlink:href="#EFALSE"/>
</op>

<!-- isequal -->

<op id="isequal" semantic="condition">
   <title>isequal</title>
   <param id="isequal.a" type="inout">
      <title>a</title>
      <any/>
   </param>
   <param id="isequal.b" type="inout">
      <title>b</title>
      <any/>
   </param>
   <raises xlink:href="#EFALSE"/>
</op>

<!-- isnotequal -->
```

```
<op id="isnotequal" semantic="condition">
   <title>isnotequal</title>
   <param id="isnotequal.a" type="inout">
      <title>a</title>
      <any/>
   </param>
   <param id="isnotequal.b" type="inout">
      <title>b</title>
      <any/>
   </param>
   <raises xlink:href="#EFALSE"/>
</op>

<!-- iszero -->

<op id="iszero" semantic="condition">
   <title>iszero</title>
   <param id="iszero.value" type="inout">
      <title>value</title>
      <any/>
   </param>
   <raises xlink:href="#EFALSE"/>
</op>

<!-- isnotzero -->

<op id="isnotzero" semantic="condition">
   <title>isnotzero</title>
   <param id="isnotzero.value" type="inout">
      <title>value</title>
      <any/>
   </param>
   <raises xlink:href="#EFALSE"/>
</op>

<!-- loop -->

<op id="loop" semantic="other">
   <title>loop</title>
   <param id="loop.max">
      <title>max</title>
      <int/>
   </param>
   <param id="loop.count" type="inout">
      <title>count</title>
      <int/>
   </param>
   <raises xlink:href="#EFALSE"/>
</op>

<!-- incr -->

<op id="incr">
   <title>incr</title>
   <param id="incr.value" type="inout">
      <title>value</title>
```

```
            <int/>
         </param>
      </op>

      <!-- decr -->

      <op id="decr">
         <title>decr</title>
         <param id="decr.value" type="inout">
            <title>value</title>
            <int/>
         </param>
      </op>

      <!-- str2int -->

      <op id="str2int" semantic="convert">
         <title>str2int</title>
         <param id="str2int.str">
            <title>str</title>
            <string/>
         </param>
         <return id="str2int.return">
            <int/>
         </return>
      </op>

      <!-- int2str -->

      <op id="int2str" semantic="convert">
         <title>int2str</title>
         <param id="int2str.int">
            <title>int</title>
            <int/>
         </param>
         <return id="int2str.return">
            <string/>
         </return>
      </op>

   </interface>

</system>
```

Anhang D

Messergebnisse der empirischen Untersuchungen

D.1 Messergebnisse der Workflow-Architektur

D.1.1 Gesamtdauer der Beispiele

GetCompNo()

	1	2	3	4	5	6	7	8	9	10	⊘
Neustart	13790										13790
Wiederstart	4777	4917	4807	4766	4857	4737	5408	5168	5608	5147	5019
Neustart nach UDTF	5478										13790

GetKompNr()

	1	2	3	4	5	6	7	8	9	10	⊘
Neustart	16396	15222									15809
Wiederstart	7050	7110	6730	6700	6660	6650	6770	6689	6599	6670	6763
Neustart nach UDTF	8412	7791									8102

GetLiefQualZuverl()

	1	2	3	4	5	6	7	8	9	10	⊘
Neustart	16114										16114
Wiederstart	7161	7110	7661	6700	7711	7120	7150	7230	7040	7511	7239
Neustart nach UDTF	8563										8563

GetLiefQualitaet()

	1	2	3	4	5	6	7	8	9	10	⊘
Neustart	17766										17766
Wiederstart	7060	7190	7661	7902	7290	7140	7621	7281	7291	7070	7351
Neustart nach UDTF	7701										7701

GetAnzahlLiefKomp()

	1	2	3	4	5	6	7	8	9	10	⌀
Neustart	19828										19828
Wiederstart	9023	8942	9153	8963	8963	9403	9153	9413	9133	9033	9118
Neustart nach UDTF	10015	9284									9650

GetLiefQualZuverl_2()

	1	2	3	4	5	6	7	8	9	10	⌀
Neustart	19198										19198
Wiederstart	9884	9623	9174	9253	9243	9103	9714	9083	9453	9864	9439
Neustart nach UDTF	9965										9965

GetAlleKompNamen()

	1	2	3	4	5	6
Neustart	23554					
Wiederstart	14651	14250	14310	14270	13940	13700
Neustart nach UDTF	16033	15622				

	7	8	9	10	⌀
Neustart					23554
Wiederstart	13860	14121	13930	14561	14159
Neustart nach UDTF					15828

D.1.2 Gesamtdauer von GetAlleKompNamen() bei variierender Datenbasis

GetAlleKompNamen()

	1	2	3	Durchschnitt	Differenz
2	8012	8452	8022	8162	
3	11056	11046	11036	11046	2884
4	13960	14100	14471	14177	3131
5	17385	17586	17726	17566	3389
6	21641	22613	21470	21908	4342
7	25126	24646	25016	24929	3021
8	28761	28951	28631	28781	3852
9	32187	32747	32617	32517	3736
10	35851	36433	37013	36432	3915

D.1.3 Startvorgang des Workflow-Controllers

	1	2	3	4	5	6	7	8	⌀
Neustart	14501	14341	14571	14631	14981	14922	14360	14711	14627
Wiederstart	11707	11597	11316	11737					11589

D.1.4 Dauer der Aktivitäten

GetAnzahl

	1	2	3	⊘
Wiederstart	1402	1402	1422	1409

GetKompName

	1	2	3	⊘
Wiederstart	1502	1482	1502	1495

GetKompNr

	1	2	3	⊘
Wiederstart	1422	1412	1402	1412

GetKompNrn

	1	2	3	⊘
Wiederstart	1442	1442	1472	1452

GetLiefNr

	1	2	3	⊘
Wiederstart	1432	1392	1402	1409

GetQualitaet

	1	2	3	⊘
Wiederstart	1402	1432	1422	1419

GetZuverlaessigkeit

	1	2	3	⊘
Wiederstart	1412	1422	1402	1412

Int2Long

	1	2	3	⊘
Wiederstart	1212	1202	1201	1205

D.1.5 Abschnittsmessungen der föderierten Funktion GetAnzahlLiefKomp()

Abschnitt	Dauer
Starten der UDTF	813
Ablauf der UDTF	954
RMI-Aufruf	398
Starten des Workflows und der Java-Umgebung	902
Ablauf der Aktivitäten	4663
Workflow	824
Workflow-Controller	431
RMI-Rückgabe	10
Beenden der UDTF	90
Gesamtdauer	9075

D.2 Messergebnisse der erweiterten SQL-UDTF-Architektur

D.2.1 Gesamtdauer der Beispiele

GetCompNo()

	1	2	3	4	5	6	7	8	9	10	⊘
Neustart	5188										5188
Wiederstart	2554	2223	2534	2113	2063	2093	2093	2093	2043	2113	2192
Neustart nach UDTF	2964										2964

GetKompNr

	1	2	3	4	5	6	7	8	9	10	⊘
Neustart	5217										5217
Wiederstart	2874	2347	2213	2634	2143	2133	2203	2654	2654	2053	2391
Neustart nach UDTF	3205										3205

GetLiefQualZuverl()

	1	2	3	4	5	6	7	8	9	10	⊘
Neustart	6149	6439									6294
Wiederstart	2364	2433	2453	2504	2974	2884	2644	2693	2573	2984	2651
Neustart nach UDTF	3456										3456

GetLiefQualitaet()

	1	2	3	4	5	6	7	8	9	10	⊘
Neustart	6198										6198
Wiederstart	2644	2674	2763	2684	2623	2674	2794	3054	2834	2749	2749
Neustart nach UDTF	3195										3195

GetAnzahlLiefKomp()

	1	2	3	4	5	6	7	8	9	10	⌀
Neustart	6970	6810									6890
Wiederstart	2934	2834	2824	2864	3335	3184	3255	3125	3094	2954	3040
Neustart nach UDTF	3795										9650

GetLiefQualZuverl_2()

	1	2	3	4	5	6	7	8	9	10	⌀
Neustart	7070										7070
Wiederstart	3234	2934	3145	3465	3114	3044	3354	2944	3144	2894	3128
Neustart nach UDTF	3465										3465

D.2.2 Abschnittsmessungen der föderierten Funktion GetAnzahlLiefKomp()

Abschnitt	Dauer
Starten der I-UDTF	332
Vorbereiten der 3 A-UDTFs	835
3 RMI-Aufrufe	737
3 Controller-Durchläufe	12
Ablauf der Aktivitäten	181
Beenden der 3 UDTFs	654
3 RMI-Rückgaben	31
Beenden der I-UDTF	272
Gesamtdauer	3054

Abbildungsverzeichnis

Literaturverzeichnis

[ABC+98] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. LeHors, G. Nicol, J. Robie, R. Sutor, C. Wilson und L. Wood. Document Object Model (DOM) Level 1 Specification, 01.10.1998. W3C Recommendation; zu beziehen über http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001.

[ABC+00] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. LeHors, G. Nicol, J. Robie, R. Sutor, C. Wilson und L. Wood. Document Object Model (DOM) Level 2 Core Specification, 13.11.2000. W3C Recommendation; zu beziehen über http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.

[ADD+91] R. Ahmed, P. DeSchedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii und M.-C. Shan. The Pegasus Heterogeneous Multidatabase System. *IEEE Computer*, 24(12):19–27, 1991.

[Adl01] S. Adler et al. Extensible Stylesheet Language (XSL) Version 1.0, 15.10.2001. W3C Recommendation; zu beziehen über http://www.w3.org/TR/xsl/.

[Bay82] R. Bayer et al. Dynamic Timestamp Allocation for Transactions in Database Systems, 1982. Distributed Database. North-Holland.

[BBG89] C. Beeri, P.A. Bernstein und N. Goodman. A Model for Concurrency in Nested Transaction Systems. *Journal of the ACM*, 36(2), 1989.

[BCE+02] T. Bellwood, L. Clement, D. Ehnebuske, A. Hately, M. Hondo, Y.L. Husband, K. Januszewski, S. Lee, B. McKee, J. Munter und C. von Riegen. UDDI Version 3.0, 2002. Zu beziehen über http://www.uddi.org/pubs/uddi-v3.00-published-20020719.htm.

[BE77] M.W. Blasgen und K.P. Eswaran. Storage and Access in Relational Data Bases. *IBM Systems Journal*, 16(4):363–377, 1977.

[BE96] O.A. Bukhres und A.K. Elmagarmid. *Object-Oriented Multidatabase Systems – A Solution for Advanced Applications*. Prentice Hall, 1996.

[BEK+00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H.F. Nielsen, S. Thatte und D. Winer. Simple Object Access Protocol (SOAP) 1.1, 2000. W3C Note; zu beziehen über http://www.w3.org/TR/SOAP/.

[BFHK94] R. Busse, P. Fankhauser, G. Huck und W. Klas. IRO-DB: An Object-Oriented Approach Towards Federated and Interoperable DBMS. In *Proc. of Int. Workshop on Advances in Databases and Information Systems*, Moskau, 1994.

[BHL99] T. Bray, D. Hollander und A. Layman. Namespaces in XML, 1999. W3C Recommendation; zu beziehen über http://www.w3.org/TR/REC-xml-names/.

[BLFM03] T. Berners-Lee, R. Fielding und L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax, 2003. IETF RFC 2396; zu beziehen über http://www.ietf.org/rfc/rfc2396.txt.

[BM01] P.V. Biron und A. Malhotra. XML Schema Part 2: Datatypes, 02.05.2001. W3C Recommendation; zu beziehen über http://www.w3.org/TR/2001/REC-xmlschema-2-20010502.

[BN97] P.A. Bernstein und Newcomer. *Transaction Processing.* Morgan Kauffman, 1997.

[BNPS89] E. Bertino, M. Negri, G. Pelagatti und L. Sbattella. Integration of Heterogeneous Database Applications through an Object-Oriented Interface. *Information Systems*, 14(5):407–420, 1989.

[BPE03] Business Process Execution Language for Web Services (BPEL4WS), 2003. Version 1.1; zu beziehen über http://www-106.ibm.com/developerworks/library/ws-bpel/.

[BPSM98] T. Bray, J. Paoli und C.M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0, 1998. W3C Recommendation; zu beziehen über http://www.w3.org/TR/1998/REC-xml-19980210.

[BST90] Y. Breitbart, A. Silberschatz und G.R. Thompson. Reliable Transaction Management in a Multidatabase System. *Proc. of ACM SIGMOD*, 19(2), 1990.

[BST92] Y. Breitbart, A. Silberschatz und G.R Thompson. Transaction Management Issues in a Failure-Prone Multidatabase System Environment. *The VLDB Journal*, 1(1):1–39, 1992.

[BSW88] C. Beeri, H.-J. Schek und G. Weikum. Multilevel Transaction Management: Theoretical Art or Practical Need? In *1st Int. Conf. on Extending Database Technology*, LNCS 303, Venice, 1988. Springer.

[CABG81] D. Chamberlin, M. Astrahan, M. Blasgen und J. Gray. A History and Evaluaton of System R. *Communications ACM*, 24(10):632–646, 1981.

[Cat94] R.G. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, 1994.

[CCMW01] E. Christensen, F. Curbera, G. Meredith und S. Weerawarana. Web Services Description Language (WSDL) 1.1, 2001. W3C Note; zu beziehen über http://www.w3.org/TR/wdsl.

[CD99] J. Clark und S. DeRose. XML Path Language (XPath) Version 1.0, 1999. W3C Recommendation; zu beziehen über http://www.w3.org/TR/xpath.

[Cha98] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proc. of ACM Symposium on Principles of Database Systems*, Seiten 4–43, Seattle, 1998.

[CHS91] C. Collet, M.N. Huhns und W.-M. Shen. Resource Integration Using a Large Knowledge Base in Carnot. *IEEE Computer*, 24(12):56–62, 1991.

[CIMP01] D. Carlisle, P. Ion, R. Miner und N. Poppelier. Mathematical Markup Language (MathML) Version 2.0, 2001. W3C Recommendation; zu beziehen über http://www.w3.org/TR/MathML2/.

[CKS+00] M.J. Carey, J. Kiernan, J. Shanmugasundaram, E.J. Shekita und S.N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *Proc. of 26th Int. Conf. on Very Large Data Bases*, Seiten 646–648, Brighton, U. K. 2000.

[Cla99] J. Clark. XSL Transformations (XSLT) Version 1.0, 1999. W3C Recommendation; zu beziehen über http://www.w3.org/TR/xslt.

[Con97] S. Conrad. *Föderierte Datenbanksysteme*. Springer-Verlag, Berlin Heidelberg, 1997.

[CS93] S. Chaudhuri und K. Shim. Query Optimization in the Presence of Foreign Functions. In *Proc. of 19th Int. Conf. on Very Large Data Bases*, Seiten 529–542, Dublin, 1993.

[DDJM99] S. DeRose, R. Daniel Jr. und E. Maler. XML Pointer Language (XPointer), 06.12.1999. W3C Working Draft; zu beziehen über http://www.w3.org/TR/1999/WD-xptr-19991206.

[DDK+95] A. Dogac, C. Dengi, E. Kilic, G. Ozhan, S. Nural, C. Evrendilek, U. Halici, B. Arpinar, P. Koksal, N. Kesim und S. Mancuhan. METU Interoperable Database System. *ACM SIGMOD Record*, 24(3):56–61, 1995.

[Dem80] R. Demolombe. Estimation of the Number of Tuples Satisfying a Query Expressed in Predicate Calculus Language. In *Proc. of 6th Int. Conf. on Very Large Data Bases*, Seiten 55–63, Montreal, 1980.

[DMO01] S. DeRose, E. Maler und D. Orchard. XML Linking Language (XLink), 27.06.2001. W3C Recommendation; zu beziehen über http://www.w3.org/TR/2001/REC-xlink-20010627.

[DSW94] A. Deacon, H.-J. Schek und G. Weikum. Semantics-based Multilevel Transaction Management in Federated Systems. In *Proc. of 10th Int. Conf. on Data Engineering*, Houston, Texas, 1994.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie und I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications ACM*, 19(11):624–63, 1976.

[EH88] A.K. Elmagarmid und A.A. Helel. Supporting Updates in Heterogeneous Distributed Database Systems. In *IEEE Proc. of the 4th Int. Conf. on Data Engineering*, 1988.

[EM99] A. Eisenberg und J. Melton. SQL:1999, formerly known as SQL3. *SIGMOD Record*, 28(1):131–138, March 1999.

[FB96] N. Freed und N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types, 1996. IETF RFC 2046; zu beziehen über ftp://ftp.ietf.org/rfc/rfc2046.txt.

[FCD02] FCD (Final Committee Draft) 9075-9:200x. Information Technology – Database Language – SQL – Part 9: Management of External Data (SQL/MED), March 2002. Derzeit unter Abstimmung; zu beziehen über http://sqlstandards.org/SC32/WG3/Progression_Documents/FCD/4FCD1-14-XML-2002-03.pdf.

[FLMS99] D. Florescu, A. Levy, I. Manolescu und D. Suciu. Query Optimization in the Presence of Limited Access Patterns. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Seiten 311–322, Philadelphia, 1999.

[FRT92] P.A. Franaszek, J.T. Robinson und A. Thomasian. Concurrency Control for High Contention Environments. *ACM Transactional Database Systems*, 17(2):304–345, 1992.

[GCO91] R. Gagliardi, M. Caneve und G. Oldano. An Operational Approach to the Integration of Distributed Heterogeneous Environments. In *Databases: Theory, Design, and Applications*, Seiten 110–124, 1991. Postconference publication of PARBASE-90, 1st Int. Conf. on Databases, Parallel Architectures and their Applications.

[GIG03] GIGA. GIGA Information Group, 2003. Zu beziehen über http://www.gigaweb.com.

[GMLY99] H. Garcia-Molina, W. Labio und R. Yerneni. Capability-Sensitive Query Processing on Internet Sources. In *Proc. 15th Int. Conf. on Data Engineering*, Seiten 50–59, Sidney, 1999.

[GMP97] P.B. Gibbons, Y. Matias und V. Poosala. Fast Incremental Maintenance of Approximate Histograms. In *Proc. of 23rd Int. Conf. on Very Large Data Bases*, Seiten 466–475, Athen, 1997.

[Gol90] C. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[GR93] J. Gray und A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kauffman, 1993.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Survey*, 25(2):73–170, 1993.

[HBH01] K. Hergula, G. Beck und T. Härder. Supporting Query Processing Across Application Systems – Aspects of Wrapper-Based Foreign Function Integration. In *Proc. of 3rd Int. Conf. on Enterprise Information Systems (ICEIS 2001)*, Seiten 110–115, Setubal, Portugal, Juli 2001.

[HBP94] A.R. Hurson, M.W. Bright und S.H. Pakzad. *Multidatabase Systems: An Advanced Solution for Global Information Sharing*. IEEE Computer Society Press, 1994.

[Her00] K. Hergula. A Mapping Language Based on XML for the Integration of Heterogeneous Application Systems. In *Proc. of the ProSTEP Science Days*, Seiten 241–251, Sindelfingen, September 2000. ISBN 3-8167-5585-2.

[Her01] K. Hergula. Wrapper und Konnektoren – geht die Rechnung auf? In *Konferenzband der 9. GI-Fachtagung „Datenbanksysteme in Büro, Technik und Wissenschaft“*, Seiten 461–466, 2001.

[HH99] K. Hergula und T. Härder. Eine Abbildungsbeschreibung zur Funktionsintegration in heterogenen Anwendungssystemen. In *Proc. 4. Workshop "Föderierte Datenbanken- Integration heterogener Informationsquellen*, Seiten 71–88, Berlin, November 1999.

[HH00] K. Hergula und T. Härder. A Middleware Approach for Combining Heterogeneous Data Sources – Integration of Generic Query and Predefined Function Access. In *Proc. of 1st Int. Conf. on Web Information Systems Engineering (WISE 2000)*, Seiten 22–29, Hongkong, Juni 2000.

[HH01] K. Hergula und T. Härder. How Foreign Function Integration Conquers Heterogeneous Query Processing. In *Proc. of 10th Int. Conf. on Knowledge and Data Engineering (CIKM'2001)*, Seiten 215–222, Atlanta, USA, November 2001.

[HH02a] K. Hergula und T. Härder. Coupling of FDBS and WfMS for Integrating Database and Application Systems: Architecture, Complexity, Performance. In *Proc. of 8th Int. Conf. on Extending Database Technology (EDBT 2002)*, Seiten 372–389, Prag, March 2002.

[HH02b] T. Härder und K. Hergula. Ankopplung heterogener Anwendungssysteme an Föderierte Datenbanksysteme durch Funktionsintegration. *Informatik – Forschung und Entwicklung*, 17(3):135–148, 2002.

[HH03] K. Hergula und T. Härder. Anfrageoptimierung bei der Funktionsintegration in Föderierten Datenbanksystemen. *Datenbank-Spektrum*, Seiten 33–42, 2003.

[HJK+92] M.N. Huhns, N. Jacobs, T. Ksiezyk, W.-M. Shen, M.P. Singh und P.E. Cannata. Enterprise Information Modeling and Model Integration in Carnot. In *Proc. of 1st Int. Conf. on Enterprise Integration Modeling*, Seiten 290–299, Cambridge, 1992.

[HR83] T. Härder und A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Survey*, 15(4):287–317, 1983.

[HR87] T. Härder und K. Rothermel. Concepts for Transaction Recovery in Nested Transactions. In *Proc. of ACM SIGMOD Conf.*, Seiten 239–248, 1987.

[IBM01] IBM. *SQL Reference*, 2001. DB2 UDB Version 7, S10J-8164.

[IBM02] IBM. Online-Hilfe der WebSphere Studio Application Developer Integration Edition, Version 4.1.1, 2002. IBM Corp.

[IBM03a] IBM. Information Integrator, 2003. Zu beziehen über http://www-3.ibm.com/software/data/integration/.

[IBM03b] IBM. WebSphere MQ Workflow, 2003. Zu beziehen über http://www-3.ibm.com/software/integration/wmqwf/.

[Ioa93] Y.E. Ioannidis. Universality of Serial Histograms. In *Proc. of 19th Int. Conf. on Very Large Data Bases*, Seiten 256–267, Dublin, 1993.

[ISO99] ISO/IEC 9075-2:1999. Information Technology – Database Language – SQL – Part 2: Foundation (SQL/Foundation), 1999. International Organization for Standardization.

[ISO02] ISO/IEC 9075-4:2002. Information Technology – Database Language – SQL – Part 4: Persistent Stored Modules (SQL/PSM), 2002. International Organization for Standardization.

[Jec03] Mario Jeckle. Informationen rund um XML, 2003. Zu beziehen über http://www.jeckle.de/xml/.

[JK84] M. Jarke und J. Koch. Query Optimization in Database Systems. *ACM Computing Survey*, 16(2):111–152, 1984.

[JMP02] A.D. Jhingran, N. Mattos und H. Pirahesh. Information Integration: A Research Agenda. *IBM Systems Journal*, 41(4):555–562, 2002.

[JS+02] V. Josifovski, , P.M. Schwarz, L.M. Haas und E. Lin. Garlic: A New Flavor of Federated Query Processing for DB2. In *Proc. of SIGMOD Conf.*, 2002.

[KDN91] M. Kaul, K. Drosten und E.J. Neuhold. ViewSystem: Integrating Heterogeneous Information Bases By Object-Oriented Views. In *Proc. of 7th IEEE Int. Conf. on Data Engineering*, Seiten 2–10, 1991.

[Kim95] A. Kim. *Modern Database Systems – The Object Model, Interoperability, and Beyond*. Addison-Wesley, 1995.

[KR81] H.T. Kung und J.T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactional Database Systems*, 6(2):213–226, 1981.

[KR90] B.W. Kernighan und D.M. Ritchie. *Programmieren in C*. Prentice Hall International, 1990.

[LDS94] B. Liskov, M. Day und L. Shira. Distributed Object Management in Thor. In *Proc. of Int. Workshop on Distributed Object Management*, Alberta, 1994.

[LR00] F. Leymann und D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, 2000.

[LR02] F. Leymann und D. Roller. Using Flows in Information Integration. *IBM Systems Journal*, 41(4):732–742, 2002.

[Lyn88] C. Lynch. Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values. In *Proc. of 14th Int. Conf. on Very Large Data Bases*, Seiten 240–251, Los Angeles, 1988.

[MD88] M. Muralikrishna und D. DeWitt. Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. In *Proc. of ACM SIGMOD Conf.*, Seiten 28–36, Chicago, 1988.

[Met03] MetaMatrix. MetaMatrix Server und MetaBase, 2003. Zu beziehen über http://www.metamatrix.com.

[MHG+92] F. Manola, D. Heiler, S. Georgakopoulos, M. Hornick und M. Brodie. Distributed Object Management. *International Journal on Intelligent & Cooperative Information Systems*, 1(1), 1992.

[Mic01] J.-E. Michel. Table Functions. SQL:200n change proposal, August 2001.

[Mit95] B. Mitschang. *Anfrageverarbeitung in Datenbanksystemen: Entwurfs- und Implementierungskonzepte*. Reihe Datenbanksysteme. Vieweg, 1995.

[MLK00] M. Murata, S.St. Laurent und D. Kohn. XML Media Types, 15.05.2000. Internet Draft; zu beziehen über ftp://ftp.ietf.org/internet-drafts/draft-murata-xml-04.txt.

[MMJ+01] J. Melton, J.-E. Michels, V. Josifovski, K. Kulkarni, P. Schwarz und K. Zeidenstein. SQL and Management of External Data. *SIGMOD Record*, 30(1), March 2001.

[MMJ+02] J. Melton, J.-E. Michels, V. Josifovski, K. Kulkarni und P. Schwarz. SQL/MED – A Status Report. *SIGMOD Record*, 31(3), September 2002.

[Mos85] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Report LCS TR 260. 1981. MIT Press, 1985.

[MRKN92] P. Muth, T.C. Rakow, W. Klas und E.J. Neuhold. A Transaction Model for an Open Publication Environment. In A.K. Elamagarmid, editor, *Database Transaction Models for Advanced Applications*, Seiten 159–218. Morgan Kauffman, 1992.

[MRW+93] P. Muth, T.C. Rakow, G. Weikum, P. Brössler und C. Hasse. Semantic Concurrency Control in Object-Oriented Systems. In *Proc. of Int. Conf. on Data Engineering*, 1993.

[MS03] Microsoft Corp. (Distributed) Common Object Model ((D)COM), 2003. Zu beziehen über http://www.microsoft.com/com/default.asp.

[MV93] K. Meyberg und P. Vachenauer. *Höhere Mathematik I.* Springer-Verlag, 1993.

[MY95] W. Meng und C. Yu. Query Processing in Multidatabase Systems. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, Seiten 551–572. ACM Press / Addison-Wesley Publishing, 1995.

[NW87] J.D. Noe und D.B. Wagner. Measured Performance of Time Interval Concurrency Control Techniques. In *Proc. of 13th Int. Conf. on Very Large Data Bases*, Seiten 359–367, Brighton, U. K. 1987.

[ODV94] T. Özsu, U. Dayal und P. Valduriez, editors. *Distributed Object Management*. Morgan Kauffman, 1994.

[OEC99] The Open Esprit Consortium. The OpenMath Standard, 1999. Version 0.3; zu beziehen über http://www.nag.co.uk/projects/OpenMath/omstd/.

[OMG03] Object Management Group. Common Object Request Broker Architecture, 2003. Zu beziehen über http://www.corba.org/.

[Ope97] The Open Group. DCE 1.2.2 Documentation – Full Set, 1997.

[Ora03] Oracle Corp. Oracle Transparent Gateway, 2003. Zu beziehen über http://www.oracle.com/gateways/.

[OV91] T. Özsu und P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Eaglewood Cliffs, New Jersey, 1991.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina und J. Widom. Object Exchange across Heterogeneous Information Systems. In *Proc. of 11th Int. Conf. on Data Engineering*, 1995.

[PI97] V. Poosala und Y.E. Ioannidis. Selectivity Estimation Without Attribute Value Independence Assumption. In *Proc. of 23rd Int. Conf. on Very Large Data Bases*, Seiten 486–495, Athen, 1997.

[REMC+88] M. Rusinkiewicz, R. El-Masri, B. Czejdo, D. Georgakopoulos, G. Karabatis, A. Jamoussi, K. Loa und Y. Li. OMNIBASE: Design and Implementation of a Multidatabase System. *Distributed Procesing Technical Committee Newsletter*, 10(2):20–28, 1988.

[RH98] F.F. Rezende und K. Hergula. The Heterogeneity Problem and Middleware Technology: Experiences with and Performance of Database Gateways. In *Proc. of 24th Int. Conf. on Very Large Data Bases*, Seiten 146–157, New York, 1998.

[RLJ99] D. Raggett, A. LeHors und I. Jacobs. HTML 4.01 Specification, 1999. W3C Recommendation; zu beziehen über http://www.w3.org/TR/1999/REC-html1401-19991224.

[RS97] M.T. Roth und P. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Proc. of Int. Conf. on Very Large Databases*, 1997.

[RSL78] D.J. Rosenkrantz, R. Stearns und P. Lewis. System Level Concurrency Control for Distributed Database Systems. *ACM Transactional Database Systems*, 3(2):178–198, 1978.

[RSS+01] J. Rütschlin, G. Sauter, J. Sellentin, K. Hergula und B. Mitschang. Komponenten-Middleware – Der nächste Schritt zur Interoperabilität von IT-Systemen. In A. Heuer, editor, *Tagungsband der GI-Fachtagung 'Datenbanksysteme in Büro, Technik und Wissenschaft' (BTW 2001)*, Seiten 322–331, Oldenburg, März 2001. Springer-Verlag.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie und T. Price. Access Path Selection in a Relational Database Management System. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seiten 251–260, Boston, 1979.

[SAP03] SAP. SAP R/3, 2003. Zu beziehen über http://www.mysap.com.

[Sau98] G. Sauter. *Interoperabilität von Datenbanksystemen bei struktureller Heterogenität*. Dissertation DISDBIS, Universität Kaiserslautern, 1998. infix-Verlag.

[SAX03] Simple API for XML (SAX), 2003. Zu beziehen über http://www.saxproject.org.

[Sch96a] W. Schaad. *Transaktionsverwaltung in heterogenen, föderierten Datenbanksystemen*. Dissertation DISDBIS, Eidgenössische Technische Hochschule Zürich, 1996. infix-Verlag.

[Sch96b] R. Schulte. Message Brokers: A Focussed Approach to Application Integration. Gartner Group, Strategic Analysis Report SSA R-401-102, 1996.

[SDR03] SDRC Corp. Metaphase, 2003. Zu beziehen über http://www.metaphasetech.com.

[SL90] A.P. Sheth und J.A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[SR93] A.P. Sheth und M. Rusinkiewicz. On Transactional Workflows. *IEEE Data Engineering Bulletin*, 16(2):37–40, 1993.

[Sri95a] R. Srinivasan. RPC: Remote Procedure Call Protocoll Specification Version 2, 1995. IETF RFC 1831; zu beziehen über ftp://ftp.ietf.org/rfc/rfc1831.txt.

[Sri95b] R. Srinivasan. XDR: External Data Representation Standard, 1995. IETF RFC 1832; zu beziehen über ftp://ftp.ietf.org/rfc/rfc1832.txt.

[SS94] A. Swami und B. Schiefer. On the Estimation of Join Result Sizes. In *Proc. of 4th Int. Conf. on Extending Data Base Technology*, Seiten 287–300, Cambridge, 1994.

[Ste02] E. Steiss. Vergleich von Konzepten zur Kooperation zwischen Datenbank- und Workflow-Systemen. Studienarbeit, Universität Stuttgart, 2002.

[Sun00] Sun Microsystems Inc. J2EE Connector Architecture Specification, 2000. Version 1.0, zu beziehen über http://java.sun.com/j2ee/connector/.

[Sun02] Sun Microsystems Inc. Java 2 Platform Enterprise Edition, 2002. Zu beziehen über http://java.sun.com/j2ee/.

[Sun03] Sun Microsystems Inc. Java RMI over IIOP, 2003. Zu beziehen über http://java.sun.com/products/rmi-iiop/.

[TBMM01] H.S. Thompson, D. Beech, M. Maloney und N. Mendelsohn. XML Schema Part 1: Structures, 02.05.2001. W3C Recommendation; zu beziehen über http://www.w3.org/TR/2001/REC-xmlschema-1-20010502.

[Tec03] Nimble Technology. Nimble Integration Suite, 2003. Zu beziehen über http://www.nimble.com.

[TRV96] A. Tomasic, L. Raschid und P. Valduriez. Scaling Heterogeneous Databases and the Design of DISCO. In *Proc. Int. Conf. on Distributed Computer Systems*, 1996.

[VL03] H. Varian und P. Lyman. How Much Information?, 2003. Zu beziehen über http://www.sims.berkely.edu/research/projects/how-much-info/.

[Vog99] M. Vogt. Entwurf von Mechanismen zur Integration heterogener Anwendungssysteme. Diplomarbeit, Universität Ulm, Ulm, 1999.

[VP97] V. Vassalos und Y. Papakonstantinou. Describing and Using Query Capabilities of Heterogeneous Sources. In *Proc. of 23rd Int. Conf. on Very Large Data Bases*, Athen, 1997.

[W3C02] World Wide Web Consortium. Web Services Activity, 2002. W3C Architecture Domain, zu beziehen über http://www.w3.org/2002/ws.

[W3C03] World Wide Web Consortium. http://www.w3.org, 2003.

[WDSS93] G. Weikum, A. Deacon, W. Schaad und H.-J. Scheek. Open Nested Transactions in Federated Database Systems. *IEEE Data Engineering Bulletin*, 16(2), 1993.

[Wei91] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transaction Database Systems*, 6(1):132–180, 1991.

[WfM03] Workflow Management Coalition, 2003. Zu beziehen über http://www.wfmc.org.

[Wis00] K. Wissmann. Erweiterte Konzepte zur Funktionsintegration. Diplomarbeit, Universität Ulm, Ulm, 2000.

[WS92] G. Weikum und H.-J. Schek. Concepts and Applications of Multi-level Transactions and Open Nested Transactions. In K. Elmagarmid, A. editor, *Database Transaction Models for Advanced Applicatons*, Seiten 515–553. Morgan Kauffman, 1992.

[WV90] A. Wolski und J. Veijalainen. 2PC Agent Method: Achieving Serializability and in Presence of Failures in a Heterogeneous Multidatabase. In N. Rishe, S. Navathe und D. Tal, editors, *Proc. of Int. Conf. on Database, Parallel Architecures, and Their Applications*, Seiten 321–330, 1990.

[WVZT90] K.-Y. Whang, B.T. Vander-Zanden und H.M. Taylor. A Time-Linear Probalistic Counting Algorithm for Database Applications. *ACM Trans. Database Systems*, 15(2):1208–229, 1990.

[WWC92] G. Wiederhold, P. Wegner und S. Ceri. Towards Megaprogramming. *Communications on ACM*, 35(11):89–99, 1992.

[ZE93] A. Zhang und A.K. Elmagarmid. A Theory of Global Concurrency Control in Multidatabase Systems. *VLDB Journal*, 2(3):331–360, 1993.

Index

Lebenslauf

Persönliche Angaben

Name:	Hergula
Vorname:	Klaudia
geboren am	11. Juli 1972
in	Göppingen
Familienstand:	ledig
Staatsangehörigkeit:	deutsch

Schulausbildung

1979 – 1983	Uhland-Grundschule in Göppingen
1983 – 1992	Mörike-Gymnasium in Göppingen, *Abschluss:* Abitur

Studium

1992 – 1998	Informatik mit Nebenfach Wirtschaftswissenschaften an der Universität Ulm *Thema der Diplomarbeit:* „Datenbank-Middleware-Systeme: eine vergleichende Untersuchung und Bewertung kommerzieller Werkzeuge“ *Abschluss:* Diplom-Informatikerin

Promotion

1998 – 2000	Doktorandin bei der DaimlerChrysler AG in Ulm. Wissenschaftliche Betreuung durch Prof. Dr. Dr. Theo Härder, Universität Kaiserslautern *Thema der Dissertation:* „Daten- und Funktionsintegration durch Föderierte Datenbanksysteme“
Ab 2000	berufsbegleitend

Berufliche Stationen

Okt. 2000 – Okt. 2001	Mitglied der Austauschgruppe (RTC/A) der DaimlerChrysler AG.
Nov. 2001 – Dez. 2002	Systemplanerin in der Abteilung ITI/TD der DaimlerChrysler AG, verantwortlich für die IT-Strategie in den Bereichen Datenbanken und Business Intelligence.
Ab Jan. 2003	Leiterin „Datenbanken und Business Intelligence“ in der Abteilung ITI/TD der DaimlerChrysler AG.

Sprachkenntnisee

Englisch	fließend und verhandlungssicher
Französisch	fließend
Slowenisch	als zweite Muttersprache
Latein	Grundkenntnisse

Göppingen, November 2003 Klaudia Hergula