Eine Abbildungsbeschreibung zur Funktionsintegration in heterogenen Anwendungssystemen

Klaudia Hergula

DaimlerChrysler AG
Forschung & Technologie
Abt. Prozeßkette Produktentwicklung
Postfach 2360, 89013 Ulm
e-mail: klaudia.hergula@DaimlerChrysler.com

Theo Härder

Universität Kaiserslautern
Fachbereich Informatik
AG Datenbanken und Informationssysteme
Postfach 3049, 67653 Kaiserslautern
e-mail: haerder@informatik.uni-kl.de

Kurzfassung: Unter Verwendung der Sprache XML wird ein Ansatz zur Abbildungsbeschreibung erarbeitet, die als Teil einer Integration von Anwendungssystemen die Funktionsabbildung mittels Abhängigkeiten beschreibt. Hierzu werden zunächst die Daten- und Funktionsintegration gegenübergestellt und anschließend eine Integrationsarchitektur vorgestellt. Es wird eine Klassifikation der API als operationale Schnittstelle entworfen, um darauf aufbauend eine generische Methodik zur Abbildungsbeschreibung vorzustellen.

Schlagworte: Funktionsintegration, API-Integration, Heterogenität, XML.

1. Einleitung

In den meisten Unternehmen liegt heutzutage eine heterogene Systemlandschaft vor. Um den Lebenszyklus eines Produktes vollständig abzudecken, werden unterschiedliche Hardware-Komponenten, Netzwerk- und Betriebssysteme, Datenbanksysteme (DBS) und Anwendungsprogramme zum Einsatz gebracht. Zur Überwindung der dabei entstehenden Heterogenität der Systeme gibt es vor allem im Bereich der heterogenen Datenbanken 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 und Multidatenbanksysteme 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 Punkte noch offen sind [SL90, BE96, Ki95, HBP94, Sa98, Da96, Co98].

Das Bild der heterogenen Datenbanklandschaft beginnt sich jedoch zu ändern. Haben sich die Unternehmen bisher bewußt für ein Datenbanksystem 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 Datenbanksystem und die zugehörigen Anwendungsprogramme 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 [SAP98] oder PDM (Produktdatenmanagement)-Systeme wie Metaphase [SDRC98] oder ENOVIAVPM [ENV98]. Im Falle von SAP können demnach die darin verwalteten Daten nicht direkt mittels SQL aus der relationalen Datenbank ausgelesen werden. Statt dessen stellt SAP sog. BAPIs (Business APIs) zur Verfügung, die dem Benutzer den Datenzugang über vordefinierte Funktionen erlaubt. Neben diesen kommerziellen Produkten gibt es aber auch häufig proprietäre, also 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) sehr häufig im Anwendungsprogramm realisiert und nicht durch das Datenbanksystem 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 die Funktionalität) nur über ein API gestattet.

Bei der DaimlerChrysler AG, Sparte PKW, werden beispielsweise Daten zu einem Auto u.a. in einem Geometrie-Informationssystem (GIS, Verwaltung der eigentlichen Geometrie), in einem Stammdatenverwaltungssystem[1] (SDVS) und in einem Dokumentverwaltungssystem (DVS, Verwaltung von 2D-Zeichnungen und sonstigen Dokumenten) gespeichert. Als Datenbanksysteme werden IMS und DB2 auf MVS-Host und Oracle auf UNIX eingesetzt. Alle Systeme sind jeweils nur über ein API zugänglich. Deshalb besteht die Notwendigkeit, nicht nur die Datenbank- bzw. Schema-Integration, sondern in Ergänzung dazu auch die Anwendungssystem- bzw. API-Integration zu unterstützen.

Während DBS generische Systeme sind, die sich durch Datenunabhängigkeit, Zugriffsflexibilität und Schemaevolution auszeichnen, sind Anwendungssysteme in der Regel auf eine vorgegebene Funktionalität zugeschnitten. Die Grundlage der Flexibilität bei der Datenintegration ist ein hoher Grad an Datenunabhängigkeit, der sich durch die dominierenden Eigenschaften der DB-Anfragesprache (deklarativ, mengenorientiert, nicht-prozedural) ergibt. Da, wie oben gezeigt, bei der Anwendungsintegration neben der Daten- auch Funktionsintegration (und vielfältige Mischformen) eingesetzt werden sollten, muß ein Lösungsansatz gefunden werden, der die beiden Integrationsformen nicht getrennt betrachtet, sondern eine Kombination der beiden Konzepte ermöglicht. Da für die Datenintegration bereits einige Lösungsansätze vorliegen, konzentrieren wir uns in dieser Arbeit zunächst auf die Integration von Funktionen bzw. APIs. Hierfür erarbeiten wir einen Lösungsansatz zur Beschreibung der Abbildung von Funktionen. So werden in Kapitel 2 zunächst die Unterschiede zwischen Daten- und Funktionsintegration herausgearbeitet, um im dritten Kapitel eine Architektur zur Daten-und Funktionsintegration vorzustellen. In Kapitel 4 soll das API als operationale Schnittstelle beschrieben werden, wobei eine Klassifikation der auftretenden Heterogenitätsformen aufgestellt wird. Basierend auf dieser Klassifikation wird in Kapitel 5 ein Ansatz vorgestellt, der unter der Verwendung von XML eine Beschreibungssprache für die Abbildung globaler Funktionen auf lokale Quellfunktionen definiert. Anschließend soll dieser Ansatz kurz zu anderen Lösungsvorschlägen abgegrenzt werden. Zum Abschluß werden die bisherigen Ergebnisse in Kapitel 7 zusammengefaßt und ein Ausblick auf zukünftige Arbeiten gegeben.

2. Daten- und Funktionsintegration

In diesem Kapitel soll herausgearbeitet werden, inwiefern sich die Datenintegration von der Funktionsintegration unterscheidet. Dazu wird zunächst in knapper Form die Datenintegration beschrieben, um anschließend die Funktionsintegration zu betrachten.

2.1 Datenintegration

Auf dem Gebiet der Datenintegration wurden bereits einige Ansätze erarbeitet, die als Föderierte Datenbanksysteme (FDBS) bekannt sind. Ausgehend von einem Szenario, in welchem es mehrere Datenbanken mit den darauf arbeitenden Anwendungsprogrammen gibt, möchte man einen einheitlichen Zugriff auf alle Datenbanken erreichen, ohne die Autonomie der zu integrierenden Datenbanksysteme einzuschränken. Alle Ansätze der Datenintegration gehen davon aus, daß die Schemata, d. h. die Objektspezifikation, die Integritätsbedingungen usw., der zu integrierenden Datenquellen bekannt sind. Ist dies nicht der Fall, so ist eine manuelle Überarbeitung oder auch Ergänzung der strukturellen und semanti-

1.) Die direkt einem Teil zugeordneten Daten, wie z.B. Identifikation, Größe, Gewicht usw., werden als Stammdaten bezeichnet.

schen Datenspezifikationen erforderlich (Reengineering). Ziel der Integration, auch bei Föderierten Datenbanksystemen, ist die Bereitstellung eines globalen Schemas, das zusammen mit einer geeigneten Anfragesprache die DB-Anwendungsprogrammierschnittstelle bildet. Diese Anfragesprache sollte (abhängig vom Datenmodell) deskriptiv, mengenorientiert und nicht-prozedural sein, weil diese Eigenschaften einen hohen Grad an Datenunabhängigkeit und damit eine große Flexibilität bei der Schemaevolution gewährleisten. Sie verkörpern eine wichtige Voraussetzung für generische Zugriffsmöglichkeiten und erlauben damit die Einführung von Ad-hoc-Anfragen.

Ein allgemein anerkanntes Modell für die Integration von Daten wird in [SL90] definiert. In der dort beschriebenen Fünf-Schichten-Architektur werden zunächst die lokalen Schemata, die auf verschiedenen Datenmodellen basieren können, in ein einheitliches Datenmodell transformiert. Aus den daraus resultierenden Komponentenschemata, die nun von der Heterogenität der zugrundeliegenden Datenbanken abstrahieren, wird ein föderiertes Schema gebildet. Dies ermöglicht globalen Anwendungen einen transparenten Zugriff auf die integrierten Datenbanken. Dieser Ansatz und andere Arbeiten zur Schemaintegration [SL90, BE96, Ki95, HBP94, Sa98, Da96, Co98] vernachlässigen jedoch die Integration von Funktionen.

2.2 Funktionsintegration

Während DBS generische Systeme sind, die sich durch Datenunabhängigkeit, Zugriffsflexibilität und Schemaevolution auszeichnen, sind Anwendungssysteme in der Regel auf eine vorgegebene Funktionalität zugeschnitten. Dies bedeutet, daß keine lokalen, strukturell objektorientierten Schemata [At+89] der zu integrierenden Systeme verfügbar sind. Außer den nach außen offengelegten Funktionen sind keine weiteren Informationen über die Daten vorhanden. Änderungen der Funktionalität rufen meist erhebliche Programmänderungen hervor. Der Einsatz neuer objektorientierter Konzepte und Techniken verspricht hier jedoch tiefgreifende Verbesserungen, da vor allem durch die Modularität und Kapselung eine einfachere Erweiterung und Wiederbenutzbarkeit der Funktionalität erreicht werden kann. Da die Funktionen nicht die Datenunabhängigkeit bieten, die eine DB-Anfragesprache auszeichnet, ist auch die Flexibilität der Funktionen eingeschränkt, d. h., Ad-hoc-Anfragen sind hier nicht möglich.

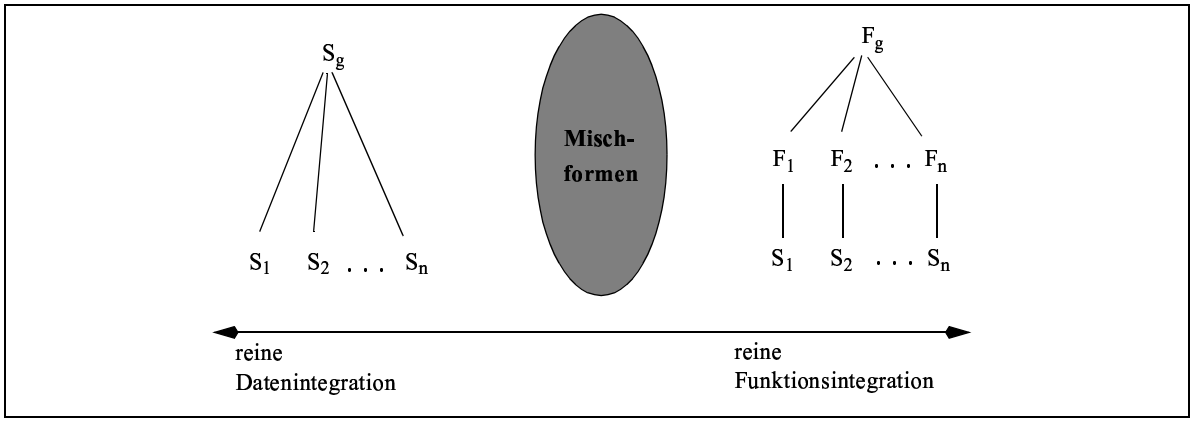

Abbildung 1: Spektrum der Integrationsformen.

Abbildung 1 verdeutlicht das Spektrum möglicher Integrationsformen, an dessen Enden die beiden Extreme Datenintegration und Funktionsintegration stehen. Links ist die reine Datenintegration dargestellt, in der die bekannten, lokalen Schemata zu einem globalen Schema integriert werden. Da diese Integrationsform generativ und evolutionär ist, kann eine generische und dadurch flexible Anfrage-

sprache zur Verfügung gestellt werden. Am anderen Ende findet sich die Funktionsintegration. Wie bereits erwähnt, sind die lokalen Schemata nicht bekannt, statt dessen werden für jedes Quellsystem eine Menge von vordefinierten Funktionen zur Verfügung gestellt. Aufbauend auf diesen Quellfunktionen können globale Funktionen definiert werden. Die globalen Funktionen können auch als virtuelle Funktionen oder Templates betrachtet werden, da sie selbst keine Funktionalität implementieren, sondern nur eine Hülle bilden, die bereits vorhandene Funktionen aufruft. Aus diesem Grund findet man auf der globalen Ebene nicht die Flexibilität der Datenintegration vor, da die Daten nur in der vordefinierten Art und Weise zugegriffen werden können.

2.3 Kombination von Daten- und Funktionsintegration

Sollen nun Datenbanksysteme sowie Anwendungssysteme in einer Architektur integriert werden, so ist die reine Datenintegration nicht mächtig genug. Statt dessen ist ein Lösungsansatz wünschenswert, der zwischen den aufgezeigten Extremen liegt und somit eine Mischform von Daten- und Funktionsintegration darstellt. Die angestrebte globale Schnittstelle soll dann dem Benutzer Daten, Funktionen und auch eine Anfragesprache zur Verfügung stellen. Im folgenden Kapitel wird eine Integrationsarchitektur vorgestellt, die einen ersten Schritt in diese Richtung macht.

3. Integrationsarchitektur

In diesem Abschnitt stellen wir eine Architektur zur Integration von Daten als auch Funktionen vor, um aufzuzeigen, wie die in dieser Arbeit vorgestellte Beschreibungssprache in diese Architektur einzuordnen ist. Es handelt sich dabei um eine Drei-Schichten-Systemarchitektur (*3-tier architecture*), die aus folgenden Komponenten besteht:

- Die unterste Schicht und damit die Server bilden die existierenden Systeme, welche die zu integrierende Funktionalität zur Verfügung stellen. Zu diesen Systemen gehören DBS, die über eine SQL-Schnittstelle verfügen, sowie die in der Einleitung beschriebenen Anwendungssysteme, die den Zugriff auf ihre Daten nur über eine API erlauben.
- Über der Server-Schicht liegt die Middleware, die eine Integrationsschicht darstellt und von uns als Integrations-Server bezeichnet wird. Hier werden die Anfragen der Clients gemäß der Informationen über die Verteilung der Daten und Funktionen zu den relevanten Systemen weitergeleitet. Des weiteren werden die heterogenen Datenformate und Funktionsaufrufe der Quellen in ein einheitliches API transformiert bzw. übersetzt.
- Die oberste Schicht, der Client, stellt eine graphische Oberfläche zur Verfügung, die als sog. *Thin Client* realisiert wird. Sie ist vollständig in Java implementiert und gewährleistet somit Plattformunabhängigkeit und Portabilität der Client-Programme. Die graphische Oberfläche basiert auf einer einheitlichen API.

Im folgenden konzentrieren wir uns auf den Integrations-Server. Wir wollen dabei aufzeigen, warum eine Beschreibungssprache für die Funktionsabbildung entwickelt wird und welche Rolle sie in dieser Architektur spielt. Momentan erfolgt die Integration der Funktionen so, daß im Client eine Schnittstelle für den Aufruf von Funktionen existiert (sozusagen ein Stub). Diese Schnittstelle greift auf die im Integrations-Server angebotenen Funktionen (globale API) zu, die jedoch nicht selbst die Funktionalitäten implementieren, sondern die eigentliche Integration der Quellfunktionen realisieren. Im einfachsten Fall bedeutet dies, daß die globale und lokale Funktion völlig übereinstimmen und somit die globale Funktion im Integrations-Server lediglich die lokale Funktion aufrufen muß. Dieses Vorgehen hat jedoch den entscheidenden Nachteil, daß die erfolgte Abbildung nicht nach außen hin sichtbar ist. Möchte man

genau wissen, wie und aus welchen lokalen Funktionen sich eine globale Funktion zusammensetzt, so ist man gezwungen, den Programmcode zu analysieren. Diese Situation soll mit der in dieser Arbeit vorgestellten Beschreibungssprache verbessert werden.

Ziel ist es, die Funktionsabbildung mittels einer Beschreibungssprache zu spezifizieren, um anschließend aus dieser vorgegebenen Spezifikation zumindest die Funktionsrümpfe im Integrations-Server zu generieren, die der Entwickler gegebenenfalls vervollständigen muß. In diesem Vorgehen sehen wir mehrere Vorteile:

- Die Informationen über die erfolgte Funktionsabbildung sind nicht mehr im Programmcode versteckt, sondern direkt über die Spezifikation zugänglich.
- Des weiteren kann der Entwicklungsaufwand reduziert werden, wenn Teile der globalen Funktionen im Integrations-Server generiert werden.
- Ein weiterer wichtiger Aspekt ist die Tatsache, daß die Integrationsarchitektur auf diese Weise sehr flexibel bleibt, wenn sich das Integrationsumfeld verändert. Werden nämlich zu einem späteren Zeitpunkt weitere Systeme integriert, so sind die Spezifikationen anzupassen und die neuen Teile inkrementell zu generieren.

In dieser Arbeit soll nun genauer analysiert werden, wie sich eine Funktionsintegration realisieren läßt. Daher wird in den nächsten Kapiteln der erste Schritt zu einer allgemeinen Methodik zur Funktionsintegration erarbeitet. Dabei konzentrieren wir uns auf die Entwicklung einer Beschreibungssprache, welche die Abbildung von Funktionen darzustellen erlaubt, so daß diese Abbildungsspezifikation als Eingabe zur Generierung der globalen Funktionen herangezogen werden kann. Bevor der Ansatz für eine solche Beschreibungssprache vorgestellt wird, entwickeln wir eine Klassifikation der auftretenden Heterogenitäten, wenn Funktionen bei einer operationalen Schnittstelle herangezogen werden.

4. Klassifikation der zu überwindenden Heterogenitäten

In diesem Abschnitt soll analysiert werden, welche Formen der Heterogenität bei einer Funktionsintegration auftreten können. Dabei ist der Fokus auf die Funktionen selbst gerichtet, d. h., es werden die Einflüsse unterschiedlicher Beschreibungsmodelle für die Signaturen, unterschiedliche Modellierungen desselben Verhaltens oder auch sich widersprechende oder fehlende Daten untersucht. Nicht betrachtet werden Heterogenitäten, die aufgrund unterschiedlicher Plattformen hervorgerufen werden, sowie Laufzeitunterstützung im Sinne von Transaktionsverwaltung oder Fehlerbehandlung. Die vorgestellte Klassifikation orientiert sich an der in [HST99] aufgezeigten Gliederung und paßt diese für den Bereich der Funktionsintegration an. So lassen sich auch im Bereich der Funktionen zwei prinzipielle Formen der Heterogenität unterscheiden, nämlich die semantische und die strukturelle Heterogenität. Semantische Heterogenität liegt vor, wenn das Verständnis über die Bedeutung und Interpretation von gleichen oder zusammengehörigen Funktionalitäten sowie über die beabsichtigte Verwendung dieser Funktionalitäten nicht übereinstimmt. Die Schwierigkeit liegt darin, daß in den meisten Fällen lediglich die Signaturen der Funktionen verfügbar sind, diese aber nicht ausreichend beschreiben, welche Funktionalität auf welchen Daten von den Funktionen angeboten werden. Häufig sind auch die Dokumentationen nicht sehr hilfreich, so daß es zu fehlerhaften Interpretationen kommen kann. Wir wollen die Diskussion dieser Heterogenitätsform hier nicht weiter vertiefen, sondern verweisen den interessierten Leser auf [HST99] und [Sa98].

Strukturelle Heterogenität wird durch unterschiedliche Arten der Repräsentation der Funktionen verursacht. Einfache Beispiele dafür sind Funktionen, die in verschiedenen Programmiersprachen implementiert sind und deren Funktionssignaturen auf unterschiedliche Art und Weise dargestellt werden. Dies wiederum bedeutet, daß die Datentypen der Funktionsparameter verschieden sein können und einer

Konvertierung unterzogen werden müssen. Im folgenden stellen wir eine Klassifikation der Faktoren auf, die strukturelle Heterogenität hervorrufen können, und geben parallel dazu eine Einschätzung der notwendigen Anforderungen zur Überwindung der Heterogenität. In den Ausführungen beschreiben die dick ausgezeichneten Abschnitte auf der linken Hälfte der Seite den **kongruenten Fall** (vollständige Überdeckung), während auf der rechten Seitenhälfte das Spektrum der möglichen Abweichungen, die bei der Funktionsintegration auftreten können (inkongruente Fälle), aufgeführt ist. Die zusätzlichen Anforderungen treten dabei auf, wenn die klassifizierten Abweichungen existieren.

- **Es existiert ein formal beschriebenes Schema der Quellfunktionen, d. h., es liegen die Funktionssignaturen sowie die durch die Funktionen bearbeiteten Datenobjekte und Beziehungen vor.**

  Das lokale Schema ist nicht bekannt, d. h., es sind nur die Funktionssignaturen verfügbar.

  Anforderung:
  Es muß Anwenderwissen herangezogen werden, um beispielsweise Integritätsbedingungen identifizieren zu können.

- **Die Quellfunktionen sind in derselben Programmiersprache implementiert und dementsprechend sind die Signaturen der Funktionen in einer gemeinsamen Sprache beschrieben.**

  Die Signaturen sind in verschiedenen Sprachen beschrieben, wie z.B. in IDL, als Java- oder C-Schnittstellen.

  Anforderung:
  Eine einheitliche Beschreibung der Schnittstellen (vergleichbar zu IDL) sowie ein Repository, das Meta-Informationen zu den lokalen APIs zur Verfügung stellt, können die Integration vereinfachen, da die lokalen Funktionsaufrufe über Wrapper in einer Sprache erfolgen können.

- **Die Zielfunktionen implementieren keine eigene Funktionalität.**

  Die Zielfunktionen rufen nicht nur Quellfunktionen auf, sondern verarbeiten die ausgelesenen Daten weiter.

  Anforderung:
  Es muß eine „vollständige“ Programmiersprache eingesetzt werden, um die benötigte Funktionalität zu implementieren. Die eigene Implementierung bedeutet natürlich einen höheren Aufwand als das Einbinden eines bereits vorhandenen Systems, das die geforderte Funktionalität anbietet.

- **Die Zielfunktionen werden in einem 1:1-Verhältnis auf die Quellfunktionen abgebildet.**

  Die Zielfunktionen werden in einem 1:n-, n:1- oder n:m-Verhältnis abgebildet.

  Anforderung:
  Die Zusammenhänge der einzelnen Funktionen müssen geklärt werden. Dies ist einfacher realisierbar, wenn strukturierte Schemata der Quellfunktionen vorliegen.

- **Es erfolgt ausschließlich lesender Zugriff, d. h., Auswirkungen auf andere beteiligte Systeme müssen nicht betrachtet werden.**

  Es wird lesender als auch schreibender Zugriff unterstützt.

Anforderung:
Für den schreibenden Fall müssen Abhängigkeiten zu anderen Systemen beachtet werden. Zudem ist die Unterstützung durch eine Transaktionsverwaltung notwendig, um die beteiligten Systeme konsistent zu halten. Durch diesen Faktor nimmt die Komplexität des Falls deutlich zu.

- **Die Programmiersprache ist homogen und nicht objektorientiert, d. h., es werden keine Objekte mit Methoden als Parameter übergeben.** Die Programmiersprachen der Funktionen sind verschieden und können auch objektorientiert sein.

  Anforderung:
  Die Übergabe von Objekten als Parameter muß unterstützt werden. Außerdem wird eine Middleware benötigt, die auch sonstige Heterogenitäten zwischen Programmiersprachen überbrükken kann (z. B. CORBA). Reichen kommerzielle Middleware-Lösungen nicht aus, so müssen die Heterogenitäten von Hand mit entsprechenden Funktionen im Zielsystem aufgelöst werden.

- **Quell- und Zielfunktionen bearbeiten den gleichen Ausschnitt der realen Welt.** Die Anwendungsbereiche der Systeme überlappen, d. h., Teile der Quellfunktionen werden bei den Zielfunktionen nicht betrachtet oder umgekehrt.

  Anforderung:
  Mechanismen zur Projektion bzw. semantischen Anreicherung werden benötigt.

- **Die Datenverteilung von Quell- und Zielfunktionen stimmt überein.** Die Datenverteilung stimmt nicht exakt überein. Dies trifft vor allem dann zu, wenn Daten in den Zielfunktionen weiterverarbeitet werden und damit neue Werte entstehen.

  Anforderung:
  Mechanismen zur Selektion bzw. Datenanreicherung werden benötigt, die vom Zielsystem unterstützt werden müssen.

- **Die Namen der Funktionen als auch Parameter stimmen überein, wenn die Funktionen in ihrer Funktionalität und die übergegebenen Datenobjekte ebenfalls übereinstimmen.** Die Namen der Funktionen oder Parameter von ansonsten kongruenten Funktionen stimmen nicht überein.

  Anforderung:
  Es werden Mechanismen zur Umbenennung benötigt. Die übereinstimmenden Komponenten können jedoch nicht automatisch ermittelt werden, ihre Identifikation erfordert vielmehr Anwendungswissen.

- **Für die Parameter werden identische Datentypen verwendet.** Übereinstimmende Parameter basieren auf unterschiedlichen Datentypen.

  Anforderung:
  Hier sind Mechanismen zur Konvertierung notwendig, die vom Zielsystem unterstützt werden müssen.

- **Die Parameter der Funktionen stehen in einem 1:1-Verhältnis, d. h., jeder Parameter einer Zielfunktion kann genau einem der Quellfunktion zugeordnet werden.** Die Parameter stehen in einem 1:n- oder n:1-Verhältnis.

  Anforderung:
  Es müssen Mechanismen zur Verfügung gestellt werden, die das Aufspalten oder Zusammenfassen von Parametern unterstützen.

- **Die Quellfunktionen sind voneinander unabhängig, d. h., sie können parallel ausgeführt werden.** Die Quellfunktionen sind voneinander abhängig.

  Anforderung:
  Die Quellfunktionen müssen in einer bestimmten Reihenfolge ausgeführt werden. Dazu muß eine Ausführungsreihenfolge ermittelt werden.

- **Die Zielfunktionen sind voneinander unabhängig, d. h., sie können parallel ausgeführt werden.** Zwischen den Zielfunktionen bestehen Abhängigkeiten.

  Anforderung:
  Die Zielfunktionen müssen in einer bestimmten Reihenfolge ausgeführt werden.

Ausgehend von der vorgestellten Klassifikation soll im nächsten Kapitel ein Lösungsansatz vorgestellt werden, der den ersten Schritt zu einer generischen Methodik zur Funktionsintegration erarbeitet, indem eine Beschreibungssprache zur Funktionsabbildung entwickelt wird.

5. Lösungsansatz einer Abbildungssprache

In diesem Kapitel wird ein Ansatz für eine Abbildungssprache zur Funktionsintegration beschrieben. Dazu wird zunächst der Begriff der Funktion geklärt und eine Notation für die Beschreibung der Funktionssignaturen und der Funktionsparameter festgelegt. Anschließend wird ein Lösungsansatz vorgestellt, der die Abbildung mittels Abhängigkeiten zu beschreiben versucht. Da ein deskriptiver Ansatz gewählt wurde, wird zur Beschreibung der Abbildung XML herangezogen, dessen Einsatz anhand eines Beispiels verdeutlicht werden soll. Der Ansatz versucht, einige der in Kapitel 4 vorgestellten Abweichungen der Funktionen zu überwinden, die am Ende dieses Kapitels aufgeführt werden. Die resultierende Abbildungsbeschreibung wird dann für die Generierung der in Kapitel 3 vorgestellten Komponenten der Integrationsarchitektur herangezogen.

5.1 Der Begriff der Funktion

Aus mathematischer Sicht wird eine Funktion wie folgt definiert: Sind D und C Mengen, so ist eine Funktion f von D nach C eine Untermenge des kartesischen Produkts von D und C, wobei die Funktion für *jedes* Element in D *eindeutig* definiert ist. Funktionen in der Programmierung wurden zwar von der mathematischen Definition abgeleitet, ihre Handhabung ist jedoch weniger streng. So ist es durchaus möglich, daß eine Funktion beim wiederholten Aufruf mit denselben Parameterwerten unterschiedliche Ergebnisse liefert und auch den Zustand des Systems verändern, also Seiteneffekte haben kann. Die bekanntesten Seiteneffekte sind hierbei Zuweisungen z.B. an globale Variablen (in diesem Sinne ist die DB eine globale Variable) sowie Ein- und Ausgaben. Eine Variante der Funktion ist die Prozedur, die

keinen Wert als Ergebnis liefert, sondern nur Seiteneffekte aufweist. Des weiteren besteht eine Funktion aus einer Signatur und einem Rumpf. Die Signatur beinhaltet die folgenden Informationen:

- Den Namen der Funktion,
- die Parameterbezeichnungen,
- die Datentypen der Parameter und
- bei Funktionen den Datentyp des Rückgabewertes.

Der Rumpf enthält den Programmcode zur Implementierung der Funktionalität. Für die Funktionsintegration ist dieser Teil der Funktion nicht weiter von Bedeutung, da dessen Inhalt nicht bekannt ist und somit als Black Box betrachtet werden kann. Eine wichtige Rolle spielt dagegen die Signatur einer Funktion, da sie die benötigten Informationen zum Aufruf einer Funktion enthält.

Von nun an bedienen wir uns folgender Notation bei der Betrachtung von Funktionen:

$$f_i(e_1, ..., e_n, a_1, ..., a_m)$$

f_i bezeichnet dabei den Funktionsnamen, $e_1, ..., e_n$ stehen für die Eingabeparameter und $a_1, ..., a_m$ für die Ausgabeparameter. Auch der Rückgabewert einer Funktion wird als Ausgabeparameter dargestellt, um Prozeduren und Funktionen in derselben Weise beschreiben zu können. Im folgenden werden wir nicht mehr zwischen Funktionen und Prozeduren unterscheiden, sondern sprechen nur noch von Funktionen. Unsere Aussagen gelten dann für beide Formen. Gibt es Unterschiede, so wird explizit darauf hingewiesen. Soll ein bestimmter Parameter einer ausgewählten Funktion benannt werden, so geschieht dies mit $f_i e_j$. Außerdem werden Ziel- und Quellfunktionen durch Groß- bzw. Kleinschreibung der Funktionsnamen unterschieden, so wird beispielsweise die Zielfunktion F_1 auf die Quellfunktionen f_1 und f_2 abgebildet.

5.2 Lösungsansatz

Bei der Funktionsintegration sollen n globale Funktionen auf m lokale Funktionen abgebildet werden. Es soll zunächst angenommen werden, daß die globalen Funktionen keine eigene Funktionalität implementieren, sondern nur auf der angebotenen Funktionalität der Quellfunktionen aufbauen. Die Beschreibung der Abbildung kann für jede globale Funktion getrennt betrachtet werden, da die Einzelabbildungen später wieder zusammengefaßt werden können. Geht man nun davon aus, daß eine Zielfunktion auf eine oder mehrere Quellfunktionen abgebildet wird, so müssen

- Eingabe- und Ausgabeparameter zwischen Ziel- und Quellsystem und
- möglicherweise Ausgabeparameter von Quellfunktionen auf Eingabeparameter anderer Quellfunktionen

abgebildet werden. Zudem muß bei mehreren Quellfunktionen eine Ablauffolge festgelegt werden, falls eine parallele Ausführung aufgrund bestehender Abhängigkeiten nicht möglich ist. Abbildung 2 veranschaulicht, daß demnach bei der Beschreibung drei unterschiedliche Abbildungsrichtungen sowie eine Ablauffolge zu definieren sind.

Versuche, diese Abbildungen in einer SQL-ähnlichen Sprache festzulegen, haben gezeigt, daß eine einheitliche Beschreibung für die unterschiedlichen Richtungen nicht möglich ist [Vo99]. Wird überdies noch eine Ausführungsreihenfolge festgelegt, so gestaltet sich die Abbildung als sehr komplex und unübersichtlich. Aus diesem Grund soll die Abbildung nun aus dem Blickwinkel der Parameter betrachtet werden. Dabei gehen wir davon aus, daß der Benutzer beim Aufruf einer Zielfunktion die Ausgabeparameter als Ergebnis erhalten möchte. Hier stellt sich nun die Frage, woher man diese Werte bekommt, d. h., von welchen anderen Werten und/oder (Zwischen-)Ergebnissen, die wiederum von Funktionen bestimmt werden, sind diese Werte abhängig. Die dabei entstehende Kette muß bei den vorgegebenen Eingabeparametern der Zielfunktion enden. Einige der Abhängigkeiten stehen dabei von

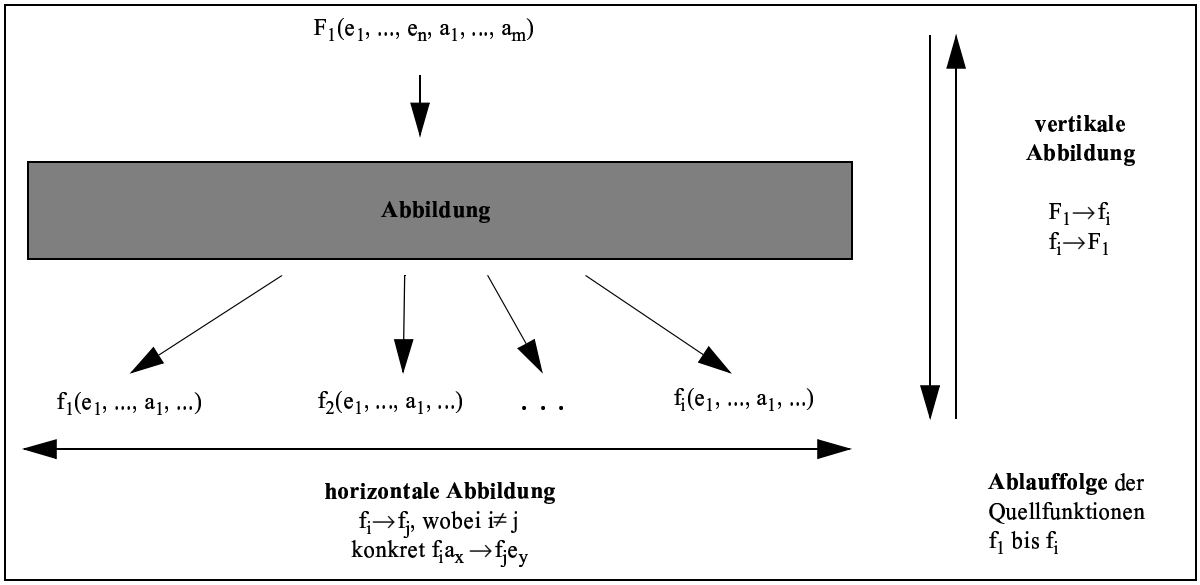

Abbildung 2: Vertikale und horizontale Abbildungen.

vornherein fest:

1. Die Ausgabeparameter der Zielfunktion hängen immer von den Ausgabeparametern der Quellfunktion(en) ab.
2. Die Ausgabeparameter einer Quellfunktion hängen immer von ihren Eingabeparametern ab.
3. Die Eingabeparameter einer Quellfunktion hängen entweder von den globalen Eingabeparametern oder von Ausgabeparametern anderer Quellfunktionen ab.
4. Die Werte der Eingabeparameter sind vorgegeben.

Dies bedeutet, daß die Abbildung vollständig mittels Abhängigkeiten beschrieben werden kann. Abbildung 3 soll dies verdeutlichen.

.

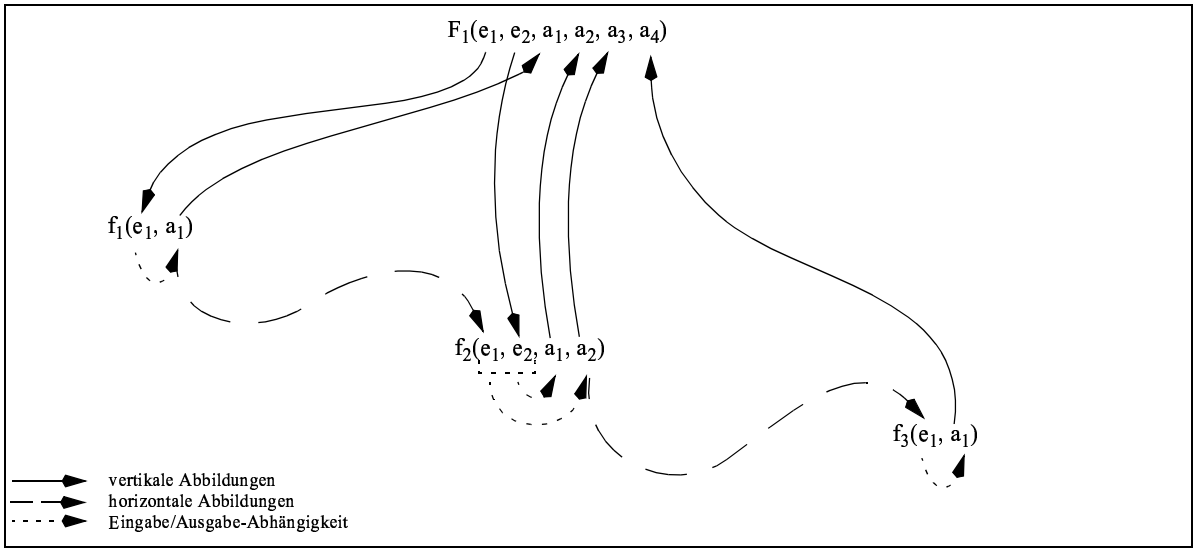

Abbildung 3: Abhängigkeiten der Parameter.

In dem gezeigten Beispiel soll die globale Funktion F_1 auf drei Quellfunktionen abgebildet werden. Die Pfeile zeigen die Abhängigkeiten der Parameter, wobei die Pfeilrichtung den Einfluß darstellt. f_3a_1 beeinflußt F_1a_4, d. h., F_1a_4 ist abhängig von f_3a_1. Betrachtet man nun die Parameter losgelöst von den Funktionen, so erhält man mit den beschriebenen Abhängigkeiten einen gerichteten azyklischen Graphen, in welchem die Parameter die Knoten und die Abhängigkeiten die Kanten darstellen. Bis zu diesem Punkt fehlt jedoch noch die Definition der Ausführungsfolge der Quellfunktionen. Bei der vorgestellten Vorgehensweise ist dies aber nicht explizit notwendig, da man auf die Graphentheorie zurückgreifen kann. Führt man nämlich auf dem gerichteten azyklischen Graphen eine topologische Sortierung durch, so 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 Funktion zwingend nach einer anderen Funktion ausgeführt werden muß. Somit ist es gelungen, die drei aufgeführten Abbildungsrichtungen sowie die Ablauffolge der Quellfunktionen in kohärenter Weise darzustellen.

Mit diesem Ansatz wurde bisher allerdings nur der Fall abgedeckt, in welchem die beiden Systeme bezüglich Parameterbezeichnung und -datentyp kongruent sind. Es fehlen noch die Abbildungsinformationen, die zur Überwindung der Heterogenitäten benötigt werden, wie z. B. für das Konvertieren von Datentypen oder Konkatenieren von Strings. Solche Operationen werden als weitere Funktionen betrachtet und können somit in die Abbildungsbeschreibung eingebettet werden. Auf diese Weise bleibt die Darstellungsform der Abbildung weiterhin homogen. Abbildung 4 erweitert das in Abbildung 3 dargestellte Szenario um zusätzliche Funktionen, welche die Parameter in irgendeiner Form verarbeiten.

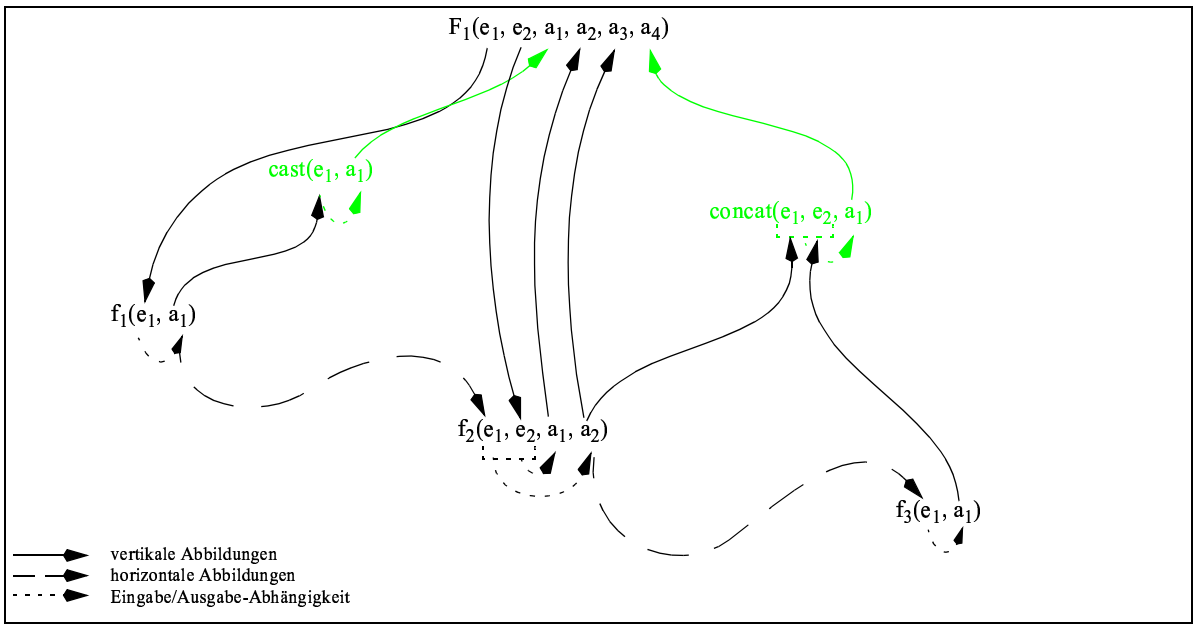

Abbildung 4: Funktionsabbildung mit zusätzlichen Funktionen zur Weiterverarbeitung der Parameter.

In dem bisher vorgestellten Ansatz sehen wir folgende Vorteile:

1. Alle Abbildungsrichtungen werden in derselben Art und Weise beschrieben (homogenes Modell).
2. Die Ablauffolge der Aufrufe der Quellfunktionen muß nicht explizit festgelegt werden, sondern kann anhand der beschriebenen Abhängigkeiten mittels einer topologischen Sortierung ermittelt werden.

3. Müssen Parameter bearbeitet werden, so können diese Operationen als weitere Funktion(en) in die Beschreibung eingebunden werden, d. h., die homogene Darstellung bleibt erhalten.
4. Es können auch Abhängigkeiten zu anderen Daten wie globalen Variablen beschrieben werden.
5. Ein Teil der Abbildung kann auf Basis der Signaturen automatisch generiert werden, z. B. die Abhängigkeiten der Ausgabeparameter der Quellfunktionen von ihren Eingabeparametern.
6. Der Lösungsansatz basiert auf der Graphentheorie und kann auf deren Erkenntnisse zurückgreifen.

Als Nachteil erweist sich bei diesem Ansatz, daß die Beschreibung sehr dokumentationsintensiv ausfällt. Daher wäre ein graphisches Werkzeug sehr hilfreich, das dem Benutzer erlaubt, die Abhängigkeiten mittels Pfeilen zu beschreiben, woraus der eigentliche Abbildungstext generiert wird.

Nachdem bisher die grundlegende Idee aufgezeigt wurde, die Abbildung mittels Abhängigkeiten der Parameter darzustellen, soll im nächsten Abschnitt eine konkrete Form der Abbildungsbeschreibung vorgestellt werden.

5.3 XML als Beschreibungssprache

Für die Beschreibung der Abbildung wurde die Sprache XML (Extensible Markup Language, [W3C98a]) gewählt, da sie ein Format aufweist, das von Menschen wie auch Rechnern einfach gelesen und manipuliert werden kann. XML stellt eine standardisierte Syntax zur Verfügung, um Informationen darzustellen und ermöglicht dem Benutzer überdies, in einer DTD (Document Type Definition) eigene Strukturformen zu definieren. Mittels XML können wir demnach zunächst festlegen, wie die Beschreibung der Funktionsabbildung aussehen soll und anschließend auch die Abbildung selbst beschreiben. Die Sprache XML soll hier nicht näher erläutert werden. Statt dessen verweisen wir den interessierten Leser auf die einschlägige Literatur zu diesem Thema [GP98, BM98, SC99].

Bevor nun die Beschreibung in XML erfolgen kann, muß zuvor eine geeignete DTD erarbeitet werden, d. h., welche Abbildungsinformationen sollen wie dargestellt werden. Zunächst werden die Systeme selbst beschrieben. Dazu gehören ein eindeutiger Bezeichner für jedes Quell- und Zielsystem, die Programmiersprache der API sowie Informationen zum Kommunikationsprotokoll. Werden für die Funktionsparameter komplexere Datentypen verwendet, so müssen auch diese vorher beschrieben werden, was hier aber nicht weiter ausgeführt wird. Dazu gehören unter anderem strukturierte Datentypen oder auch geschachtelte Aggregationstypen. Anschließend erfolgt die Beschreibung der Funktionen, d. h. deren Name sowie die Parameterbezeichnungen und -datentypen. Außerdem muß festgehalten werden, welche Parameter zur Ein- bzw. Ausgabe dienen. Bis zu diesem Punkt könnte dann eine DTD folgendermaßen aussehen:

```
<!ELEMENT    system_desc (sys_name, description?, prog_language, communication, function+)>
<!ATTLIST    system_desc
             id                 ID                          #REQUIRED
             type               (source|target|helper)      #IMPLIED>

<!ELEMENT    sys_name           (#PCDATA)>
<!ELEMENT    description        (#PCDATA)>
<!ATTLIST    description
             type               (system|function)           #REQUIRED
             language           (de|en|fr|...)              #REQUIRED>

<!ELEMENT    prog_language      (#PCDATA)>
<!ELEMENT    communication      (os, ip, port)>
<!ELEMENT    os                 (#PCDATA)>
<!ELEMENT    ip                 (#PCDATA)>
<!ELEMENT    port               (#PCDATA)>

<!ELEMENT    function           (func_name, description?, parameter+)>
<!ATTLIST    function
             id                 ID                          #REQUIRED>
<!ELEMENT    func_name          (#PCDATA)>
<!ELEMENT    parameter          (para_name|datatype)>
<!ATTLIST    parameter
             id                 ID                          #REQUIRED
```

```
              type                (in|out|inout|return)      #REQUIRED>
<!ELEMENT     para_name           (#PCDATA)>
<!ELEMENT     datatype            (#PCDATA)>
```

Die hier vorgestellte DTD beschreibt ein System durch den Namen des Systems (`sys_name`), die verwendete Programmiersprache der API (`prog_language`), Daten zum Kommunikationsprotokoll (`communication`) sowie den angebotenen Funktionen (`function`). Überdies kann eine textuelle Dokumentation des Systems (`description`) hinzugefügt werden, dessen Sprache über das Attribut `language` angegeben wird. Das Element `system_desc` enthält zwei Attribute, die einen eindeutigen Bezeichner (`id`) sowie eine optionale Typisierung des Systems als Ziel-, Quell- oder Hilfsfunktion. Das Element `communication` ist momentan auf das Netzwerkprotokoll TCP/IP ausgerichtet und untergliedert sich in die Komponenten des vorliegenden Betriebssystems (`os`), der IP-Adresse (`ip`) und dem zugehörigen Port (`port`). Die Funktionen des Systems werden durch ihre Signatur beschrieben, d. h. durch den Funktionsnamen (`func_name`) sowie den Parametern (`parameter`). Wie beim System kann auch bei der Funktion eine textuelle Dokumentation hinzugefügt werden. Daher enthält das Element `description` ein weiteres Attribut `type`, das angibt, ob die Dokumentation ein System oder eine Funktion beschreibt. Auch jede Funktion bekommt über ein Attribut einen eindeutigen Bezeichner zugeordnet (`id`). Jeder Parameter besteht aus dessen Namen sowie dem Datentyp. Um festzulegen, ob es sich um einen Rückgabewert, Eingabe- oder Ausgabeparameter oder auch beides handelt, kann das Attribut `type` mit den Werten `return`, `in`, `out` oder `inout` belegt werden. Des weiteren wird den Funktionen ebenfalls ein eindeutiger Bezeichner (`id`) zugeordnet.

Nachdem die beteiligten Systeme beschrieben wurden, erfolgt nun die eigentliche Abbildung. Hierbei müssen die Abhängigkeiten der einzelnen Parameter in geeigneter Art und Weise dargestellt werden. In unserem Ansatz ziehen wir hierfür den aktuellen Entwurf für die Verknüpfungssprache XLink (XML Linking Language, [W3C99a]) heran. Dieser Verknüpfungsmechanismus erweitert das bereits allgemein bekannte Link-Konzept von HTML. In dieser Arbeit soll jedoch nicht die ganze Funktionalität von XLink erläutert werden, sondern nur die für die Abbildung relevanten Ausschnitte. Wir bedienen uns dabei der sog. erweiterten Verknüpfung (extended link). Dieser Mechanismus erlaubt es, eine beliebige Anzahl von Ressourcen zu verknüpfen, ohne daß diese ausgehende Verknüpfungen enthalten. Dies bedeutet in unserem Fall, daß die in den Systembeschreibungen aufgeführten Parameter verknüpft werden können, ohne daß dies explizit in der Systembeschreibung enthalten ist. Eine erweiterte Verknüpfung besteht aus sog. Locators und Arcs, wobei Locators die an einer Verknüpfung beteiligten Quellen definieren und Arcs das Traversierungsverhalten festlegen.

Sollen nun in der Abbildung die Abhängigkeiten der Parameter beschrieben werden, so geschieht dies mit der erweiterten Verknüpfung, indem die beteiligten Parameter als Locators und die Abhängigkeiten als Arcs dargestellt werden. Die DTD für die Abbildungsbeschreibung gestaltet sich wie folgt:

```
<!ELEMENT map       (node+, dependency+)>
<!ATTLIST map
      xmlns:xlink   CDATA                           #FIXED       "http://www.w3.org/XML/XLINK/0.9"
      xlink:type    (simple|extended|locator|arc)   #FIXED       "extended">

<!ELEMENT node      EMPTY>
<!ATTLIST node
      xmlns:xlink   CDATA                           #FIXED       "http://www.w3.org/XML/XLINK/0.9"
      xlink:type    (simple|extended|locator|arc)   #FIXED       "locator"
      id            ID                              #REQUIRED
      xlink:href    IDREF                           #REQUIRED>

<!ELEMENT dependency EMPTY>
<!ATTLIST node
      xmlns:xlink   CDATA                           #FIXED       "http://www.w3.org/XML/XLINK/0.9"
      xlink:type    (simple|extended|locator|arc)   #FIXED       "arc"
      from          IDREF                           #REQUIRED
      to            IDREF                           #REQUIRED>
```

Eine Abbildung `map` besteht demnach aus Knoten (`node`) und Abhängigkeiten (`dependency`) und wird über ihre Attribute als erweiterte Verknüpfung definiert. Mit dem Element `node` wird über eine Verknüpfung auf diejenigen Parameter verwiesen, die im Abhängigkeitsgraphen enthalten sind. Der dabei angewandte Mechanismus ist die Verknüpfungssprache XPointer (XML Pointer Language, [W3C99b]), die es

ermöglicht, interne Strukturen eines XML-Dokuments zu adressieren. Mit Hilfe von XPointer werden die Parameter identifiziert, um sie als Locators der erweiterten Verknüpfung darzustellen. Des weiteren wird jedem Knoten eine eindeutige ID zugeordnet, die bei der Beschreibung der Abhängigkeiten benötigt wird. Diese werden mit dem Element `dependency` dargestellt, das als Arc fungiert. Mit den Attributen `from` und `to` wird definiert, welches die beiden Endpunkte der Abhängigkeit sind.

Nun fehlen noch die Operationen, die benötigt werden, um die Parameter zu verarbeiten, d. h. beispielsweise Datentypen zu konvertieren oder Parameter zu konkatenieren. Für jede dieser Operationen muß vom Zielsystem eine Funktion zur Verfügung gestellt werden, die ebenfalls in einer separaten Systembeschreibung aufgeführt wird. Der vorgestellte Ansatz soll im nächsten Abschnitt anhand eines Beispiels verdeutlicht werden.

5.4 Ein Beispiel

Das hier gezeigte Beispiel beschreibt die in Abbildung 4 dargestellte Funktionsabbildung in XML. Dazu werden zunächst das Ziel- und Quellsystem sowie die Funktionen beschrieben, wobei in diesem Beispiel nur ein Quellsystem vorliegen soll. Es werden nicht alle Funktionen ausführlich beschrieben, da die anderen in der gleichen Art und Weise aufzuführen sind. Das XML-Dokument für das Zielsystem sieht dann wie folgt aus:

```
<?xml version="1.0"?>
<!DOCTYPE system_desc SYSTEM "system_desc.dtd">
<system id="Z" type="target">
    <sys_name> Zielsystem </sys_name>
    <description type="system" language="de">
        Dies ist das Zielsystem, das eine globale Funktion zur Verfügung stellt.
    </description>
    <prog_language> C++ </prog_language>
    <communication>
        <os> HP-UX </os>
        <ip> 555.555.555.555 </ip>
        <port> 7000 </port>
    </communication>
    <function id="F1">
        <func_name> F1 </func_name>
        <description type="function" language="de">
            Diese Funktion besteht aus zwei Eingabe- und vier Ausgabeparametern.
        </description>
        <parameter id="F1e1">
            <para_name> e1 </para_name>
            <datatype> string </datatype>
        </parameter>
        <parameter id="F1e2">
            <para_name> e2 </para_name>
            <datatype> real </datatype>
        </parameter>
        <parameter id="F1a1">
            <para_name> a1 </para_name>
            <datatype> string </datatype>
        </parameter>
        . . .
    </function>
</system>
```

Das Quellsystem ist folgendermaßen beschrieben:

```
<?xml version="1.0"?>
<!DOCTYPE system_desc SYSTEM "system_desc.dtd">
<system id="Q" type="source">
    <sys_name> Quellsystem </sys_name>
    <description type="system" language="de">
        Dies ist das Quellsystem, das drei lokale Funktionen zur Verfügung stellt.
    </description>
    <prog_language> Java </prog_language>
    <communication>
        <os> AIX </os>
        <ip> 444.444.444.444 </ip>
        <port> 7100 </port>
    </communication>
```

```
    <function id="f1">
        <func_name> f1 </func_name>
        <description type="function" language="de">
            Diese Funktion besteht aus einem Eingabe- und einem Ausgabeparameter.
        </description>
        <parameter id="f1e1">
            <para_name> e1 </para_name>
            <datatype> string </datatype>
        </parameter>
        <parameter id="f1a1">
            <para_name> a1 </para_name>
            <datatype> real </datatype>
        </parameter>
    </function>
    . . .
</system>
```

In derselben Form werden auch die zusätzlichen Funktionen (`cast` und `concat`) für die Bearbeitung der Parameter beschrieben. In unserem Beispiel gehen wir des weiteren davon aus, daß keine komplexen Datentypen eingesetzt werden, so daß diese nicht explizit beschrieben werden müssen. Nun folgt die eigentliche Beschreibung der Abhängigkeiten. Hierzu müssen zunächst alle benötigten Parameter als Locators und anschließend alle Abhängigkeiten als Arcs bestimmt werden:

```
<?xml version="1.0"?>
<!DOCTYPE map SYSTEM "map.dtd">
<map>
<!-- Parameter der Zielfunktion -->
    <node id="F1e1" href="c:\data\Zielsystem.xml#id("F1e1")/self::parameter"/>
    <node id="F1e2" href="c:\data\Zielsystem.xml#id("F1e2")/self::parameter"/>
    <node id="F1a1" href="c:\data\Zielsystem.xml#id("F1a1")/self::parameter"/>
    . . .
<!-- Parameter der Quellfunktionen -->
    <node id="f1e1" href="c:\data\Quellsystem.xml#id("f1e1")/self::parameter"/>
    <node id="f1a1" href="c:\data\Quellsystem.xml#id("f1a1")/self::parameter"/>
    <node id="f2e1" href="c:\data\Quellsystem.xml#id("f2e1")/self::parameter"/>
    . . .
<!-- Parameter der zusätzlichen Funktionen: h1 für cast, h2 für concat -->
    <node id="h1e1" href="c:\data\helper.xml#id("h1e1")/self::parameter"/>
    <node id="h1a1" href="c:\data\helper.xml#id("h1a1")/self::parameter"/>
    <node id="h2e1" href="c:\data\helper.xml#id("h2e1")/self::parameter"/>
    . . .
<!-- Abhängigkeiten -->
    <dependency from="F1e1" to="f1e1"/>
    <dependency from="F1e2" to="f2e2"/>
    <dependency from="f1a1" to="h1e1"/>
    <dependency from="h1a1" to="F1a1"/>
    <dependency from="f1a1" to="f2e1"/>
    . . .
</map>
```

Es sei angemerkt, daß die Abhängigkeiten der Ausgabeparameter einer Quellfunktion von ihren Eingabeparametern nicht explizit aufgeführt werden müssen, da diese bereits durch die Beschreibung der Funktionen implizit gegeben sind.

Wurde die Abbildung vollständig in XML beschrieben, so können die Informationen mittels der Sprache XSLT (Extensible Stylesheet Language Transformations, [W3C99c]) in eine andere Form transformiert werden, die für die weitere Verarbeitung der Daten notwendig ist. Es wäre beispielsweise denkbar, daß für die Durchführung der topologischen Sortierung die Abhängigkeiten anders beschrieben werden. Außerdem kann aufgrund der Abbildungsbeschreibung Code generiert werden, der im Zielsystem eingesetzt wird, z.B. der Funktionsaufruf in Java.

Wir möchten anmerken, daß es sich bei der hier skizzierten DTD zur Abbildungsbeschreibung um einen rudimentären Ansatz handelt, da nicht alle Möglichkeiten von XML und verwandten Standards und Vorschlägen ausgeschöpft wurden. Ebenso enthält die Beschreibung der Systeme lediglich die notwendigen Informationen für den vorgestellten Ansatz der Funktionsabbildung. Um die Systeme genauer zu beschreiben, sollten noch mehr Daten zu einem dahinterliegenden Datenmodell sowie der enthaltenen Semantik bereitgestellt werden. Bei der Erarbeitung einer geeigneten DTD bedarf es somit zukünftig noch eingehender Betrachtungen.

5.5 Erzielter Leistungsumfang der Beschreibungssprache

Abschließend soll analysiert werden, inwiefern die vorgestellte Beschreibungssprache dazu beiträgt, die in Kapitel 4 aufgeführten Abweichungen der abzubildenden Funktionen zu überwinden. Dieser Ansatz ermöglicht in erster Linie die Bearbeitung der Abweichungen, welche die Parameter betreffen. Dazu gehören die Umbenennung und Datentypkonvertierung der Funktionsparameter. Ebenso kann der Fall beschrieben werden, wenn Parameter weiterverarbeitet werden müssen und dazu zusätzliche Funktionen notwendig sind. Betrachtet man die Funktionen, so ist es möglich, die Abbildung der Zielfunktionen in einem 1:1-, n:1- und n:m-Verhältnis darzustellen, wenn die Zielfunktionen selbst nicht voneinander abhängig sind. Liegen Abhängigkeiten zwischen Quellfunktionen vor, so kann aus der Beschreibung eine Ausführungsreihenfolge ermittelt werden.

Ein wesentlicher Punkt, den der Lösungsansatz noch nicht betrachtet hat, ist der schreibende Zugriff. Zudem wurde noch nicht genauer untersucht, wie eine Art Schema für die lokalen APIs eingearbeitet werden kann, das nicht nur die Signaturen der Funktionen enthält. Weiterhin wurde nicht darauf eingegangen, welche Konsequenzen das Einsetzen von Objekten mit Methoden als Parameter haben kann und wie sich Abhängigkeiten zwischen den Zielfunktionen auswirken können.

6. Vergleichbare Ansätze

Viele Arbeiten, die neben den Daten auch die Integration von Funktionen unterstützen wollen, wählen einen objektorientierten Ansatz [GCO90, BNS95, HD92, RS97]. Dieser ermöglicht es, neben den strukturellen Eigenschaften einer Quelle auch das Verhalten der Instanzen mittels der Definition von Methoden oder Funktionen zu beschreiben. Diese Ansätze verfolgen neben der strukturellen Abbildung der reinen Datenintegration auch die operationale Abbildung. Von struktureller Abbildung spricht man, wenn Übereinstimmungen zwischen konzeptuellen Dateneinheiten festgelegt werden, die in den Schemata der zu integrierenden Systeme auftreten. Diese Form schlägt jedoch fehl, sobald keine Schemata oder nur Fragmente davon vorliegen. Im Gegensatz dazu werden bei der operationalen Abbildung Übereinstimmungen zwischen Operationen auf unterschiedlichen Stufen festgesetzt. Basierend auf der operationalen Abbildung erweitert die operationale Integration den Anwendungsbereich der Integration von der Wiederverwendung von Daten zur Wiederverwendung von Daten und Anwendungssoftware. Existierende Anwendungen können entweder als Methoden gehandhabt werden, die mit spezifischen Klassen verbunden werden, oder als neue Objekte für diejenigen Anwendungen, die nicht das Objekt-Konzept unterstützen. Bei den referenzierten Ansätzen wird jedoch keine allgemeine Methodik zur API-Integration, vergleichbar etwa zu [SL90] für die Schemaintegration, vorgegeben. Statt dessen werden in den Implementierungen der globalen Methoden alle Plattformheterogenitäten (Programmiersprachen, Netzwerkprotokolle usw.) proprietär überwunden. Weiterhin werden oftmals keine oder nur wenige Sprachmittel zur Modellierung von Semantik in dem globalen Schema (Vor- und Nachbedingungen, Abhängigkeiten zwischen Objekten, Invarianten usw.) zur Verfügung gestellt. Eine umfangreiche Spezifikationssprache kann jedoch den Integrationsprozeß erleichtern und zum Verständnis der System- und Datenabhängigkeiten beitragen.

Ein weiterer Ansatz wird in [WWC92] und [MBSW99] vorgestellt. Unter dem Begriff *Megaprogramming* wird die Zusammensetzung von Komponenten betrachtet, die von heterogenen, autonomen und verteilten Software-Modulen, den sog. *Megamodules*, als Methoden angeboten werden. Das Ziel hierbei ist, diese Methoden zusammenzuführen, um daraus neue Anwendungen zu erstellen und gleichzeitig die Autonomie der Software-Module zu erhalten. Während unser Ansatz jedoch eine vertikale Integration anstrebt, konzentriert sich das Megaprogramming eher auf den horizontalen Ansatz, d. h. das Zusammenführen von Komponenten und nicht der integrierte Zugriff auf ausgewählte Funktionalität der integrierten Quellsysteme. Des weiteren wird nicht in Betracht gezogen, daß sich die Ausschnitte der

realen Welt in den Quellsystemen überlappen und dabei auch Abhängigkeiten entstehen können. Ebenso wird keine Anfragesprache angeboten, da die Module nur dazu genutzt werden, um neue Anwendungen zu schreiben. Somit ist keine flexible Interaktion des Benutzers möglich.

7. Zusammenfassung und Ausblick

In dieser Arbeit wurde eine Beschreibungssprache zur Funktionsintegration in heterogenen Anwendungssystemen vorgestellt, die auf einer generischen Methodik basiert. Hierzu wurde zunächst die Funktionsintegration von der Datenintegration abgegrenzt, die bereits Gegenstand vieler Forschungsarbeiten ist. Überdies wurde aufgezeigt, daß eine Kombination der beiden Integrationsformen notwendig ist, wenn Datenbank- und Anwendungssysteme integriert werden sollen. Anschließend wurde in knapper Form eine Integrationsarchitektur vorgestellt, in deren Rahmen die Funktionsabbildung erfolgen soll. Nach der Erarbeitung einer Klassifikation der zu integrierenden APIs als operationale Schnittstelle wurde ein Lösungsansatz für eine Beschreibungssprache vorgestellt, mit welcher die Abbildung von globalen auf lokale Funktionen spezifiziert werden kann. Dieser Ansatz basiert auf der Beschreibung von Abhängigkeiten zwischen den Funktionsparametern und deckt somit größtenteils die Punkte der beschriebenen Klassifikation ab, welche die Funktionsparameter betreffen. Als Beschreibungssprache wurde XML gewählt, wobei auch eine einfache DTD zur Festlegung der Beschreibungsweise vorgestellt wurde. Ein Beispiel sollte den Lösungsansatz verdeutlichen. Die so erstellte Abbildungsbeschreibung soll schließlich zur teilweisen Generierung von Komponenten der Integrationsarchitektur eingesetzt werden. Abschließend wurde der vorgestellte Ansatz zu anderen verwandten Arbeiten abgegrenzt.

In zukünftigen Arbeiten muß eine DTD entworfen werden, welche die Möglichkeiten von XML optimal ausschöpft und somit eventuell auch kompakter gestaltet werden kann. Darüber hinaus muß analysiert werden, inwiefern die Transformation der Abbildungsinformation mit XSLT ausgeführt und wie die Methoden des DOM (Document Object Model, [W3C98b]) genutzt werden können. Neben dem Beschreibungsmodell, das in dieser Arbeit vorgestellt wurde, muß noch ein geeignetes Ausführungsmodell erarbeitet werden. Dieses könnte beispielsweise einen Operatorbaum aufbauen, der gleichzeitig die Betrachtung einer möglichen Optimierung der Ausführung erlauben würde. Da das Ziel die Kombination von Daten- und Funktionsintegration ist, muß eine Verbindung zwischen den beiden Formen realisiert werden. Es bleibt zu klären, ob dies ebenfalls über die Beschreibung von Abhängigkeiten möglich ist. Überdies muß noch untersucht werden, welche Schritte der Beschreibung generiert werden können, um dem Benutzer die Arbeit zu erleichtern. Denkbar wäre z.B. ein graphisches Werkzeug, das einen Teil der XML-basierten Abbildungsbeschreibung selbst erzeugen kann.

Literatur

$At^{+}89$	M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, S. Zdonik: *The Object Oriented Database System Manifesto*, Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases, 1989, 40-58
BE96	O.A. Bukhres, A.K. Elmagarmid (Hrsg.): *Object-Oriented Multidatabase Systems - A Solution for Advanced Applications*, Prentice Hall, 1996.
BM98	H. Behme, S. Mintert: *XML in der Praxis*, Addison Wesley, 1998.
BNS95	E.Bertino, M.Negri, L.Sbattella: *An Overview of the Comandos Integration System*, Object-Oriented Multidatabase Systems (O.Bukhres and A.Elmagarmid, eds.), Prentice-Hall, 1995.
Co98	S. Conrad: *Föderierte Datenbanksysteme - Konzepte der Datenintegration*, Springer-Verlag, 1997
Da96	P. Dadam: *Verteilte Datenbanken und Client/Server-Systeme*, Springer-Verlag, 1996.
ENV98	Enovia Corp.: *ENOVIAVPM PDM II Solutions*, 1998; zu beziehen über http://www.enovia.com/products/html/enoviavpm.html.
GCO90	R. Gagliardi, M. Caneve, G. Oldano: *An Operational Approach to the Integration of Distributed Heterogeneous Environments*, Databases: Theory, Design, and Applications, postconference publication of PARBASE-90, First International Conference on Databases, Parallel Architectures and their Applications, 1991, S. 110-124.
GP98	C.F. Goldfarb, P. Prescod: *The XML Handbook*, Prentice Hall PR, 1998.

HBP94 A.R. Hurson, M.W. Bright, S.H. Pakzad (Hrsg.): *Multidatabase Systems: An Advanced Solution for Global Information Sharing*, IEEE Comp. Society Press, 1994.

HD92 M. Härtig, K. R. Dittrich: *An object-oriented integration framework for building heterogeneous database systems*, Proceedings of the IFIP DS-5 Conference on Semantics in Interoperable Database Systems, Lorne, Australia, S. 33-53, 1992.

HST99 T. Härder, G. Sauter, J. Thomas: *The Intrinsic Problems of Structural Heterogeneity and an Approach to their Solution*, VLDB Journal 8(1), 1999, 25-43.

Ki95 W. Kim (Hrsg.): *Modern Database Systems - The Object Model, Interoperability, and Beyond*, Addison-Wesley, 1995.

MBSW99 L. Melloul, D. Beringer, N. Sample, G. Wiederhold: *CPAM, A Protocol for Software Composition*, Proc. 11th Int. Conf. on Advanced Information Information Systems Engineering (CAISE), Heidelberg, 1999, S. 11-25.

NW94 M. Norrie, M. Wunderli et al.: *Coordination Approaches for CIM*, Proceedings European Workshop on Integrated Manufacturing Systems Engineering (IMSE), 1994, S. 223-232.

OMG99 Object Management Group: *The Common Object Request Broker Architecture: Architecture and Specification*, Revision 2.3, 1999; zu beziehen über http://www.omg.org/corba/corbaiiop.html

RH98 F.d.F. Rezende, K. Hergula: *The Heterogeneity Problem and Middleware Technology: Experiences with and Performance of Database Gateways*, Proc. 24th Int. Conf. on VLDB, New York, 1998, S. 146-157.

RS97 Mary Tork Roth, Peter M. Schwarz: *Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources*, VLDB 1997; S. 266-275.

Sa98 G. Sauter: *Interoperabilität von Datenbanksystemen bei struktureller Heterogenität*, Dissertation, infix-Verlag, 1998.

SAP98 SAP AG: *Das R/3 System*, 1998; zu beziehen über http://www.sap-ag.de/products/r3/.

SC99 S. St. Laurent, E. Cerami: *Building XML Applications*, McGraw Hill, 1999.

SDRC98 SDRC Corp.: *Metaphase*, 1998; zu beziehen über http://www.metaphasetech.com/.

SL90 A.P. Sheth, J.A. Larson: *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases*, ACM Computing Surveys 22 (3), 1990, S. 183-236.

Vo99 M.Vogt: *Entwurf von Mechanismen zur Integration von Anwendungssystemen*, Diplomarbeit, Universität Ulm, 1999.

W3C98a World Wide Web Consortium: *Extensible Markup Language (XML) 1.0*, W3C Recommendation,1998; zu beziehen über http://www.w3.org/TR/REC-xml.

W3C98b World Wide Web Consortium: *Document Object Model (DOM) Level 1 Specification Version 1.0*, W3C Recommendation, 1998; zu beziehen über http://www.w3.org/TR/REC-DOM-Level-1

W3C99a World Wide Web Consortium: *XML Linking Language (XLink)*, W3C Working Draft, 1999; zu beziehen über http://www.w3.org/TR/xlink.

W3C99b World Wide Web Consortium: *XML Pointer Language (XPointer)*, W3C Working Draft, 1999; zu beziehen über http://www.w3.org/TR/WD-xptr.

W3C99c World Wide Web Consortium: *XSL Transformations (XSLT) Version 1.0*, W3C Working Draft, 1999; zu beziehen über http://ww.w3.org/TR/xslt.

WWC92 G. Wiederhold, P. Wegner, S.Ceri: *Towards Megaprogramming*, Communications of the ACM 35(11), 1992, S. 89-99.