Der FDBS-Demonstrationsprototyp $\mathbf{SIGMA}_{DEMO}$[1]

Michael Höding Guido Grohmann Eyk Hildebrandt

Institut für Technische Informationssysteme
Otto-von-Guericke-Universität Magdeburg
Postfach 4120, D–39016 Magdeburg
Tel.: ++49-391-67-12220 Fax: ++49-391-67-12020
E-mail: {hoeding|grohmann|eykhilde}@iti.cs.uni-magdeburg.de

März 1997

[1]Diese Arbeit wurde gefördert vom Land Sachsen-Anhalt unter FKZ: 1987A/0025 "Föderierung heterogener Datenbanksysteme und lokaler Datenhaltungskomponenten zur systemübergreifenden Integritätssicherung" – "$\mathbf{SIGMA}_{FDB}$ (**S***chema* **I***ntegration and* **G***lobal integrity* **M***aintenance* **A***pproach* *for* **F***ederated* **D***ata* **B***ases*)"

Zusammenfassung

In diesem Bericht stellen wir den im Rahmen des Projektes SIGMA$_{FDB}$ entwickelten FDBS-Demonstrationsprototypen SIGMA$_{DEMO}$ vor. Dabei handelt es sich um ein Softwarepaket, anhand dessen Funktionsweise und Wirkprinzipien eines föderierten Datenbanksystems (FDBS) demonstriert werden können. Die vorgestellte Lösung realisiert ein zur Illustration der FDBS-Entwurfsmethodik auf Basis des *Generischen Integrations Modells* GIM genutztes Beispiel, das dem Bereich Maschinenbau entliehen wurde.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Abschnitt möchten wir einen kurzen Einblick in das Umfeld der in diesem Beitrag vorgestellten FDBS-Demonstrationssoftware SIGMA$_{DEMO}$ geben. Dazu erfolgt eine Erläuterung der Projektziele und des Projektumfeldes. Durch die Diskussion projektinterner und projektexterner Anforderungen wird der Bedarf nach einer FDBS-Demonstrationssoftware motiviert. Abschließend stellen wir die Kapitelstruktur dieses Berichtes vor.

1.1 Das Projekt SIGMA$_{FDB}$

Das Projekt **SIGMA**$_{FDB}$[1] wird im Rahmen des Forschungsschwerpunktes "Neue Produktionssysteme/Experimentelle Fabrik" als Grundlagenforschungsprojekt seit September 1996 durch das Land Sachsen-Anhalt gefördert. Hauptziel ist es, wichtige Grundlagen für die Schaffung einer Informationsinfrastruktur für die durchgängige Fabrikgestaltung zu erforschen. Dabei geht es in erster Linie darum, verschiedenartige Softwaresysteme, die in den Phasen der Fabrikplanung im besonderen bzw. der Produktgestaltung im allgemeinen zum Einsatz kommen, sinnvoll zu verknüpfen. Aufgrund der Feststellung, daß die Heterogenität der eingesetzten Softwaresysteme weder durch ein allumfassendes monolithisches Datenhaltungssystem noch durch die Umstellung aller vorhandenen Applikationen auf geeignete Standards überwunden werden kann, gilt es Wege zu finden, die ein Leben mit dieser Heterogenität ermöglichen. Der von uns zur Lösung dieses Problems vorgeschlagene Einsatz föderierter Datenbanksysteme erlaubt eine Integration heterogener Datenhaltungssysteme bei weitgehender Wahrung der lokalen Autonomie. Dies bedeutet, daß die Datenhaltungssysteme der integrierten Softwarekomponenten nicht oder nur wenig verändert werden müssen, so daß lokale Applikationsprogramme unbeeinflußt weiter arbeiten können.

Dieser Ansatz soll es nun erlauben, verschiedene Softwaresysteme zur Fabrikplanung

[1] **SIGMA**$_{FDB}$ (**S***chema* **I***ntegration and* **G***lobal integrity* **M***aintenance* **A***pproach* $_{for}$ **F**$_{ederated}$ **D**$_{ata}$ **B**$_{ases}$)

sinnvoll zu verknüpfen, ohne sie deshalb "umprogrammieren" zu müssen. Um das Anwendungsgebiet und die Probleme, die z.B. auf dem Gebiet der softwarebasierten Fabrikplanung existieren, kennenzulernen, arbeiten Informatiker, Maschinenbauer und Geisteswissenschaftler, die an insgesamt sieben vom Land Sachsen-Anhalt geförderten Projekten mitwirken, im Rahmen der Projektgruppe Metop[2], kooperativ zusammen. Neben den in regelmäßigen Metop-Treffen diskutierten strategischen Fragen, dienen vor allem direkte Treffen zwischen Projektmitarbeitern auf bilateraler Ebene der konzentrierten Zusammenarbeit. Ein grundsätzliches, aber mehr und mehr gelöstes Problem ist dabei das Finden einer gemeinsamen Sprache. Wesentlich sind dabei Referenzbeispiele, die für alle Beteiligten verständlich sind und damit eine Diskussion untschiedlicher Sichten und Herangehensweisen unterstützen. Damit wird es möglich, voneinander und miteinander zu lernen.

1.2 Interne und externe Projektziele

Betrachtet man das Gebiet föderierter Datenbanksysteme, so findet man bereits eine Reihe von Arbeiten, die sich mit Architektur und Entwurf von FDBSen auseinandersetzen [SL90, HM85]. Trotzdem müssen wir konstatieren, daß es noch wesentliche Grundlagen zu erforschen gilt, ehe FDBSe für den breiten Einsatz als Basistechnologie zur Integration heterogener Informationssysteme zur Verfügung stehen. Wir konzentrieren uns im Rahmen des Projektes vor allem auf folgende Fragestellungen:

- **Methodik zum Entwurf von FDBSen:**
  Wesentlich für den erfolgreichen Einsatz von FDBSen ist die Verfügbarkeit einer Methodik, die den korrekten Entwurf föderierter Datenbanksysteme ermöglicht. Dabei spielen vor allem Fragen der Konfliktlösung durch Schematransformation und Schemaintegration eine Rolle. Um diesen Prozeß zu unterstützen wurde im Rahmen des Projektes ein speziell für den FDBS-Entwurf konzipiertes Datenmodell entworfen [Sch95]. Dieses GIM (Generic Integration Model) genannte Datenmodell ermöglicht die automatische Ableitung integrierter Schemata. Die dafür in [SS96a, SS96b, SS96c] vorgestellten Algorithmen sind bereits teilweise im Rahmen von FDBS-Entwurfswerkzeugen implementiert.

- **Einbeziehung von Dateienverbünden in FDBSe:**
  Ein wichtiges zu lösendes Problem ist auch die Integration von Dateiensystemen im Rahmen von FDBSen. Viele der bisherigen Ansätze gehen davon aus, daß alle zu integrierenden Komponentensysteme auf der Datenbankmanagementsystemschnittstelle aufsetzen. Diese Situation ist aber durch eine Vielzahl leistungsfähiger Spezialapplikationen, die vor allem im PC-Umfeld anzutreffen sind, nicht gegeben. Erste Vorschläge für eine systematische und damit effiziente Dateienintegration wurden in [Höd96b, Köl96, Höd96a] vorgestellt.

[2] Mensch-Technik-Organisation-Planung

- **Systemübergreifende Konsistenzsicherung durch aktive Mechanismen:** Ein wichtiges Ziel einer Integration ist es, neben der lokalen Konsistenz der Daten, die durch die lokalen Datenbankmanagementsysteme garantiert wird, auch ein hohes Maß an systemweiter Integrität zu gewährleisten. Das bedeutet u.a., daß Änderungen von Daten über ein Objekt der realen Welt, das in mehreren Datenbanken enthalten ist, durch das FDBS in allen Datenbanken vollzogen werden müssen. Dazu wurde in [TC95, CT97] die Nutzung aktiver Mechanismen vorgeschlagen.

Weitere im Rahmen des Projektes erforschte Grundlagenfragen betrachten *Transaktionskonzepte für FDBSe* und *Sicherheit in FDBSen*.

Neben der für einen zukünftigen Praxiseinsatz wesentlichen Klärung von grundlegenden konzeptionellen Fragen gibt es noch eine Reihe von externen Zielen, die vor allem im Zusammenhang mit der interdisziplinären Zusammenarbeit im Rahmen der Projektgruppe Metop zu sehen sind. Diese betreffen:

- die Aufarbeitung der spezifischen Anforderungen und Problemstellungen der Anwendungsgebiete aus Sicht der Datenbankforschung,
- die Erarbeitung von Diskussionsbeispielen, die FDBSe und Anwendungsgebiete verknüpfen,
- die Unterstützung anderer Projektgruppen mit Datenbankwissen und
- die Darstellung erzielter Forschungsergebnisse als Angebot an andere Forschungsgruppen oder an Unternehmen der Region.

Ein wesentliches Mittel, um bereits gewonnene Erkenntnisse im Sinne der oben genannten Ziele synergetisch zu verknüpfen, ist der in den letzten Monaten im ITI entwickelte und in dieser Arbeit vorgestellte FDBS-Demonstrationsprototyp SIGMA$_{DEMO}$. Dabei handelt es sich um ein Softwaresystem, das zwei heterogene lokale Informationssysteme eines fiktiven Fördertechnik-Unternehmens integriert. Dies sind das Informationssystem der Konstruktionsabteilung auf Basis eines objektorientierten Datenbanksystems und das Informationssystem der Vertriebsabteilung auf Basis eines relationalen Datenbanksystems. Das Beispielszenario zeichnet sich durch eine hohe Komplexität bei geringem Umfang aus. Das heißt, die Schemata der lokalen Systeme enthalten nur wenige Klassen, die aber die meisten typischen Datenmodell- und Schemakonflikte enthalten. Die Lösung dieser Konflikte erfolgt durch unsere Entwurfsmethodik (vgl. [CHJ$^+$96]). Das als Ergebnis des Entwurfsprozesses entstandene objektorientierte integrierte Schema wurde im SIGMA$_{DEMO}$-Prototyp implementiert. Anhand einer auf diesem Föderierungskern aufsetzenden, globalen Applikation können die komplexen Abläufe in einem FDBS veranschaulicht werden. Zum Beispiel kann man erkennen, wie sich lokale Änderungsoperationen global auswirken. Interessant sind insbesondere die durch den Föderierungskern unterstützten Mechanismen zur globalen Konsistenzsicherung. Desweiteren wird eine globale Applikation, die eine online-Sicht auf die integrierten Datenbanken auf Basis eines Datenmodells, das Nullwerte erlaubt, in Form eines (simplifizierten) WWW-Produktkataloges realisiert. Neben diesen der Veranschaulichung und

Diskussionsunterstützung dienenden Aspekten der SIGMA$_{DEMO}$, kann der Prototyp im weiteren zur Validierung von Architekturvorschlägen, insbesondere zur Erprobung von Konsistenzsicherungs- und Transaktionsmechanismen genutzt werden.

1.3 Vorgehen und Kapitelstruktur

Ziel dieses Berichtes ist es, Grundlagen, Funktion und Implemetierungsaspekte der FDBS-Demonstrationssoftware SIGMA$_{DEMO}$ zu präsentieren.

Nach der in diesem Kapitel gegebenen Motivation werden im Kapitel 2 wichtige Grundbegriffe zum Forschungsgebiet föderierte Datenbanksysteme aufgearbeitet. Dabei geht es um die Einordnung in das Gebiet der Multidatenbanksysteme, Architekturaspekte, den Entwurf und den Betrieb föderierter Datenbanksysteme. Im folgenden dritten Kapitel wird das Beispielszenario ausführlich vorgestellt. In Kapitel 4 folgt die Erläuterung der weitgehend aus [CHJ^{+}96] übernommenen Anwendung der GIM-Methodik zum Entwurf von FDBS auf unser Beispielszenario. Insbesondere wird dabei der Aspekt der Integration vorhandener Integritätsbedingungen diskutiert. Eine detaillierte Diskussion von Implementierungskonzepten wird im fünften Kapitel durchgeführt. Dabei werden systematisch die verschiedenen lokalen Systeme, der Föderierungskern und die globalen Applikationen vorgestellt. In Kapitel 6 werden wesentliche Ergebnisse und Erkenntnisse des Berichtes zusammengefaßt. Abschließend folgt ein Ausblick auf zukünftige Arbeiten.

Weitere Details zu Systemvorraussetzungen für SIGMA$_{DEMO}$ (vor allem bezüglich vorrauszusetzender Software) sowie Hinweise zur Konfiguration und Bedienung der vorhandenen Version der SIGMA$_{DEMO}$-Software können den Anhängen entnommen werden.

Kapitel 2

Föderierte Datenbanksysteme

Für das Verständnis der SIGMA$_{DEMO}$–Software ist ein grundlegender Einblick in das Gebiet der föderierten Datenbanksysteme notwendig. Dieser Einblick soll im folgenden Kapitel vermittelt werden. Im ersten Abschnitt wird der Einsatz von FDBSen motiviert. Danach erfolgt eine kurze Einführung in die Grundlagen von FDBSen. Im Anschluß wird eine Referenzarchitektur für FDBSe vorgestellt. Abschließend wird eine Einteilung und Einordnung von FDBSen vorgenommen.

2.1 Heterogeniät in der Datenverwaltung

Die stürmische Entwicklung der letzten Jahrzehnte in der Informatik und speziell in der Welt der Datenbanksysteme haben bewirkt, daß die derzeitige Datenverwaltung in Unternehmen von Heterogenität geprägt und ein gesamtheitlicher Blick nicht möglich ist. So kommen oft in ein und demselben Unternehmen eine Reihe von Datenbankmanagementsystemen (DBMSen) zum Einsatz, die sich insbesondere hinsichtlich Datenmodell, Anfragesprache, Hardwareanforderungen, Sicherheitsmodell etc. unterscheiden. Die Anschaffung und Wartung von DBMSen stellt für kommerzielle Benutzer in der Regel eine erhebliche Investition sowohl in die Systeme als auch in die Erfassung und Verwaltung der Daten dar. So werden ältere Systeme weiterverwendet und gleichberechtigt mit modernen leistungsfähigeren Systemen, die auch neue Anwendungsfelder erschließen, eingesetzt. Dadurch können sich erhebliche Redundanzen bei der Datenhaltung ergeben.

Der Drang nach Rationalisierung der Arbeitsabläufe erfordert, alle Daten eines Unternehmens unter einem Blickwinkel zu betrachten und systemübergreifende Applikationen zu etablieren. In solchen Applikationen müssen die unterschiedlichen Schnittstellen und Datenformate der zugrundeliegenden DBMSe berücksichtigt werden. Es wäre daher wünschenswert, wenn über eine einheitliche Schnittstelle und unter Verwendung einer einheitlichen Sicht auf die Daten zugegriffen und dabei die Herkunft der Daten verborgen werden könnte. Ebenfalls könnte eine solche Schnittstelle Redundanzen verbergen und systemübergreifende Beziehungen sichtbar machen. Ziel ist ein integrierter Zugriff auf die Unternehmensdaten, um den ganzheitlichen Aspekt der Unternehmensverwaltung auch im Bereich der Datenverarbeitung unterstützen zu können. Eine offensichtliche Lösung wäre die Anschaffung eines neuen Systems und die Migration aller Daten und Applikationen in das neue einheitliche Format. Kosten, Aufwand und Risiko stehen jedoch in keinem Verhältnis zu dem zu erwartenden Nutzen.

Als akzeptabler Lösungsansatz wurde in den letzten Jahren die Idee der sogenannten föderierten Datenbanksysteme (u.a. [HM85, SL90]) entwickelt, die einerseits die Forderungen nach einer systemübergreifenden Sicht erfüllen und andererseits die kostspielige Umstellung von Daten und Applikationen vermeiden soll. Vielfältige Forschungsarbeit wurde auf diesem Gebiet bereits geleistet. Dabei wurden sowohl Prototypen als auch erste einsatzfähige Produkte entwickelt (vgl. [TBD+87, BFHK94, RBB+95]). In den folgenden Abschnitten sollen wichtige Begriffe definiert sowie die grundlegenden Konzepte, Charakteristiken und die Architektur von FDBSen erläutert werden.

2.2 Begriffe, Charakteristiken und Konzepte

Über die Grundlagen von FDBSen wurden in den letzten Jahren eine Reihe von Artikeln mit ausführlichen Betrachtungen veröffentlicht. Basis des nachfolgend Erläuterten ist der Artikel von [SL90], der allgemein als Ausgangspunkt für die FDBS-Forschung akzeptiert wird. Bevor ein tieferer Einstieg in das Gebiet der FDBSe erfolgen kann, gilt es nachfolgend die wichtigsten damit verbundenen Begriffe zu klären.

- **Föderiertes Datenbanksystem**

  Ein FDBS ist ein Datenbanksystem (DBS), bestehend aus einer Sammlung kooperierender, autonomer und möglicherweise heterogener *Komponentendatenbanksysteme* (CDBSe), das eine einheitliche Schnittstelle für in den CDBSen gespeicherte Daten anbietet.

- **Föderiertes Datenbankmanagementsystem (FDBMS)**

  Das FDBMS, auch als *Föderierungsschicht* oder *–dienst* bezeichnet, koordiniert die Zusammenarbeit der CDBSe innerhalb des FDBSs und bietet *globalen Applikationen* einheitliche Operationen auf der Basis einer homogenisierten und integrierten Gesamtsicht auf die in den CDBSen verwalteten Daten an.

- **Gemeinsames Datenmodell (GDM)**

  Als GDM bezeichnet man das Datenmodell, d.h. die Modellierungs- und Anfragesprache, auf dem die homogenisierte und integrierte Gesamtssicht basiert.

Der Zusammenhang zwischen den definierten Begriffen wird durch die prinzipielle Architektur, wie sie Abbildung 2.1 zeigt, verdeutlicht.

Nach der Definition wichtiger Begriffe sollen nun die Charakteristiken von FDBSen genauer diskutiert werden. Dies soll anhand der drei folgenden orthogonalen Dimensionen geschehen, die Systeme, die aus mehreren DBSen bestehen, wozu auch die FDBSe gehören, prägen:

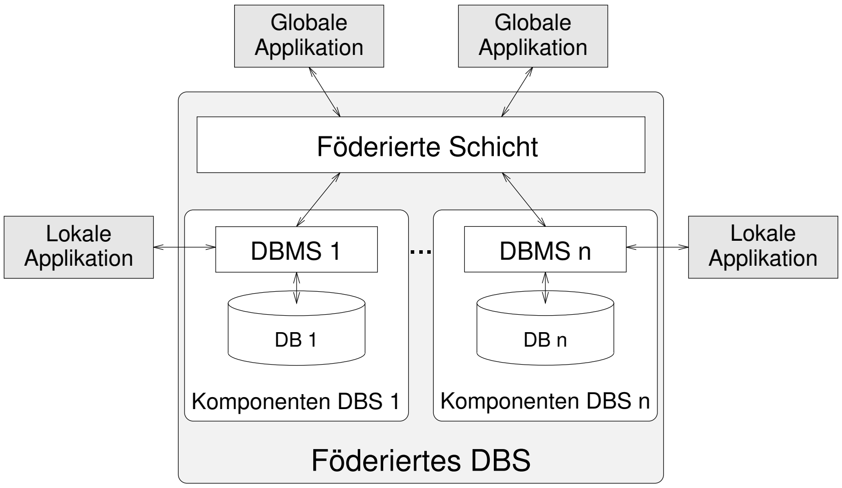


Abbildung 2.1: Föderiertes Datenbanksystem

- Verteilung
- Heterogenität
- Autonomie

Neben jeweils einer kurzen Erläuterung, welche Ausprägungen diese Dimensionen annehmen können, werden die Grundkonzepte von FDBSen abgeleitet.

Die Daten, auf die mittels eines FDBSs zugegriffen werden kann, sind in verschiedenen Datenbanken unter der Kontrolle unterschiedlicher CDBSe gespeichert, d.h. sie sind verteilt. In FDBSen wird es angestrebt, diese Verteilung für die globalen Nutzer so weit wie möglich transparent [1] zu halten.

Wie bereits eingangs erwähnt, ist die Hetrogenität von Datenhaltungssystemen in Unternehmen, die die CDBSe eines FDBSs bilden, ein Hauptgrund, eine einheitliche Schnittstelle anzubieten. Heterogenität äußert sich in einer Vielzahl von Arten, so daß eine Klassifikation notwendig wird. Grundsätzlich lassen sich zwei Bereiche abstrahieren. Für den ersten Bereich – die syntaktische Heterogenität – lassen sich u. a. die folgenden Arten erkennen.

- Art des Datenhaltungssystems
  - Zentrales DBS
  - Verteiltes DBS
  - Föderiertes DBS
  - Dateissystem

[1] Unter *Transparenz* wird hierbei, wie im Bereich der verteilten Systeme allgemein üblich, Durchsichtigkeit bzw. Durchschaubarkeit [Dud96] in der Bedeutung von *nicht sichtbar* verstanden.

- Art des Datenmodells
  - hierarchisch
  - relational
  - objektorientiert
  - ...
- Art des Datenbankmodells

Der zweite Bereich ist die semantische Heterogenität der konzeptionellen Schemata. Sie entsteht durch die Abstraktion, Modellierung und Definition von Daten durch die Betreiber der CDBSe. Überlappende Weltausschnitte können dabei (eventuell trotz gleichem Datenmodell) unterschiedlich dargestellt werden. In einem FDBS können die CDBSe unterschiedliche Grade von Heterogenität aufweisen. Die Überwindung der Heterogenität ist das wichtigste Ziel für ein FDBS. Ein FDBS mit heterogenen Komponenten zeigt Abbildung 2.2.

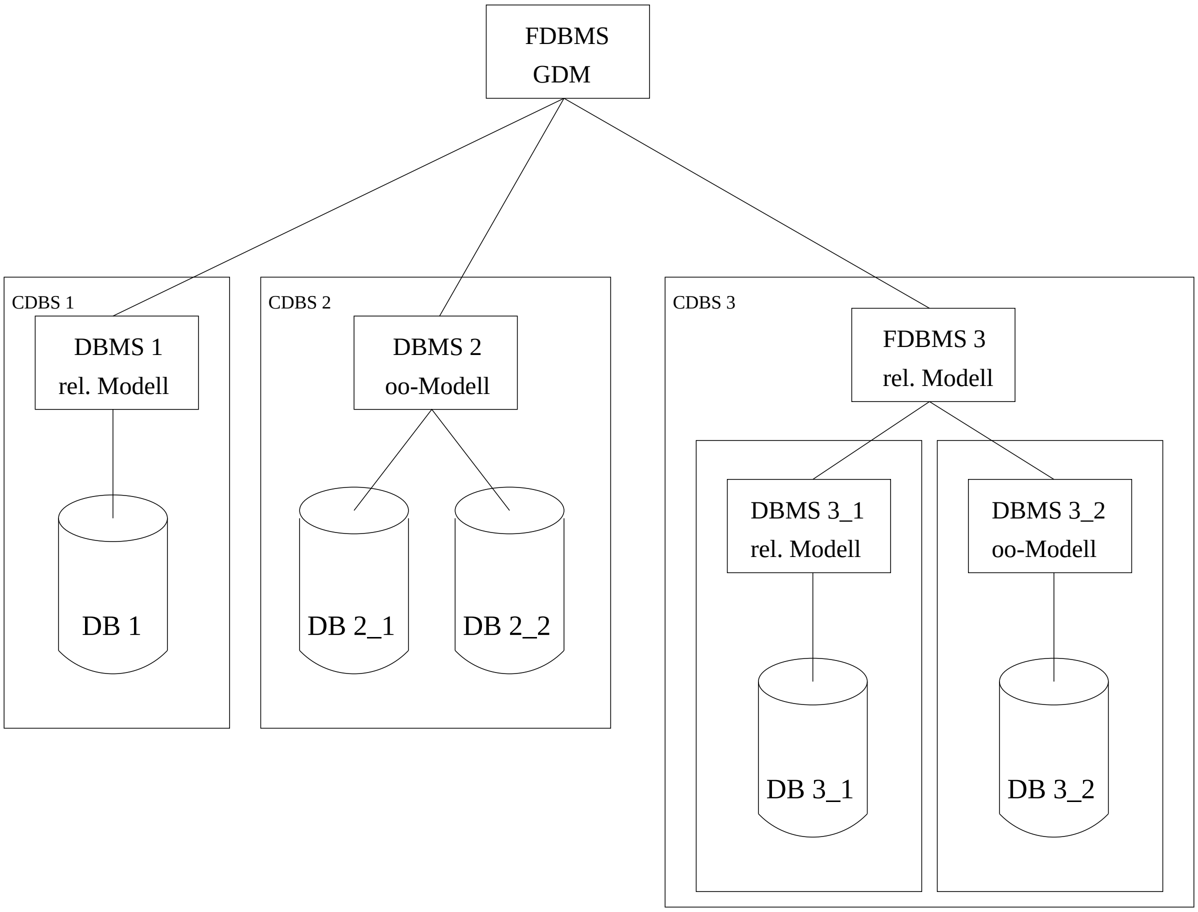


Abbildung 2.2: FDBS mit heterogenen Komponenten

Ein wichtiges Charakteristikum von Datenhaltungssystemen in Unternehmen ist die Autonomie, d.h. die Systeme besitzen i.A. die uneingeschränkte Kontrolle über die von ihnen verwalteten Daten. Die in Tabelle 2.1 beschriebenen vier Arten von Autonomie werden unterschieden. Ein wichtiges Ziel für FDBSe ist es den Schutz aller Arten von Autonomie der CDBSe so weit wie möglich sicherzustellen, so daß der Betrieb eines CDBSs unabhängig von seiner Teilnahme an einer Föderation weiterlaufen kann. Insbesondere sollen die lokalen Applikationen der CDBSe unabhängig und ohne Beeinträchtigung vom FDBS ihre Aufgaben ausführen können.

Designautonomie	Kontrolle über Modellierung und Repräsentation der Daten
Kommunikationsautonomie	Kontrolle über Kommunikation mit anderen Systemen
Ausführungsautonomie	Kontrolle über Ausführung aller Operationen
Assoziationsautonomie	Kontrolle über Teilung der Ressourcen und Funktionalität

Tabelle 2.1: Autonomiearten

Zusammenfassend lässt sich sagen, daß FDBSe einen akzeptablen Ansatz bieten, um eine Integration heterogener, autonomer Datenhaltungssysteme unter einem vernünftigen Kosten-Nutzen-Verhältnis zu erreichen. Aber FDBSe können auch weitere Einsatzzwecke erfüllen, wofür die folgenden drei Beispiele stehen sollen:

- Datenmigration zur Eliminierung von DBSen ([RS94])
- Realisierung eines OODBMSs auf der Basis eines relationalen DBSs
- FDBS als sicheres Frontend für unsichere CDBSe ([JD94])

2.3 Referenzarchitektur

Im folgenden Abschnitt soll nun die interne Realisierung von FDBSen eingehender betrachtet werden. Als Ausgangspunkt soll dabei die 5-Schemaebenenarchitektur von [SL90] dienen, die eine allgemein akzeptierte Referenzarchitektur für FDBSe darstellt.

Basis der Referenzarchitektur ist das 5-Schemaebenenmodell, das in Anlehnung an das ANSI/X3/SPARC 3-Schichtenmodell für Datenbanksysteme ([TK78, Dat86]) entwickelt wurde. Das Modell basiert auf den für ein FDBS notwendigen Schemata, die in 5 Ebenen hierarchisch angeordnet sind. Im folgenden sollen die Schemata und die notwendigen Übergangsprozesse zwischen in der Hierarchie benachbarten Schemata kurz erläutert werden.

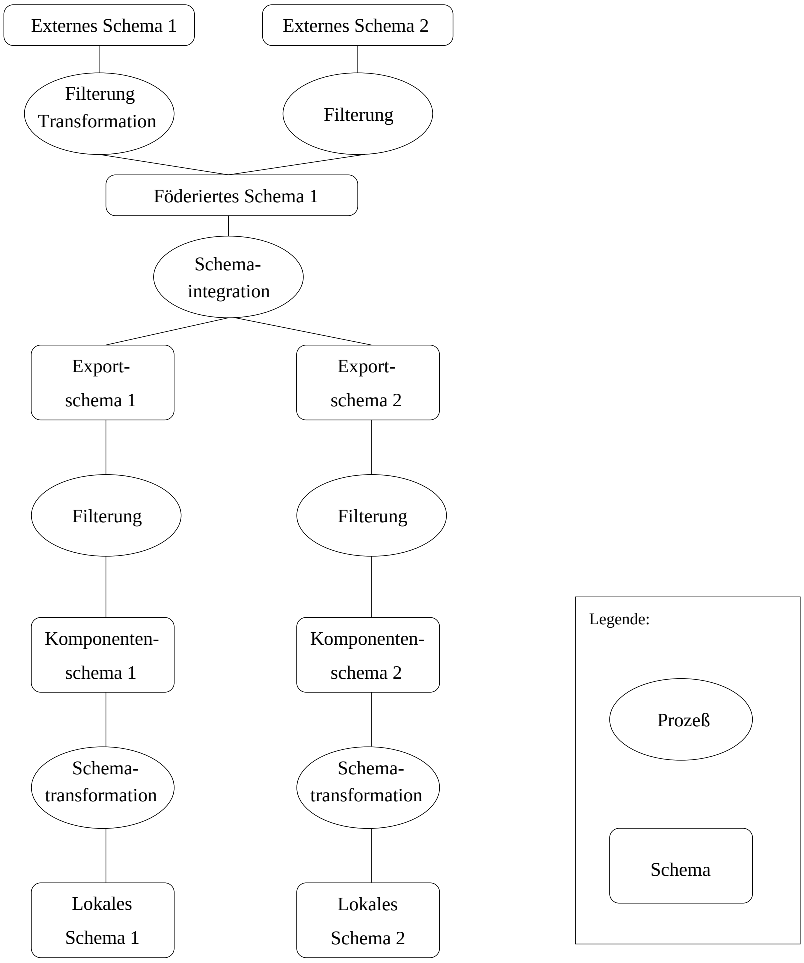


Abbildung 2.3: 5-Schemaebenenmodell nach [SL90]

Lokale Schemata: Als lokale Schemata (LS) werden die konzeptionellen Schemata der CDBSe bezeichnet. Sie liegen in den originären Datenmodellen der zugrundeliegenden DBMSe vor.

Schematransformation: Beim Prozeß der Schematransformation werden die LS in das GDM des FDBSs übersetzt, wodurch die syntaktische Heterogenität überwunden wird. Dieser Prozeß kann zusätzlich mit einer semantischen Anreicherung versehen werden, wenn das Datenmodell des lokalen Schemas nicht die Ausdrucksfähigkeiten des GDMs besitzt (vgl. [HK95]).

Komponentenschemata: Als Komponentenschemata (KS) werden die in das GDM transformierten LS bezeichnet.

Filterung: Bei diesem Prozeß werden aus den KS die Informationen gefiltert, die dem FDBS zur Verfügung gestellt werden sollen. Dieser Prozeß begründet sich durch die

Autonomie- und Sicherheitsanforderungen der CDBSe, d.h. sie bestimmen, welche der von ihnen gespeicherten Daten, globalen Nutzern zugänglich sein sollen.

Exportschemata: Die durch den Prozeß der Filterung entstandenen Schemata werden als Exportschemata (ES) bezeichnet.

Schemaintegration: Bei diesem Prozeß werden die ES durch Auflösung von Schemakonflikten, wodurch die semantische Heterogenität überwunden wird, integriert. Zu Herangehensweisen, Problemen und Lösungsansätzen vergleiche man u. a. [BLN86, SPD92, Dup94, Sch95].

Föderierte Schemata: Die föderierten Schemata (FS) entstehen durch den Schemaintegrationsprozeß. Das Schemaebenenmodell erlaubt die Bildung von einem oder mehreren FS innerhalb eines FDBSs. Dies ist davon abhängig, ob ein eng gekoppelter oder ein lose gekoppelter Ansatz gewählt wird (vgl. Abschnitt 2.4).

Ableitung Externer Schemata: Aus den FS können Externe Schemata zur Spezifizierung von Benutzersichten, zur Hinzufügung zusätzlicher Integritätsbedingungen oder zur werteabhängigen Zugriffskontrolle abgeleitet werden. Sollen die Externen Schemata in einem anderen Datenmodell als dem GDM formuliert werden, ist eine weitere Schematransformation notwendig.

Unter Umständen können einzelne Ebenen wegfallen, z.B. ist keine Schematransformation notwendig, wenn das Datenmodell eines CDBSs und das GDM übereinstimmen. Der Zusammenhang zwischen Schemata und Prozessen wird noch einmal in Abbildung 2.3 verdeutlicht.

Das 5-Schemaebenenmodell wird ergänzt durch eine Systemarchitektur, die die grundlegenden Systemkomponenten und Abläufe innerhalb eines FDBSs aufzeigt, ohne eine konkrete Realisierung vorzugeben. Für genauere Betrachtungen sei auf [SL90] verwiesen.

2.4 Einteilung und Einordnung

Wie bereits in Abschnitt 2.2 gesagt wurde, können FDBSe verschiedene Einsatzzwecke erfüllen. Ausgehend von der Referenzarchitektur kann für jeden Zweck ein spezielles FDBS entwickelt werden. Zwei grundlegende Systemkonzepte für FDBSe lassen sich jedoch unterscheiden und sollen nachfolgend kurz diskutiert werden.

Bei *lose gekoppelten* Systemen wird keine einheitliche Betrachtungsweise für die Daten angestrebt, d.h. die Heterogenität wird nur soweit wie für den angestrebten Zweck notwendig überwunden. Häufig werden hierbei mehrere föderierte Schemata erstellt, d.h. für jeden Nutzer oder Nutzergruppe wird ein eigenes föderiertes Schema gebildet. Die CDBSe kooperieren dabei nur soweit, daß ein koordinierter Zugriff auf ihre Daten möglich ist. Schutz der Autonomiearten steht im Vordergrund. Lose gekoppelte Systeme finden

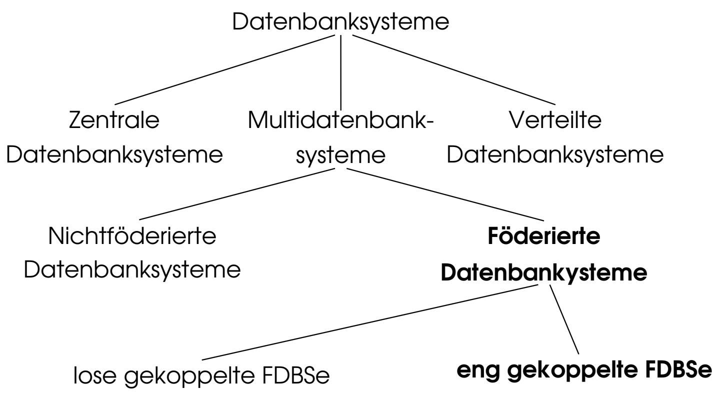


Abbildung 2.4: Einordnung von FDBSen nach [SL90]

ihren Einsatz u. a. beim Information Retrieval über große Datenbanken auf der Basis einer schnellen Anfragesprache ([Rah94]).

Eng gekoppelte Systeme streben eine gesamtheitliche Sicht auf die von den CDBSen bereitgestellten Daten an, d. h. es wird nur ein föderiertes Schema propagiert. Das bedeutet weiterhin, daß eine vollständige Überwindung der syntaktischen und semantischen Heterogenität sowie eine hohe Verteilungstransparenz angestrebt wird. Um den entsprechend notwendigen Kooperationsumfang der CDBSe zu erreichen, müssen Kompromisse beim Schutz der Autonomie eingegangen werden. Eng gekoppelte Systeme bieten im Gegensatz zu lose gekoppelten Systemen die Möglichkeit zur systemübergreifenden Konsistenzsicherung (vgl. u.a. [TC95, TC96, SCH+96]). Sie eignen sich vor allem für den in Abschnitt 2.1 motivierten Einsatzzweck. Auch der in dieser Arbeit vorgestellte Prototyp basiert auf diesem Systemkonzept.

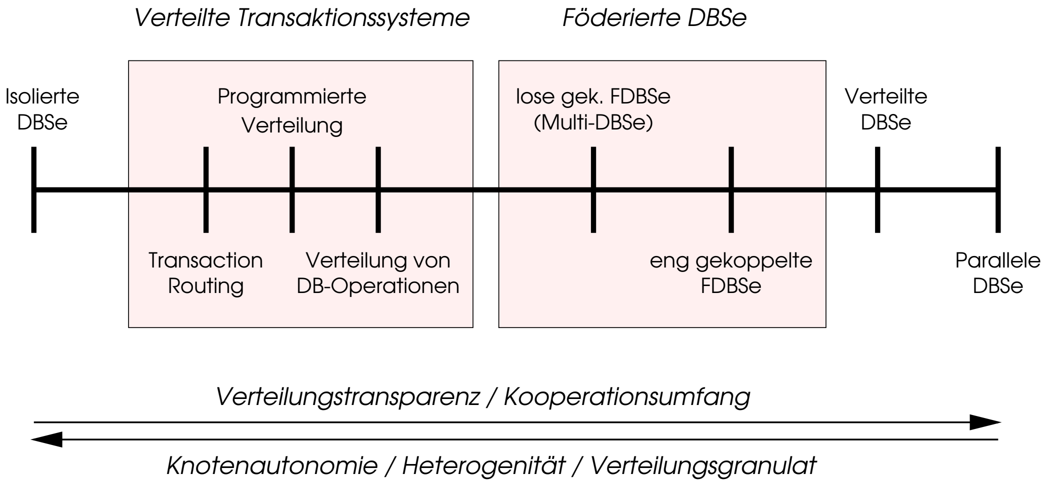


Abbildung 2.5: Einordnung von FDBSen nach [Rah94]

Abschließend soll eine Einordnung der FDBSe in den Bereich der Datenbanksysteme vorgenommen werden. Abbildung 2.4 zeigt die Einordnung nach [SL90]. FDBSe sind hiernach eine Teilmenge der sogenannten *Multidatenbanksysteme* (MDBSe), die sich von Zentralen und Verteilten DBSen unterscheiden. MDBSe ermöglichen Operationen auf verschiedenen beteiligten Datenbanksystemen. Zu den MDBSen gehören noch die *Nichtföderierten* DBSe, die eine Integrationsform nichtautonomer CDBSe darstellen.

Abbildung 2.5 zeigt nun die Einordnung nach [Rah94], die auf einer Abgrenzung von DBSen anhand der drei Dimensionen Verteilung, Autonomie und Heterogenität beruht. Isolierte DBSe entsprechen hierbei den Zentralen DBSen aus der Einordnung von [SL90]. *Verteilte Transaktionssysteme* sind ein weiterer Ansatz zum Zugriff auf heterogene Datenbestände. Diese ermöglichen eine verteilte Transaktionsverarbeitung unter der Kontrolle von Monitoren außerhalb der heterogenen DBSe ([Rah94]). Sie bilden eine Alternative zu FDBSen, wenn ein geringerer Kooperationsumfang zwischen den beteiligten DBSen angestrebt wird oder der Aufwand für die Realisierung eines FDBSs zu groß gegenüber dem erstrebten Nutzen erscheint.

Kapitel 3

Anwendungszenario

In diesem Kapitel wird zunächst das Anwendungsgebiet, ein produzierendes Unternehmen umrissen. Desweiteren werden die in ein FDBS zu integrierenden und als vorhanden angenommenen Informationssysteme betrachtet. Hierbei werden insbesondere die Datenbankschemata beleuchtet.

3.1 Das Beispielunternehmen

Grundlage der Betrachtungen bildet ein (fiktives) produzierendes Unternehmen. Dieses Unternehmen bietet komplette Materialflußanlagen für Stückgüter an. Zum Transport von Stückgütern dienen hier Rollenförderer. Das Tätigkeitsfeld umfaßt daher die Konstruktion und Produktion solcher Rollenförderer sowie den Vertrieb eigener und fremdgefertigter Transportmaschinen, wozu z.B. auch Gabelstapler gehören. Innerhalb des Unternehmens gibt es u.a. eine Konstruktions- und eine Vertriebsabteilung, die jeweils über eigene Datenhaltungssysteme verfügen.

Von der Konstruktions- und Planungsabteilung werden Rollenförderer (`Roller_Conveyor, RC`) konstruiert sowie der zugehörige Ablauf der Produktion vorbereitet. Ein wichtiges Merkmal eines solchen Rollenförderers ist das Vorhandensein oder Fehlen eines eigenen Antriebs. Hat ein Rollenförderer einen eigenen Antrieb vorliegt, müssen weitere technische Daten in der Konstruktionsdatenbank berücksichtigt werden. Daher werden noch angetriebene (`Driven_Roller_Conveyor, DRC`) und nichtangetriebene Rollenförderer unterschieden.

Die beim Entwurf oder bei der Konstruktion gewonnenen und benötigten Daten werden von einer CAD/CAP-Anwendung in einer CAx-Datenbank verwaltet. In der Konstruktionsabteilung ist hierzu ein objektorientiertes Datenbanksystem zur Verwaltung von CAD- und anderen technischen Daten von Rollenförderern als Komponenten kompletter Materialflußanlagen etabliert. Eine hier vorhandene Datenbankanwendung erlaubt die Erfassung und Manipulationen der gespeicherten Daten. Ein darüber hinaus vorhandener Datenbankbrowser kann natürlich auch benutzt werden.

Im Unterschied zur Konstruktionsabteilung kommt in der Vertriebsabteilung ein relationales Datenbanksystem zum Einsatz. In diesem Datenbanksystem sind u.a. alle aktuell produzierten Rollenförderer und alle anderen von der Firma angebotenen Transportmaschinen mit ihren Preisen und der Lieferzeit erfaßt. Ein Anwendungsprogramm erlaubt auch hier Einfügen, Löschen und Ändern von Datensätzen in der Datenbank der Vertriebsabteilung. Ähnlich wie in der Konstruktionsabteilung kann hier eine interaktive SQL-Schnittstelle, wie z.B. das `sqlplus` von ORACLE, benutzt werden.

Aus den bis jetzt beschriebenen Sachverhalten bezüglich Tätigkeitsfeld und Unternehmensstruktur ist ersichtlich, daß insbesondere zu bereits fertig konstruierten Rollenförderern sowohl Konstuktionsdaten auch Daten in der Verkaufsdatenbank vorhanden sind. Es existieren also Daten zu in der Realwelt identischen Objekten in den verschiedenen Datenbanken. Andererseits kann es auch vorkommen, daß die Konstruktionsdaten eines Rollenförderers gelöscht werden, aber seine Verkaufsdaten bleiben erhalten, bis die betreffenden Exemplare verkauft sind.

3.2 Die Datenbankschemata

Nachfolgend werden die für die weitere Arbeit relevanten Sichten auf die Schemata der beiden - im Hinblick auf die in Abb. 2.3 vorgestellte Architektur nun als lokale Datenbanksysteme bezeichneten - Informationssysteme näher vorgestellt. Hierzu werden auch Auszüge aus Quelltexten herangezogen. Zunächst wird das Objektorientierte Schema der CAx-Datenbank betrachtet.

3.2.1 Das Schema der Konstruktionsdatenbank

Das objektorientierte Datenbankschema der CAx-Datenbank wird in Abbildung 3.1 graphisch sowie in C++-Notation dargestellt. (Anstelle von ... stehen im tatsächlich implementierten Quelltext Anweisungen bzw. Ausdrücke, die für die Verwendung der C++-Klassen als Datenbankklassen erforderlich sind, aber keine Relevanz bei der Betrachtung der lokalen Schemata haben.)

```
enum PROJECT_STATES { IN_DESIGN = 1, IN_CONSTRUCTION, COMPLETED };
... class roller_conveyor : ...
{
 private:  // all Attributes with conditions
    char                product_no[6] ; // key
    int                 length ;
    int                 width ;
    PROJECT_STATES      projectstate ;
  public:   // Attributes without conditions
    char                product_name[20] ;
```

```
    // Constructor
    roller_conveyor(char*        _product_no,
           char*                 _product_name,
           int                   _length,
           int                   _width,
           PROJECT_STATES        _projectstate);
   ...
    ~roller_conveyor();
}
```

Zunächst sei die Klasse der Rollenförderer vorgestellt. Für einen jeden solchen Rollenförderer werden eine unternehmensweit einheitliche Produktnummer (ein Schlüssel also), eine Bezeichnung, Länge und Breite sowie der Projektstatus gespeichert. Die Bezeichnung (der Name) kann aus Groß- und Kleinbuchstaben gebildet werden. Ziffern und Sonderzeichen sind ebenfalls erlaubt. Der Projektstatus gibt an, ob der betreffende Rollenföderer gerade entworfen (`IN_DESIGN`) oder konstruiert wird (`IN_CONSTRUCTION`), oder ob die Konstruktion abgeschlossen (`COMPLETED`) ist. Dabei soll der Zustand des Entwerfens nur zu Beginn der Objektlebensdauer zugelassen sein, vom Zustand `COMPLETED` kann aber wieder in den Status `IN_CONSTRUCTION` übergegangen werden, was z.B. bei nachträglich festgestellten Konstruktionsmängeln sinnvoll ist. Länge und Breite sind Beispiele für Konstruktionsdaten. Aus technischen Gründen gibt es Einschränkungen im Verhältnis zwischen Länge und Breite eines Rollenförderers. Daraus wurde die Bedingung abgeleitet, daß der Längenwert eines solchen RC-Objektes nicht 15 mal größer sein darf als sein Breitenwert, was durch eine Methode (hier: `change()`) überprüft wird.

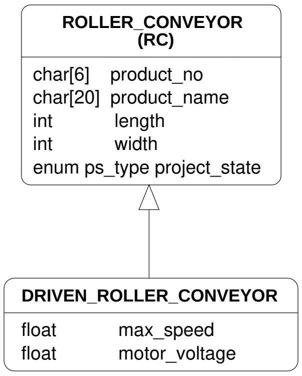


Abbildung 3.1: CAD/CAP-Schema

Die Werte für Länge und Breite können nur mittels dieser Methode verändert werden. Dieses objektorientierte Konzept wird als Kapselung bezeichnet und muß dann angewendet werden, wenn Attribute eines Objektes bestimmten Bedingungen genügen sollen. Die im Zusammenhang mit der Produktnummer erwähnte Schlüsselbedingung wird dann auch in einer Methode (`set_product_no()`) überprüft. Hierbei wird zumindest innerhalb des Datenbestandes der Konstruktionsabteilung garantiert, daß keine Produktnummern mehrfach auftreten. Die Realisierung von Integritätsbedingungen in Klassenmethoden ist auch für objektorientierte Datenbanksysteme ein durchaus vertretbarer und weit verbreiteter Ansatz.

```
  int      change(int _length, int _width);
    {
    if (_length <= 15 * _width)
      {
      //Attribute setzen
      length = _length ;
      width = _width;
      return 0;
      }
    else return 1;
    }
```

eingebetteter Integritäsbedingung

Durch Spezialisierung entsteht aus der Klasse `Roller_Conveyor` die Klasse der angetriebenen Rollenförderer (`Driven_Roller_Conveyor`). Sie haben neben den bereits bekannten Eigenschaften der Rollenförderer auch einige nur den Antrieb betreffende Parameter. Hier sind die Höchstgeschwindigkeit, die konstruktiv bedingt ist, und die Höhe der Versorgungsspannung für den Motor, die je nach Anwendungsgebiet völlig unterschiedliche Werte annehmen kann, zu nennen. Da es sich um ein objektorientiertes Schema handelt, können die bereits vorhandenen Eigenschaften einfach von den Rollenförderern geerbt werden.

```
class driven_roller_conveyor : public roller_conveyor
{
  public:
    float      max_speed;
    float      motor_voltage;
    driven_roller_conveyor( char*           _product_no,
                            char*           _product_name,
                            int             _length,
                            int             _width,
                            PROJECT_STATES  _projectstate,
                            float           _max_speed,
                            float           _motor_voltage
                           );
    ~driven_roller_conveyor();
};
```

Im nächsten Abschnitt wird nun das Schema der relationalen Vertriebsdatenbank vorgestellt.

3.2.2 Das Schema der Vertriebsdatenbank

Die Vertriebsabteilung des betrachteten Unternehmens verwaltet alle zu veräußernden Transportmaschinen in einer relatinalen Datenbank, deren Schema jetzt Gegenstand der Betrachtungen ist. In der Datenbank wird eine Transportmaschine mit ihrer Produktnummer, einer Bezeichnung, dem Preis der Lieferdauer und einem weiteren Attribut, das eine Klassifikation der gespeicherten Transportmaschinen ermöglicht, abgelegt. Das Schema der Datenbank wurde in einer für relationale Datenbanksysteme gebräuchlichen Datenbeschreibungssprache notiert und ist wie aus dem folgenden ersichtlich aufgebaut.

```
Create Table transport_machine (
   product_no char(6) primary Key,
   product_name vchar(40),
   price number(10,2),
   delivery_period number(3),
   machine_class vchar(30)) );
```

Die Bezeichnung (oder der Name) einer Transportmaschine wird aufgrund einer Vereinbarung innerhalb der Abteilung in großen Buchstaben geschrieben, während in der CAx-Datenbank auch kleine Buchstaben bei der Benennung zugelassen sind.

Auch in diesem Schema ist die Produktnummer ein Schlüssel. Im Gegensatz zum objektorientierten System der Konstruktionsabteilung werden zur Überwachung der Schlüsselbedingung Konsistenzsicherungsmechanismen des Datenbanksystems benutzt, wie das Schlüsselwort `primary key` erkennen läßt. Das Attribut Preis ist ein Gleitkommawert, während die Geltungsdauer in vollen Tagen anzugeben und somit ganzzahlig ist. Die Maschinenklasse einer Transportmaschine muß, wenn es sich um einen eigenproduzierten Rollenförderer handelt, einen der aus Tabelle 3.1 zu entnehmenden Werte haben.

Maschinenklasse	Transportmaschine
RC	Rollenförderer (ohne Antrieb)
DRC	angetriebener Rollenförderer
FLK	Gabelstabler

Tabelle 3.1: Maschinenklassen für Rollenförderer in der Verkaufsdatenbank

Die bis jetzt beschriebenen Datenbanken, Datenbankschemata und Applikationen sind nun Ausgangspunkt für eine vom Unternehmen gewünschte Integration, die im folgenden Kapitel beschrieben wird.

Kapitel 4

FDBS-Entwurf und SIGMA$_{DEMO}$

Das im SIGMA$_{DEMO}$-Projekt realisierte Beispiel wurde zu wesentlichen Teilen aus einem Papier übernommen, in dem auf das Verhalten und die Nutzung von Integritätsbedingungen beim Entwurf föderierter Datenbanksysteme eingegangen wird [CHJ$^+$96]. Da die Methodik von großer Bedeutung für das Verständnis des SIGMA$_{DEMO}$-Prototyps ist, möchten wir die Schritte der Schemaintegration und Schematransformation an Hand des in Kapitel 3 vorgestellten Anwendungsszenarios diskutieren. Zuvor gehen wir kurz auf die wesentlichen Konzepte des für die Entwurfsmethodik zentralen Integrations-Datenmodells GIM ein.

4.1 Konzepte von GIM

Die richtige Wahl des gemeinsamen Datenmodells (*GDM*) ist von entscheidender Bedeutung für den Integrationsprozeß. Diese Entscheidung beeinflußt vor allem die Qualität des Integrationsschrittes, des wichtigsten Schritts im FDBS-Entwurfsprozeß. Vorgeschlagene GDMe reichen von semantisch armen Datenmodellen, z.B. dem relationalen Modell, bis hin zu semantisch reichen Datenmodellen, z.B. dem objektorientierten Modell. Die meisten in jüngerer Zeit vorgeschlagenen Ansätze bevorzugen objektorientierte Datenmodelle. Es wird oft argumentiert, daß nur semantisch reiche Datenmodelle als GDM in Frage kommen, weil dadurch kein oder nur ein geringer semantischer Verlust bei der Transformation von einem objektorientierten Datenmodell in das gemeinsame Datenmodell auftreten kann [SCG91]. Andererseits bedingen semantisch reiche Datenmodelle mehr Schemaheterogenität als semantisch arme Datenmodelle [SPD92]. Diese Schemaheterogenität erhöht wiederum die Komplexität des Integrationsprozesses. Dieses Problem ist in auf objektorientierte Datenmodellen basierenden Methodiken, wie sie z.B. in [SPD92, RPRG94, PBE95, GSC95] beschrieben wurden, zu beobachten.

Deshalb wurde von I. Schmitt in [Sch95] erstmalig das semantisch arme Datenmodel GIM (*Generic Integration Model*) für den FDBS-Entwurf vorgeschlagen. In weiteren Beiträgen, u.a. [SS96b, SS96c, CHJ$^+$96, SS96b], wurde die auf GIM beruhende Entwurfsmethodik weiterentwickelt und zum Teil formalisiert. In GIM können nur Schemainformationen, die für den Integrationsprozeß notwendig sind, ausgedrückt werden. Hingegen sind Schemainformationen, die eine spezifische Sicht einer Applikation definieren, nicht durch GIM darstellbar. Weiterhin ermöglicht ein in GIM formuliertes Schema die Ableitung verschiedener externer Schemata. Die sichtabhängige Semantik lokaler Schemata, die nicht in GIM ausgedrückt werden kann, kann dann bei der Ableitung externer Schemata wieder genutzt werden. Dieser Aspekt wird in den folgenden Abschnitten bei der Erläuterung des Beispiel ausführlicher diskutiert.

Im Folgenden werden die Grundkonzepte von GIM kurz aufgelistet und erläutert:

- *Einfache Datentypen*: Das Konzept der Relation in erster Normalform wurde von den relationalen Datenbanksystemen übernommen, um die Intension einer GIM-Klasse zu definieren.

- *Nichtüberlappende Klassenextensionen*: Die Extensionen beliebiger Paare von GIM-Klassen müssen entweder identisch oder elementfrei sein. Untermengenbeziehungen oder teilweise Überlappung zwischen Klassen sind nicht erlaubt.

- *Objektidentifikatoren*: In GIM wird das OID-Konzept genutzt (vgl. auch [KC86]).

- *Bidirektionale binäre Referenzen*: Eine Referenz drückt die Beziehung zwischen exakt zwei Klassen aus. GIM unterstützt hierfür verschiedene Kardinalitäten (1:1, 1:n, m:n) für Referenztypen zwischen Klassen. Alle Referenzen in GIM sind bidirektional, das heißt, jede Referenz hat eine zugehörige inverse Referenz. Referenzen werden als Referenzattribut einer Klasse ausgedrückt. Wenn ein Referenzattribut auf mehr als ein Objekt einer Klasse verweisen kann, so ist das Referenzattribut mehrwertig. Nur Referenzattribute können mehrwertig sein.

- *Integritätsbedingungen*: GIM unterstützt Integritätsbedingungen vergleichbar dem SQL2-Standard. Dazu gehören z.B. Constraints auf Referenzkardinalitäten, Uniqueness-Constraints und Constraints, die einen festen Attributwert für alle Elemente einer Klasse definieren.

Wegen der geringen semantischen Ausdruckskraft der aufgelisteten Konzepte sollte GIM nicht als Schnittstellendatenmodell für Applikationsprogramme dienen. Deshalb müssen integrierte bzw. externe Schemata, die auf anderen Datenmodellen basieren, aus dem integrierten GIM-Schema abgeleitet werden. Die Ableitung eines objektorientierten Schemas aus einem semantisch armen Datenmodell ist u.a. in [BKNW91] beschrieben. Ein Algorithmus, der die automatische Ableitung von objektorientierten Schemata aus GIM-Schemata unterstützt, wurde in [SS96a] und [SS96b] vorgestellt.

Zu viele Schematransformation reduzieren die Leistung eines FDBS. Aus diesem Grund wird in [CHJ+96] vorgeschlagen, ein Entwurfswerkzeug zu benutzen, das unabhängig vom FDBS ist. Ein derartiges Tool kann die Ableitung integrierte bzw. externer Schemata graphisch unterstützen. Transformationsinformationen für spezielle FDBS können aus der entworfenen Schemahierarchie abgeleitet werden, nachdem unnötige

Transformationsschritte entfernt und andere Transformationsschritte optimiert worden sind.

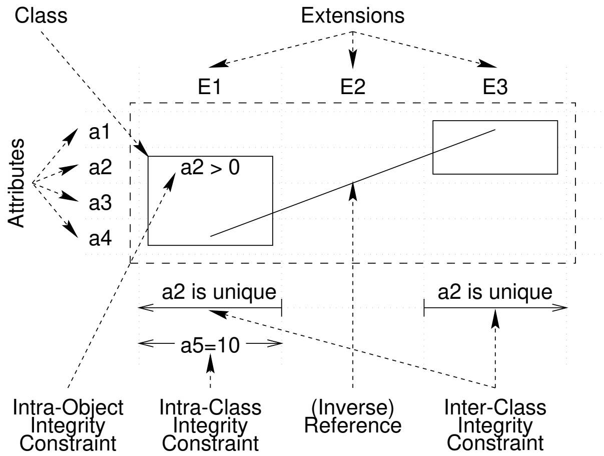


Abbildung 4.1: GIM-Diagramm-Konventionen

GIM-Schemata werden in Form von GIM-Diagrammen dargestellt (vgl. Abb. 4.1). Um die GIM-Diagramme verstehen zu können, möchten wir im folgenden kurz die Semantik der graphischen Elemente erläutern. Die horizontale Dimension eines GIM-Diagramms beschreibt die extensionalen Aspekte eines Schemas, wohingegen die vertikale Dimension die intensionalen Aspekte der GIM-Klassen beschreibt. Eine Klasse selbst wird durch ein Rechteck repräsentiert. Eine Referenz zwischen zwei Klassen wird durch eine Linie, die die beiden entsprechenden Referenzattribute verbindet, dargestellt. Auch die Darstellung von Integritätsbedingungen ist in GIM-Diagrammen möglich. Ist die Integritätsbedingung eine "intra-class"- und "single object"-Bedingung, wird sie innerhalb des die Klasse bzw. das Attribut beschreibenden Rechtecks positioniert. Anderenfalls kann die Integritätsbedingung unter dem Diagramm gefunden werden. Die extensionale Ausdehnung einer solchen Integritätsbedingung wird durch zwei Pfeile begrenzt. Integritätsbedingungen, die einen bestimmten Wert für ein Klassenattribut festlegen (derartige Attribute gehören nicht zur Intension einer Klasse) werden ebenfalls unter dem Diagramm positioniert. In Abbildung 4.1 ist die Integritätsbedingung `a5=10` eine derartige Integritätsbedingung für das Klassenattribut `a5`. Die Namen der Extensionen und die Namen der Klassenattribute werden entsprechend über dem bzw. links vom Diagramm aufgeführt.

In den nächsten Abschnitten wollen wir die Integrationsmethodik an Hand unseres Beispielszenarios genauer diskutieren. In unserem Beispiel werden Exportschemata nicht berücksichtigt.

4.2 Schematransformation in GIM

In diesem Abschnitt stellen wir kurz die Schematransformation nach GIM für das CAx-Schema und das Verkaufs-Schema unseres Beispielszenarios vor.

Einfache Datentypen
Auf die Benutzung mengenwertiger Attribute wurde bei der Definition der lokalen Schemata des Beispielszenarios verzichtet. Die lokalen Schemata liegen in erster Normalform vor. Treten mengenwertige Attribute auf, so ist vergleichbar zum Normalisierungsprozeß, der aus dem relationalen Datenbankentwurf bekannt ist, die Ableitung einer zusätzlichen Klasse sowie entsprechender Referenzattribute und Integritätsbedingungen vorzunehmen. Wir wollen hierauf an dieser Stelle nicht näher eingehen. Dieses Problem wurde tiefergehend in [CHJ+96] diskutiert.

Grundsätzlich ist zu beachten, daß Probleme bei der Transformation auftreten können, wenn es keine isomorphe Abbildung zwischen globalen Datentypen und GIM-Datentypen gibt. In diesem Fall müssen zusätzliche Integritätsbedingungen, z.B. zur Definition von Ober- bzw. Untergrenzen oder der Genauigkeit, abgeleitet werden. In unserem Beispiel haben wir die neuen Integritätsbedingungen `A1` und `B1` erzeugt, durch die die Länge der Zeichenketten, die Produktnamen repräsentieren, begrenzt wird.

Nichtüberlappende Klassenextensionen
Das GIM-Konzept der nichtüberlappenden Klassenextensionen steht im Widerspruch zum Konzept der Spezialisierung in objektorientierten Datenmodellen. Auch in relationalen Schemata kann diese GIM-Anforderung verletzt werden, obwohl im relationalen Datenmodell keine Spezialisierung möglich ist. Eine derartige Verletzung kann durch extensionale Dekomposition gelöst werden. Wenn sich die Extensionen der Klassen `A` und `B` überlappen, sind die drei Klassen `A−B`, `B−A` und `A∩B` (unter Anwendung der üblichen Mengenoperationen $-$ und $\cap$) zu erzeugen. Diese Klassen haben keine überlappenden Extensionen mehr. Die Intensionen der Klasse `A−B` bzw. der Klasse `B−A` sind dieselben, wie die der Klasse `A` bzw. `B`. Die Intension der Klasse `A∩B` enthält alle Attribute der beiden Klassen `A` und `B`.

Weiterhin müssen nach dem Ausführen der extensionalen Dekomposition die entsprechenden Integritätsbedingungen mittransformiert werden. "Intra-object"-Integritätsbedingungen können direkt auf die aus der Transformation resultierenden Klassen übertragen werden. Im Gegensatz dazu müssen "intra-class"-Integritätsbedingungen in "inter-class"-Integritätsbedingungen, die alle entstandenen Klassen überdecken, umgewandelt werden. Es ist zu beachten, daß Integritätbedingungen, die ursprünglich auf einer einzelnen Klasse definiert waren, nach der extensionalen Dekomposition zu "inter-class"-Bedingungen werden, da die ursprüngliche Klasse in unterschiedliche Klassen unterteilt wurde.

In unserem Beispiel sind das Ergebnis der Dekomposition der Klassen `roller_conveyor` und `driven_roller_conveyor` die Klassen `DRC` und `RCnotDRC`. Dabei enthält die Extension der Klasse `RCnotDRC` alle Rollenförderer, die *nicht* angetrieben sind. Hingegen enthält die Extension der Klasse `DRC` **nur** angetriebene Rollenförderer.

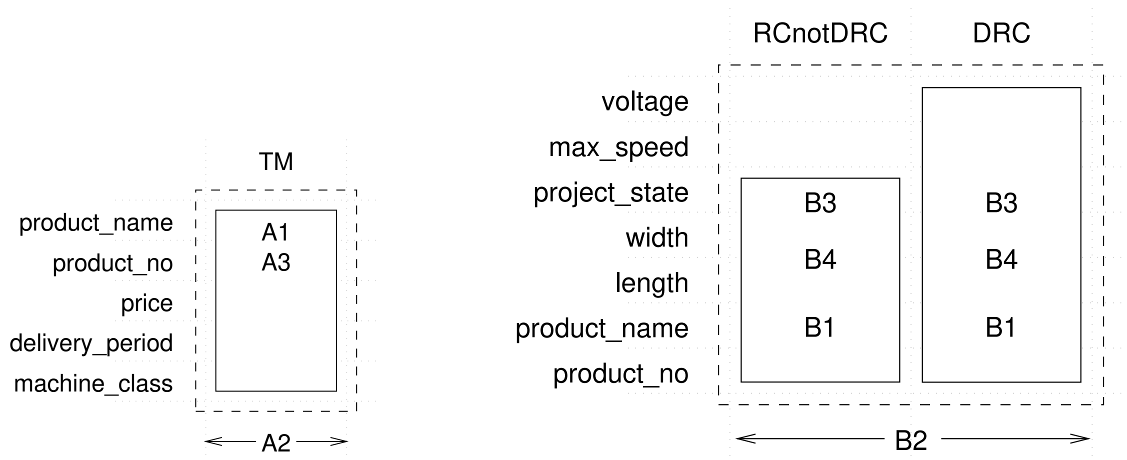


Abbildung 4.2: In GIM transformierte Schemata

Nr	Attribute	Extension	Beschreibung	Bemerkung
A1	prod_name	TM	length<20	intra-object
A2	prod_no	TM	unique	intra-class
A3	prod_no	TM	not null	intra-object
B1	prod_name	RCnotDRC, DRC	length<40	intra-object
B2	prod_no	RCnotDRC ∪ DRC	unique	inter-class
B3	project_state	RCnotDRC, DRC	in_design⇒in_construction⇒completed	intra-object
B4	length, width	RCnotDRC, DRC	length<15∗width	intra-object

Tabelle 4.1: Integritätsbedingungen der Komponentenschemata

Die Intension der Klasse `RCnotDRC` wird von der Klasse `roller_conveyor` übernommen, während die Intension der Klasse `DRC` die Attribute der beiden Klassen `roller_conveyor` und `driven_roller_conveyor` enthält. Außerdem wird die Schlüsselbedingung der Klasse `roller_conveyor` in die "inter-class" unique-Bedingung `B2` transformiert, die gegenüber der Vereinigung der Extensionen der beiden Klassen `RCnotDRC` und `DRC` zu überprüfen ist.

Die transformierten Schemata sind in Abbildung 4.2 als GIM-Diagramme dargestellt. Die Integritätsbedingungen für diese Komponentenschemata sind in der folgenden Tabelle aufgelistet.

Die abgeleiteten Schemata und die aufgelisteten Integritätsbedingungen dienen als Eingangsinformationen für die im nächsten Abschnitt beschriebene Schemaintegration.

4.3 Schemaintegration in GIM

Nachdem alle lokalen Schemata in das gemeinsame Datenmodell transformiert worden sind, ist das Problem der Datenmodellheterogenität gelöst. Um im Weiteren auch die

Schemaheterogenität zu überwinden, müssen die erzeugten Komponentenschemata integriert werden. Die Schemaintegration ist eine sehr komplexe Aufgabe. Viele Arten von Schemakonflikten, die zwischen den zwei zu integrierenden Schemata auftreten können, sind zu lösen. Voraussetzung dafür ist natürlich, daß alle Schemakonflikte erkannt werden können. Das Erkennen von Schemakonflikten kann nicht vollständig automatisch erfolgen. Es kann aber durch ein Integrationswerkzeug unterstützt werden [SEHT96, SS96a].

Der Integrationsprozeß ist entsprechend der folgenden Schritte vorzunehmen:

1. Lösung von Metakonflikten,
2. Lösung von Attributkonflikten,
3. Lösung von intensionalen Konflikten,
4. Lösung von extensionalen Konflikten,
5. Ableitung eines objektorientierten integrierten Schemas.

Nachdem die Schritte eins bis vier ausgeführt wurden, sind alle Konflikte in der GIM-Repräsentation gelöst. Als Ergebnis dieser Schritte erhalten wir das *normalisierte* Schema, das zur Ableitung objektorientierter Schemata genutzt werden kann.

Lösung von Metakonflikten

Ein Metakonflikt tritt auf, wenn Objekte einer Klasse in dem einen Schema durch unterschiedliche Werte eines bestimmten Attributes, aber in dem anderen Schema durch unterschiedliche Klassenzugehörigkeit unterschieden werden. Die Werte des ersten Schemas korrespondieren mit den Objekten des anderen Schemas [KLK91, CL94].

In unserem Beispielszenario korrespondiert der Wert "roller conveyor" des Attributs `machine_class` der Klasse `TM` mit den Objekten der Klassen `RCnotDRC` und `DRC`. Mit anderen Worten, nur für Transportmaschinen, bei denen der Attributwert von `machine_class` auf "roller conveyor" gesetzt ist, existieren korrespondierende Objekte in den Klassen `RCnotDRC` und `DRC`.

Metakonflikte können durch Transformation der Klasse von der Attributvariante in die korrespondierenden Klassenvariante gelöst werden. Diese Transformation wird durch Aufspaltung der Klasse entsprechend einer Vergleichsoperation drchgeführt. Dies geschieht in unserem Beispiel wie folgt: Die Klasse `TM` wird in die Klassen `TMnotRC` und `TMisRC` aufgesplittet. Dabei enthält die Klasse `TMnotRC` alle Objekte bei denen das Attribut `machine_class` ungleich dem Wert "roller conveyor" ist. Hingegegen enthält die Klasse `TMisRC` ausschließlich Objekte bei denen dieses Attribut gleich "roller conveyor" ist. Die Intensionen beider resultierender Klassen werden von der Klasse `TM` übernommen.

Wenn eine Klasse durch eine Vergleichsoperation aufgespalten wird, so müssen neue Integritätsbedingungen, die die Attributwerte einschränken, erzeugt werden. In unserem Beispiel wird der Wert des Attributes `TMisRC.machine_class` durch die Integritätsbedingung `A4` auf den Wert "roller conveyor" festgelegt. Damit kann dieses Attribut als

Klassenattribut betrachtet und aus der Intension der Klasse `TMisRC` entfernt werden. Die Integritätsbedingung `A5` der Klasse `TMnotRC` verbietet den Attributwert "roller conveyor".

Wie bereits oben diskutiert, können bei der extensionalen Zerlegung die "single-object"-Bedingungen `A1` und `A3` für die abgeleiteten Klassen adaptiert werden. Weiterhin wird die "intra-class"-unique Bedingung `A2` in eine "inter-class"-Bedingung überführt. `A2` ist auf der Vereinigung der Extensionen der durch Zerlegung entstandenen Klassen definiert.

Die resultierenden GIM Klassen des Konstruktions-Schemas sind in Abbildung 4.3 dargestellt.

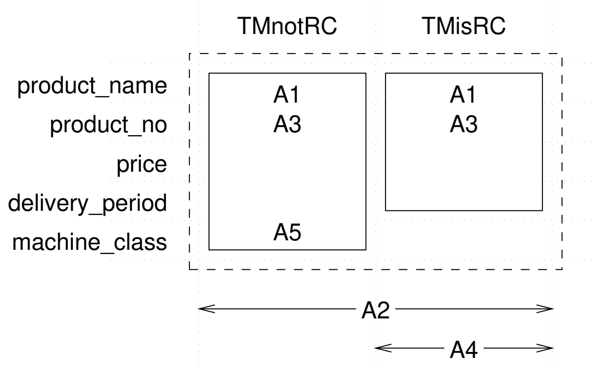


Abbildung 4.3: Verkaufs-Schema nach Auflösung des Metakonfliktes

Lösung von Attributkonflikten

Um die Klassen verschiedener Komponentenschemata zusammensetzen zu können, müssen ihre Attribute miteinander verglichen werden. Ein oder mehrere Attribute können in einem normalisierten Schema als eins betrachtet werden, wenn sie semantsich äquivalent sind. Die semantischen Äquivalenzen müssen dabei vom Administrator der Föderation festgelegt werden. Dabei können Attributkoflikte auftreten, wenn zuvor als semantisch äquivalent erklärte Attribute über nicht kompatible Wertebereiche verfügen (vgl. auch [DeM89]). Solche Konflikte können, im Falle der Existenz einer isomorphen Abbildung, durch die Definition von Wertepaaren in Form von "lookup"-Tabellen gelöst werden. Eine andere Möglichkeit ist die Nutzung von prozeduralen Abbildungsfunktionen. Falls keine isomorphe Abbildung existiert, kann mitunter der Attributkonflikt durch das Einführen neuer Integritätsbedingungen in einen Integritätsbedingungskonflikt überführt werden.

In unserem Beispiel nehmen wir an, daß alle Attribute, die denselben Namen haben, semantisch äquivalent und kompatibel sind. Die einzige Ausnahme ist die unterschiedliche Stringlänge des Attributes `product_name`. Weil die maximale Stringlänge im Schema der Verkaufsdatenbank auf 20 Zeichen beschränkt ist, während sie in der Konstruktionsdatenbank bis zu 40 Zeichen betragen kann, sind die Wertebereiche dieser Attribute nicht kompatibel und es existiert keine isomorphe Abbildung. Dieser Attributkonflikt wurde bereits bei der Transformation in GIM durch Einführung der Integritätsbedingungen `A1` und `B1` in einen Integritätsbedingungskonflikt überführt. Auf die Behandlung dieses Konfliktes wird später bei der Ableitung des objektorientierten integrierten Schemas eingegangen.

Zusammenfassung der transformierten Klassen in einem Diagramm
Nach Lösung vorhandener Attributkonflikte können die Klassen vertikal (intensional) in einem GIM-Diagramm angeordnet werden. Aus den extensionalen Beziehungen zwischen den lokalen Klassen (vgl. Abb. 4.4) und den bis zu dieser Stelle ausgeführten Restrukturierungsoperationen können die extensionalen Beziehungen der transformierten Klassen automatisch abgeleitet werden. Das Ergebnis ist in Abbildung 4.4 dargestellt.

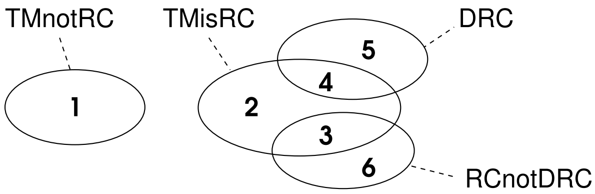


Abbildung 4.4: Extensionen der GIM-Klassen

Nun haben wir einen Punkt erreicht, an dem die Klassen beider Schemata gemeinsam in einem GIM-Diagramm dargestellt werden können(vgl. Abb. 4.5). Um alle extensionalen Überlappungen mit der Extension von `RCnotDRC` darstellen zu könnnen, wurde diese Klasse in der Abbildung links und rechts der Extension von `DRC` dargestellt. Diese graphische Dekomposition kann nicht als Restrukturierung des Schemas betrachtet werden. Sie ist lediglich für die graphische Darstellung notwendig.

Wie wir in Abbildung 4.5 erkennen können, überlappen sich Intensionen und Extensionen der Klassen, z.B. bei den Klassen `TMisRC` und `DRC`. Da in GIM keine extensionalen Überlappunge erlaubt sind werden wir im weiteren die Auflösung von Überlappungen durchführen.

Lösung intensionaler Konflikte
Intensionale Überlappungen behindern die flexible Ableitung externer Schemata. Dieser Konflikt wird durch intensionale Dekomposition gelöst. Darunter verstehen wir die Gruppierung von Attributen derart, daß die Attribute zur gleichen Menge von Klassen gehören. Durch Anwendung dieses Kriteriums werden Äquivalenzklassen über der Menge der Attribute erzeugt. Entsprechend dieser Äquivalenzklassen können die GIM-Klassen intensional zerlegt werden. Die intensionale Dekomposition wird graphisch durch horizontale Linien veranschaulicht. Ist eine Klasse intensional zerlegt worden, so sind die Extensionen der erzeugten Klassen gleich. Die Intensionen sind jedoch disjunkt. Die intensionale Dekomposition beeinflußt in unserem Beispiel keine Integritätsbedingungen.

Lösung extensionaler Konflikte
Die Lösung extensionaler Konflikte ist wegen der komplizierteren Behandlung von Integritätsbedingungen komplexer als die Lösung intensionaler Konflikte. Um der GIM-

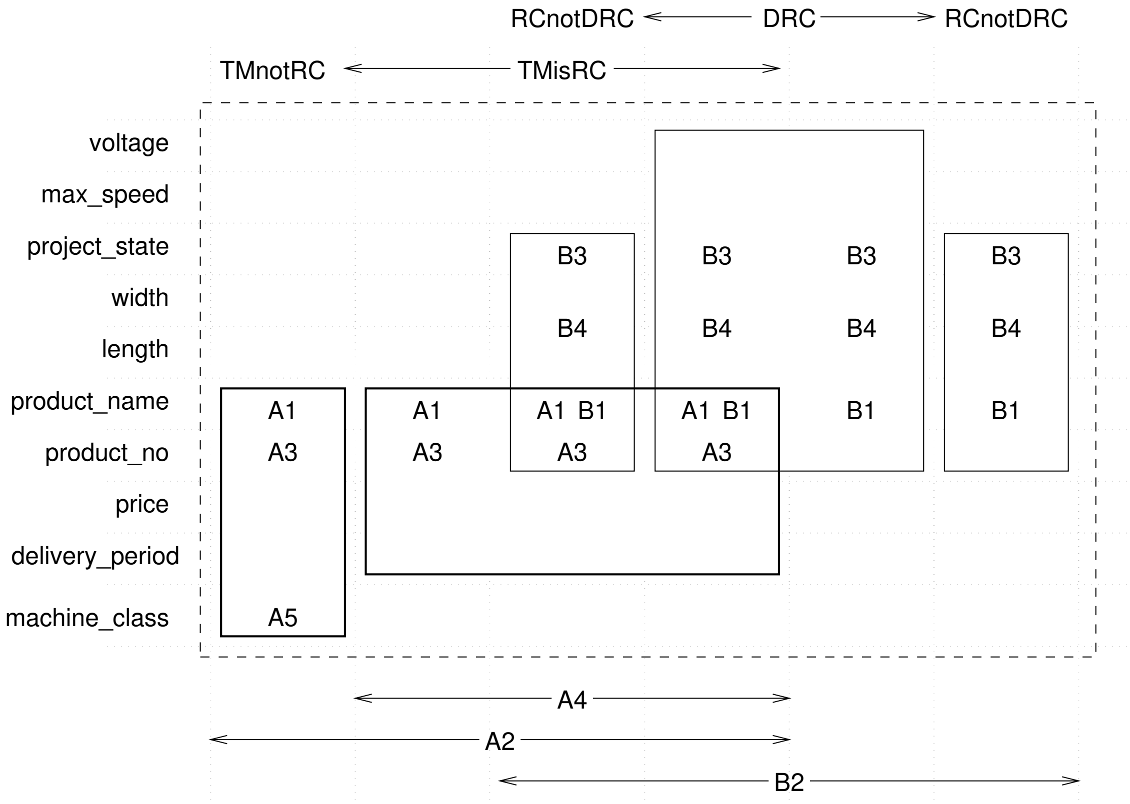


Abbildung 4.5: GIM-Klassen in einem Diagramm

Forderung nach nicht überlappenden oder identischen Klassenextensionen zu genügen, müssen die Klassen extensional in nichtüberlappende Klassen zerlegt werden. Dieser Zerlegungsprozeß ist detailiert in [SS96b, SS96c] beschrieben.

Für unser Beispielszenario ist das Ergebnis der Zerlegung in Abbildung 4.6 dargestellt. Beispielsweise wurden wegen der teilweisen Überlappung der Klassen `TMisRC` und `DRC` (vgl. Abb. 4.5) diese Klassen in die Klassen `TMisRC`, `DRCisTM` und `DRCnotTM` mit den Attributen `product_no` und `product_name` zerlegt. Den durch Zerlegung entstandenen Klassen müssen noch manuell Namen vergeben werden. Dies ist im allgemeinen eine recht schwierige Aufgabe. Deshalb benutzen wir für die Benennung Bezeichnungen, die sich an die Mengenoperationen anlehnen ("is" steht für vollständige Überlappung und "not" für Differenz).

Bei der Zerlegung von GIM-Klassen muß, wie bereits oben erwähnt, auch die Anpassung von Integritätsbedingungen vorgenommen werden. Zum Beispiel wird die "intra-object"-Bedingung `B4` der Klasse `DRC` ohne Probleme für die Extensionen der Klassen `DRCisTM` und `DRCnotTM` übernommen. Problematischer ist die Behandlung von Integritätsbedingungen bei sich (vollständig) überlappenden Klassen, d.h. für Klassen, für die sowohl in der Konstruktions-Datenbank, als auch in der Verkaufs-Datenbank Objekte existieren (können). Dies ist z.B. der Fall bei der durch Zerlegung entstandenen Klasse `DRCisTM`. Hier müssen die Integritätsbedingungen der Klassen `DRC` und `TMisRC` durch

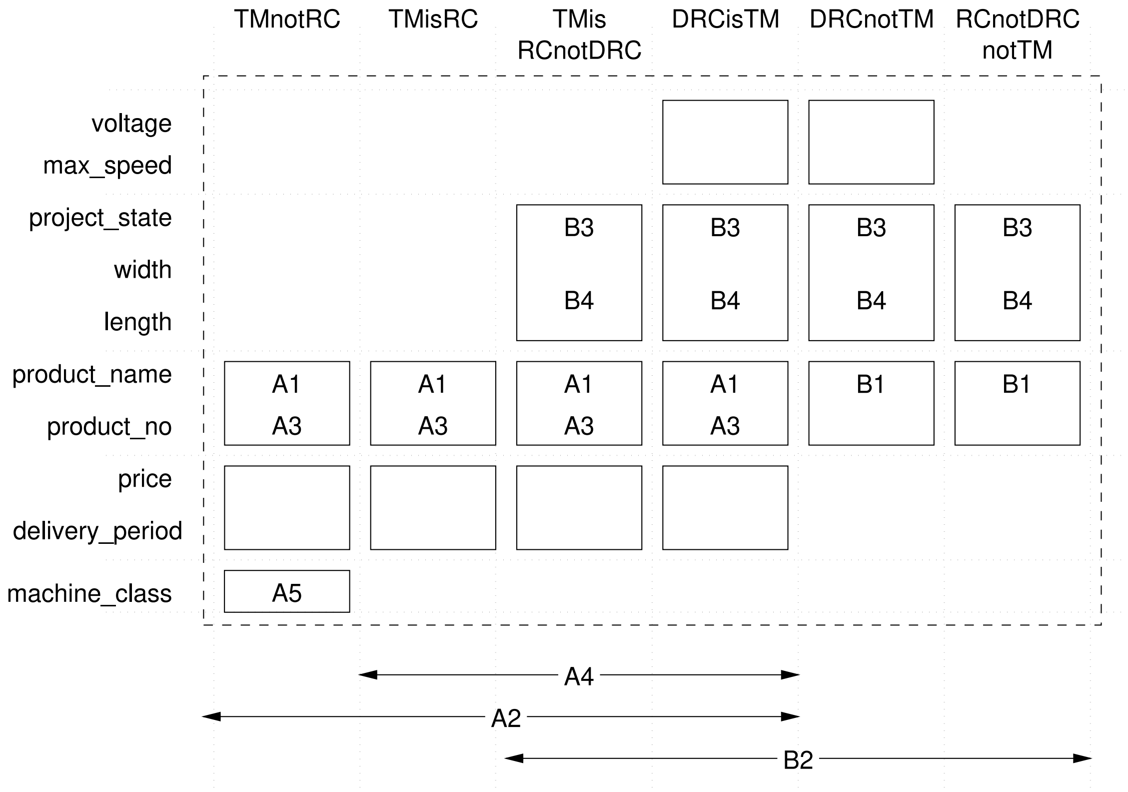


Abbildung 4.6: GIM-Schema nach der intensionalen und extensionalen Zerlegung

logische Konjunktion verknüpft werden, d.h. für das Attribut `product_no` die Integritätsbedingungen `A1` und `B1`. Da die Bedingung `A1` stärker ist als `B1`, kann `B1` weggelassen werden.

Die Klasse `DRCisTM` verbirgt Redundanz. Die Werte der Attribute `product_no` und `product_name` sind für `DRCisTM` in beiden lokalen Datenbanken gespeichert. Im allgemeinen können die Attributwerte in beiden Datenbanken unterschiedlich sein. Wie sind nun derartige Inkonsistenzen zu behandeln? In [CHJ+96] wird die Einführung neuer Integritätsbedingungen vorgeschlagen, die die Äquivalenzbedingung für alle Attribute der davon betroffenen Klassen definiert. Solche Integritätsbedingungen müssen, entsprechend der Forderung nach Transparenz, globalen Applikationen verborgen bleiben. Um derartige Integritätsbedingen zu garantieren, ist im allgemeinen die Abschwächung der Forderung nach lokaler Autonomie unumgänglich. Wenn lokale Anwendungen den Zustand einer Datenbank ändern, muß diese Änderung dem FDBMS mitgeteilt werden, das die Informationen über semantisch gleiche Objekte verwaltet. Außerdem müssen alle Inkonsistenzen gelöst sein, bevor es einer globalen Anwendung erlaubt wird, Daten zu lesen. Diese Konzepte zur Überwindung der Datenheterogenität sind bei der Konzipierung der SIGMA$_{DEMO}$-Software berücksichtigt und zum größten Teil bereits in der vorliegenden Version umgesetzt worden.

Ableitung eines objektorientierten integrierten Schemas

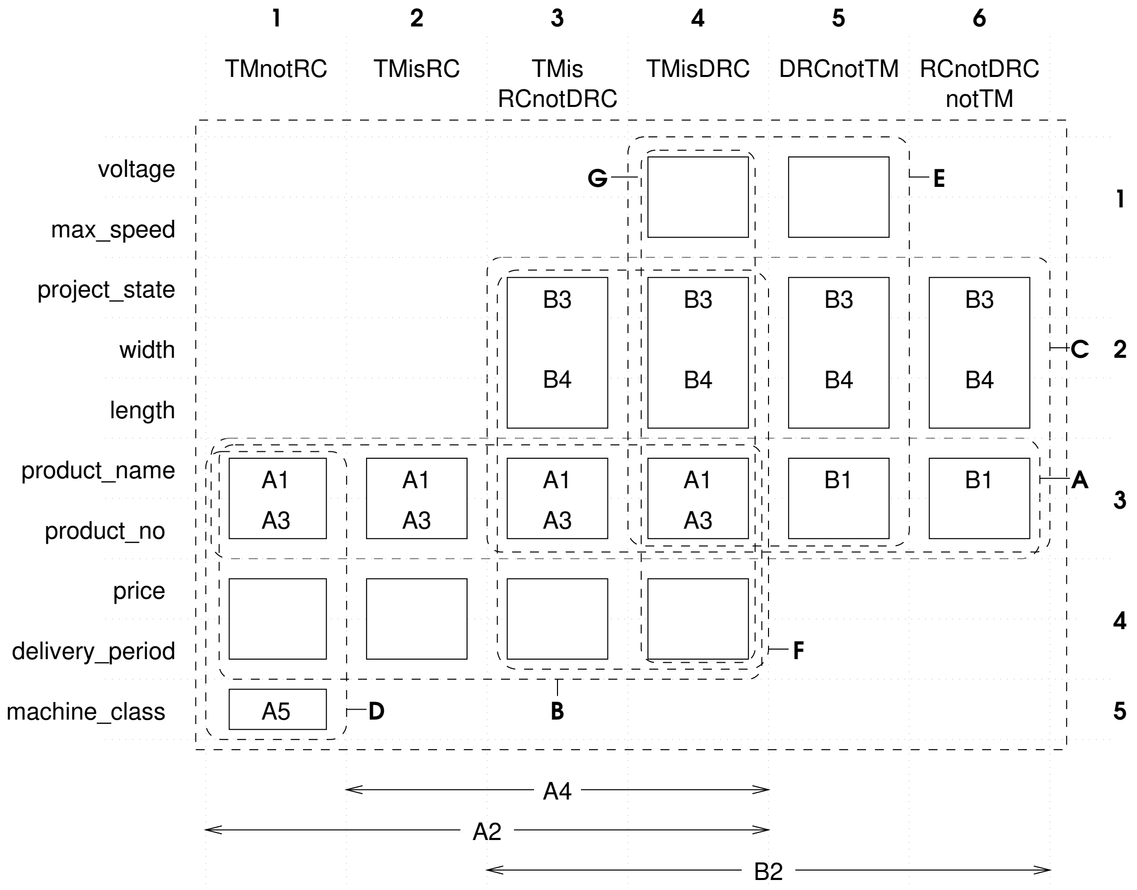


Abbildung 4.7: Ableitung eines objektorientierten integrierten Schemas

Auf Basis des normalisierten GIM-Schemas ist es nun möglich, ein objektorientiertes integriertes Schema abzuleiten, das als Schema der Föderierungsschicht dienen soll. Der Ableitungsalgorithmus ist in [SS96b] ausführlich beschrieben. An dieser Stelle wird der Ableitungsprozeß nur kurz skizziert.

Die Ableitung von objektorientierten Schemata erfolgt durch Zusammensetzung von GIM-Klassen zu größeren Klassen. Dabei kann man zwischen intensionaler und extensionaler Zusammensetzung unterscheiden. Ausschließlich Klassen mit gleicher Extension können intensional zusammengestzt werden. Falls eine "inter-class"-Integritätsbedingung zwischen Attributen der beteiligten Klassen besteht, wird diese in eine "intra-class"-Bedingung überführt.

Die extensionale Zusammensetzung ist schwieriger als die intensionale. Probleme treten bei der Komposition von Klassen, für die wiedersprüchliche Integritätsbedingungen definiert sind, auf. In [CHJ+96] wird die disjunktive Verknüpfung derartiger Integritätsbedingungen vorgeschlagen. Die dabei entstehenden Integritätsbedingungen sind schwächer als ihre Komponenten. Dadurch sind alle lokal verfügbaren Objekte an der globalen Schnittstelle nutzbar.

Da das von uns genutzte objektorientierte Datenmodell das Spezialisierungskonzept anbietet, können sich Klassen extensional und intensional überkreuzen. *Eine Superklasse überkreuzt ihre Subklasse extensional, wohingegen eine Subklasse ihre Superklasse intensional überkreuzt.* Durch Anwendung dieser Regeln lassen sich verschiedenen Vererbungshierarchien ableiten (vgl. [SS96c]).

In Abbildung 4.7 wird verdeutlicht, wie die GIM-Klassen des normalisierten Schemas zu den Klassen des objektorientierten Schemas zusammengesetzt werden. In GIM-Diagramm wird jede durch Zusammensetzung entstehende Klasse durch gestrichelte Linien gekennzeichnet. Anhand der Regeln zum Ableiten von Vererbungshierachien erhält man das in Abbildung 4.8 dargestellte Schema.

Der Tabelle 4.2 kann man Name, Intensionen und Extensionen jeder abgeleiteter Klasse sowie die zugehörigen Integritätsbedingungen entsprechend dem in Abbildung 4.7 dargestellten Ableitungsprozesses entnehmen.

Abgeleitete Klassen		*Extensionen*	*Intensionen*	*Integritätsbedingungen*
A	**Product/AC**	1,2,3,4,5,6	4	X1
B	**TM**	1,2,3,4	4,5	X2,X3,X4
C	**RC**	3,4,5,6	3,4	X1,X5,X6,X7
D	**TMnotRC**	1	4,5,6	X2,X3,X4,X8
E	**DRC**	4,5	2,3,4	X1,X5,X6
F	**TMisRC**	3,4	3,4,5	X2,X3,X5,X6
G	**TMisDRC**	4	2,3,4,5	X2,X3,X5,X6

Tabelle 4.2: Abgeleitete Klassen und Integritätsbedingungen

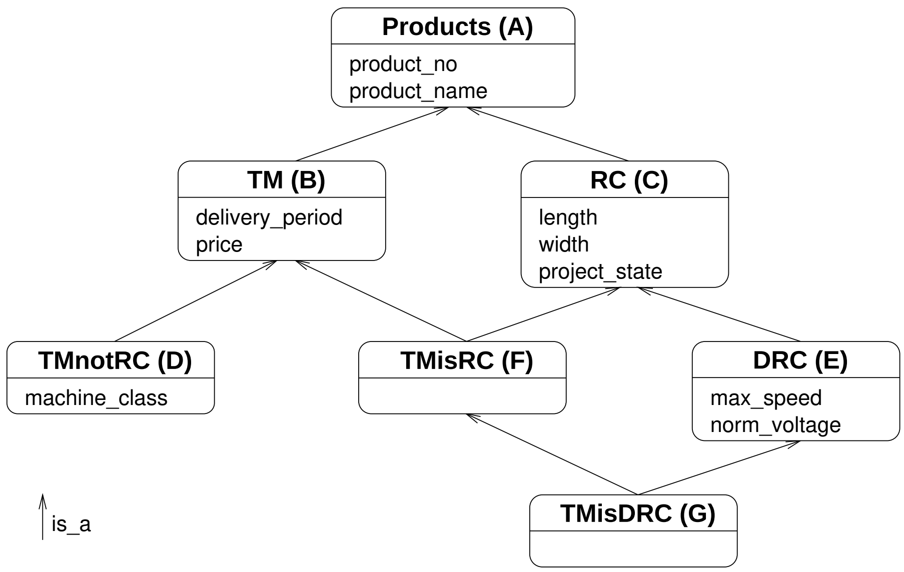


Abbildung 4.8: Abgeleitetes objektorientiertes Schema

Die Tabelle 4.3 beschreibt die Bedeutung der auf globaler Ebene definierten Integritätsbedingungen.

No	Attribute	Extensionen	Beschreibung	Ursprung	Bemerkung
X1	product_name	Product/AC	length()<40	A1∨B4	intra-object
X2	product_name	TM	length()<20	A1	intra-object
X3	product_no	TM	not null	A3	intra-object
X4	product_no	TM	unique	A2	intra-class
X5	project_state	RC	in_design⇒in_construction⇒completed	B6	intra-object
X6	length, width	RC	length<15∗width	B7	intra-object
X7	product_no	RC	unique	B5	intra-class
X8	machine_class	TMnotRC	≠'roller conveyor'	A5	intra-object

Tabelle 4.3: Integritätsbedingungen des integrierten Schemas

Somit haben wir ein integriertes Schema abgeleitet, das auf einem objektorientierten Datenmodell ohne Rollenkonzept aber mit Mehrfachvererbung basiert. Die Extensionen und Intensionen werden nun im Einzelnen kurz erläutert.

AC: (Produkt) - eine abstrakte Klasse. Die Intension dieser Klasse enthält die bei jedem Produkt vorhandene Produktnummer und eine Produktbezeichnung.

RC: Die Extension dieser Klasse enthält alle Konstruktionsdaten von Rollenförderern, zu denen keine Einträge in der Verkaufsdatenbank existieren.

DRC: Die Extension dieser Klasse besteht aus allen angetriebenen Rollenförderern, zu denen nur Konstruktionsdaten verwaltet werden.

TM: Hier finden sich die Transportmaschinen aus der Verkaufsdatenbank wieder, aus deren Maschinenklasse ersichtlich ist, daß sie Rollenförderer sind.

TMnotRC: Die Extension dieser Klasse enthält also alle Transportmaschinen, die *nur* in der Verkaufsdatenbank verwaltet werden können.

TM_RC: In der Extension dieser Klasse befinden sich alle die Rollenförderer ohne Antrieb, zu welchen Konstruktionsdaten *und* Verkaufsdaten verfügbar sind. Implizit ergibt sich, daß die Maschinenklasse eines solchen Objektes immer den Wert `RC` hat.

TMisDRC: Dieser Klasse ist der vorangegengen Klasse sowohl bezüglich extension als auch Intension recht ähnlich. Der Unterschied ist nur, daß hier angetriebene Rollenförderer verwaltet werden.

Basierend auf diesem objektorientierten Schema wurde die nun die Implementation vorgenommen. Die oben beschriebenen Schemaklassen sind in der Föderierungsschicht der Demonstrationssoftware wiederzufinden. Ihre Funktion wird bei der Benutzung des im nächsten Kapitel beschriebenen Admintools demonstriert.

Kapitel 5

Implementierung der SIGMA$_{DEMO}$

Dieses Kapitel beschreibt die Softwarekomponenten von SIGMA$_{DEMO}$. Alle Programme wurden in C++ auf Sun-Workstations geschrieben, mit Ausnahme des Teils, der embedded SQL verwendet und daher in C programmiert ist. Zunächst wird eine Übersicht über die einzelnen Komponenten gegeben (vgl. auch Abb. 5.1):

- lokale Informationssysteme der Abteilungen:
  - Informationssystem der Konstruktionsabteilung: Verwendung eines objektorientierten Datenbanksystems zur Speicherung der Produkt- und Projektdaten mit einer eigenen lokalen Datenbankanwendung.
  - Informationssystem der Verkaufsabteilung: Verwaltung der Verkaufsdaten unter Verwendung eines RDBSes mittels einer eigens dafür geschriebenen Applikation, wobei die Benutzung einer vorhandenen SQL-Schnittstelle auch möglich ist.
- die Föderierungsschicht: die Klassen des integrierten Schemas, auf deren Basis einfach und schnell globale Anwendungen erstellt werden können.
- Anwendungen mit globaler Sicht auf die Unternehmensdaten:
  - ein Administrationstool, mit dem Objekte der integrierten Schemaklassen erzeugt, angezeigt und manipuliert werden können,
  - ein Produktkatalog im WWW, basierend auf einer Datenbankanfrage mit globaler Sicht auf die Daten beider Abteilungen.

Die erste vollständig funktionsfähige Version von SIGMA$_{DEMO}$ läuft auf einer Workstation mit dem Betriebssystem Sun-OS. Als objektorientiertes Datebanksystem wurde ONTOS DB 3.0 [Ont92, Ont94] ausgewählt; das relationale Datenbanksystem ORACLE [Stü93] wurde aufgrund bereits gesammelter Erfahrung mit der embedded SQL-Schnittstelle `ProC` verwendet.

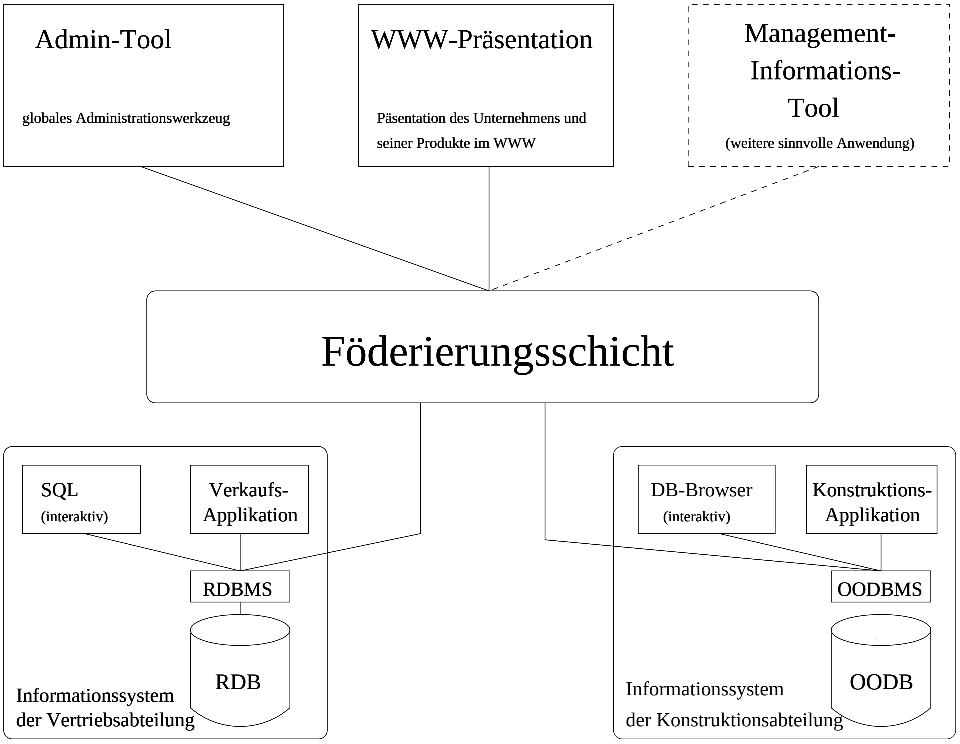


Abbildung 5.1: Übersicht über die Softwarekomponenten von SIGMA$_{DEMO}$

Die Demonstrationssoftware SIGMA$_{DEMO}$ wurde anhand der in den vorangegangenen Abschnitten behandelten Gesichtspunkte implementiert. Zunächst wurden die Datenbankmanagementsysteme ONTOS und ORACLE unter dem Betriebssystem SunOS eingesetzt. Diese Version hatte jedoch zwei entscheidene Nachteile: die ONTOS-Softwarelizenz ist an bestimmte Rechner gebunden und die Performance dieser Rechner und des eingesetzten Betriebssystem waren unzureichend. Aus diesen Gründen sollte versucht werden, statt der beiden kommerziellen Datenbankmanagementsysteme frei verfügbare DBMSe in der SIGMA$_{DEMO}$ zu integrieren. Die Wahl fiel auf Minerva SQL (MiniSQL) als relationales Datenbanksystem sowie ODE 4.1 als objektorientiertes Datenbanksystem.

Im weiteren Verlauf dieses Abschnittes wird die Implementation der lokalen und globalen Anwendungen beschrieben. Dabei wird auf alle Bestandteile einzeln eingegangen.

5.1 Lokale Informationssysteme

Wie bereits mehrfach erwähnt, gehören zur SIGMA$_{DEMO}$-Software u.a. zwei lokale Datenbanken, eine CAx-Datenbank und eine Verkaufsdatenbank, mit jeweils einer zugehörigen Datenbankanwendung. Der Aufbau dieser lokalen Anwendungen wird nun beschrieben. Auf Besonderheiten im Zusammenhang mit den verwendeten Datenbanksystemen wird hierbei in jeweils separaten Abschnitten eingegangen. Zunächst wird das Informationssystem der Konstruktionsabteilung betrachtet.

5.1.1 Das Informationssystem der Konstruktionsabteilung

In der Konstruktionsabteilung kommt ein objektorientiertes Datenbankmanagementsystem zum Einsatz. Die Generierung von Datenbankschemata in OODBMSen läßt sich nur schwer von der Erstellung von Datenbankanwendungen trennen, da die Generierung meist auf der Basis von Quelltexten erfolgt. Der folgende Abschnitt beschäftigt sich nun mit der Implementation der Schemaklassen.

Ergänzende Details zum Datenbankschema

Die (später genauer betrachtete) CAx-Datenbankanwendung wurde in C++ geschrieben, wobei die Klassendefinitionen jeweils in je einer eigenen Header- und Quelltextdatei zu finden sind. Auf der Grundlage dieser Headerdateien werden in vielen objektorientierten Datenbanksystemen die Datenbankschemata generiert.

Innerhalb des Schemas der CAx-Datenbank gibt es die beiden Klassen `Roller_Conveyor` und `Driven_Roller_Conveyor`. Deshalb gibt es hier die folgenden Dateien:

- `roller_conveyor.h` und `roller_conveyor.C`, sowie
- `driven_roller_conveyor.h` und `driven_roller_conveyor.C`.

Nachfolgend wird nun die Implementation der beiden Klassen mit ihren Methoden detailierter beschrieben, wobei auch kleine Unterschiede zu den im Abschnitt 3 vorgesehenen Klassen deutlich werden. Gründe für diese Abweichungen sind in Besonderheiten der verwendeten OODBMSe zu suchen.

1. Klasse `Roller_Conveyor`

   Die Klasse `Roller_Conveyor` wurde im wesentlichen wie bei der Betrachtung der lokalen Datenbankschemata beschrieben vereinbart. Sie wurde um eine Methode `rc_constructor()` erweitert, die bereits durch den Constructor erzeugte Objekte mit Daten füllt und eine Information darüber zurück gibt, ob diese Daten eine Integritätsbedingung verletzen oder nicht. Für den Fall, daß eine Integritätsbedingung beim Speichern des bearbeiteten Objektes in der Datenbank verletzt würde, wird ein Integerwert ungleich 0, im anderen Fall eine Null zurückgegeben. Zum schreibenden Zugriff auf Attribute, für die Integritätsbedingungen gelten, gibt es Methoden, die die Einhaltung dieser Bedingungen überwachen. Die Produktnummer kann nur durch die Verwendung der Methode `set_product_no()` verändert werden. Hierbei wird die Einhaltung der Schlüsselbedingung getestet und eine Veränderung der Produktnummer nur vorgenommen, wenn die Schlüsselbedingung erfüllt ist. In diesem Fall gibt auch diese Funktion den Integerwert 0 zurück, ansonsten einen anderen, von 0 verschiedenen Wert. Analog dazu funktioniert die Methode `changer1(length,width)`, der die Werte für `length` und `width` zu übergeben sind. Diese Werte werden nur in das Objekt geschrieben, wenn sie die Integritätsbedingung bezüglich Länge und Breite eines Rollenförderers erfüllen. Die Rückgabewerte entsprechen denen der Funktion `set_product_no()`. Der Projektstatus kann nur mit der Methode `set_project_state()` gesetzt werden. Auch hier wird sichergestellt, daß nur sinnvolle Werte eingetragen werden und daß nur erlaubte Zustandsübergänge stattfinden. Erwähnenswert ist noch, daß alle Attributwerte mit `get`-Methoden angefordert werden können.

2. Klasse `Driven_Roller_Conveyor`

   Diese Klasse wurde genauso wie bereits angemerkt implementiert und erbt alle Eigenschaften von `Roller_Conveyor`. Zusätzlich werden für die neuen Attribute `max_speed` und `motor_voltage` `set`- und `get`-Methoden implementiert. Die Methode `drc_constructor()` ersetzt die Methode `rc_constructor()` der Klasse `Roller_conveyor`.

Die Konktrete Implementation -insbesondere der Klassenmethoden- variiert jedoch infolge der Heterogenität der benutzten Datenbankmanagementsysteme. Es wurde jedoch im Zuge der Entwicklung der SIGMA$_{Demo}$-Software Wert darauf gelegt, daß eine Portabilität des Gesamtsystems erreicht werden soll. Konkret bedeutet das, daß sowohl lokale Datenbankanwendungen wie auch die Föderierungsschicht auf verschiedenen DBMSen ohne gravierende Quelltextanpassungen arbeiten sollen. Eine Lösung dieses Problems stellt die Programmierung einer Schnittstelle zwischen DBMS und Anwendung dar. Die Programmierschnittstelle zu den verwendeten OODBMSen ist Gegenstand des nächsten Abschnittes.

Die allgemeine OODB-Schnittstelle

Im vorangegangenen Abschnitt wurde bereits auf die Notwendigkeit einer Programmierschnittstelle zu den eingesetzten OODBMSen hingewiesen. Da sowohl die lokale Anwendung als auch die später noch beschriebene Föderierungsschicht auf der objektorientierten Datenbank dieselben Operationen durchführen, wurden die wichtigsten dieser Operationen in einem eigenständigen Modul mit der Bezeichnung `oocaxdb` untergebracht. Diese Schnittstelle bietet den Vorteil, daß sich die Implementation (lokaler) Anwendungsprogramme und auch der Föderierungsschicht relativ unabhängig von den unterliegenden DBMSen gestaltet. Eine solche Implementation hat den Vorteil der Portabilität. Gegenstand dieses Abschnittes ist nun der Funktionsumfang der Programmierschnittstelle zur CAx-Datenbank.

Die Funktionen zum Zugang zur Datenbank heißen `oo_login()` und `oo_logout()`. Sie benötigen keine Paramater. Zur Initialisierung der Datenbank mit den im Anhang aufgeführten Datensätzen kann die Funktion `oo_initialisieren()` aufgerufen werden. Um ein Objekt der Klassen `roller_conveyor` bzw. `driven_roller_conveyor` in der

Datenbank zu speichern, gibt es die Funktionen `oo_(d)rc_eingabe()`, die einen Zeiger auf ein entsprechendes Objekt als Parameter benötigen.

Das Ändern eines `(driven_)roller_conveyors` erfolgt, indem das betreffende Objekt zunächst mittels der Funktion `oo_(d)rc_return(char *)` aus der Datenbank geholt wird. Dieser Funktion ist die Produktnummer zu übergeben. Bei Verwendung des OODBMS ONTOS wird ein Zeiger auf das entsprechende Objekt zurückgegeben, bei Verwendung von ODE ist zu beachten, daß die betreffenden Objekte als `persistent` zu deklarieren sind. Diese Besonderheiten resultieren aus weiter unten behandelten Unterschieden zwischen den Persistenzkonzepten beider Datenbanksysteme. Mit `oo_(d)rc_aendern` können die Objekte wieder in die Datenbank zurückgeschrieben werden. Die Funktion `oo_loeschen(char *)` kann anhand der übergebenen Produktnummer ein Objekt aus der Datenbank löschen.

Zum Aufbau der Objektlisten, die innerhalb der Föderierungsschicht und auch von der lokalen Beispielanwendung zur Anzeige aller Instanzen gebraucht werden, dienen die verbleibenden drei Funktionen. Die Struktur der Objektlisten zeigt Abb. 5.2. Die hier

```
typedef struct (D)RC_list_element

  (driven_)roller_conveyor *data;
  struct (D)RC_list_element *next;
  ;
```

Abbildung 5.2: Struktur der temporären Objektlisten

besprochenen Funktionen liefern einen Zeiger auf das erste Element einer solchen Liste von Objekten. Je nachdem, welche der drei Funktionen aufgerufen wurde, sind in dieser Liste alle (nichtangetriebenen) `roller_conveyor`, alle `driven_roller_conveyor` oder alle in der Cax-Datenbank vorhandenen Objekte, letztere jedoch nur mit den Attributen von `roller_conveyor`, vorhanden. Nach der Beschreibung der Programmierschnittstelle wird nun auf die bereits mehrfach erwähnte lokale Beispielanwendung eingegangen.

Eine lokale Datenbankanwendung

In diesem Abschnitt soll die Struktur eines lokalen Anwendungsprogramms für die Konstruktionsabteilung beschrieben werden. Dazu werden die C++-Funktionen genannt, welche die aus dem im Anhang näher beschriebenen Menü aufrufbaren Funktionen bereitstellen. Die Grundstruktur des Programms mit den verschiedenen zugehörigen Quelldateien ist aus Abb. 5.3 ersichtlich. Hier ist auch zu erkennen, wie die oben beschriebene Programmierschnittstelle benutzt wird.

Der Zugriff auf Attribute eines Objektes erfolgt über Methoden, die in den Dateien `roller_conveyor.C` und `driven_roller_conveyor.C` implementiert sind. Wenn der

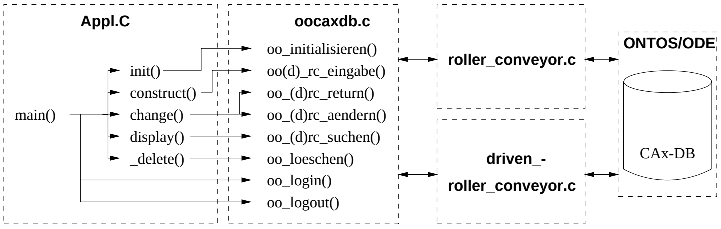


Abbildung 5.3: Struktur der lokalen Anwendung

Nutzer z.B. den Menüpunkt `Initialize database` auswählt, wird die C++-Funktion `init()` aufgerufen, die dann die Funktion `oo_initialisieren()` der Programmierschnittstelle aufruft. Die Zuordnung der in Abbildung 5.3 aufgeführten Funktionen zu den Menüpunkten sollte dem Leser keine Probleme bereiten. Mit diesen Informationen und den Headerdateien sowohl der Schnittstelle (`oocaxdb.h`) als auch der Schemaklassen sollte eine Erweiterung bzw. Nachnutzung des Programms ohne größere Schwierigkeiten möglich sein. Die DBMS-spezifischen Besonderheiten der Beispielanwendung sind Gegenstand des folgenden Abschnittes.

Implementierungsaspekte für Ode 4.1

Anwendungen für das OODBMS ODE werden mit einem erweiterten C++, das O++ genannt wird. geschrieben. Der zum Übersetzen von O++ Programmen mitgelieferte (Prä)Compiler trägt die Bezeichnung `OO` und erzeugt C++-Code, der dann mit dem Compiler `CC` weiterverarbeitet wird. Der Programmierer merkt davon i.d.R. nichts, er arbeitet mit dem `OO` genauso wie bei der reinen C++-Programmierung mit dem Compiler `CC`. Werden jedoch Datenbankklassen übersetzt, generiert der Precompiler automatisch das Datenbankschema. Treten hierbei Fehler auf, werden Hinweise gegeben, die die Beseitigung der Fehler ermöglichen. Die Besonderheit dieser Hinweise liegt darin, daß die Fehler durch Anweisungen innerhalb eines durch den Programmierer selbst zu erstellenden O++-Programmes behoben werden müssen.

In OODBSen gibt es verschiedene Persistenzkonzepte. Die zur Realisierung der SIGMA$_{DEMO}$ benutzten OODMSe unterschieden sich in diesem Punkt betrachtlich, weshalb nun zunächst Ausführungen zum Persistenzkonzept des OODBMS ODE erfolgen. Sollen Objekte einer Datenbankklasse persistent gemacht werden können, so ist der Klassendefinition das Schlüsselwort `persistent` voranzustellen. Es muß also bereits zum Erzeugungzeitpunkt eines Objektes entschieden werden, ob es persistent sein soll. Daß die soeben beschriebenen Sachverhalte unter den benutzten OODBMSen, ist aus dem jetzt folgenden Abschnitt zu Implementierungsaspekten für das OODBMS ONTOS ersichtlich.

Implementierungsaspekte für ONTOS DB 3.0

In der Sun-OS Version von SIGMA$_{DEMO}$ wurde als objektorientiertes Datenbanksystem für die Konstruktionsabteilung ONTOS-DB 3.0 ausgewählt. ONTOS-Anwendungen werden mit dem zugehörigen Compiler `CPLUS` übersetzt. Auch hier wird beim Übersetzen der Datenbankanwendungen das Datenbankschema aus Headerdateien generiert. Dieser Vorgang wird als `classify` bezeichnet. Ein weiterer Precompilerlauf schließt sich an. Was nun bei der Compilierung der lokalen Beispielanwendung und der Programmierschnittstelle geschieht, zeigt Abb.5.4.

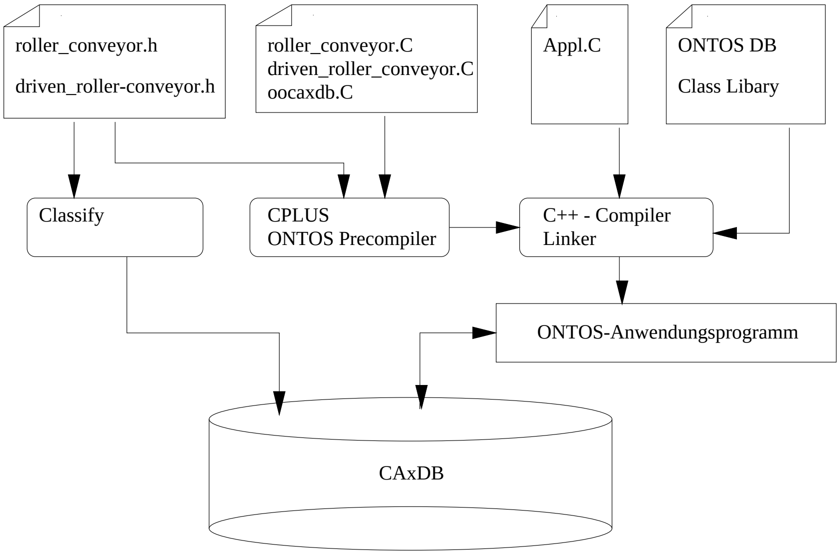


Abbildung 5.4: Erstellung der (lokalen) Anwendung unter ONTOS

ONTOS unterstützt - im Gegensatz zu dem OODBMS ODE - das Konzept der Persistenz durch Vererbung. Dies bedeutet, daß Datenbankklassen, von denen persistente, d.h. dauerhafte Objekte erzeugt werden sollen, die Persistenzeigenschaft von einer Klasse erben müssen. Diese Klasse heißt im Fall des OODBMS ONTOS `OC_Object`.

Eine weitere ONTOS-spezifische Besonderheit ist die `getDirectType()`-Funktion, welche den Objekttyp zurückliefert. Diese wird verwendet, wenn Objekte mittels Iteratoren aus der Datenbank geholt werden. Die Datenbankanfragen, die z.B. zur Erstellung der im Zusammenhang mit der Programmierschnittstelle `oocaxdb` erwähnten Objektlisten sienen, wurden i.a. mit solchen Iteratoren realisiert.

Nachdem nun das Informationssystem der Konstruktionsabteilung beschrieben wurde, soll das Informationssystem der Vertriebsabteilung betrachtet werden.

5.1.2 Das Informationssystem der Vertriebsabteilung

Die Vertriebsabteilung benutzt ein relationales Datenbanksystem. Die auch hier vorhandene lokale Beispielapplikation wurde in C implementiert. Ein Schema kann in den meisten RDBMSen durch einige Befehle innerhalb eines - bei beiden verwendeten RDBMSen vorhandenen - interaktiven Tools erstellt werden. Da dieser Prozeß bei dem in der SIGMA$_{Demo}$ verwendeten Beispielschema keine besonderen Komplikationen birgt, wird auf Ausführungen zum Datenbankschema verzichtet. Die auch in diesem Informationssystem vorhandene Programmierschnittstelle wird zunächst beschrieben. Daran anschliessend wird eine lokale Datenbankanwendung betrachtet. Implementaionsaspekte bezüglich der verwendeten RDBMSe schließen die Beschreibung der Implementaion der Software des Informationssystems der Vertriebsabteilung ab.

Die allgemeine RDB-Schnittstelle

In diesem Abschnitt werden die Funktionen der Programmierschnittstelle `oraprodukt` zum Zugriff auf die relationale Datenbank näher beschrieben. Um die Tupel der Relation TM der zugrundeliegenden Datenbank in einer C++-Anwendung in Objekte zu überführen, wurde zunächst eine Struktur mit der Bezeichnung `tm` vereinbart:

```
typedef struct tm_
  char        product_no[7];
  char        product_name[41];
  float       price;
  int         delivery_period;
  char        machine_class[31];
  struct tm_* next;
 tm ;
```

In einem solchen Datensatz können alle Daten eines Tupels der Relation TM zwischengespeichert werden. Variablen vom Typ `tm` werden in der Mehrzahl der `ora*()`-Funktionen zum Datenaustausch benutzt. Diese `ora*()`- Funktionen erledigen im einzelnen die folgenden Aufgaben:

- Datensätze suchen:

  ```
  int orasuchen(int what,
                char * _product_no,
                tm **tm_return);
  ```

  Mit dem Parameter `what` wird bestimmt, nach welchen Datensätzen gesucht werden soll. Er kann folgende Werte annehmen:

  – `PR_SUCH_PNO`
    Über den Parameter `_product_no` wird ein Datensatz mit einer bestimmten Produktnummer gesucht.

- `PR_SUCH_ALL`
  Alle Datensätze werden zuruückgegeben.
- `PR_SUCH_ALL_TM`
  Alle Datensätze mit der Maschinenklassifikation RC werden gesucht.
- `PR_SUCH_ALL_TM_DRC`
  Alle Datensätze mit der Maschinenklassifikation DRC werden gesucht.
- `PR_SUCH_ALL_TMnotRC`
  Alle Datensätze mit einer Maschinenklassifikation ungleich RC werden gesucht.

Der Paramter `tm_return` liefert einen Zeiger auf eine Liste von gefundenen `tm`-Tupeln zurück.

- Datensätze eingeben:

```
int  oraeingabe(tm tm_neu);
```

Der Parameter `tm_neu` enthält ein neu einzufügendes Tupel. In dieser Funktion wird die Schlüsselbedingung überprüft und gegebenenfalls das Einfügen des Tupels zurückgewiesen.

- Datensätze ändern:

```
int oraaendern(char* _product_no,
               tm tmae);
```

Das Tupel mit der Produktnummer `_product_no` wird mit den Daten von *tmae* überschrieben. Auch beim Ändern wird die Schlüsselbedingung überpüft und gegebenenfalls ein Änderungswunsch zurückgewiesen.

- Datensätze löschen:

```
int oraloeschen(char* _product_no);
```

Das Tupel mit der Produktnummer `_product_no` wird gelöscht.

- Initialisieren der Datenbank:

```
int orainitialisieren(void);
```

Hier werden alle Tupel aus der Relation TM gelöscht und die für die Demonstration notwendigen Tupel in die Relation TM eingefügt.

- Datensätze zählen:

```
int oracount(void);
```

  In dieser Funktion werden alle Datensätze der Relation TM gezählt und die Anzahl wird ausgegeben.

- Login in die Datenbank:

```
int oralogin(void);
```

- Logout aus der Datenbank:

```
int oralogout(void);
```

Auch diese Funktionen werden sowohl durch lokale Anwendungen als auch von der Föderierungsschicht aufgerufen. Eine solche lokale Datenbankanwendung ist Gegenstand des nun folgenden Abschnittes.

Eine lokale Datenbankanwendung

Die lokale DB-Anwendung `produkt` ermöglicht Zugriffe auf die Verkaufsdatenbank. Der Quelltext dieser Anwendung ist in den folgenden Dateien zu finden:

- produkt.c
- allprodukt.c und allprodukt.h

Die Abbildung 5.5 zeigt, welche Teile der Applikation in welcher Datei implementiert wurden.

Der Funktionsumfang der lokalen Verkaufsapplikation umfaßt Funktionen wie Anzeigen von Datensätzen, Eingabe von neuen Daten, Löschen und Zählen von Datensätzen. Die Menüs sind ähnlich denen der CAx-Anwendung gestaltet, wodurch sich die Bedienung der beiden lokalen Anwendungen nur wenig unterscheidet. Wird ein Menüpunkt aufgerufen, werden, sofern erforderlich, Daten von Nutzer erfragt. Dann wird die entsprechende Funktion aus dem im vorangegangenen Abschnitt genauer beschriebenen Modul aufgerufen und eventuell ermittelte Ergebnisse werden ausgegeben. Die beiden folgenden Abschnitte befassen sich mit Besonderheiten der beiden implementierrten Versionen, deren Ursache in der Heterogenität der beiden RDBMSe ORACLE und MiniSQL zu suchen sind.

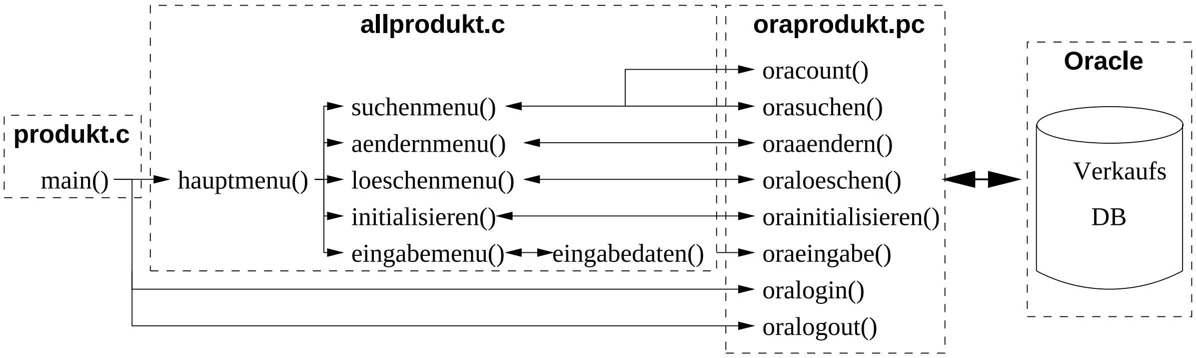


Abbildung 5.5: Struktur der lokalen Verkaufsanwendung

Implementierungsaspekte für MiniSQL

Die Version des Verkaufsapplikation für SIGMA$_{DEMO}$ unter dem Betriebssystem SOLARIS benutzt das RDBMS MiniSQL. Auf die Datenbank wird via Call-Level-Interface zugegriffen. Um aus dem bei beiden Versionen der Applikation identischen Quelltext die Interface-Funktionen aufzurufen, befinden sich die Quellen dieser Funktionen in der Datei `oraprodukt.c`. Sie stellen dieselbe Funktionalität wie der bei der ORACLE-Version in der Datei `oraprodukt.pc` implementierten Funktionen zur Verfügung.

Darüberhinaus mußte die Datenstruktur `tm` umbenannt werden, da es in einer der zum OODBS ODE gehörenden Bibliotheken eine gleichnamige Datenstruktur gibt. Dieser Konflikt verhindert jedoch die Übersetzung der Quelltexte für die Funktionen der Föderierungsschicht. Daraufhin wurde entschieden, die Struktur `tm` in `tmdata` unzubennenen, was auch Konsequenzen für die Implementation der einem späteren Abschnitt beschriebenen Föderierungsschicht hat. Welche Besonderheiten bei der Implemention der SIGMA$_{DEMO}$-Software unter Verwendung des RDBMS ORACLE zu beachten sind, beschreibt der folgende Abschnitt.

Implementierungsaspekte für ORACLE

Zur Programmierung der Interface-Funktionen zum Zugriff auf die relationale Verkaufsdatenbank mittels Embedded-SQL wurde der ORACLE-eigene Präcompiler `ProC` verwendet. Dieser Präprozessor übersetzt die Datei `oraprodukt.pc` in C-Quelltext, der in der Datei `oraprodukt.c` zu finden ist. Diese Datei muß dann mit einem C-Compiler wie z.B. `gcc` übersetzt werden. Im Gegensatz zu den Präcompilern der verwendeten OODBMSe ODE bzw. ONTOS liegt findet nur ein Vorübersetzungsprozeß statt, sodaß weitere Ausführungen nicht erforderlich sind. Es Bleibt noch zu erwähnen, daß sowohl aus dem Quellcode der lokalen Anwendung heraus als auch von der Föderierungsschicht immer auf die in der Datei `oraprodukt.c` befindlichen Funktionen zugegriffen wird. Die Föderierungsschicht ist nun Gegenstand der folgenden Abschnitte.

5.2 Die Föderierungschicht

Im diesem Abschnitt wird nun die Föderierungsschicht mit ihrem integrierten Schema und ihrer Funktionalität beschrieben. Hierzu wird jede Klasse des integrierten Schemas kurz beschrieben. Das Funktionsprinzip der Schemaklassen wird anschließend vorgestellt. Den wichtigsten Funktionen sind eigene Abschnitte gewidmet. Die Föderierungsschicht hat die Aufgabe, globalen Anwendungen die Klassen des integrierten Schemas mit ihren Intensionen und vor allem Extensionen zur Verfügung zu stellen. Die Abb. 5.6 zeigt das komplette integrierte Schema noch einmal im Überblick. Die Klassen dieses Schemas

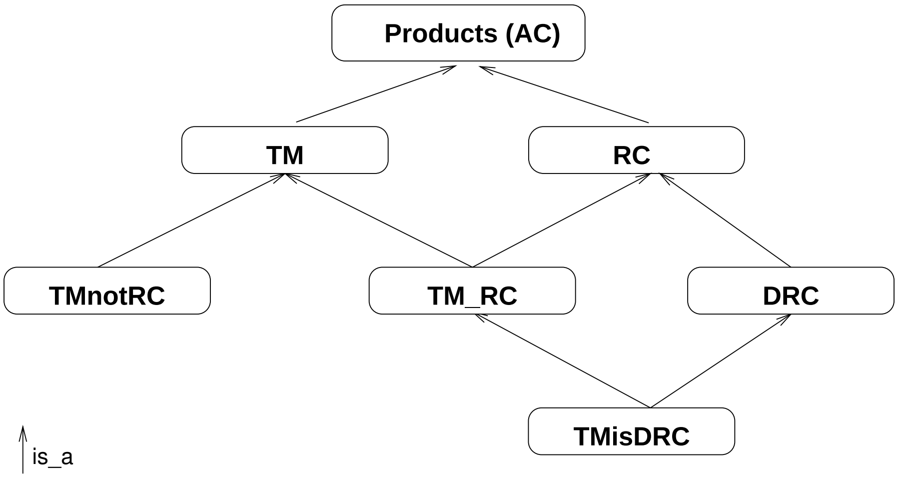


Abbildung 5.6: Das integrierte Schema der Föderierungsschicht

werden nun beschrieben:

AC: (Produkt) - eine abstrakte Klasse. Die Intension dieser Klasse enthält die bei jedem Produkt vorhandene Produktnummer und eine Produktbezeichnung. Die Produktnummer ist ein Schlüssel, d.h. es existiert hierfür eine globale Schlüsselbedigung. Die Bezeichnung eines Produktes ist auf eine Länge von 20 Zeichen begrenzt worden, wie es bereits im objektorientierten Datenbankschema der CAx-Datenbank der Fall war. Bei Manipulationen an Produktbezeichnungen bestehender Datensätze sind aufgrund der in der Verkaufsdatenbank (in der Relation `Transport_Maschine`) geforderten Großschreibung Zeichenkettenkonvertierungen notwendig.

RC: Die Extension dieser Klasse enthält alle Konstruktionsdaten von Rollenförderern, zu denen keine Einträge in der Verkaufsdatenbank existieren. In der Intension dieser Klasse finden sich neben den ererbten Attributen und Integritätsbedingungen die geometrische Daten enthaltenden Attribute Länge und Breite wieder. Die bereits im lokalen Schema der Konstruktionsdatenbank definierten Integritätbedingung für diese beiden Attribute gibt es hier auch. Der Projektstatus eines Rollenförderers mit seiner transitionalen Integritätsbedingung komplettiert die Intension dieser Klasse.

DRC: Die Extension dieser Klasse besteht aus allen angetriebenen Rollenförderern, zu denen nur Konstruktionsdaten verwaltet werden. Die Intension unterscheidet sich von der Intension der Klasse `RC` durch die neu hinzugekommenen Attribute max_speed und motor_voltage. Diese Attribute stammen von der Klasse Driven_Roller_Conveyor aus dem lokalen Schema der CAx-Datenbank. Sie beschreiben die Höchstgeschwindigkeit und die Höhe der Versorgungsspannung des Antriebs. An diese Attribute sind keine Integritätsbedingungen geknüpft.

TM: Hier finden sich die Transportmaschinen aus der Verkaufsdatenbank wieder, aus deren Maschinenklasse ersichtlich ist, daß sie Rollenförderer sind. Sie werden mit ihrem Preis, der Lieferzeit und der Maschinenklasse verwaltet. Die Produktnummer und die Bezeichnung erben sie - wie alle anderen Klassen auch - von der abstrakten Klasse AC.

TMnotRC: Die Intension dieser Klasse gleicht der der vorangegangenen Klasse, mit Ausnahme der Bedingung, daß Transportmaschinen, die aufgrund ihrer Maschinenklasse *nicht* in der Konstruktionsdatenbank vertreten sein können, in diese Klasse gehören. Das Attribut Maschinenklasse darf also *nicht* den Wert RC oder DRC enthalten. Die Extension enthält also alle Transportmaschinen, die *nur* in der Verkaufsdatenbank verwaltet werden können.

TM_RC: Die Intension dieser Klasse enthält zunächst alle Attribute und Bedingungen der Klasse TM. Alle Attribute der Klasse RC, mit Ausnahme von Produktnummer und Bezeichnung, die ja schon vorhanden sind, kommen dazu. Das gilt natürlich auch für die Integritätsbedingungen, wie die Schlüsselbedingung der Produktnummer und die Längen-Breiten-Bedingung oder die transitionale Bedingung für den Projektstatus. Folgerichtig befinden sich in der Extension dieser Klasse all die Rollenförderer ohne Antrieb, zu welchen Konstruktionsdaten *und* Verkaufsdaten verfügbar sind. Implizit ergibt sich, daß die Maschinenklasse eines solchen Objektes immer den Wert `RC` hat.

TMisDRC: Diese Klasse ist der vorangegangenen Klasse sowohl bezüglich Extension als auch Intension recht ähnlich. Der Unterschied ist nur, daß hier angetriebene Rollenförderer verwaltet werden. Die Intension enthält daher zusätzlich zu den Attributen und Bedingungen der Klasse TM_RC die von der Klasse DRC ererbten Attribute maxspeed und motor_voltage. Es ergibt sich aus den vorrangegangenen Fakten die Restriktion, daß als Maschinenklasse `DRC` eingetragen sein muß.

Der Zugriff auf die Objekte dieser Klassen erfolgt über "Meta"-Klassen, die den Präfix `class_` haben. Damit immer der aktuelle Zustand der Objekte in der globalen Anwendung sichtbar ist, werden die Metaobjekte bei Bedarf (d.h. bei globalen Operationen) dynamisch erzeugt und nach der Benutzung wieder vernichtet. Jede "Meta"-Klasse (außer `class_i_AC`, sie hat als abstrakte Klasse eine Sonderstellung) hat folgende Methoden:

- 2 Konstruktoren,

- Methoden zur Realisierung eines Scans:
  `open()`, `fetch()`,`close()`,`eoo()`,
- eine Methode zur Objekterzeugung: `create()`,
- eine Methode zum Objektlöschen: `delete()`,
- eine Methode zum Ändern: `change()`,
- einen Destruktor.

Anhand der "Meta"-Klasse `class_i_TM_RC` soll beschrieben werden, wie eine Metaklasse die oben geforderte Funktionalität mittels ihrer Methoden zur Verfügung stellen kann. Die Objekte der entsprechenden Schemaklasse werden intern in Listen verwaltet. Für jede Schemaklasse kann jeweils eine Objektliste vorhanden sein. Die Listen werden von einem Konstruktor der Metaklasse dynamisch erzeugt. Im Zuge dieses Prozesses muß diese Methode die Produktnamen aus der lokalen DB aufgrund der Datenheterogenität umwandeln. [1] Da die Extensionen der (integrierten Schema-) Klassen disjunkt sein sollen, muß bei ihrer Erzeugung eine zusätzliche Überprüfung erfolgen. Diese Überprüfung wird bei der hier betrachteten Klasse folgendermaßen durchgeführt: Der Konstruktor erzeugt jeweils die "Meta"-Klasse der Objekte, die unter der Klasse `i_TM_RC` in Richtung der Blätter des Typhierarchiebaumes stehen, d.h. von `class_i_TM_RC` erben. In diesem Fall betrifft das die Metaklasse `class_i_TM_DRC`). Durch einen Scan über der Objektliste der Klasse `class_i_TM_DRC` erkennt der Konstruktor, welche Objekte er aus seiner internen Liste streichen kann, da sie schon von (einer) anderen Metaklasse(n) verwaltet werden. Dieses Verfahren wird dann bis zu den Blättern rekursiv durchgeführt.

Ein zweiter vorhandener Konstruktor erstellt die o.g. Listen so, daß auch die Objekte aus Subklassen mit in die Liste aufgenommen werden, d.h. die Maßnahmen zur Erreichung der Disjunktheit werden nicht durchgeführt. Dieser Konstruktor darf *nur* zur Anzeige einer Klasse des integrierten Schemas mit Subklassen benutzt werden und wurde lediglich zur Vereinfachung der Programmierung der Anzeige aller Objekte einer Klasse inklusive ihrer Subklassen (für Demonstrationszwecke) geschaffen.

Zum Zugriff auf die Objekte in den Listen hat jede Metaklasse die Methoden `open()`, `fetch()`, `eoo()` und `close()`. Die Bezeichner dieser Methoden wurden in Anlehnung an das Cursor-Konzept von PL-SQL gewählt. Die Benutzung dieser Methoden wird in Abb. /refCursor an einem Beispiel demonstriert:

Dabei setzt `open()` einen internen Zeiger auf den Anfang der (internen) Objektliste. Mit `eoo()` kann festgestellt werden, ob der interne Zeiger am Ende der Objektliste steht. Die Methode `fetch()` liefert einen Zeiger auf dasselbe Objekt, auf das der interne Zeiger zeigt. Die Methode `close()` gibt es nur der Vollständigkeit halber. Dieses Iterieren über den Objektlisten wird in den globalen Anwendungen zum Zugriff auf globale Objekte durchgeführt.

[1] In der Vertriebsdatenbank wird eine Produktbezeichnung immer in Großbuchstaben dargestellt. Im Schema der Konstruktionsdatenbank existiert diese Beschränkung nicht(vgl. 3.2).

```
class_i_TM_RC *circ1;
i_TM_RC *itmrc;
citmrc1 = new class_i_TM_RC(); // Listenaufbau - Metaobjekt erzeugen
citmrc1->open(); // internen Zeiger auf Listenanfang setzen.
while(!circ1->eoo()) // Ende der Liste abfragen

  irc = citmrc1->fetch(); // Zeiger zuweisen
  // Mit dem i_TM_RC-Objekt arbeiten
  cout << strlen(itmrc->get_product_no()) << endl;
  itmrc->display(); // mal alle Attribute anzeigen lassen
  cout << endl;

citmrc1->close(); // nur pro Forma
```

Abbildung 5.7: Arbeit mit einer Objektliste

Zum globalen Einfügen eines Objektes muß die Insert-Methode auf das Metaobjekt (die interne Liste von TM_RCs also) der (Meta)-Klasse angewendet werden. Dabei muß überprüft werden, ob irgendein Objekt mit der entsprechenden Produktnummer bereits existiert. Dazu wird die Methode `lookup` (siehe hierzu auch Abschnitt 5.2) verwendet. Wurde zu der Produktnummer in keiner Datenbank ein Eintrag gefunden, kann das Einfügen fortgesetzt werden. Der Produktname wird dann gemäß den Vereinbarungen der lokalen Datenabnkschemata konvertiert (s.a. 3.2). Anschliessend wird ein Objekt zur Speicherung in der CAx-Datenabnk erzeugt und ein Einfügen in der relationalen Datenbank durchgeführt. Hierzu dienen ausschliesslich Funktionen, die durch die Programmierschnittstellen zur Verfügung gestellt werden, sowie die `set`-Methoden der objektorierten lokalen Schemas. Beim Löschen sind keine weiteren Besonderheiten zu beachten.

Um Objekte zu erzeugen, zu löschen oder zu ändern, gibt es jeweils eine Funktion, die der Metaklasse (hier wieder am Beispiel von `class_i_TM_RC`) zugeordnet ist:

- Objekterzeugung: `create (i_TM_RC* neu)`: Hier muß ein Zeiger auf ein Objekt vom Typ `i_TM_RC` übergeben werden, aus welchem die Attributwerte ausgelesen und unter Benutzung der in `oocaxdb.h` und in `oraprodukt.h`) beschriebenen Funktionen in ein neu zu erzeugendes, (bezüglich des OODBMS) persistentes Objekt kopiert werden.

- Objekt löeschen: `_delete (char* pno)`: Die Variable `pno` enthält eine Produktnummer, anhand derer das zugehörige Objekt aus den beiden lokalen Datenbanken entfernt wird.

- Update: `change (i_TM_RC* alt,i_TM_RC* neu)`: Die Variable `alt` ist ein Zeiger auf das zu verändernde globale Objekt. Die neu zu setzenden Attributwerte werden dem Objekt, auf das die Zeigervariable `neu` zeigt, entnommen. Die Variable `neu` existiert nur zum temoprären Zwischenspeichern.

Mit Hilfe dieser Ausführungen sollte die Erstellung einer globalen Anwendung möglich sein. Der folgende Abschnitt erklärt einige weitere Funktionen, mit deren Hilfe die Programmierung der Föderierungsschicht etwas erleichert wurde.

Weitere globale Funktionen

In diesem Abschnitt werden einige Methoden beschrieben, die nur bei der Klasse `class_i_AC` vorhanden sind. Eine solche globale Funktion dient zum Initialisieren der beiden lokalen Datenbanken. Diese Funktion, sie heißt `initdb()`, benötigt keine Parameter. Im SIGMA$_{DEMO}$-Admintool wird sie z.B. aufgerufen, wenn der Menüpunkt `Init DBs` ausgewählt wird. Da sie alle Objekte der (gedachten) Extension dieser Klasse betrifft, erscheint eine Zuordnung zu `classs_i_AC` als vertretbar.

Viele der im vorangegangenen Abschnitt behandelten Methoden der Metaklassen müssen feststellen, ob zu einer Produktnummer global ein Objekt existiert. Mitunter stellt sich für so ein Objekt auch die Frage, welcher Klasse es angehört. Hierfür gibt es die Funktion `int lookup(char* pnr)`. Mit ihrer Hilfe kann nun festgestellt werden, ob zu einer übergebenen Produktnummer global schon ein Objekt vorhanden ist. Diese Funktion wird zur Gewährleistung der globalen Schlüsselbedingung gebraucht. Ihr Rückgabewert gibt an, ob, und wenn ja in welcher Klasse, ein Objekt mit der entsprechenden Produktnummer existiert. Aus dem Rückgabewert lassen sich die folgenden Informationen entnehmen:

- 0: Zur angegebenen Produktnummer gehört kein Objekt.
- 1: Zur angegebenen Produktnummer gehört ein `RC`.
- 2: Zur angegebenen Produktnummer gehört ein `DRC`.
- 3: Zur angegebenen Produktnummer gehört ein `TM`.
- 4: Zur angegebenen Produktnummer gehört ein `TMnotRC`.
- 5: Zur angegebenen Produktnummer gehört ein `TM_RC`.
- 6: Zur angegebenen Produktnummer gehört ein `TMisDRC`.

Anhand dieser Rückgabewerte kann durch eine Methode einer Metaklasse eine bevorstehende Verletzung der globalen Schlüsselbedingung erkannt und verhindert werden. Darüber hinaus kann diese Funktion auch aus einer globalen Anwendung heraus direkt aufgerufen werden. Sinnvoll ist dies, wenn eine Bewegung von Objekten innerhalb der Vererbungshierarchie des integrierten Schemas mittels Löschen und anschliessendem Einfügen simuliert wird. Dieser Weg wurde bei der Implementierung des Admin-Tools gewählt. Die Implementation globaler Anwendungen, wie z.B. des o.g. Admintools, wird im nächsten Abschnitt beschrieben.

5.3 Globale Anwendungen

Nachdem nun die Föderierungsschicht mit ihrer Struktur und ihrem Funktionsumfang beschrieben wurde, sollen nun die beispielhaft erstellten globalen Anwendungen vorgestellt werden. Diese globalen Anwendungen arbeiten auschliesslich mit den Klassen des integrierten Schemas. Eine dieser Anwendungen mit einer globalen Sicht auf die Daten unseres fiktiven Beispielunternehmens ist das SIGMA$_{Demo}$-Admintool, welches im folgenden Abschnitt vorgestellt werden soll.

5.3.1 Das SIGMA$_{Demo}$-Admin-Tool

Um zu zeigen, wie mit den Klassen des integrierten Schemas gearbeitet wird, wurden globale Anwendungen geschrieben. Eine solche globale Anwendung ist z.B. ein Programm, mit dessen Hilfe alle globalen Objekte erzeugt und manipuliert werden können. Dieses Programm wurde mit einer graphischen Benutzeroberfläche ausgestattet, wie Abb. 5.8 zeigt. Struktur und Funktionsumfang dieses Programmes werden im nachfolgenden

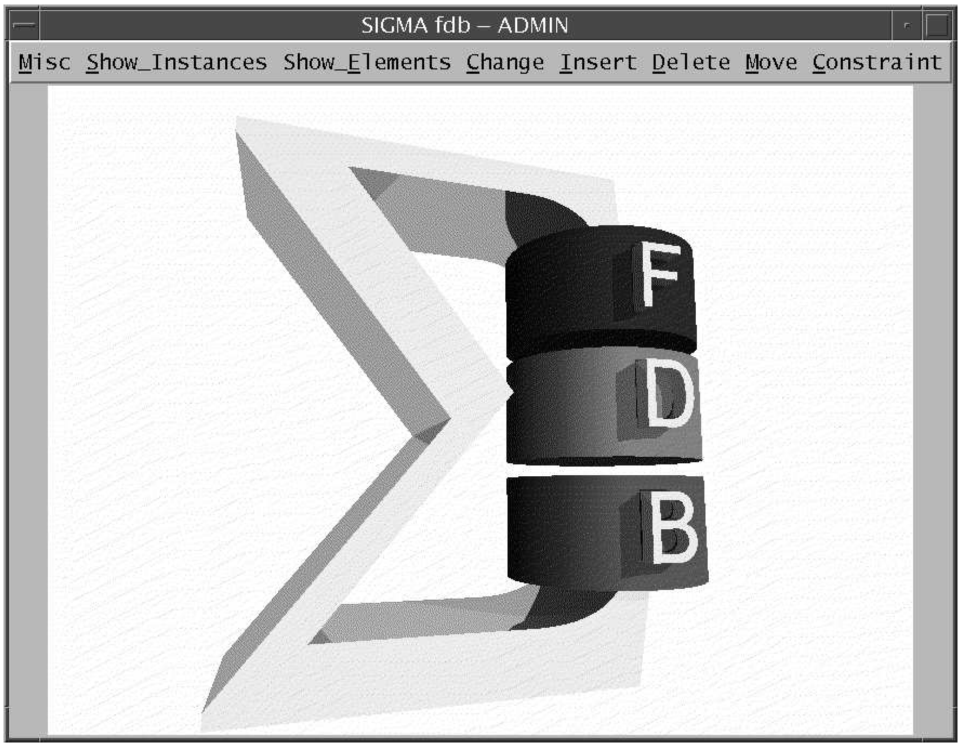


Abbildung 5.8: Das Admin-Tool im Einsatz

Abschnitt besprochen.

Die Realisierung des Admin-Tools mit dem Tk/Tcl-Toolkit

Damit Das ADmin-Tool intuitiv bedient werden kann, wurde es mit einer graphischen Benutzeroberfläche ausgestattet. Dazu wurde das Tk/Tcl-Toolkit [Ous94] verwendet. Nach dem Start durch die Eingabe von `start` wird das Menü angezeigt.

Aus diesem Menü können die folgenden Menüpunkte aufgerufen werden: *Misc*, *ShowInstances*, *ShowElements*, *Change*, *Insert*, *Delete*, *Move* und *Global Integrity Constraint*. Hinter *Misc* verbergen sich nur die Exit-Funktion zum Verlassen des Programms, sowie eine Funktion zur Initialisierung beider lokaler Datenbanken zu Demonstrationszwecken.

Mit *ShowElements* können die Elemente jeder Klasse inclusive ihrer Subklassen angezeigt werden. *ShowInstances* zeigt hingegen die Instanzen jeder Klasse (ohne Subklassen) an. Mit *Delete* können Elemente einer Klasse aus der bzw. den Datenbanken entfernt werden. Dies bedeutet, daß z.B. mit *DeleteTM_RC* die zugehörigen Einträge aus beiden Datenbanken gelöscht werden.

Durch Auswahl von *Insert* können neue Objekte in einer oder in beiden lokalen Datenbanken - abhängig von der gewünschten Klassenzugehörigkeit - eingefügt werden. Hierbei werden eventuelle Integritätsverletzungen verhindert, indem die Bedingungen global getestet werden. Soweit es nötig (oder sinnvoll) ist, werden Migrationen von Objekten in der globalen Klassenhirarchie angeboten. Soll z.B. ein `Roller_Conveyor`, der bereits in `i_RC` enthalten ist, in der Klasse `i_TM` eingefügt werden, wird die Migration zu einem `i_TM_RC` angeboten, wenn das einzufügende Objekt und das vorhandene in der Realwelt ein- und dasselbe Objekt sind.

Der Menüpunkt *Change* erlaubt die Änderung bereits vorhandener Objekte. Bezüglich Integritätssicherung gilt hier das bereits zu *Insert* Gesagte. MIt dem Menüpunkt *Move* lassen sich einzelne Objekte entlang der Vererbungshirarchie verschieben. So ist es hier z.B. möglich, durch die Migration eines `TM_RC`-Objektes zu einem `RC`-Objekt nur die Verkaufsdaten eines Rollenförderers, nicht aber seine Konstruktionsdaten zu verwerfen.

Der Menüpunkt *(Global Integrity) Constraint* dient der (geplanten) Aktivierung bzw. Deaktivierung einer ECA-Regel.

Die C++-Funktionen zu den Menüpunkten befinden sich in den Dateien `show.C`, `input.C`, `delete.C`, `change.C` sowie `rule.C`. In `show.C` befinden sich alle Funktionen der Menüpunkte *ShowInstances* und *ShowElements*. Die Datei `input.C` enthält die Insert-Funktionen. In `delete.C` stehen die Delete-Funktionen für die einzelnen Klassen, während die Datei `change.C` die Funktionen des gleichnamigen Menüpunktes enthält. In `rule.C` soll die Aktivierung der im Szenario (vgl. hierzu Anhang D) beschriebenen globalen Integritätsregeln bzw. deren Deaktivierung implementiert werden.

Implementation einer Funktion

Die konkrete Implementation einer Funktion wird am Beispiel der Funktion `CreateDialogRC(..)` beschrieben. Auf Unterschiede zu den anderen Funktionen wird hierzu bisweilen aufmerksam gemacht. Der Kopf und die Variablendeklaration der besprochenen Funktion sehen folgendermaßen aus:

```
int InputDialogDRC(...,Tcl_Interp *interp,int argc,char *argv[])

  static zustand=-1;
...
  char   parameter1[100];
```

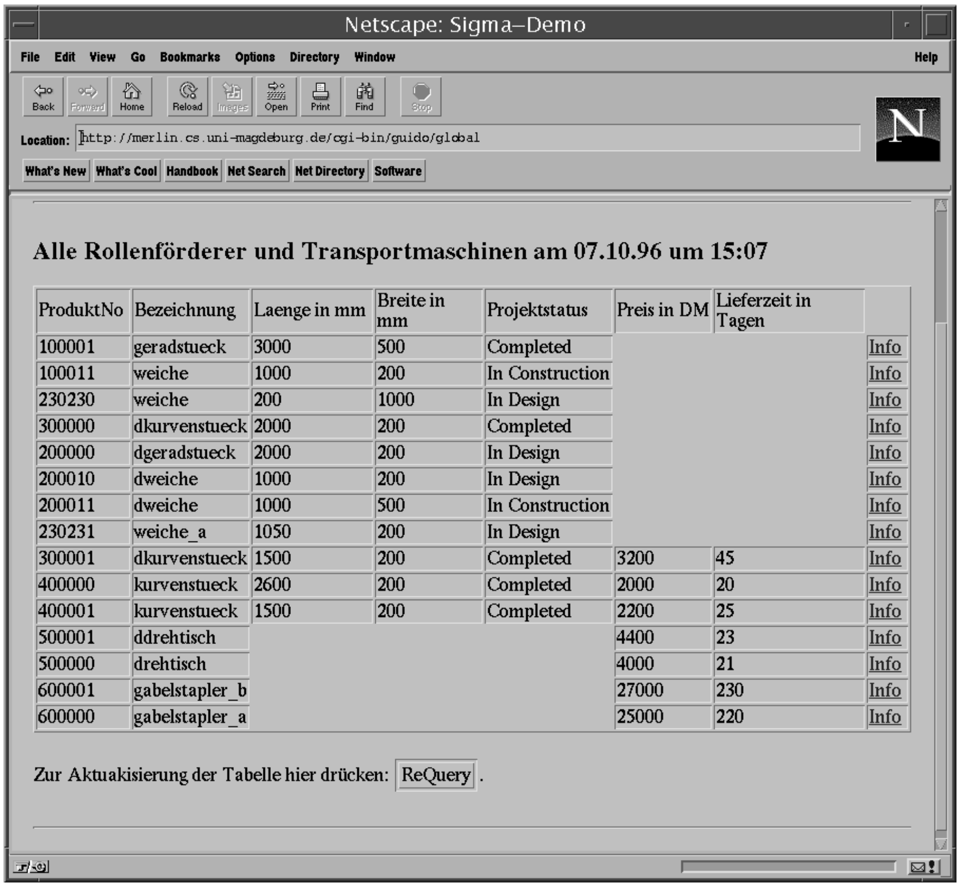
In der Variable `zustand` wird der interne Zustand der Funktion gespeichert. Bei einer Aktion, wie z.B die Auswahl eines Buttons, wird die Funktion aus sich selber hinaus erneut aufgerufen, weshalb diese Zustandsvariable als `static` deklariert ist. Der Initialzustand ist immer -1. In diesem Zustand wird hier das Eingabefenster erzeugt. Wird nun einer der beiden Buttons ausgewählt, erfolgt ein erneuter Aufruf der Funktion, die sich jetzt im Zustand 1 befindet. Bei Auswahl des [Ok]-Buttons werden nun die Tcl-Variablen ausgelesen und ausgewertet und, wenn kein Fehler auftritt, in der Datenbank gespeichert. In diesem Fall wird der Zustand auf -1 gesetzt und die Funktion wieder verlassen. Im Fehlerfall wird ein Fenster mit der entsprechenden Meldung angezeigt und ein anderer Zustand vergeben. Der Ablauf beim Betätigen der nun sichbaren Buttons ist bereits bekannt.

Die Variablen `parameter1` bzw. `parameter2` werden zum Aufruf der Tcl-Interpreterfunktionen gebraucht. Zu weiteren Details sei auf den Quelltext der Funktion `ChangeDialogRC` in der Datei `change.C` verwiesen, der ausführlich kommentiert wurde.

Eine völlig andere globale Applikation entsteht, wenn mittels eines Web-Browsers aktuelle Datenbankanfragen durchgeführt werden sollen. Eine solche Anwendung, die einen stets aktuellen Produktkatalog unseres fiktiven unternehmens bereitstellt, beschreibt der folgende Abschnitt.

5.3.2 Ein Produktkatalog im WWW

Eine weitere Anwendung mit einer globalen Sicht auf die Daten eines Unternehmens entsteht, wenn unseser Beispielunternehmens einen Katalog seiner Produkte im WWW anbietet. Ein solcher katalog spiegelt idealerweise den aktuellen Datenbestand der Datenbanken des Unternehmens wieder. Dazu wurde eine Webseite erstellt, von der aus eine Datenbankanfrage über alle Produkte mit ihren technischen Daten und Lieferbedingungen durchgeführt werden kann. Das Ergebnis einer solchen Anfrage zeigt Abb. 5.9. Zu jedem Produkt kann dann ein Datenblatt generiert werden, welches u.U. noch weitergehende Daten als das Ergebnis der generellen globalen Anfrage enthält. Um dies zu erreichen, müssen WWW-Seiten dynamisch generiert werden. Hierzu wurden CGI-Skripte verwendet. Dabei handelt es sich um Programme, die aus einer WWW-Seite heraus aufgerufen werden und die Daten von einem WWW-Browser übernehmen können. Die CGI-Skripte können dann dem Web-Server Daten z.B. in Form von HTML-Code wieder übergeben, sodaß sie vom Web-Browser dargestellt werden können.



Alle Rollenförderer und Transportmaschinen am 07.10.96 um 15:07

ProduktNo	Bezeichnung	Laenge in mm	Breite in mm	Projektstatus	Preis in DM	Lieferzeit in Tagen	
100001	geradstueck	3000	500	Completed			Info
100011	weiche	1000	200	In Construction			Info
230230	weiche	200	1000	In Design			Info
300000	dkurvenstueck	2000	200	Completed			Info
200000	dgeradstueck	2000	200	In Design			Info
200010	dweiche	1000	200	In Design			Info
200011	dweiche	1000	500	In Construction			Info
230231	weiche_a	1050	200	In Design			Info
300001	dkurvenstueck	1500	200	Completed	3200	45	Info
400000	kurvenstueck	2600	200	Completed	2000	20	Info
400001	kurvenstueck	1500	200	Completed	2200	25	Info
500001	ddrehtisch				4400	23	Info
500000	drehtisch				4000	21	Info
600001	gabelstapler_b				27000	230	Info
600000	gabelstapler_a				25000	220	Info

Zur Aktuakisierung der Tabelle hier drücken: ReQuery .

Abbildung 5.9: Eine globale Datenbankanfrage

Die zur Ausführung einer (globalen) Datenbankfrage nötigen Programme entstanden in C++ und benutzen - wie das Admintool - nur die Klassen des integrierten Schemas der Föderierungsschicht. [2] Zur Realisierung der globalen Anfrage dient das Programm `global`, welches:

- einen gültigen http-Header sendet,
- die Environmentvariablen `HOME` und `ODE_DEFAULT_AREA` setzt,
- den Kopf der HTML-Seite ausgibt,
- alle Klassen des integrierten Schemas durchiteriert,
- daraus eine HTML-3-Tabelle generiert,
- und den restlichen Code der Seite ausgibt.

[2] Bei der Ausführung eines CGI-Skriptes ist besonders zu beachten, daß evtl. zur Verwendung einer Datenbankschnittstelle benötigte Environmentvariablen durch das Programm selbst zu setzen sind. Insbesondere betrifft dies das hier verwendete objektorientierte Datenbanksystem ODE.

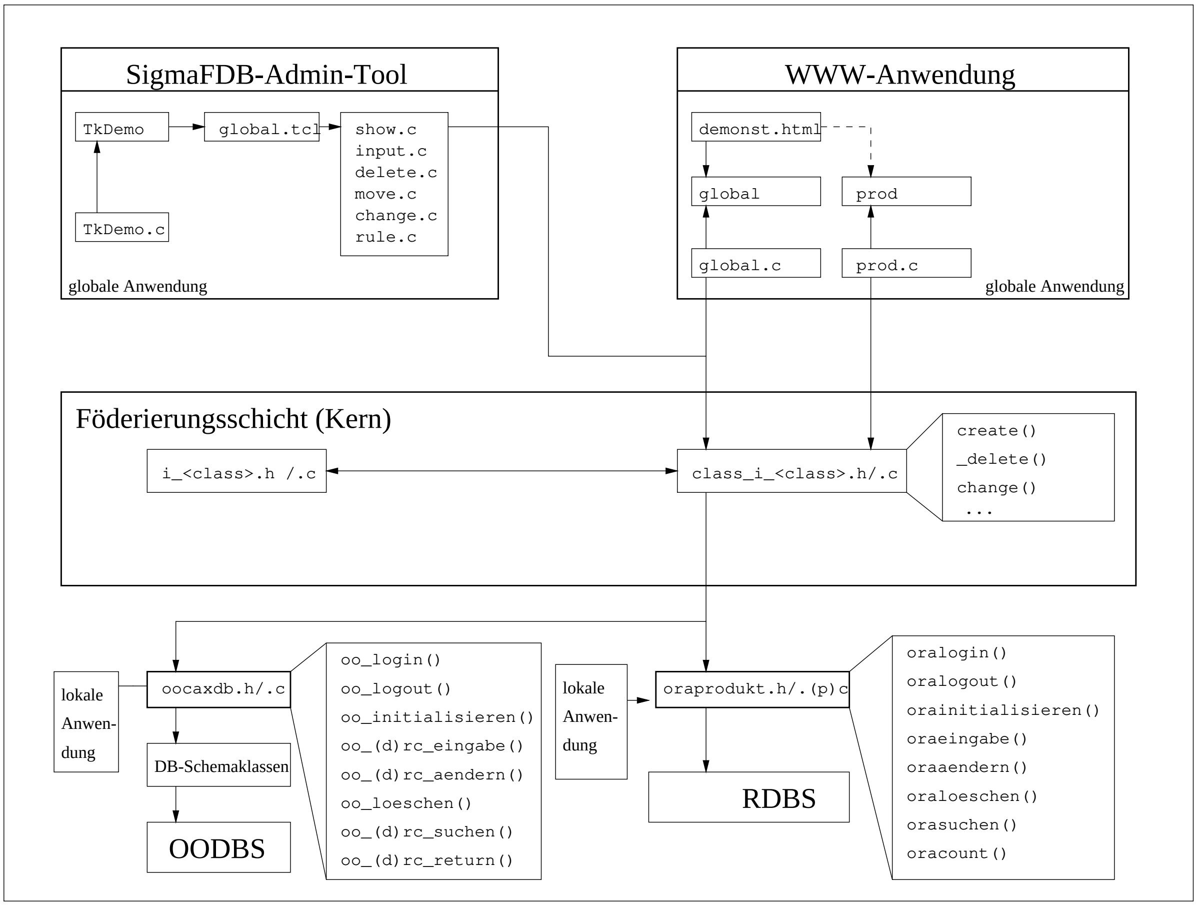


Abbildung 5.10: Überblick über alle Module

Die vom Web-Browser aufgebaute Tabelle enthält in jeder Zeile einen Link auf ein Datenblatt, welches von `prod` generiert wird. Dieser Skript bekommt dazu eine Produktnummer als Parameter (d.h. ein Query String in HTML) übergeben und holt alle zugehörigen Daten aus den Datenbanken. Eventuell vorhandene Abbildungen werden mit in die generierte Seite eingebunden. Die Funktionsweise von `prod` entspricht in etwa der von `global`.

5.4 Zusammenfassung

Abschliessend soll eine Übersicht über die wichtigsten Quelltexte gegeben werden, um ihr Zusammenwirken zu erläutern. Der Abb. 5.10 können folgende Sachverhalte entnommen werden:

- Programme mit einer globalen Sicht auf die Daten (wie das Admin-Tool) benutzen nur die Metaklassen des integrierten Schemas (`class_i_RC` usw.).

- Die Funktionen der Metaklassen greifen auf die lokalen Datenbanksysteme nur mittels der Module `oocaxdb` sowie `oraprodukt` zu.
- Die lokalen Anwendungen der Vertriebs- und Konstruktionsabteilung benutzen zum Zugriff auf die jeweilige Datenbank die Funktionen derselben Module. Dies ist nicht zwingend nötig, erleichterte jedoch die durchgeführte Portierung von SIGMA$_{DEMO}$.

Einen genauen Aufschluß über den Funktionsumfang eines jeden Moduls liefert gegebenenfalls die zugehörige Headerdatei, sofern dies nicht in den entsprechenden Kapiteln beschriebenen wurde.

Kapitel 6

Zusammenfassung und Ausblick

Im vorliegendem Bericht haben wir einen detaillierten Überblick über den FDBS-Demonstrationsprototypen **SIGMA**$_{DEMO}$ gegeben. Nachdem wir eingangs die durchgeführten Arbeiten kurz motiviert haben, sind wir im weiteren auf wesentliche Grundlagen föderierter Datenbanksysteme eingegangen. Danach erfolgte in Kapitel 3 einer Beschreibung des Anwendungsszenarios die Vorstellung der zu integrierenden lokalen Informationssysteme. Es handelt sich dabei um ein Beispiel, das dem Förderanlagenbau entnommen wurde und die Integration zweier Informationssysteme beschreibt. Als zugrundeliegende Datenbanksysteme kommen dabei ein relationales und ein objektorientiertes Datenbanksystem zum Einsatz. Die Integration der Schemata der Komponentensysteme erfolgte auf Basis der GIM-Methodik, deren Anwendung im Kapitel 4 anhand des Anwendungsszenarios vorgestellt wurde. Bei der Diskussion der Schritte Schematransformation, Schemaintegration und Ableitung eines objektorientierten integrierten Schemas wurde insbesondere auf die Entwicklung von Integritätsbedingungen während des FDBS-Entwurfs eingegangen. Danach erfolgte eine detaillierte Schilderung von Aspekten der Implementation des Demonstrationsprototypen. Sowohl die lokalen Systeme, bestehend aus Datenbank und Anwendungsprogrammen, als auch die globale Schicht und die darauf aufsetzenden Anwendungen wurden allgemein beschrieben. Auf Details, die für die unterschiedlichen Datenbanksysteme (Oracle6 vs. MiniSQL und Ontos vs. ODE) von wesentlicher Bedeutung sind, wurde kurz eingegangen. Weitere Implementierungsdetails und eine Beispielsitzung sind im Anhang beschrieben.

Wir können konstatieren, daß uns mit der vorliegenden Version der **SIGMA**$_{DEMO}$ ein leistungsfähiges Mittel zur Verfügung steht, um Möglichkeiten und Probleme föderierter Datenbanksysteme zu veranschaulichen. Nach Vorführungen des Demonstrationsprototypen kommt es immer wieder zu interessanten Diskussionen, sowohl mit Informatikern als auch mit Anwendern, z.B. Maschinenbauern. Ergebnis sind, neben einem tieferen Verständnis der Interessenten für die FDBS-Anwendung, auch neue inhaltliche Anregungen für die weitere Forschung im Rahmen des Projektes **SIGMA**$_{FDB}$.

Aufbauend auf der aktuellen Version der **SIGMA**$_{DEMO}$ können weitere Forschungsideen durch prototypische Implementation validiert werden. So ist die Erprobung aktiver

Mechanismen zur systemübergreifenden Konsistenzsicherung in FDBS geplant. Auch die praktische Untersuchung einiger globaler Transaktionskonzepte für FDBS kann im Rahmen der **SIGMA**$_{DEMO}$ erfolgen. Ein weiterer wesentlicher Aspekt sind Konzepte zur Datensicherheit und zum Datenschutz, die im Rahmen von FDBS zu berücksichtigen sind.

Weiterhin ist zu untersuchen, inwieweit sich das globale objektorientierte Schema und die Zugriffsmethoden automatisch generieren lassen. Sie sollten dabei als Output einer mit den Werkzeugen der **SIGMA**$_{Bench}$ [SEHT96] (eines am ITI entwickelten Integrationstools) durchgeführten Schemaintegration erzeugt werden. Ebenfalls sinnvoll ist die semi-automatische Generierung von Sicherheitsmodellen oder ECA-Regeln für das integrierte Schema.

Abschließende möchten wir uns bei den an Entwurf und Implementierung beteiligten Studenten und Mitarbeitern Sven Janssen, Andreas Stephanik und Steffen Thorhauer bedanken. Ein Dankeschön geht auch an die Mitarbeiter des **SIGMA**$_{FDB}$ -Teams Steffan Conrad, Steffen Geschke, Gunter Saake, Ingo Schmitt und Kerstin Schwarz für die zahlreichen Anregungen und die hilfreiche Kritik.

Anhang A

Voraussetzungen für SIGMA$_{DEMO}$

A.1 Varianten von SIGMA$_{DEMO}$

Die bis jetzt entstandenen Varianten von SIGMA$_{DEMO}$ werden zunächst einmal in Tabelle A.1 zusammengefaßt. In den nachfolgenden Abschnitten dieses Anhangs wird dann nur noch von der SOLARIS oder Sun-OS-Version gesprochen:

Betriebssystem	RDBMS	OODBMS
Sun-OS 4.1.3	ORACLE 6.0	ONTOS DB 3.0
SOLARIS 2.3	MiniSQL	Ode 4.1

Tabelle A.1: Varianten von SIGMA$_{DEMO}$ und zugehörige Datenbanksysteme

A.2 Verzeichnisse und Quelltexte im Überblick

Ausgangspunkt der folgenden Übersicht sei ein Home-Pfad für SIGMA$_{DEMO}$. In diesem Verzeichnis muß ein Unterverzeichnis mit den Quelltexten liegen, welches `Sources` heißt. Folgende weitere Unterverzeichnisse gibt es dort:

- msql: Quelltexte der Verkaufsapplikation (nur SOLARIS-Version).
- oracle: Quelltexte der Verkaufsapplikation (nur Sun-OS-Version).
- ontos: Quelltexte der CAx-Anwendung (nur Sun-OS-Version).
- ode: Quelltexte der CAx-Anwendung (nur SOLARIS-Version).
- global: Alle Funktionen der Föderierungsschicht, einschließlich einer kleinen Bibliothek mit Stringkonvertierungsfunktionen (`strself.C`). Für jede Klasse bzw. Metaklasse des integrierten Schemas gibt es eine Header- und eine Quelltextdatei.

- tcl1: Quellcodes des Admin-Tools. Hier ist jedem Untermenü eine Quelltextdatei zugeordnet. Eine Ausnahme bilden die beiden Show-Menüs. Ihre Funktionen befinden sich in einer gemeinsamen Datei.
- www-appl: Die C++-Quellen für die CGI-Skripte.
- start: 2 Batchdateien (nur in der Solaris-Version vorhanden)
  - `makedemo`: ruft alle `make`s auf - Compilierung des Gesamtsystems.
  - `start`: startet beide globalen Anwendungen, ein interaktives MSQL-Tool, das Admin-Tool und einen WWW-Browser.

Jedes Unterverzeichnis enthält die zugehörigen Makefiles. In jedem Makefile ist gesondert vermerkt, wozu es gebraucht wird.

A.3 Voraussetzungen für SIGMA$_{DEMO}$ unter Sun-OS

Compiler und Precompiler:

- CPLUS, der zum Datenbanksystem ONTOS DB 3.0 gehört.
- ProC (für Oracle 6.0): Vorübersetzer für C-Programme, die embedded-SQL verwenden,
- gcc zur Weiterverarbeitung der durch o.g. Precompiler erzeugten C-Quelltexte.

Umgebungsvariablen, Nutzer und Passwort für die Datenbanksysteme:

- `TWO_TASK=T:...:demo`: der ORACLE-Connectstring,
- User: Mustermann / Passwort: Musterpasswd: die bekannten ORACLE-Standardvorgaben,
- für ONTOS DB 3.0: `setenv DBNAME demo2db`: die zu benutzende Datenbank ist anzugeben.

A.4 Compilierung des Gesamtsystems auf Sun-OS Workstations

1. ONTOS-Applikation compilieren: `Demo/Sources/ontos/make`

2. Verkaufsanwendung `produkt` (nutzt ORACLE) compilieren: `Demo/Sources/oracle/make`
3. Festlegung, für welches System Föderierungsschicht / Admin-Tool übersetzt werden soll: in Datei `Demo/Sources/global/demosys.h` eintragen, welches System ausgewählt wurde. Hier muß `#define SUN` aktiviert sein. Alle möglichen Eintragungen sind bereits vorhanden, es sollten nur Kommentare gesetzt oder entfernt werden.
4. Quellen für Föderierungsschicht compilieren: `Demo/Sources/global/make`
5. Admin-Tool übersetzen: `Demo/Sources/tcl1/make`.

Hinweise:

- Eventuell sind vor der Compilierung bereits vorhandene Objectfiles (*.o) zu löschen.
- Die WWW-Anwendung wurde nicht auf dem Betriebssystem Sun-OS erprobt. Sie ist prinzipiell aber benutzbar, wenn:
  - Makefile(s) analog ADMIN-Tool geschrieben und
  - betriebssystemabhängige Quelltextänderungen (in C++- und auch HTML-Dateien) vorgenommen werden,
  - sowie ein WWW-Server unter Sun-OS vorhanden ist.

A.5 Voraussetzungen für die Installation von SIGMA$_{DEMO}$ bei Verwendung von SOLARIS 2.X

Zum Übersetzen aller C++-Quellen, wie die lokale CAx-Anwendung, die Föderierungsschicht usw. wird ein C++-Compiler, am besten `CC` benötigt. Einige Environmentvariablen müssen gesetzt werden:

- HOME (wenn nicht da, wie im Falle der CGI-Skripte),
- ODE_DEFAULT_AREA ... dbs/ode_db_sigma.

Achtung: Da alle O++(Ode)-Programme mit der Option `-private` übersetzt werden, ist es **nicht** notwendig, den Server `odeserver` zu starten. Zu weiteren Details sei auf das ODE-Manual verwiesen. Bei Benutzung des DBSs MSQL sollte jedoch sichergestellt sein, daß `msqld`, der Serverprozeß, läuft.

A.6 Compilierung von SIGMA$_{DEMO}$ (SOLARIS-Version)

1. Festlegung, für welches System Föderierungsschicht / Admin-Tool übersetzt werden soll: in Datei `Demo/Sources/global/demosys.h` gewünschtes System eintragen.

2. alles übersetzen: `Demo/Sources/start/makedemo`: Diese Batchdatei erledigt alle Übersetzunsgaufgaben, inklusive dem Löschen vorhandener Objektfiles.

3. WWW-Applikation: Die ausführbaren Dateien (`global` und `prod`) in den `cgi-bin`-Pfad des WWW-Servers (!) kopieren. Die Ausführungsrechte für diese beiden Dateien sind freizugeben.

   Beispiel für einen CGI-Pfad: server:/.../httpd/cgi-bin/nutzer/.

4. **Achtung:** Die Anpassung der HTML-Dateien und der Quelltexte der WWW-Anwendung (HTML FORM-Tags) ist nötig, um den benutzten WWW-Server korrekt einzutragen.

Anhang B

Hinweise zur Bedienung

Bei Verwendung der SOLARIS-Version können alle für eine Demonstration benötigten Programme durch den Aufruf der Datei `start` im Startverzeichnis (vgl. Anhang A.1) gestartet werden. Diese Datei funktioniert nicht für die Sun-OS-Variante. Hierzu sind einige Pfadnamen in der Datei zu ändern. Alternativ hierzu können die Anwendungen auch einzeln gestartet werden.

B.1 Funktionsumfang der DB-Anwendung der Konstruktionsabteilung

Nach dem erfolgreichen Start der lokalen Applikation `Appl` wird das folgende Menü aktiviert, aus dem durch die Eingabe der Ziffern 0…4 sowie 9 die gewünschte Funktion ausgewählt werden kann:

```
               CAD Database: Roller Conveyor
=========================================================
Initialize database                             '0'
Contruct a new roller_conveyor                  '1'
Change an existing roller_conveyor              '2'
Show all roller_conveyors                       '3'
Delete a roller_conveyor                        '4'
Exit program                                    '9'
```

Der Nutzer muß nun entscheiden, welche Operationen er auf der Datenbank ausführen will. Initialisieren (Punkt 0) bedeutet hier, alle vorhandenen Objekte zu löschen und 10 neue Objekte mit Standardwerten zu generieren. Diese Objekte mit ihren Standardwerten sind dem Szenario im Anhang zu entnehmen.

Desweiteren kann ein neuer `(Driven_)Roller_Conveyor` erzeugt (konstruiert) werden (Punkt 1). Dabei kann der Nutzer entscheiden, ob der Rollenförderer angetrieben sein soll oder nicht (d.h. ob nicht etwa ein `Driven_Roller_Conveyor` entstehen soll).

Vorhandene Objekte können auch verändert werden (`change`, Punkt 2). Außerdem kann man durch Auswahl des Menüpunktes 4 Objekte löschen. Die Auswahl eines zu verändernden bzw. zu löschenden Rollenförders erfolgt durch Eingabe der Produktnummer.

Die Ausgabe einer Liste aller momentan erfaßten `(Driven_)Roller_Conveyor` ist auch möglich. Zu einigen Zeitpunkten der Programmabarbeitung, u.a. beim Löschen oder Speichern von Objekten werden Sicherheitsabfragen vorgenommen, die mit `y` oder `n` zu beantworten sind.

B.2 Funktionsumfang der lokalen Verkaufsanwendung

Die lokale Verkaufsapplikation der Vertriebsabteilung hat einen ähnlichen Funktionsumfang wie die CAx-Applikation. Nach dem erfolgreichen Start erscheint das folgende Menü:

```
Hauptmenu
---------

1  Anzeigen
2  Aendern
3  Einfuegen
4  Loeschen
5  Initialisieren
6  Ende
```

Die Auswahl eines Menüpunktes erfolgt durch die Eingabe der ihm voranstehenden Ziffer. Außer beim Punkt `Initialisieren` wird dann immer ein Untermenü angezeigt, daß meistens auch die Rückkehr ins Hauptmenü anbietet. So kann man nach der Auswahl des Punktes `Anzeigen` entscheiden, ob man alle, einen bestimmten, durch die Produktnummer anzugebenden Datensatz oder nur die Anzahl der gespeicherten Datensätze angezeigt bekommen soll. Bei Auswahl eines der Punkte `Aendern`, `Einfuegen` oder `Loeschen` enthält das Untermenü jeweils zwei Punkte, von denen einer ins Hauptmenü zurück führt. Eine Initialisierung wird ohne eine Anzeige am Bildschirm durchgeführt. Wurde eine Liste am Bildschirm ausgegeben, gelangt man mit der `RETURN`- bzw. `ENTER`-Taste wieder ins Hauptmenü.

Die Anwendung startet auch, wenn kein Datenbankserver bzw. Serverprozeß läuft. In diesem Fall wird eine Fehlermeldung angezeigt. Das Programm sollte daraufhin beendet werden, um z.B. den Datenbankserver `msqld` (SOLARIS-Version) zu starten. Erst dann kann die Anwendung erneut gestartet werden.

B.3 Die Bedienung des Admintools

Das SIGMA$_{DEMO}$-Admintool bietet eine graphische Benutzeroberfläche. Alle Funktionen können aus einer Menüleiste gewählt werden. Der Funktionsumfang ist bereits im Implementationskapitel beschrieben. Hier sollen nur einige ergänzende Hinweise zur Bedienung folgen. Als Bestätigungen von Eingaben werden nur die Auswahl der Buttons, z.B. [OK] oder [CANCEL] akzeptiert. Zur Benutzung einzelner Menüpunkte werden nun einige Hinweise gegeben:

- Menüpunkt *Others*: Hier findet man die Exit-Funktion zum Verlassen des Programms sowie den Menüpunkt Init DBs zum Initialisieren beider Datenbanken.
- Den Menüpunkt *Change* benutzt man wie folgt:
  1. Objekttyp wählen im (Unter-)Menü
  2. Produktnummer eingeben (vorher in das Feld klicken), dann mit OK-Button bestätigen.
  3. Attributwerte ändern, danach [OK] bzw. [Cancel] auswählen.
- Die Benutzung von *Insert* oder *Delete* ist vergleichbar mit der von *Change*.
- Beim Punkt *Move* ist folgendermaßen vorzugehen:
  1. Objekttyp wählen (z.B. *Move RC*),
  2. Produktnummer des zu verschiebenden Objektes angeben und bestätigen (mittels Mausklick),
  3. Zielklasse auswählen (Die möglichen Klassen des integrierten Schemas werden angeboten.),
  4. bei Wechsel in Subklasse die fehlenden Attribute ergänzen,
  5. bei Wechsel in Superklasse werden überflüssige Attribute gelöscht.
- Constraint: Die Benutzung aktiver Regeln zur Konsistenzsicherung wurde noch nicht realisiert.

Hinweis: Durch Auswahl des mehrfach angebotenen [Cancel]-Buttons werden z.T. unterschiedliche Reaktionen ausgelöst. In einigen Fällen kehrt man in ein eventuell noch offenes vorhergehendes Fenster zurück, manchmal wird die Aktion auch komplett abgebrochen. Letzteres ist z.B der Fall, wenn nach der Eingabe einer Produktnummer bei Delete, Change oder Move der Cancel-Button betätigt wurde.

Anhang C

Beispieldaten zur Initialisierung

Da es sich bei SIGMA$_{DEMO}$ um ein Demonstrationsbeispiel handelt, wurde für Vorführungen der Software ein Szenario ausgearbeitet, um die Leistungsfähigkeit des Systems zu veranschaulichen. Eventuelle Unzulänglichkeiten werden bei einer Vorführung auch deutlich. Für eine Demonstration kann mit dem Menupunkt `Init` des Admintools eine Initialisierung beider lokaler Datenbanken mit Standardwerten vorgenommen werden. Darüber hinaus besteht mittels der beiden lokalen Anwendungen die Möglichkeit, jeweils eine Datenbank einzeln zu initialisieren. Die nachfolgenden Übersichten zeigen die Initialwerte:

Produktno.	Bezeichnung	Länge	Breite	Status
100000	geradstueck	2000	200	IN_DESIGN
100001	geradstueck	3000	500	COMPLETED
100011	weiche	1000	200	IN_CONSTRUCTION
400000	kurvenstueck	2000	200	COMPLETED
400001	kurvenstueck	1500	200	COMPLETED

Tabelle C.1: Nichtangetriebene Rollenförderer (`RC`) in der CAx-Datenbank

Produktno.	Bezeichnung	Länge	Breite	Status	v_{max}	U_{Motor}
200000	dgeradstueck	2000	200	IN_DESIGN	10.5	42
200010	dweiche	1000	200	IN_DESIGN	5.0	230
200011	dweiche	1000	500	IN_CONSTRUCTION	5.5	230
300000	dkurvenstueck	2000	200	COMPLETED	2.0	42
300001	dkurvenstueck	1500	200	COMPLETED	2.5	220

Tabelle C.2: Angetriebene Rollenförderer (`DRC`) in der CAx-Datenbank

Produktno.	Bezeichnung	Preis	Lieferzeit	Maschinenklasse
300000	DKURVENSTUECK	3000.00	40	DRC
300001	DKURVENSTUECK	3200.00	45	DRC
400000	KURVENSTUECK	2000.00	20	RC
400001	KURVENSTUECK	2200.00	25	RC
500000	DREHTISCH	4000.00	21	RC
500001	DDREHTISCH	4400.00	23	DRC
600000	GABELSTAPLER_A	25000.00	220	FLK
600001	GABELSTAPLER_B	27000.00	230	FLK

Tabelle C.3: Transportmaschinen in der Verkaufs-Datenbank

Bei der näheren Betrachtung dieser Tabellen wird auffallen, daß alle Objekte, die derselben Klasse des integrierten Schemas angehören, einem eigenen Nummernbereich zugeordnet wurden. Diese Bereiche werden durch die führende Ziffer der Produktnummer gebildet. Diese Unterteilung wurde nur aus Gründen der Übersichtlichkeit vorgenommen. Somit gibt es keine Abhängigkeit von einer Klassenzugehörigkeit bezüglich des integrierten Schemas, was durch die Erzeugung eines Objektes mit z.B. der Produktnummer `777777` unter Verwendung des Admintools leicht nachzuvollziehen ist. Das so erzeugte Objekt einer frei wählbaren Klasse kann dann zur Untermauerung noch Klassenwechsel vollziehen. Auf den nun tabellarisch vorliegenden Beispieldaten setzt das nachfolgend beschriebene Szenario auf.

Anhang D

Szenario

Dieses Szenario soll zeigen, aus welchen Vorgängen im Geschäftsbetrieb unseres Unternehmens sich entsprechende Anwendungsfälle des hier vorgestellten Demonstrationsbeispieles SIGMA$_{DEMO}$ ergeben. Teilweise geht das Szenario über die Möglichkeiten der tatsächlich implementierten Programme hinaus und zeigt so Probleme und auch das Potential von Konsistenzsicherungsmechanismen in föderierten Datenbanksystemen auf.

D.1 Lokale Datenbestände

Die lokalen Datenbestände der Konstruktionsdatenbank und der Verkaufsdatenbank werden initialisiert. Hierzu kann das Admintool dienen.

D.2 Ausgangssituation

Zur Sichtbarmachung der Ausgangssituation werden mit Hilfe der lokalen Applikationen CAD und Verkauf die Daten der beiden Komponentendatenbanken wie folgt angezeigt:

CAD:
: alle Instanzen der Klasse `Roller_Conveyor` (inklusive aller Instanzen der Unterklasse `Driven_Roller_Conveyor`)

Verkauf:
: alle Tupel der Relation `transport_machine`.

Zusätzlich wird jeweils die Anzahl der existierenden Instanzen angezeigt. Es ist zu zeigen, daß eine intensionale und extensionale Überlappung zwischen den Objekten der Konstruktionsdatenbank und den Tupeln der Verkaufsdatenbank besteht.

Die Existenz von Konflikten der drei Arten sollte wie folgt dargestellt werden:

Datenmodellkonflikt:
objektorientiertes Modell (ONTOS) in der Konstruktionsdatenbank vs. relationales Modell (ORACLE) in der Verkaufsdatenbank

Schemakonflikte: extensionale Überlappung von `RC/DRC` und `TM`

Beschreibungskonflikt bei product_name: char[20] vs. char[40]

Datenkonflikte: Groß- / Kleinschreibung bei `product_name`

D.3 Integration der Komponentendatenbanken

Mit Hilfe des Admintools wird gezeigt, wie durch das föderierte Datenbanksystem eine Integration der Komponentendatenbanken vorgenommen wird und die Konflikte gelöst werden können.

Alle Instanzen der abstrakten Klasse `i_AC` werden mit Hilfe des Admintoolsangezeigt. Besonders ist auf die Objekte in den Klassen `TM_RC` und `TMisDRC` hinzuweisen, die auf realweltäquivalente Objekte in beiden Komponentendatenbanken verweisen. Die Integration kann durch Anzeigen der Anzahl der Instanzen von `i_AC` und Vergleich mit der Summe der lokalen Instanzen verdeutlicht werden.
Ebenfalls wird die Konfliktlösung beim Attribut `product_name` gezeigt:

NAME vs. name: alle globalen Attributwerte in kleinen Buchstaben und Abbildungsfunktion zwischen globaler Applikation und Verkaufsdatenbank (hier alles Großbuchstaben)

char(20) vs. char(40): global char(40), Restriktionen bei Werteeingabe und Werteänderungen

D.4 Auswirkungen lokaler Datenbankoperationen auf das föderierte System

Die Auswirkungen lokaler Datenbankoperationen auf das föderierte System sollen mit Hilfe der lokalen und globalen Applikationen gezeigt werden.

1. Ein neuer Rollenförderer wird in der Konstruktionsabteilung entworfen. Folgende Operationen werden dazu in den einzelnen Applikationen durchgeführt:

   (a) Mit der CAD-Applikation werden die einzelnen Extensionen ohne Subtypen der Konstruktionsdatenbank angezeigt.

(b) Mit der CAD-Applikation wird der Datensatz
`RC(#100010, weiche, 1000, 500, IN_DESIGN)`
erzeugt.

(c) mittels des Admintools wird die Extension der Klasse `RC` ohne Subtypen angezeigt.

2. In der Konstruktionsabteilung wird der Name für einen Rollenförderer, der auch in der Verkaufsdatenbank enthalten ist, geändert. Eine solche Änderung wird nicht nur global, sondern auch in der Verkaufsdatenbank propagiert. Dazu werden folgende Operationen ausgeführt:

   (a) Mit der CAD-Applikation wird der Datensatz
   `RC(#400000, kurvenstueck, 2000, 200, COMPLETED)`
   in
   `RC(#400000, geradstueck, 2000, 200, COMPLETED)`
   geändert.

   (b) In einer globalen Applikation wird die Extension der Klasse `TM_RC` ohne Subtypen angezeigt.

   (c) In der Applikation Verkauf wird der Datensatz aus `TM` mit dem Schlüssel #400000 gesucht und angezeigt.

3. Da ein Produkt nicht mehr produziert wird, muß es aus der Verkaufsdatenbank gelöscht werden. Die Konstruktionsdaten sollen jedoch erhalten bleiben. Ein dynamischer Klassenwechsel eines Objektes innerhalb des globalen Schemas ist die Folge. Folgende Operationen werden ausgeführt:

   (a) Mit dem Admintool wird die Extension der Klasse `TMisDRC` angezeigt.

   (b) Mit der Verkaufsapplikation wird der Datensatz
   `TM(#300000, DKURVENSTUECK, 3000.00, 40, DRC)`
   gelöscht.

   (c) Mit dem Admintool werden die Extensionen der Klassen `TMisDRC` und `DRC` angezeigt.

D.5 Auswirkungen globaler Datenbankoperationen auf das föderierte System

Die Auswirkungen globaler Datenbankoperationen auf das föderierte System und die Komponentendatenbanken werden mit Hilfe der globalen und lokalen Applikationen gezeigt.

1. Eine neue Transportmaschine wird in das Verkaufsprogramm aufgenommen. Ein Sachbearbeiter nimmt den zugehörigen Datenbankeintrag mit Hilfe der globalen Applikation vor. Folgende Operationen werden ausgeführt:

   (a) Der Datensatz
   `TMnotRC(#600002, gabelstapler_c, 30000.00, 220, FLK)`
   wird global mit `insert TMnotRC` eingefügt.

   (b) In der Applikation Verkauf wird der Datensatz aus TM mit dem Schlüssel #600002 gesucht und angezeigt.

2. Durch die Angliederung eines kleinen Unternehmens, das angetriebene Geradstücke produziert, werden die kompletten Konstruktionsunterlagen mitgeliefert. Die noch auf Lager vorhandenen Geradstücken der Firma werden jetzt über die Mutterfirma verkauft. Alle Daten werden mit der globalen Applikation eingefügt. Folgende Operationen werden ausgeführt:

   (a) Der Datensatz
   `TMisDRC(#200001, dgeradstueck, 200, 10, COMPLETED, 12.5, 220, 230, 240, 466.00, 127)`
   wird global mit `insert TMisDRC` eingefügt.

   (b) Mit den lokalen Applikationen wird der entsprechende Eintrag in der jeweiligen Datenbank angezeigt.

3. Ein Produkt wird mit allen Konstruktions- und Verkaufsdaten an ein anderes Unternehmen zur Weiterführung verkauft. Das Löschen des zugehörigen globalen Objektes bewirkt das Löschen der entsprechenden Einträge in beiden Komponentendatenbanken. Folgende Operationen werden ausgeführt:

   (a) Der Datensatz
   `TM_RC(#400001, kurvenstueck, 1500, 200, COMPLETED, 2200, 25)`
   wird global mit `delete TM_RC` gelöscht.

   (b) In der CAD-Applikation wird die Extension der Klasse `RC` angezeigt.

   (c) In der Applikation Verkauf werden alle Tupel von `TM` angezeigt.

D.6 Lokale und globale Integritätsbedingungen

Die Auswirkungen lokaler und globaler Integritätsbedingungen auf das föderierte System und die Komponentendatenbanken werden mit Hilfe der globalen und lokalen Applikationen gezeigt.

1. Mit der globalen Applikation wird versucht, die Attribute einer Instanz der Klasse RC zu ändern. Dabei wird eine lokale Integritätsbedingung der Konstruktionsdatenbank verletzt, so daß der Versuch zurückgewiesen wird. Folgende Operationen werden ausgeführt:

   (a) Die Attribute `length` und `width` des Datensatzes
   `RC(#100001, geradstueck, 3000, 500, COMPLETED)`
   werden global mit `change RC` auf die Werte 30000 und 1000 gesetzt.

   (b) Die globale Applikation weist den Änderungsversuch mit einer Anzeige zurück, da die lokale Integritätsbedingung
   `length < (width * 15)`
   verletzt wird.

2. Durch die Firmenübernahme werden die Konstruktionsdaten eines neu entwickelten Gurtbandes mit übernommen. Beim globalen Einfügen der Daten kommt es zu einer Verletzung globaler Integritätsbedingungen. Folgende Operationen werden ausgeführt:

   (a) Der Datensatz
   `RC(600001, gurtband, 2000, 300, COMPLETED)`
   wird global mit `insert RC` eingefügt.

   (b) Das Admintool weist den Einfügeversuch per Anzeige zurück, da
      i. ein globaler Datensatz mit dem Schlüssel #600001 existiert und
      ii. aufgrund der Disjunktheit der Klassen `RC` und `TMnotRC` kein Zusammenführen beider Datensätze erfolgen kann.

   An dieser Stelle kann der Nutzen von globalen Integritätsbedingungen erklärt werden.

3. Ein bereits im Verkauf befindlicher Drehtisch soll nach aufgetretenen Fehlern in der Konstruktionsabteilung überarbeitet werden. Da er nicht mehr in der Konstruktionsdatenbank enthalten ist, erfolgt ein globales Einfügen. Dabei ist die globale Applikation gefordert, Mechanismen zum Zusammenführen realweltäquivalenter Objekte zur Verfügung zu stellen. Folgende Operationen werden ausgeführt:

   (a) Der Datensatz
   `RC(#500000, drehtisch, 1000, 500, IN_DESIGN)`
   wird global eingefügt.

   (b) Das Das Admintool weist den Einfügeversuch per Anzeige zurück, da ein globaler Datensatz mit dem Schlüssel #500000 existiert. Dem Benutzer wird aber vorgeschlagen, den Datensatz einzufügen und mit dem vorhandenen globalen Datensatz aus der Klasse `TM` zu matchen.

(c) Der Benutzer akzeptiert und der Datensatz wird mit `insert()` eingefügt.

(d) Es folgt eine Anzeige der Instanzen der Klassen `RC`, `TM` und `TM_RC`.

D.7 Komplexszenario

Abschließend soll ein Komplexszenario folgende Fragen beantworten:

1. Wie können globale Integritätsbedingungen bei lokalen Datenbankoperationen überwacht werden?

2. Welche Auswirkungen hat das Einfügen von globalen Integritätsbedingungen im laufenden Betrieb auf das föderierte System?

Zur Veranschauliching dieser Problematik können folgende Operationen unter Verwendung aller lokalen und globalen Anwendungen ausgeführt werden:

1. Mit Hilfe der Applikation Verkauf wird der Datensatz

   `TM(#100011, WEICHE, 230.00, 10, RC)`

   in die Verkaufsdatenbank eingefügt.

2. Da ein realweltäquivalentes Objekt bereits in der Konstruktionsdatenbank enthalten ist, wird die globale Schlüsselbedingung verletzt.

   Über eine Anfrage (z.B. durch einen globalen Regelmanager) bestätigt der lokale Benutzer, daß beide Datensätze dasselbe Objekt abbilden.

   Hinweis: Da in der derzeitigen Version von SIGMA$_{DEMO}$ keine Regelverarbeitung implementiert ist, lassen sich die daraus resultierenden Auswirkungen recht gut erkennen und unterstreichen die Notwendigkeit solcher aktiven Systeme. An dieser Stelle sollte erläutert werden, wie mit aktiven Schichten die globalen Integritätsbedingungen überwacht werden können. Die notwendigen Eingriffe in die lokale Autonomie sind zu diskutieren.

   Es wird erkannt, daß nicht fertig konstruierte Werkstücke nicht verkauft werden sollten. Daraus ergibt sich folgende neue globale Integritätsbedingung: Nur mit dem Projektstatus `COMPLETED` versehene Werkstücke dürfen auch in der Verkaufsdatenbank enthalten sein.

   Zur Verdeutlichung dieser Bedingung werden folgende weitere Operationen ausgeführt:

3. Mit dem Admintool soll diese Bedingung definiert und dem globalen Schema hinzugefügt werden.

4. Der globale Regelmanager muß nun alle bereits vorhandenen Datensätze der Klassen `TM_RC` und `TMisDRC` auf Einhaltung der neuen Bedingung überprüfen. Er erkennt dann, daß die folgenden Datensätze diese Bedingung nicht erfüllen, und wiest darauf hin:

   (a) TM_RC(#100011, weiche, 1000, 200, IN_CONSTRUCTION, 230.00, 10, RC)

   (b) TM_RC(#500000, drehtisch, 1000, 500, IN_DESIGN, 4000.00, 23, RC)

   Für jeden Datensatz werden dem Benutzer vom Admintool folgende beide Vorschläge unterbreitet:

   (c) Ändern des Attributs project_state in `COMPLETED`.

   (d) Löschen des zugehörigen Datensatzes aus der Verkaufsdatenbank.

5. Der Benutzer entscheidet sich bei (a) für (c) und bei (b) für (d).

6. Es kann nun mit einer globalen Applikation eine Anzeige der Instanzen der Klassen `RC`, `TM` und `TM_RC` vorgenommen werden.

7. In der CAD-Applikation werden die Instanzen der Klasse `RC` angezeigt. In der Applikation Verkauf werden die Tupel von `TM` angezeigt.

8. Mit Hilfe der Applikation Verkauf wird der Datensatz

   `TM(#200000, DGERADSTUECK, 3500.00, 44, DRC)`

   in die Verkaufsdatenbank eingefügt.

9. Da ein realweltäquivalentes Objekt bereits in der Konstruktionsdatenbank enthalten ist, wird die globale Schlüsselbedingung verletzt.

   Ein globales Einfügen kann ebenfalls nicht erfolgen, da die gerade eingefügte globale Integritätsbedingung verletzt würde. Deshalb würde das Einfügen von einem globalen Regelmanager per Anzeige zurückgewiesen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[BFHK94] R. Busse, P. Fankhauser, G Huck, and W. Klas. IRO-DB: An Object-Oriented Approach Towards Federated and Interoperable DBMS. In *Proc. of the Int. Workshop on Advances in Databases and Information Systems (ADBIS'94), Moscow, Russia*, pages 178–186. Russian Academy of Sciences, May 1994.

[BKNW91] T. Barsalou, A. Keller, Siambela N., and G. Wiederhold. Updating Relational Databases through Object-Based Views. In J. Clifford and R. King, editors, *Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data, Denver, Colorado, SIGMOD RECORD 20(2)*, pages 248–257. ACM Press, June 1991.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[CHJ+96] S. Conrad, M. Höding, S. Janssen, G. Saake, I. Schmitt, and C. Türker. Integrity Constraints in Federated Database Design. Preprint Nr. 2, Fakultät für Informatik, Universität Magdeburg, April 1996.

[CL94] J. Chomicki and W. Litwin. Declarative Definition of Object-Oriented Multidatabase Mappings. In M. T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*, pages 375–392. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[CT97] S. Conrad and C. Türker. Unternehmensweite Datenkonsistenz durch Integration bestehender Informationssysteme. In H. Krallmann, editor, *Wirtschaftsinformatik'97: Internationale Geschäftstätigkeit auf der Basis flexibler Organisationsstrukturen und leistungsfähiger Informationssysteme*, pages 345–356. Physica-Verlag, Heidelberg, 1997.

[Dat86] DAFTG Database Architecture Framework Task Group (DAFTG) of the ANSI/X3/SPARC Database System Study Group. Reference Model for DBMS Standardization. *ACM SIGMOD RECORD*, 15(1):19–58, 1986.

[DeM89] L. DeMichiel. Resolving Database Incompatibility: An Approach to Performing Relational Operations over Mismatched Domains. *IEEE Transactions on Knowledge and Data Engineering*, 1(4):485–493, December 1989.

[Dud96] Dudenredaktion. *Duden*. Dudenverlag, 1996.

[Dup94] Y. Dupont. Resolving Fragmentation Conflicts in Schema Integration. In P. Loucopoulos, editor, *Entity-Relationship Approach — ER'94, Proc. of the 13th Int. Conf. on the Entity-Relationship Approach, Manchester, UK*, pages 513–532. LNCS 881, Springer-Verlag, Berlin, December 1994.

[GSC95] M. Garcia-Solaco, F. Saltor, and M. Castellanos. A Structure Based Schema Integration Methodology. In P. S. Yu and A. L. P. Chen, editors, *Proc. of the 11th IEEE Int. Conf. on Data Engineering (ICDE'95), Taipei, Taiwan, March 1995*, pages 505–512. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[HK95] U. Hohenstein and C. Körner. Semantische Anreicherung relationaler Datenbanken. In G. Lausen, editor, *Proc. GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft" (BTW'95), Dresden, März 1995*, pages 130–149. Informatik aktuell, Springer-Verlag, Berlin, 1995.

[HM85] D. Heimbigner and D. McLeod. A Federated Architecture for Information Management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.

[Höd96a] M. Höding. An Approach to Integration of File Based Systems into Database Federations. In *Heterogeneous Information Management, Prague, Czech Republic, 4–5 November 1996, Proc. of the 10th ERCIM Database Research Group Workshop*, pages 61–71. ERCIM-96-W003, European Research Consortium for Informatics and Mathematics, 1996.

[Höd96b] M. Höding. Föderierung von Dateien: Finden, Verwalten und Nutzen von Metainformationen. In S. Conrad, G. Saake, I. Schmitt, and C. Türker, editors, *Kurzfassungen – 8. Workshop "Grundlagen von Datenbanken", Friedrichsbrunn/Harz (28.05.–31.05.96)*, pages 46–50. Institut für Technische Informationssysteme, Universität Magdeburg, May 1996.

[JD94] D. Jonscher and K.R. Dittrich. An Approach For Building Secure Database Federations. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 24–35, Santiago, Chile, September 1994. Morgan Kaufmann Publishers, San Francisco, CA.

[KC86] S. Khoshafian and G. P. Copeland. Object Identity. In N. Meyrowitz, editor, *Proc. of the 1st Int. Conf. on Object Oriented Programming Systems,* *Languages and Applications (OOPSLA'86), Portland, Oregon, SIGPLAN Notices 21(11)*, pages 406–416. ACM Press, November 1986.

[KLK91] R. Krishmamurthy, W. Litwin, and W. Kent. Interoperability of Heterogeneous Databases with Schematic Discrepancies. In Y. Kambayashi, M. Rusinkiewicz, and A. Sheth, editors, *Proc. of the 1st Int. Workshop on Interoperability in Multidatabase Systems (IMS'91), Kyoto, Japan*, pages 144–151. IEEE Computer Society Press, April 1991.

[Köl96] A. Köller. Dateien in föderierten Datenbanksystemen - Untersuchungen an Systemen zur Fabrikplanung. Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, September 1996.

[Ont92] Ontos, Inc., Burlington, MA, USA. *ONTOS 2.1 First Time User's Guide*, 1992.

[Ont94] Ontos, Inc., Burlington, MA, USA. *Introduction to ONTOS DB 3.0*, 1994.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[PBE95] E. Pitoura, O. Bukhres, and A. K. Elmagarmid. Object Orientation in Multidatabase Systems. *ACM Computing Surveys*, 27(2):141–195, June 1995.

[Rah94] E. Rahm. *Mehrrechner-Datenbanksysteme.* Addison-Wesley, Bonn, 1994.

[RBB+95] E. Radeke, R. Böttger, B. Burkert, Y. Engel, G. Kachel, S. Kolmschlag, and D. Nolte. Efendi: Federated Database System of Cadlab. In M. J. Carey and D. A. Schneider, editors, *Proc. of the 1995 ACM SIGMOD Int. Conf. on Management of Data, San Jose, CA, SIGMOD RECORD 24(2)*, page 481. ACM Press, June 1995.

[RPRG94] M. P. Reddy, B. E. Prasad, P. G. Reddy, and A. Gupta. A Methodology for Integration of Heterogeneous Databases. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):920–933, December 1994.

[RS94] E. Radeke and M. H. Scholl. Federation and Stepwise Reduction of Database Systems. In W. Litwin and T. Risch, editors, *Applications of Databases, Proc. of the 1st Int. Conf., ADB-94, Vadstena, Sweden, June 1994*, pages 381–399. LNCS 819, Springer-Verlag, Berlin, 1994.

[SCG91] F. Saltor, M. Castellanos, and M. Garcia-Solaco. Suitability of Data Models as Canonical Models for Federated Databases. *ACM SIGMOD RECORD*, 20(4):44–48, December 1991.

[Sch95] I. Schmitt. Flexible Integration and Derivation of Heterogeneous Schemata in Federated Database Systems. Preprint Nr. 10, Fakultät für Informatik, Universität Magdeburg, November 1995.

[SCH+96] G. Saake, S. Conrad, M. Höding, S. Janssen, I. Schmitt, and C. Türker. Föderierung heterogener Datenbanksysteme und lokaler Datenhaltungskomponenten zur systemübergreifenden Integritätssicherung — Grundlagen und Ziele des Projektes **SIGMA**$_{FDB}$. In *Integrationskonzepte für Mensch, Technik, Organisation, und Planung — Grundlagen und Projektziele der Projektgruppe METOP, Bericht zum 1. Kolloquium*, pages 13–24. Preprint Nr. 5, Fakultät für Maschinenbau, Universität Magdeburg, 1996.

[SEHT96] I. Schmitt, A. Ebert, M. Höding, and C. Türker. **SIGMA**$_{Bench}$ – Ein Werkzeug zum Entwurf föderierter Datenbanken. In W. Hasselbring, editor, *Kurzfassungen zum 2. Workshops "Föderierte Datenbanken", Dortmund, 12.-13. Dezember 1996*, pages 19–26. SWT Memo Nr. 90, Fachbereich Informatik, Universität Dortmund, 1996.

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

[SPD92] S. Spaccapietra, C. Parent, and Y. Dupont. Model Independent Assertions for Integration of Heterogeneous Schemas. *The VLDB Journal*, 1(1):81–126, July 1992.

[SS96a] I. Schmitt and G. Saake. Flexible Generation of Global Integrated Schemata using GIM. In S. Conrad, M. Höding, S. Janssen, and G. Saake, editors, *Kurzfassungen des Workshops "Föderierte Datenbanken", Magdeburg, 22.04–23.04.96*, pages 29–43. Bericht 96–01, Institut für Technische Informationssysteme, Universität Magdeburg, April 1996.

[SS96b] I. Schmitt and G. Saake. Integration of Inheritance Trees as Part of View Generation for Database Federations. In B. Thalheim, editor, *Conceptual Modelling — ER'96, Proc. of the 15th Int. Conf., Cottbus, Germany, October 1996*, pages 195–210. LNCS 1157, Springer-Verlag, Berlin, 1996.

[SS96c] I. Schmitt and G. Saake. Schema Integration and View Generation by Resolving Intensional and Extensional Overlappings. In K. Yetongnon and S. Hariri, editors, *Proc. of the 9th ISCA Int. Conf. on Parallel and Distributed Computing Systems (PDCS'96), Dijon, France*, pages 751–758, September 1996.

[Stü93] G. Stürner. *ORACLE7 – Die verteilte semantische Datenbank.* DBMS Publishing, Weissach, 2 edition, 1993.

[TBD+87] M. Templeton, D. Brill, S. K. Dao, E. Lund, P. Ward, A. L. P. Chen, and R. MacGregor. Mermaid: A Front-End to Distributed Heterogeneous Databases. *Proc. of the IEEE*, 75(5):695–708, May 1987.

[TC95] C. Türker and S. Conrad. Aktive Mechanismen zur Konsistenzsicherung in föderierten Datenbanksystemen. In C. Eckert, H.-J. Klein, and T. Polle, editors, *7. GI-Workshop "Grundlagen von Datenbanken", Bad Salzdetfurth*, pages 148–152. Hildesheimer Informatik Berichte, June 1995.

[TC96] C. Türker and S. Conrad. Using Active Mechanisms for Global Integrity Maintenance in Federated Database Systems. In S. Conrad, M. Höding, S. Janssen, and G. Saake, editors, *Kurzfassungen des Workshops "Föderierte Datenbanken", Magdeburg, 22.04–23.04.96*, pages 51–65. Bericht 96–01, Institut für Technische Informationssysteme, Universität Magdeburg, April 1996.

[TK78] D. C. Tsichritzis and A. Klug. The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Database Management Systems. *Information Systems*, 3(3):173–191, 1978.