Otto-von-Guericke-Universität Magdeburg


Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

# Nutzung des GIM-Ansatzes zum Datenbankimport von XML-Daten

Verfasser:

Jan Reidemeister

24. März 2002

Betreuer:

Dr.-Ing. Ingo Schmitt

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg
Germany

**Reidemeister, Jan:**
*Nutzung des GIM-Ansatzes zum Datenbankimport von XML-Daten*
Diplomarbeit
Otto-von-Guericke-Universität Magdeburg, 2002.

## Hintergrund der Aufgabenstellung

Datenbanksysteme sind allgemein akzeptierte Systeme zur effizienten Verwaltung von Daten. Durch den aktuellen Erfolg der Sprache XML und deren Akzeptanz entstehen momentan weltweit große in XML kodierte Datenmengen. Um diese Datenmengen effizient in Datenbanksystemen verwalten zu können, ist eine Transformation im Rahmen eines Datenbankimports notwendig. Bestehende Transformationsansätze versuchen dabei hauptsächlich, eine XML-nahe Datenbankstruktur zu finden, aus der wieder effizient XML-Daten zurückgewonnen werden können. Durch den Einsatz der GIM-Methode soll hingegen versucht werden, adäquate Datenbankschemata aus XML-Daten zu generieren. Das heißt, der Datenbanknutzer arbeitet mit den herkömmlichen Datenbanksprachen auf der Datenbank, ohne dass die XML-Herkunft der Datenbank ersichtlich wird.

## Geplante Etappen

- Einarbeitung in die GIM-Methode
- Literaturrecherche und -aufarbeitung von Arbeiten zum Thema XML-Daten und Datenbanken
- Aufstellen von Anforderungen bzgl. eines Datenbankimports und -exports
- Erstellen oder Nutzung zweier Anwendungsszenarien zur praktischen Evaluierung
- Entwurf des Datenbankimportwerkzeuges unter Berücksichtigung von Nutzerpräferenzen
- Implementierung von Import- und Exportwerkzeugen unter Java
- Dokumentationserstellung
- Evaluierung der GIM-Methode

## Zusammenfassung

Durch die zunehmende Bedeutung der Extensible Markup Language (XML) als Austauschformat im Internet ist eine Integration in bestehende Anwendungen, welche in den meisten Fällen (relationale) Datenbanken zur Verwaltung der Daten nutzen, notwendig. Um eine Integration zu ermöglichen, müssen Methoden zur Speicherung von XML-Dokumenten in Datenbanken, zum Propagieren von Updates in beide Richtungen und zur Ausgabe der Daten als XML gefunden werden.

In dieser Diplomarbeit wurde ein Algorithmus zur datenbankunabhängigen Schemageneration für XML-Dokumente basierend auf einer DTD konzipiert. Durch verschiedene alternative Abbildungen und Transformationen kann das generierte Schema für einen konkreten Einsatzzweck optimiert werden. Das Ergebnis der Schemageneration ist ein optimiertes Schema und eine Beschreibung, wie die XML-Konstrukte auf dieses Schema abgebildet werden.



**Stichworte:** Extensible Markup Language (XML), Generisches Integrationsmodell (GIM), Schemageneration, Mapping, Datenbanksysteme (DBS)



## Abstract

The Extensible Markup Language (XML) is an emerging standard for exchanging data on the internet. This requires an integration with existing applications, that often use (relational) databases for data management. The task for such an integration involves the storage of XML in databases, the propagation of updates in both directions and the publishing of data as XML documents.

This paper introduces a new approach for generating a database independent schema for XML documents based on a DTD. The generated schema can be optimized with the help of different alternate mappings and transformations. The result of the schema generation is an optimized schema and a description of the mapping from XML to this schema.



**Keywords:** Extensible Markup Language (XML), Generic Integration Model (GIM), Schemageneration, Mapping, database systems (DBS)

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| **DBMS** | Datenbankmanagementsystem |
| **DBS** | Datenbanksystem |
| **DCD** | Document Content Descriptor |
| **DDL** | Data Definition Language |
| **DOM** | Document Object Model |
| **DTD** | Document Type Definition |
| **FDBS** | Förderiertes Datenbanksystem |
| **GIM** | Generisches Integrationsmodell |
| **HTML** | HyperText Markup Language |
| **ISO** | International Organisation for Standardiziation |
| **PDF** | Portable Document Format |
| **RDBS** | Relationales Datenbanksystem |
| **SAX** | Simple API for XML-Processing |
| **SDBS** | Spezialdatenbanksysteme |
| **SGML** | Standard Generalized Markup Language |
| **SQL** | Structured Query Language |
| **SVG** | Scalable Vector Graphics |
| **URI** | Uniform Resource Identifier |
| **URL** | Uniform Resource Locator |
| **WWW** | World Wide Web |
| **$W_3C$** | World Wide Web Consortium |
| **XHTML** | Extensible HyperText Markup Language |
| **XML** | Extensible Markup Language |
| **XSL** | Extensible Stylesheet Language |
| **XSLT** | Extensible Stylesheet Language Transformations |
| **XSL-FO** | Extensible Stylesheet Language - Formatting Objects |

# Kapitel 1

# Einleitung

Das Konzept der semistrukturierten Daten, besonders in der Form XML, ist mittlerweile allgemein akzeptiert. XML ist eine einfach zu schreibende, einfach zu parsende, erweiterbare und plattformunabhängige Auszeichnungssprache, die sich zum de facto Standard für den Datenaustausch im Internet entwickelt hat.

Auf der anderen Seite existiert eine Vielzahl von Anwendungen, welche verschiedene Datenbanksysteme (DBS) zur Verwaltung der Daten verwenden. Besonders relationale Datenbanksysteme (RDBS) haben sich in den letzten zwanzig Jahren zur beherrschenden Technologie entwickelt. Dies wird auch in der absehbaren Zukunft aufgrund der Zuverlässigkeit, Skalierbarkeit und Performance der RDBSe so bleiben.

Um das volle Potenzial der beiden Technologien auszuschöpfen, ist eine Integration von XML und Datenbanken notwendig. Dazu müssen Methoden zum Speichern von XML-Daten, dem Propagieren der Updates in beide Richtungen und der Ausgabe der Daten als XML gefunden werden.

Die vorliegende Arbeit beschäftigt sich mit einem Teilproblem der Speicherung von XML-Daten, der Generierung eines adäquaten Schemas für die Datenbank.

## 1.1 Gliederung

Nach einer Einleitung wird im zweiten Kapitel eine Einführung in XML und die damit verbundenen Konzepte und Technologien gegeben. Ein zentrales Thema ist dabei der Aufbau von XML-Dokumenten sowie Möglichkeiten zur Beschreibung der Struktur.

Das dritte Kapitel gibt einen Überblick über existierende Möglichkeiten zur Speicherung von XML-Dokumenten in Datenbanken. Stellvertretend werden zwei verschiedene Ansätze zur Speicherung in einer relationalen Datenbank ausführlich vorgestellt. Abschließend werden diverse Alternativen oder Erweiterungen diskutiert.

Das Generische Integrationsmodell wird im vierten Kapitel präsentiert. Vorgestellt werden ein praktikabler Algorithmus zur Ableitung eines integrierten Schemas und verschiedene Möglichkeiten zur Modifikation des generierten Schemas.

Im fünften Kapitel wird ein Algorithmus zur Generierung eines Schemas für XML-Dokumente entwickelt. Durch verschiedene alternative Abbildungen und die Verwendung des Generischen Integrationsmodells ist es möglich, das generierte Schema auf einen Anwendungszweck hin zu optimieren. Das Ergebnis der Schemageneration ist ein optimiertes Schema und eine Beschreibung, wie die XML-Konstrukte auf das Schema abgebildet werden.

Das sechste Kapitel fasst abschließend die Arbeit zusammen.

## 1.2 Notation

In der vorliegenden Arbeit werden einige typographische Konventionen benutzt, um bestimmte Textteile hervorzuheben. Neu eingeführte Begriffe, Wörter in Fremdsprachen (meist Englisch) oder Hervorhebungen werden *kursiv* notiert. In einer nichtproportionalen Schrift werden `Programmtexte` und daraus entnommene `Bezeichner` notiert.

Fußnoten reflektieren teilweise die (nicht immer begründeten) Ansichten des Autors. Für das Verständnis der Arbeit sind Grundkenntnisse aus dem Bereich Datenbanken empfehlenswert.

# Kapitel 2

# Extensible Markup Language

In diesem Kapitel wird ein einführender Überblick über *XML* (**Ex**tensible **M**arkup **L**anguage) und die damit verbundenen Konzepte und Technologien gegeben.

## 2.1 XML

XML hat sich in kürzester Zeit einen festen Platz im Internet erobert. Zentrales Anliegen für den Einsatz von XML ist es, Inhalte maschinell zugänglich, auffindbar und manipulierbar zu machen. Um das zu erreichen, wird mit XML eine Möglichkeit gegeben, Inhalte über kennzeichnende Markierungen in funktionale Blöcke zu untergliedern. Daher ist XML hervorragend zur Speicherung und zum Austausch von strukturierten und semistrukturierten Daten geeignet.

Die folgenden Ausführungen basieren auf [Wor98], [Bra98] und [BM98].

### 2.1.1 Historische Entwicklung

Die Sprache XML besitzt zwei Wurzeln, SGML und HTML.

Schon lange existiert die Idee, die Struktur eines Dokumentes innerhalb des Dokumentes selbst zu kennzeichnen. Umgesetzt wurde sie in größerem Maßstab erstmals mit der Entwicklung von *SGML* (**S**tandard **G**eneralized **M**arkup **L**anguage, [ISO86]), welches 1986 von der ISO standardisiert wurde. Eine weite Verbreitung fand SGML in der Verwaltung von „sehr großen“ Datenbeständen, wie beispielsweise der Dokumentation im Flugzeugbau, für Enzyklopädien oder Gesetzestexte.

SGML ist eine sehr mächtige Auszeichnungssprache. Jedoch behinderte genau dieser Umfang und die Komplexität die Verbreitung von SGML. Programme, die SGML (auch nur teilweise) nutzen, sind sehr aufwändig zu entwickeln, so dass vergleichbare proprietäre Anwendungen oft günstiger und weniger fehleranfällig sind und sich daher besser am Markt durchsetzten. Aus diesem Grund wurde eine vereinfachte Form von SGML zum Auszeichnen von Dokumenten gesucht.

Die zweite Wurzel von XML liegt in *HTML* (**H**ypertext **M**arkup **L**anguage). Als Anfang der 90er Jahre des letzten Jahrhunderts das *WWW* (**W**orld **W**ide **W**eb) entstand, entwickelte Tim Berners-Lee eine Beschreibungssprache für das Darstellen und Verknüpfen von Informationen. Durch spezielle, fest vorgegebene Auszeichnungselemente (*Markup Tags*) ist es möglich, eine hierarchische Struktur für Dokumente festzulegen und Texte für die Darstellung zu formatieren. Die HTML-Dokumente können durch Verweise (*Links*) untereinander und mit verschiedenen sonstigen Ressourcen, wie z. B. Multimediadaten, verknüpft werden.

Die Syntax von HTML entspricht größtenteils der von SGML, alle anderen Prinzipien von SGML wurden jedoch weggelassen. Dadurch ist HTML eine sehr einfache Sprache geworden und konnte sich als Grundlage des WWW durchsetzen. Jedoch ist diese Einfachheit der Grund, weshalb HTML-Dokumente nicht oder nur sehr schwer von computergesteuerten Anwendungen interpretiert und verarbeitet werden können. Mehrere Eigenschaften von HTML erschweren die automatische Erschließung der Struktur von HTML-Dokumenten.

- HTML ist darauf ausgelegt, das Aussehen der Daten zu beschreiben, die Bedeutung der Daten wird ignoriert.
- HTML ist nicht erweiterbar. Anwendungsspezifische Erweiterungen, um Dokumente strukturell anzureichern, sind nicht vorgesehen.
- Es gibt keine Möglichkeit HTML-Dokumente zu validieren. HTML bietet keine Möglichkeiten, ein gegebenes Dokument auf Gültigkeit (d.h. ob es einem entsprechenden Format entspricht) zu überprüfen.

Diese Beschränkungen von HTML und die Bemühungen SGML zu vereinfachen, führten schließlich zur Entwicklung von XML. 1996 wurde durch das $W_3C$ (World Wide Web-Konsortium), welches auch den HTML-Standard pflegt, mit der Definition von 10 Designzielen die Entwicklung einer neuen Auszeichnungssprache namens XML beschlossen. XML basiert grundlegend auf SGML[1], übernimmt jedoch einige Besonderheiten von HTML und besitzt einige neue Möglichkeiten, die speziell für das WWW notwendig sind. Im Februar 1998 wurde die Version 1.0 von XML ([Wor98], [Wor99a]) veröffentlicht. In der Folge entstanden rund um XML verschiedene Erweiterungen, wie XPointer ([Wor01b]), XLink ([Wor01f]) oder XSL ([Wor01g]).

### 2.1.2 Aufbau von XML-Dokumenten

XML bietet die Möglichkeit, Daten so zu strukturieren, dass sie Regeln entsprechen, die man selbst festlegen darf. Auf diese Weise strukturierte Daten werden als Dokumente bezeichnet. Sie besitzen eine logische und eine physische Struktur. Die logische Struktur wird durch die Elemente dargestellt, die physische durch die Entities. Physisch kann

[1] Es existieren Techniken, um SGML zu XML zu transformieren ([Cla97]).

ein XML-Dokument in verschiedene Komponenten aufgeteilt werden, die voneinander getrennt gespeichert werden.

**Beispiel 2.1.** - XML-Dokument

```
1  <?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
2  <!DOCTYPE purchase SYSTEM "purchase.dtd">
3  <purchase id="p001">
4          <!-- generated 2001-06-23 -->
5          <customer id="cust123"/>
6          <product id="prod345" color="&blue;">
7                  <amount>7</amount>
8                  <price>8.67</price>
9                  <descr>
10                         For details see
11                         <link>http://www.company.com/prod345</link>.
12                 </descr>
13         </product>
14 </purchase>
```

Beispiel 2.1 zeigt ein einfaches XML-Dokument. Auf den ersten Blick fällt auf, dass mittels in eckiger Klammern eingefasster sogenannter *Elemente* eine logische Struktur geschaffen wird. Einige der Elemente, wie „`purchase`“, besitzen *Attribute*, die zusätzliche Informationen zu den Elementen geben.

XML-Dokumente bestehen aus Auszeichnungen (*Markup*) und Inhalt (*Content*). Es gibt sechs verschiedene Arten von Auszeichnungen: Elemente (und ihre Attribute), Kommentare, Deklarationen, Processing Instructions, Entityreferenzen und Marked Sections (CDATA). Sie werden nachfolgend detaillierter betrachtet.

### Elemente

XML-Dokumente bestehen aus Elementen. Die Elemente werden durch sogenannte *Tags* begrenzt. Alle Daten, die zwischen dem öffnenden und dem schließenden Tag liegen, sind der Elementeinhalt. Öffnende Tags beginnen mit einem „`<`“ gefolgt von dem Elementnamen und einem „`>`“, schließende Tags beginnen mit einem „`</`“. Eine Ausnahme bilden leere Elemente, sie werden verkürzt in der Form „`<Elementname/>`“ dargestellt.

Der Elementname sollte den Inhalt beschreiben und ist frei wählbar. Wird bei der Vergabe der Namen darauf geachtet, dass diese aussagekräftig sind, kann man sagen, dass ein daraus bestehendes Dokument selbstbeschreibend ist.

Der Elementeinhalt kann entweder nur weitere Elemente, nur Text (`PCDATA`), beides (*Mixed Content*) oder einen beliebigen Inhalt (`ANY`) enthalten oder leer sein.

### Attribute

Elementen können Attribute zugeordnet werden, um zusätzliche beschreibende Informationen abzubilden. Weiterhin sparen Attribute Hierarchiestufen und machen ein Dokument übersichtlicher. Sie werden vor allem dann verwendet, wenn es sicher ist, dass der Inhalt auf dieser Ebene maximal einmal vorkommt (dadurch werden weitere Kindelemente vermieden). Attribute werden innerhalb des öffnenden Tags aufgeführt und bestehen aus ihrem Namen und dem jeweiligen Wert. Im Beispiel 2.1 besitzt unter anderem das Element „`purchase`“ das Attribut „`id`“ mit dem Wert „p001“.

### Kommentare

Durch „`<!--`“ und „`-->`“ gekennzeichnete Kommentare können an beliebiger Stelle innerhalb eines Dokumentes stehen. Im Beispiel 2.1 ist ein Kommentar nach dem Starttag des Elementes „`purchase`“ zu sehen.

### Deklarationen

Mittels Deklarationen kann der Typ von XML-Dokumenten festgelegt oder Elemente, Attribute und Entities definiert werden (siehe Abschnitt 2.2). Deklarationen werden mittels „`<!`“ und „`>`“ gekennzeichnet. Die zweite Zeile im Beispiel 2.1 zeigt die Deklaration einer externen DTD namens „purchase.dtd“ für dieses XML-Dokument.

### Processing Instructions

Processing Instructions erlauben es, Anweisungen an verarbeitende Programme in Dokumente einzubinden. Sie werden in der Form „`<?PITarget instruction?>`“ notiert. Trifft ein Programm während der Verarbeitung eines Dokumentes auf eine Processing Instruction, die es kennt, kann es auf die in „`instruction`“ stehende Anweisung reagieren.

Processing Instructions werden für die *XML Deklaration* zu Beginn eines XML-Dokumentes verwendet. Durch das für den XML-Standard reservierte `PITarget` „xml“ können die Version des verwendeten Standards sowie Informationen zum benutzten Zeichensatz und der Art der Dokumenttypdefinition angegeben werden. Im Beispiel 2.1 zeigt die erste Zeile die Anwendung dieser Processing Instruction.

### Entities

Entities erlauben es, die physische Struktur von der logischen zu trennen. Sie ermöglichen die getrennte Benennung und Speicherung von Komponenten des Dokumentes, das Einbinden von binären Daten (z.B. Multimediadaten) und die Wiederverwendung von Komponenten. Es gibt verschiedene Typen von Entities.

- **Vordefinierte Entities** existieren für die Zeichen „`<`“, „`>`“, „`&`“, „`&apos;`“ und „`&quot;`“, da diese für die Auszeichnung in XML vorbelegt sind.

- **Zeichen-Entities** werden für Zeichen, die nicht mit einer normalen Tastatur eingeben werden können, genutzt.
- **Interne Entities** können zum Abkürzen von häufig genutzten Textpassagen verwendet werden.
- **Externe Entities** verweisen auf außerhalb des Dokumentes gespeicherte Daten, die in das Dokument eingebunden werden.

Entities werden in Dokumente über Entityreferenzen eingebunden. Die Referenzierung erfolgt in der Form „`&Name;`“ über den Namen. Dabei ist eine Referenz keine Verknüpfung, wie beispielsweise die Links in HTML, sondern der Entityinhalt wird bei der Verarbeitung anstelle der Referenz eingefügt.

#### CDATA Sections

Da diverse Zeichen, wie beispielsweise „`<`“ oder „`&`“, für die Auszeichnung mit XML reserviert sind, ist es nicht möglich, diese Zeichen im Inhalt zu benutzen. Umgehen kann man dies entweder durch die Verwendung der vordefinierten Entities oder durch eine CDATA Section. Der Inhalt einer solchen Section wird von einem verarbeitenden Programm nicht interpretiert und kann deshalb nahezu alle beliebigen Zeichen enthalten. CDATA Sections werden durch „`<![CDATA[`“ und „`]]>`“ begrenzt.

### 2.1.3 Zugriff auf XML-Daten

XML-Daten werden normalerweise in einfachen Textdateien gespeichert. Für den programmtechnischen Zugriff auf die Daten werden sogenannte *Parser* verwendet. Ein XML-Parser liest die Textdaten ein, verarbeitet sie und stellt sie den Anwendungen über eine geeignete Schnittstelle zur Verfügung. Für die Verarbeitung von XML-Daten ist eine DTD nicht zwingend notwendig, sie unterstützt nur die Überprüfung (siehe 2.1.4).

Für den Zugriff auf die vom Parser verarbeiteten XML-Dokumente haben sich zwei verschiedene Schnittstellen etabliert. Die beiden Schnittstellen *SAX* und *DOM* stehen jedoch nicht in Konkurrenz zueinander, sondern ergänzen sich hervorragend, da sie jeweils andere Intensionen verfolgen.

#### SAX

SAX (**S**imple **A**PI for **X**ML-Processing) stellt einen ereignisbasierten Zugriff auf die XML-Daten zur Verfügung. Dazu registriert sich die Anwendung für die sie interessierenden Informationen am Parser. Dieser verarbeitet dann sequentiell das Dokument und informiert die Anwendung über das Auftreten von bestimmten XML-Teilen, wie beispielsweise den Beginn und das Ende des Dokumentes oder von Elementen. Die Anwendung erhält so die Möglichkeit, nur auf die sie interessierenden Informationen zu reagieren.

### DOM

DOM (**D**ocument **O**bject **M**odel, [Wor01a]) ist ein objektorientiertes Zugriffsmodell auf XML- oder HTML-Daten, welches vom $W_3C$ entwickelt wurde. Hierzu wird das jeweilige XML-Dokument als Baum interpretiert. Die Baumstruktur wird durch die Schachtelung der Elemente gebildet. Der Parser verarbeitet (zum Beispiel mittels SAX) das gesamte Dokument und baut einen entsprechenden Baum auf. Die Anwendung erhält als Ergebnis das *Wurzelelement* (*root*-Element) des Dokumentes, welches einen navigierenden Zugriff auf die Daten ermöglicht.

Stellt man sich das Wurzelelement als Wurzel eines Stammbaumes vor, so sind alle Elemente im Dokument Nachfahren (Nachkommen) der Wurzel. Dabei ist zu beachten, dass die Reihenfolge wichtig ist. Im XML-Jargon wird deshalb auch von Eltern-Elementen, Kind-Elementen, Geschwister-Elementen etc. gesprochen. Die Abbildung 2.1 zeigt einen vereinfachten Baum für das XML-Dokument aus dem Beispiel 2.1.

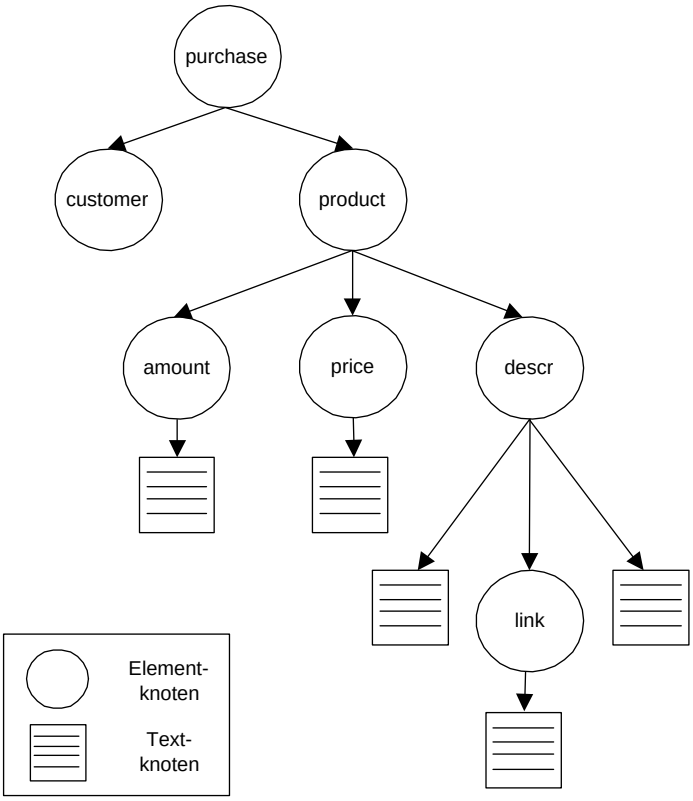


Abbildung 2.1: vereinfachter XML-Baum zum Beispiel 2.1

Das Parsen mittels SAX ist im Vergleich zu DOM sehr schnell und ressourcensparend. Nur der gerade benötigte Teil des XML-Dokumentes wird geladen. Zum Aufbau eines DOM-Baumes muss das ganze Dokument geladen werden, was deutlich mehr Ressourcen benötigt.

Daher werden die beiden Schnittstellen für verschiedene Zwecke eingesetzt. SAX erlaubt die performante Verarbeitung auch großer XML-Dateien, während DOM einen deutlich einfacheren Zugriff bietet.

### 2.1.4 Validierung von XML-Dokumenten

Neben der Verarbeitung kann ein Parser die XML-Dokumente auch überprüfen. Diese Überprüfung umfasst zwei Stufen, die *Wohlgeformtheit* und die *Gültigkeit*.

#### Wohlgeformte Dokumente

Damit ein XML-Dokument wohlgeformt (*well-formed*) ist, muss es die folgenden Eigenschaften erfüllen:

- Das Dokument folgt der XML-Syntax.
- Attribute sind nur einmal pro Element im Starttag notiert.
- Alle Elemente im Dokument sind richtig geklammert.
- Leere Elemente besitzen immer ein Endtag (z.B. `<br>` ist nicht erlaubt, es muss `<br/>` heißen).
- Sämtliche Entities werden vor ihrer Benutzung definiert.
- Attribute besitzen keine externen Entities.
- Es gibt keine Rekursion in den Entities.

Die angegebenen Eigenschaften können von einem XML-Parser problemlos überprüft werden. Zur Wohlgeformtheit gehört weiterhin, dass jedes XML-Dokument ein Wurzelelement (*root tag*) hat, welches das gesamte Dokument umschließt. Alle Auszeichnungen, die im Inneren des Dokumentes angewendet werden, sind hierarchisch in das Wurzelelement eingeschlossen[2].

XML-Dokumente sollten immer wohlgeformt sein, da ansonsten Probleme bei der Verarbeitung auftreten.

#### Gültige Dokumente

Die zweite Stufe der Validierung ist die Gültigkeit. Gültige (*valid*) XML-Dokumente müssen zum einen wohlgeformt sein und außerdem eine XML-Deklaration und eine DTD (siehe Abschnitt 2.2) besitzen und dieser folgen. Dazu überprüft der Parser, ob alle Elemente, Attribute und Entities den Deklarationen in der DTD entsprechen, wobei diese Überprüfung teilweise sehr aufwändig sein kann.

[2]Das Dokument kann als Baum dargestellt werden.

## 2.2 DTD

Im Gegensatz zu HTML ist in XML die Elementmenge nicht vorgegeben. Jeder Nutzer kann Elemente nach seinen Anforderungen erstellen und verwenden. Diese Eigenschaft erschwert das automatische Verarbeiten von XML-Dokumenten durch Programme. Aus diesem Grund existiert ein optionaler Mechanismus, um die logische Struktur einer Dokumentklasse zu definieren.

Die *DTD* (**D**ocument **T**ype **D**efinition) eines Dokumentes erlaubt es, Elemente, ihre Attribute und die erlaubte Verschachtelung der Elemente festzulegen. Wird ein Dokument mittels der Einbindung einer DTD oder einer lokalen Definition einer Dokumentklasse zugeordnet, kann ein validierender XML-Parser das Dokument auf Gültigkeit überprüfen.

Es gibt vier verschiedene Deklarationen in XML: Elementtyp-, Attribut-, Entity- und Notationsdeklarationen. Auf diese wird im Folgenden detaillierter eingegangen.

### 2.2.1 Elementtypen

Elementtypdeklarationen (*element type declarations*) bestimmen die Namen und den Inhalt von Elementen. Die Deklaration

```
1 <!ELEMENT purchase (customer, product+)>
```

identifiziert ein Element namens „`purchase`“. Nach dem Namen folgt der erlaubte Inhalt des Elementes (*content model*). In diesem Fall muss ein `purchase`-Element genau ein `customer`- und ein oder mehrere `product`-Elemente enthalten. Alle aufgeführten Elemente müssen auch deklariert werden, damit der Parser das Dokument überprüfen kann.

Die folgenden Wertigkeiten können für das Auftreten von Elementen innerhalb anderer Elemente definiert werden. Sie werden direkt hinter dem Element notiert.

- **ohne** - Das Element muss exakt einmal enthalten sein.
- **?** - Das Element ist optional, kann jedoch maximal einmal enthalten sein.
- + - Das Element kann beliebig oft, muss aber mindestens einmal enthalten sein.
- * - Das Element ist optional und kann beliebig oft enthalten sein.

Die Elemente können mittels „`,`“ (AND, siehe Beispiel 2.2 Zeile 1) oder „`|`“ (XOR, siehe Beispiel 2.2 Zeile 6) verknüpft und beliebig geklammert werden. Werden die Elemente mittels des Kommas verknüpft, so wird explizit eine Reihenfolge angegeben, in der die Elemente dann in einem XML-Dokument auftreten müssen.

Die obige Deklaration definiert ein Element vom Typ *element content*. Weitere Typen sind *mixed content* (`PCDATA`, Parseable Character Data, beliebiger Text), *empty content*

(`EMPTY`, leeres Element ohne Endtag) oder *any content* (`ANY`, beliebiger Inhalt). Die Verwendung des `ANY`-Typs sollte möglichst vermieden werden, da dadurch sämtliche Validierungen für dieses Element deaktiviert werden.

Das folgende Beispiel zeigt die komplette Deklaration der Elemente für Beispiel 2.1.

**Beispiel 2.2.** - Elementdeklarationen für das Beispiel 2.1

```
1 <!ELEMENT purchase (customer, product+)>
2 <!ELEMENT customer EMPTY>
3 <!ELEMENT product  (amount, price, descr?)>
4 <!ELEMENT amount   (#PCDATA)>
5 <!ELEMENT price    (#PCDATA)>
6 <!ELEMENT descr    (#PCDATA | link)*>
7 <!ELEMENT link     (#PCDATA)>
```

### 2.2.2 Attribute

Für die Elemente können Attribute definiert werden. Die Deklaration legt dabei fest, welche Attribute ein Elemente haben kann (oder muss), welche Werte die Attribute annehmen können und setzt Standardwerte (*default value*), die dann gelten, wenn das Attribut am Elemente nicht vorhanden ist.

**Beispiel 2.3.** - Attributdeklarationen für das Beispiel 2.1

```
1 <!ATTLIST purchase id     ID #REQUIRED
2                    status ( accepted | delivered ) 'accepted'>
3 <!ATTLIST customer id     ID #REQUIRED>
4 <!ATTLIST product  id     CDATA #REQUIRED
5                    color  ENTITY #IMPLIED
6                    size   CDATA #IMPLIED>
```

Beispiel 2.3 zeigt die Attributdeklarationen für die Elemente aus dem Beispiel 2.1. Für das Element „`purchase`“ werden die zwei Attribute „`id`“ und „`status`“ festgelegt. Das Attribut „`id`“ ist als ID definiert und ein Pflichtattribut. Das Attribut „`status`“ besitzt eine fest vorgegebene Werteliste („`accepted`“ oder „`delivered`“) und den Standardwert „`accepted`“. Für das Element „`product`“ ist unter anderem das Attribut „`color`“ deklariert, welches ein Entity enthalten kann und optional ist.

Eine Attributdeklaration besitzt pro Attribut drei Elemente: den Namen, den Typ und den Standardwert. Es gibt sechs mögliche Attributtypen:

- `CDATA` - CDATA-Attribute enthalten beliebigen Text.
- `ID` - Über IDs können Elemente benannt werden. Die ID muss dokumentweit eindeutig sein. Elemente dürfen nur ein ID-Attribut haben.

- `IDREF` - Mittels IDREF-Attributen können Verweise auf andere Elemente im selben Dokument (über die ID) erstellt werden. IDREFS-Attribute fassen mehrere Verweise zusammen.
- `ENTITY` - ENTITY-Attribute halten eine Entityreferenz, ENTITIES-Attribute mehrere.
- `NMTOKEN` - NMTOKEN-Attribute sind eine restriktivere Form der CDATA-Attribute. Sie dürfen nur einen „Namen“ (ohne Leerzeichen) enthalten, NMTOKENS-Attribute entsprechend mehrere Namen durch Leerzeichen getrennt.
- `Werteliste` - Die möglichen Werte eines Attributes können festgelegt werden.

Für die Standardwerte gibt es vier verschiedene Werte:

- `#REQUIRED` - Es handelt sich um ein Pflichtattribut.
- `#IMPLIED` - Das Attribut ist optional.
- `#FIXED Wert` - Das Attribut ist eine Konstante mit dem festgelegten Wert.
- `ein Wert` - Das Attribut ist optional. Wenn es nicht angegeben wird, gilt dieser Wert als Standardwert.

Attributwerte werden bei der Verarbeitung vom Parser normalisiert, was bedeutet, dass Referenzen auf Zeichen oder Entities aufgelöst und an der entsprechenden Stelle eingefügt werden.

### 2.2.3 Entities

Mittels Entitydeklarationen kann man bestimmte Inhaltsfragmente benennen und an beliebigen Stellen im Dokument wiederverwenden. Die Fragmente können Text, Referenzen auf externe Dokumente beliebiger Art oder Teile einer DTD sein.

Beispiel 2.4 zeigt einige typische Entitydeklarationen.

**Beispiel 2.4.** - Entitydeklarationen

```
1 <!ENTITY blue "#0000FF">
2 <!ENTITY UNI "Otto-von-Guericke-Universität Magdeburg">
3 <!ENTITY legal SYSTEM "/include/legalnotice.xml">
4 <!ENTITY otto SYSTEM "/include/otto.jpg" NDATA JPEG>
```

Es gibt drei verschiedene Typen von Entitydeklarationen: interne, externe und Parameter Entities.

### Interne Entities

Interne Entities definieren einen Namen für einen beliebigen Text. Das erste und das zweite Entity im Beispiel 2.4 sind interne Entities. Wird im Dokument die Referenz „`&UNI;`“ benutzt, so wird sie durch „Otto-von-Guericke-Universität Magdeburg“ ersetzt. Interne Entities werden für häufig genutzten Text oder Text, der sich häufig ändert, wie die Version oder der Status eines Dokumentes, benutzt.

### Externe Entities

Externe Entities definieren Verweise auf den Inhalt von anderen Dateien. Diese Dateien können Text oder binäre Daten enthalten.

Das dritte Entity im Beispiel 2.4 ist ein externes Entity mit einer Referenz auf eine Textdatei. Wird im Dokument die Referenz „`&legal;`“ benutzt, so wird an dieser Stelle der *Inhalt* der Datei „/include/legalnotice.xml“ eingefügt. Der Inhalt wird vom XML Parser genau so interpretiert, als ob er wirklich an dieser Stelle stehen würde.

Das vierte Entity im Beispiel 2.4 verweist auf eine binäre Datei. Binäre externe Entities sind an der zusätzlichen Notation („`NDATA`“) zu erkennen (siehe Abschnitt 2.2.4).

### Parameter Entities

Parameter Entities können nur in der DTD und nicht im Dokument verwendet werden. Sie werden durch ein „`%␣`“ (Prozent und Leerzeichen) gekennzeichnet, das Prozentzeichen wird auch anstelle des Ampersand zur Referenzierung verwendet. Parameter Entities können zum einen verwendet werden, um in der DTD semantisch gleiche Sachverhalte zu verdeutlichen, und zum anderen, um DTDs zu modularisieren (die Parameter Entities funktionieren als Include-Anweisung für externe DTDs).

Gegeben seien die folgenden Elementdeklarationen.

```
1 <!ELEMENT description (#PCDATA | br | link | b | i)*>
2 <!ELEMENT details (#PCDATA | br | link | b | i)*>
```

Die beiden Elemente „`description`“ und „`details`“ sind bis auf den Namen gleich. Um zu verdeutlichen, dass die Elemente den gleichen Inhalt haben, oder um bei möglichen Änderungen an der DTD sicherzustellen, dass sie immer den gleichen Inhalt haben, kann der folgende Parameter Entity verwendet werden.

```
1 <!ENTITY % textcontent "#PCDATA | br | link | b | i">
```

Die Elemente werden dann wie folgt definiert.

```
1 <!ELEMENT description (%textcontent;)*>
2 <!ELEMENT details (%textcontent;)*>
```

### 2.2.4 Notationen

Notationsdeklarationen bestimmen einen Typ für externe binäre Daten. Zusätzlich kann eine externe Anwendung (*helper application*) definiert werden, welche in der Lage ist, die Daten zu verarbeiten, wie beispielsweise ein Programm zum Anzeigen eines Bildes. Beispiel 2.5 definiert eine Notation namens „`JPEG`“ (die im Beispiel 2.4 in der vierten Zeile verwendet wird) und definiert als *helper application* das Programm „`/usr/bin/xview`“.

**Beispiel 2.5.** - Notationsdeklarationen

```
1 <!NOTATION JPEG PUBLIC "-//JPEG//EN" "/usr/bin/xview">
```

### 2.2.5 Lokalisierung von Ressourcen

Für die Lokalisierung von externen Ressourcen existieren zwei verschiedene Mechanismen.

Über das Schlüsselwort „`SYSTEM`“ wird ein sogenannter *System-Identifier* bestimmt, der den Speicherort der gewünschten Information angibt. Dabei kann es sich um einen relativen oder absoluten *URI* (**U**niform **R**esource **I**dentifier, ein Verweis auf eine Ressource) handeln. Zu sehen ist die Verwendung des System-Identifiers im Beispiel 2.4.

Die zweite Möglichkeit stellt ein *Public-Identifier* dar, der über das Schlüsselwort „`PUBLIC`“ angegeben wird. Hierbei übernimmt ein Entity-Manager mittels eines Kataloges die Abbildung des Public-Identifiers auf einen System-Identifier. Anwendungen für Public-Identifier sind unternehmensweite oder gar globale Ressourcen, die jedoch ihren Speicherort ändern können. Ein Public-Identifier minimiert dabei die Pflege aller XML-Dokumente, die diese Ressource nutzen. Um jedoch kompatibel zu XML-Parsern zu sein, die einen Public-Identifier nicht auflösen können, muss immer ein System-Identifier mit angegeben werden. Im Beispiel 2.5 wird ein Public-Identifier zur Identifizierung einer Anwendung zum Anzeigen von JPEG-Graphiken verwendet. Im Katalog des Entity-Managers ist unter dem Schlüssel „`-//JPEG//EN`“ der Verweis auf eine entsprechende Anwendung hinterlegt. Kann dieser Schlüssel durch den XML-Parser nicht aufgelöst werden, so ist zusätzlich mit „`/usr/bin/xview`“ eine Alternative angegeben.

## 2.3 Namensräume

Um XML-Dokumente sinnvoll verarbeiten zu können, ist die Angabe einer DTD sehr sinnvoll. Jedoch wird es in der Praxis oft vorkommen, dass ein Dokument Elemente und Attribute aus verschiedenen DTDs verwendet. So kann man sich ein technisches Dokument vorstellen, welches eine eigene DTD besitzt, jedoch zusätzlich einige mathematische Gleichungen in MathML und verschiedene Graphiken in SVG enthält.

Damit eine Anwendung den Elementen (die durchaus in mehreren DTDs unterschiedlich definiert sein können) die richtige Semantik zuordnet, muss eine Angabe der richtigen

DTD möglich sein. In XML wird dieses mittels sogenannter Namensräume (*Namespaces*, siehe [Wor99a]) gelöst.

Ein Namensraum wird durch das Attribut „`xmlns`“, welches als Wert den URI der DTD[3] besitzt, definiert. Bei der Verwendung von Namensräumen gibt es zwei Möglichkeiten.

Die erste Möglichkeit besteht in der Verwendung von „Dateninseln“. Hierbei gilt der Namensraum für das Element, welches ihn enthält, und alle untergeordneten Elemente. In einem untergeordneten Element kann dann wieder ein anderer Namensraum definiert werden, welcher den übergeordneten überschreibt.

Die zweite Alternative erlaubt das Mischen von Namensräumen. Das Attribut „`xmlns`“ wird durch einen Doppelpunkt und einen lokalen Namen für den Namensraum erweitert.

**Beispiel 2.6.** - Verwendung von Namensräumen

```
1 <section xmlns:ml="http://www.w3.org/TR/REC-MathML/">
2         <para>
3                 The fraction 3/4 can be expressed in MathML as:
4                 <ml:cn type="rational">3<ml:sep/>4</ml:cn>.
5         </para>
6 </section>
```

Der Name des Namensraumes wird dann im Weiteren als Präfix (separiert durch einen Doppelpunkt) vor den Elementen oder Attributen angegeben. Im Beispiel 2.6 wird in der ersten Zeile der Namensraum „`ml`“ definiert, die dazugehörige DTD findet sich unter „http://www.w3.org/TR/REC-MathML/“. Innerhalb dieses `section`-Elementes kann der Namensraum durch Angabe des Präfixes „`ml:`“ verwendet werden, wie in Zeile 4 zu sehen.

Die Verwendung von Namensräumen im Zusammenspiel mit DTDs ist jedoch nicht so einfach, wie es auf den ersten Blick scheint. Sollen die Dokumente mittels einer DTD validiert werden, so müssen in der DTD auch die Elemente und Attribute (inclusive der Definition des Namensraumes), die zu einem anderen Namensraum gehören, definiert sein.

## 2.4 XML Schema

XML Schema (siehe [Wor01c], [Wor01d] und [Wor01e]) ist eine weitere Empfehlung (*recommendation*) des $W_3C$, um eine bessere Beschreibung der Dokumentstruktur als dies

[3]Hierbei ist anzumerken, dass es sich bei der URI nicht um eine real existierende Ressource handeln muss. Es ist natürlich sinnvoll, wenn unter der angegebenen URI die DTD, welche für den Namensraum gilt, zu finden ist, zwingend vorgeschrieben ist es jedoch nicht. Der einzige Zweck der URI-Angabe bei der Definition eines Namensraumes besteht in der (weltweit) eindeutigen Identifikation. Aus diesem Grund wird die Verwendung von Domainnamen, für die eine Eindeutigkeit garantiert ist, in der URI empfohlen.

mittels DTDs vorgenommen werden kann, zu ermöglichen. XML Schema, welches selber wieder in XML notiert wird, kann als Ersatz oder Erweiterung für DTDs verwendet werden.

Neben der Beschreibung der logischen Struktur von XML-Dokumenten bietet XML Schema unter anderem die folgenden Möglichkeiten:

- (Neben-)bedingungen für Aufbau der Dokumentstruktur (z.B. Anzahl bestimmter Elemente)
- Bedingungen über Datentypen für Elemente und Attribute (z.B. Integer, Float, URL)
- Objektorientierung und Vererbung
- Identity Constraints (*key*, *keyref*, *unique*, zusammengesetzte Schlüssel)
- Einschränkung oder Erweiterung des Datentyps durch Facetten (Muster, Aufzählungen, Längenbeschränkungen, Wertebereichsbeschränkungen, Genauigkeit bei Zahlen)
- Anmerkungen (*annotation*) sowohl als Dokumentation für den Nutzer (*documentation*) als auch für Anwendungen (*appinfo*)

Das Beispiel 2.7 zeigt das XML Schema[4] für das XML-Dokument aus dem Beispiel 2.1.

**Beispiel 2.7.** - (Unvollständiges) XML Schema für das Beispiel 2.1.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
3   <xsd:element name="purchase">
4    <xsd:complexType>
5     <xsd:sequence>
6      <xsd:element name="customer" type="empty" minOccurs="1" maxOccurs="1">
7       <xsd:attribute name="id" type="xsd:ID" use="required"/>
8      </xsd:element>
9      <xsd:element name="product" type="product" minOccurs="1" maxOccurs="unbounded"/>
10    </xsd:sequence>
11    <xsd:attribute name="id" type="xsd:ID" use="required"/>
12   </xsd:complexType>
13  </xsd:element>
14  <xsd:complexType name="product">
15   <xsd:sequence>
16    <xsd:element name="amount" type="xsd:positiveInteger" minOccurs="1" maxOccurs="1"/>
```

[4]Die Definition des `descr`- und `link`-Elementes wurde zur Erhöhung der Übersichtlichkeit weggelassen.

```
17    <xsd:element name="price" type="xsd:double" minOccurs="1" maxOccurs="1"/>
18    <!-- Beschreibung des descr-Elementes ausgelassen -->
19   </xsd:sequence>
20   <xsd:attribute name="name" type="xsd:String" use="required"/>
21   <xsd:attribute name="color" type="xsd:String"/>
22  </xsd:complexType>
23 </xsd:schema>
```

## 2.5 Anfragen auf XML-Daten

Es gibt mittlerweile diverse Anfragesprachen, wie zum Beispiel Lorel [AQM+97], Quilt [CRF00], UnQL [BDHS96], XML-QL [DFLS], XQL [RLS98] oder YATL [CJS99], um Anfragen auf XML-Daten zu realisieren.

### 2.5.1 XPath

Um Informationen in einem XML-Dokument zu adressieren, wird die *XML Path Language* (XPath, siehe [Wor99b]) verwendet. Dabei ist XPath keine Anfragesprache, da die Adressierung nur ein Teil einer Anfragesprache ist. Der andere umfasst alle Möglichkeiten, (Zwischen-)Ergebnisse zu transformieren und (mit anderen) zu integrieren. Jedoch bildet XPath die Basis für viele Erweiterungen um XML, wie XSLT (siehe Abschnitt 2.7.1) oder XPointer (siehe Abschnitt 2.6.2).

Der Zugriff auf Informationen mittels XPath basiert auf einer Baumdarstellung der XML-Daten (vgl. Abbildung 2.1). Das Dokumentmodell, welches für XPath verwendet wird, besteht aus sieben verschiedenen Knotentypen: dem Wurzelelement (*root*), Elementen, Text, Attributen, Kommentaren, Processing Instructions und Namespaces. Durch dieses abstrakte Modell werden die Daten normalisiert, das heißt, es besteht kein Unterschied zwischen einem Text, der in einer CDATA-Section definiert, und einem Text, der über ein externes Entitiy eingebunden wurde.

Das Beispiel 2.8 zeigt verschiedene XPath-Ausdrücke. Diese werden im Folgenden genauer betrachtet.

**Beispiel 2.8.** - Verschiedene XPath-Ausdrücke

```
1 /purchase
2 purchase[3]/product[1]
3 //purchase[customer]
4 //customer[.='Company1']
5 id('p001')//product[@name='prod345']
```

Knoten aus diesem XML-Baum werden mittels Pfadausdrücken (ähnlich den Pfaden in Dateisystemen) selektiert. Die Syntax orientiert sich stark an Unix-Dateisystemen:

- Der Schrägstrich („/“) wird als Pfadseparator verwendet.
- Das Wurzelelement wird mittels eines Schrägstriches am Anfang des Pfadausdruckes referenziert.
- Das übergeordnete Element wird durch zwei Punkte (“..“) gekennzeichnet.
- Der aktive Knoten wird durch einen Punkt (“.“) symbolisiert.

Sonstige Ausdrücke sind immer relative Pfade vom aktuellen Knoten ausgehend. In der Zeile 1 des Beispiels 2.8 werden alle `purchase`-Elemente, die direkt unter dem Wurzel-Element notiert sind, selektiert.

Das Ergebnis einer XPath-Anfrage ist eine Menge von Knoten aus dem XML-Baum. Um auf bestimmte Knoten zugreifen zu können, kann ein Index in eckigen Klammern angegeben werden. Im Beispiel 2.8 wird in der zweiten Zeile vom aktuellen Knoten aus vom dritten `purchase`-Element das erste `product`-Element selektiert.

Die eckigen Klammern werden nicht nur für den Indexzugriff verwendet, sondern können weitergehende Filter enthalten. So ist es möglich, auf die Existenz oder die Werte von Elementen oder Attributen zu prüfen. In der dritten Zeile des Beispiels 2.8 werden alle `purchase`-Elemente[5] selektiert, die ein Subelemente namens „`customer`“ besitzen. Die vierte Zeile gibt alle `customer`-Knoten zurück, die den String-Wert[6] „`Company1`“ besitzen.

Weitere Möglichkeiten von XPath umfassen Wildcards, Vergleiche sowie einfache Rechenoperationen. Zusätzlich werden diverse Funktionen angeboten, wie beispielsweise „`id(...)`“ (Zugriff über die ID), Positionsfunktionen („`last()`“, „`position()`“ oder „`count(...)`“), Stringoperationen („`substring(string, idx)`“, „`string-length()`“, „`concat(string1, string2, ...)`“ oder „`normalize-space()`“) und Funktionen zum Konvertieren zwischen verschiedenen Datentypen („`string(...)`“ oder „`boolean(...)`“). Eine Funktion wird im Beispiel 2.8 in der fünften Zeile verwendet. Zuerst wird das Element mit der ID „`p001`“ selektiert. Anschließend wird in allen Subelementen (egal wie viele Hierarchieebenen dazwischen liegen) namens „`product`“ das Attribut „`name`“ geprüft. Ist der Wert gleich „`prod345`“, wird das Element in die Ergebnismenge aufgenommen. Würde diese Anfrage auf das Beispiel 2.1 angewendet werden, würde das Ergebnis der Abfrage die Zeilen 6 bis 13 aus Beispiel 2.1 enthalten.

### 2.5.2 XML-QL

Wie die meisten Erweiterungen rund um XML arbeitet auch XML-QL ([DFLS]) auf einem abstrakten Datenmodell. Die XML-Daten werden als Baum dargestellt, wobei

[5] Der doppelte Schrägstrich („//“) selektiert alle Subelemente, egal auf welcher Hierarchieebene sie sich befinden.

[6] Um den String-Wert eines Elementes zu bestimmen, werden sämtliche Tags entfernt. Übrig bleiben alle Textinhalte des Elementes und aller Subelemente. Die aneinandergehängten Textfragmente bilden den String-Wert.

Kommentare und Processing Instructions sowie die relative Ordnung der Elemente ignoriert werden. Für jeden Knoten wird (wenn noch nicht vorhanden) eine ID vergeben. Die Blätter des Baumes enthalten nur Textdaten.

Anfragen werden in XML-QL nach einer Art *Query-by-Example* gestellt. Sie bestehen aus den beiden Klauseln `WHERE` (die Selektionsbedingungen) und `CONSTRUCT`, welche angibt, wie das Ergebnis gebildet werden soll.

Die Selektion in XML-QL erfolgt durch *element patterns*. Im Beispiel 2.9, welches eine einfache XML-QL-Anfrage[7] zeigt, werden alle `book`-Elemente selektiert, die mindestens ein `title`-, ein `author`- und ein `publisher`-Element, dessen `name`-Element gleich „Addison-Wesley“ ist, enthalten. Für jedes gefundene `book`-Element werden der/die Autor/en (Vor- und Nachname) und der Titel an die Variablen „`a`“ beziehungsweise „`t`“ gebunden.

**Beispiel 2.9.** - XML-QL-Anfrage

```
1  WHERE
2    <bib>
3      <book>
4        <publisher><name>"Addison-Wesley"</name></publisher>
5        <title> $t </title>
6        <author> $a </author>
7      </book>
8    </bib> IN "bib.xml"
9  CONSTRUCT
10   <result>
11     <author> $a </author>
12     <title> $t </title>
13   </result>
```

Die `CONSTRUCT`-Klausel einer Abfrage gibt an, in welcher Form das Ergebnis ausgegeben werden soll. Die in der Selektion verwendeten Variablen werden an dieser Stelle in der gewünschten Ausgabeform aufgeführt. Der komplette Text der `CONSTRUCT`-Klausel wird für jeden Treffer der Abfrage mit den entsprechenden Werten ausgegeben. Im Beispiel 2.9 wird für jedes gefundene `book`-Element der/die Autor/en und der Titel durch ein `result`-Element gruppiert. Das Ergebnis der Abfrage ist jedoch nur ein XML-Fragment, da kein *root*-Element vorhanden ist, sondern nur eine Auflistung von `result`-Elementen mit den entsprechenden Werten.

Neben diesen einfachen Anfragen unterstützt XML-QL auch kompliziertere Selektionsbedingungen (Tag-Variablen, um auf Tags zuzugreifen, *regular path expressions*, um Selektionen über Schachtelungstiefen oder Pfadausdrücke durchzuführen), geschachtelte Anfragen, Generierung von Objekt-IDs über Variablen, Element-Joins über Werte, Nutzung von verschiedenen Datenquellen oder benutzerdefinierte Funktionen.

[7] Die hier vorgestellte Anfrage ist [DFLS] entnommen, wo auch detaillierter die zugrunde liegenden XML-Daten beschrieben werden. Es handelt sich dabei um eine bibliographische Auflistung von Büchern und Artikeln mit Angaben bezüglich Titel, Autor(en) (Vor- und Nachname), Verleger und Jahr.

## 2.6 Verknüpfen von XML-Dokumenten

Die Möglichkeit, Dokumente durch Verweise (*Links*) untereinander zu vernetzen, ist bereits aus HTML bekannt. Auch in XML sind hierfür Technologien vorgesehen, die weit über die Beschränkungen aus HTML hinausgehen.

Das $W_3C$ hat zum Verlinken von XML-Dokumenten XLink und XPointer vorgesehen.

### 2.6.1 XLink

XLink (siehe [Wor01f]) bietet zwei verschiedene Möglichkeiten, XML-Daten zu verlinken, einfache (*simple*) und erweiterte (*extended*) Links. Dabei können zum Verknüpfen von XML-Dateien entweder die durch XLink spezifizierten Elemente oder eigene Elemente mit XLink-Attributen verwendet werden. Um die Verweise zu verwenden, muss der XLink-Namensraum eingebunden werden.

```
1 <my_element xmlns:xlink="http://www.w3.org/1999/xlink"> ...
```

Werden nicht die Elemente aus XLink verwendet, so stellt XLink eine Menge von *globalen Attributen* zur Verfügung, um zu spezifizieren, ob ein selbstdefiniertes Element ein Verweis ist und um verschiedene Eigenschaften für die Verweise (z.B. wann die verknüpften Ressourcen geladen oder wie sie dargestellt werden) festzulegen. Folgende Attribute werden durch XLink definiert:

- Typ Definition: `type`
- Lokalisierungsattribute: `href`
- Semantische Attribute: `role`, `arcrole`, `title`
- Verhaltensattribute: `show`, `actuate`
- Verknüpfungsattribute: `label`, `from`, `to`

Im Folgenden werden einige Attribute und ihre Verwendung genauer betrachtet.

Das `type`-Attribut kann sechs verschiedene Werte annehmen. Diese entsprechen den in XLink definierten Elementen. `simple` für einen einfachen bzw. `extended` für einen erweiterten Link (z.B. multiple Verweise auf unterschiedliche Ressourcen), `locator` für einen Verweis auf eine externe und `resource` für eine interne Resource, `arc`, um Beziehungen zwischen Ressourcen abzubilden und `title`, um ein anderes Linkelement zu beschriften.

Das `type`-Attribut (oder das entsprechende Element) bestimmt, welche Attribute und Subelemente notwendig bzw. erlaubt sind. So ist ein einfacher Verweis („`simple`“) mit einem `href`-Attribut vollständig, weitere Attribute sind zum Teil optional (z.B. „`title`“ oder „`role`“) oder nicht erlaubt (z.B. „`from`“) und Subelemente werden nicht betrachtet, während bei einem erweitertem Verweis („`extended`“) mindestens ein `locator`- oder `resource`-Subelement vorhanden sein muss.

### Einfache Verweise

Einfache Verweise entsprechen mit einigen kleinen Erweiterungen den Links aus HTML.

Ein einfacher Verweis ist in Beispiel 2.10 zu sehen (der Namensraum „`xlink`“ ist wie oben definiert), wobei in diesem Beispiel das Element beliebig heißen kann, da der XLink mittels des Attributes „`xlink:type`“ definiert ist. Der Wert des Attributes „`xlink:href`“ kann eine URL, eine Abfrage oder ein XPointer (siehe 2.6.2) sein. Der gleiche Link bei Verwendung des `simple`-Elementes aus XLink ist im Beispiel 2.11 zu sehen.

**Beispiel 2.10.** - Einfacher Verweis durch Attribute

```
1 <slink xlink:type="simple" xlink:href="locator" xlink:title="simple link">
2         some text
3 </slink>
```

**Beispiel 2.11.** - Einfacher Verweis durch Elemente

```
1 <xlink:simple href="locator" title="simple link">
2         some text
3 </xlink:simple>
```

### Erweiterte Verweise

Erweiterte Verweise gibt es in dieser Form in HTML nicht. Mit erweiterten Verweisen können Beziehungen zwischen mehreren Ressourcen dargestellt werden.

Das Beispiel 2.12 zeigt einen erweiterten Verweis für einen literarischen Text. Neben dem reinen Text (Zeile 2) sind mehrere Anmerkungen (Zeilen 3 und 4) sowie eine Kritik (Zeile 5) in dem Verweis enthalten.

**Beispiel 2.12.** - Erweiterter Verweis

```
1 <xlink:extended role="annotation">
2         <xlink:locator href="text">The Text</xlink:locator>
3         <xlink:locator href="annot1">Annotations</xlink:locator>
4         <xlink:locator href="annot2">More Annotations</xlink:locator>
5         <xlink:locator href="litcrit">Literary Criticism</xlink:locator>
6 </xlink:extended>
```

## 2.6.2 XPointer

Die Lokalisierung der Ressourcen in XLink ist entweder global (die gesamte Datei) oder basiert auf dem ID/IDREF-Mechanismus von XML (siehe Abschnitt 2.2.2). Diese Möglichkeiten sind jedoch nicht immer ausreichend, besonders wenn man auf Elemente

von Dokumenten verweisen möchte, die nicht mit IDs gekennzeichnet sind oder für die man kein Schreibrecht besitzt.

Um diese Beschränkungen zu umgehen, wird vom $W_3C$ XPointer (*Extended Pointer*, siehe [Wor01b]) entwickelt. XPointer erlaubt die Lokalisierung von Ressourcen (in XML-Dokumenten) durch Traversieren durch den Elementbaum. XPointer basiert dabei auf XPath (siehe Abschnitt 2.5.1). Zum Beispiel lokalisiert der XPointer „`http://www.foo.org/bar.xml#xpointer(/article/section[position()<=5])`“[8] die ersten fünf `section`-Elemente vom *root*-Element („`article`“) in dem angegebenen Dokument.

Neben einzelnen Elementen können mittels XPointer auch Regionen oder einzelne Zeichen aus Textknoten selektiert werden. Der Link auf die selektierten Ressourcen wird dann mittels XLink erstellt, *ohne* dass das Zieldokument verändert werden muss.

## 2.7 XSL

Die **E**xtensible **S**tylesheet **L**anguage (XSL, siehe [Wor01g] und [Hol01]) bietet Möglichkeiten zum Anzeigen von XML-Daten. Dabei können die Informationen für verschiedene Zielgruppen unterschiedlich formatiert werden. Das Umwandeln der Daten wird bei XSL in zwei Prozesse unterteilt, XSLT zum Transformieren der XML-Daten in eine passende Zwischenrepräsentation und XSL-FO zum Formatieren der transformierten Daten (siehe Abbildung 2.2).

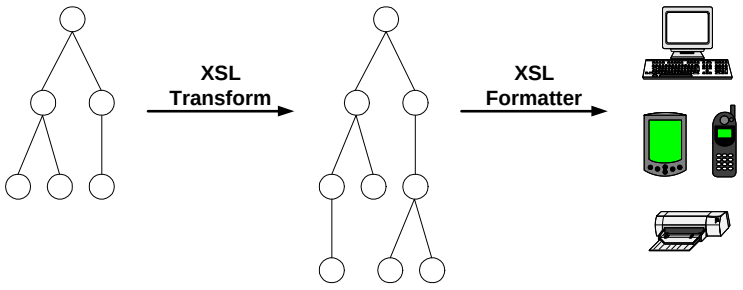


Abbildung 2.2: Der XSL-Prozess: Transformieren und Formatieren (aus [Wor01g], modifiziert)

[8] Das Kleiner-Als-Zeichen („`<`“) wird, da es in XML schon für die Kennzeichnung der Tags belegt ist, durch das vordefinierte Entity „`<`“ ausgedrückt (siehe Abschnitt 2.1.2).

### 2.7.1 Transformation

Bei XSLT (**E**xtensible **S**tylesheet **L**anguage **T**ransformations, siehe [Wor99c]) handelt es sich um eine Transformationssprache zum Umwandeln von XML-Daten. Ein sogenanntes *Stylesheet* (welches selbst ein XML-Dokument ist) beschreibt die Transformation, welche auf *Templates* basiert. Der XSLT-Prozessor verarbeitet die XML-Daten und das XSLT-Stylesheet und erstellt den Templates entsprechend inkrementell das Ergebnis.

XSLT wurde hauptsächlich entwickelt, um XML-Daten in eine andere XML-Repräsentation zu transformieren. Zusätzlich kann das Ergebnis auch als HTML oder reiner Text ausgegeben werden.

Ein Template beschreibt, wie ein XML-Wert ausgegeben und unter welchen Bedingungen es durch den XSLT-Prozessor auf die XML-Daten angewendet werden soll. Dabei werden die Bedingungen durch XPath-Ausdrücke (siehe Abschnitt 2.5.1) dargestellt. Innerhalb des Templates kann dann entweder die Verarbeitung wieder an den XSLT-Prozessor abgegeben werden, welcher dann weitere Templates verarbeiten kann, oder explizit andere Templates aufgerufen werden. XSLT-Templates sind in den Beispielen 2.13 und 2.15 zu sehen. In beiden wird ein `customer`-Element formatiert, im Beispiel 2.13 zu HTML und im Beispiel 2.15 zu XSL-FO, welches beispielsweise als PDF ausgegeben werden kann.

**Beispiel 2.13.** - XSLT-Template zum Transformieren zu HTML.

```
1 <xsl:template match="customer">
2         <p>
3                 <xsl:text>Customer: </xsl:text>
4                 <b><xsl:value-of select="@id"/></b>
5         </p>
6 </xsl:template>
```

Würde man ein komplettes Stylesheet, welches das Template aus Beispiel 2.13 enthält, auf das Beispiels 2.1 anwenden, so würde das (nur durch das Template erzeugte) Ergebnis wie im Beispiel 2.14 aussehen.

**Beispiel 2.14.** - Ergebnis des XSLT-Templates aus Beispiel 2.13 bei der Anwendung auf das Beispiel 2.1.

```
1         <p>Customer: <b>cust123</b></p>
```

Auch die XSLT-Verarbeitung durch den XSLT-Prozessor basiert auf einer Baumrepräsentation der XML-Daten. Der Prozessor traversiert durch den XML-Baum und prüft jeweils, ob ein Template auf das aktuelle Element angewendet werden muss.

Neben dem Template-System können mittels XPath-Selektionen beliebige Knoten aus dem XML-Baum selektiert und iterativ, zum Beispiel durch eine `for-each`-Schleife (wenn gewünscht auch sortiert), verarbeitet werden.

### 2.7.2 Formatierung

Der zweite Teil des XSL-Prozesses beschäftigt sich mit dem Formatieren der Daten aus dem XSLT-Prozess. Für die Formatierung gibt es diverse Möglichkeiten. Wenn wie in Beispiel 2.13 die Daten zu HTML umgewandelt wurden, ist eine Formatierung in dem Sinne nicht mehr notwendig, da der Browser die Darstellung übernimmt.

Sollen die Daten in anderen Formaten, wie zum Beispiel durch Darstellung am Bildschirm, einen Ausdruck oder auch durch Vorlesen präsentiert werden, so bietet XSL ein abstraktes Datenmodell zur Beschreibung des Ergebnisses. Diese Beschreibung umfasst unter anderem einen sogenannten *area tree*, welcher die Positionierung aller Objekte beschreibt. Die *formatting objects* bestimmen dann durch vordefinierte Klassen und zusätzliche Attribute die genaue Darstellung. Das Beispiel 2.15 zeigt ein XSLT-Template zum Umwandeln eines `customer`-Elements zu einem *formatting object*-Block.

**Beispiel 2.15.** - XSLT-Template zum Transformieren zu XSL-FO.

```
1 <xsl:template match="customer">
2         <fo:block space-before.optimum="20pt" font-size="20pt">
3                 <xsl:text>Customer: </xsl:text>
4                 <fo:inline-sequence font-weight="bold">
5                         <xsl:value-of select="@id"/>
6                 </fo:inline-sequence>
7         </fo:block>
8 </xsl:template>
```

Das Ergebnis der Transformation des Beispiels 2.1 durch das Template in Beispiel 2.15 ist im Beispiel 2.16 zu sehen.

**Beispiel 2.16.** - Ergebnis des XSLT-Templates aus Beispiel 2.15 bei der Anwendung auf das Beispiel 2.1.

```
1 <fo:block space-before.optimum="20pt" font-size="20pt">
2         Customer:
3         <fo:inline-sequence font-weight="bold">cust123</fo:inline-sequence>
4 </fo:block>
```

Die Formatierungsbeschreibung wird dann von einer *render engine* verarbeitet, welche die wirkliche Ausgabe übernimmt. Das Beispiel 2.15 könnte beispielsweise als ein PDF-Dokument oder auf einem Drucker ausgegeben werden.

# Kapitel 3

# Speicherung von XML in Datenbanken

Aufgrund der wachsenden Bedeutung von XML ist eine Integration von XML in Datenbanksysteme notwendig ([CFP00, Wid99]). Bezüglich des verwendeten Systems gibt es vier verschiedene Alternativen ([Bou00, FK99, TDCZ00])[1] [2]:

- Spezialdatenbanksysteme (SDBS), wie Rufus ([Se93]), Lore ([GMW99]), Strudel ([FLM99]), Natix ([KM00]), eXcelon ([Obj99]) oder Tamino ([SW00]), die für semistrukturierte Daten oder explizit für XML entwickelt wurden

- objektorientierte Datenbanksysteme ([BA96]), welche aufgrund der guten Modellierungsmöglichkeiten XML zum Beispiel als DOM-Bäume oder in der Dokumentstruktur entsprechenden Objekten speichern können

- objekt-relationale Datenbanksysteme, welche durch Schachtelungen die Abbildung der Struktur von XML-Dokumenten unterstützen

- relationale Datenbanksysteme

Um zu entscheiden, welche Technologie angewendet werden soll, wird oft die Unterscheidung zwischen daten- und dokumentorientierten XML-Dokumenten angeführt ([Bou00]).

Datenorientierte Dokumente besitzen eine ausgeprägte Struktur, feingranulare Datenelemente (Daten treten nur in PCDATA-Elementen oder Attributen auf) und selten

[1] Zur Speicherung von XML-Dokumenten können neben dem Dateisystem auch *Dokument* oder *Content Management Systeme* verwendet werden. Da diese aber kaum dem Begriff Datenbank entsprechen, werden sie nicht weiter betrachtet.

[2] Die Reihenfolge, in der die verschiedenen Technologien vorgestellt werden, entspricht der Eignung zur *reinen Speicherung* von XML-Daten (jedoch sollte aus dieser Reihenfolge keine Wertung abgeleitet werden).

Mixed Content. Die Reihenfolge der Elemente ist oft irrelevant. Diese Art von XML-Dokumenten findet man oft, wenn XML als Austauschformat zwischen verschiedenen Anwendungen genutzt wird.

Im Gegensatz dazu haben dokumentorientierte XML-Dokumente wenig Struktur, grobe Datenelemente (Elemente mit Mixed Content oder gar das gesamte Dokument selbst) und viele Elemente mit Mixed Content. Die Reihenfolge in dokumentorientierten Dokumenten ist fast immer relevant. Dies Art von Dokumenten wird oft von Menschen erstellt und genutzt (Bücher, Emails, jedes XHTML-Dokument, ...).

Die Grenze zwischen daten- und dokumentorientiert ist sehr fließend. So können datenorientierte Dokumente teilweise kaum strukturierte Elemente mit Mixed Content enthalten, zum Beispiel eine detaillierte Beschreibung der Teile in einem Angebot. Andererseits gibt es dokumentorientierte XML-Dokumente, die fein strukturierte Daten[3] enthalten, wie beispielsweise bibliographischen Angaben (Autor, Title, Verlag, ...) in einem Buch.

Die vier oben aufgeführten Datenbanktechnologien eignen sich unterschiedlich gut für daten- bzw. dokumentorientierte Dokumente.

**Spezialdatenbanksysteme** ([Se93, GMW99, FLM99, KM00, Obj99, SW00]) eignen sich sehr gut für dokumentorientierte XML-Dokumente, da nicht versucht werden muss, für semi- oder nicht strukturierte Daten eine Struktur zu finden. Auch für datenorientierte XML-Dokumente können SDBS verwendet werden.

Die Vorteile von SDBS sind die Speicherung von beliebigen XML-Dokumenten, die gute Unterstützung von XML-Anfragesprachen und die Verarbeitungsgeschwindigkeit. SDBS können XML-Dokumente (physisch) als Ganzes speichern, wodurch keinerlei Transformationen bei der Speicherung und der Ausgabe notwendig sind. Weiterhin kann bei Anfragen das ganze Dokument mit einem Zugriff erreicht werden ([TDCZ00]) im Gegensatz zu x Anfragen mit y Verbundoperationen, die beispielsweise in einem RDBS nötig sind, um das Dokument wieder zusammenzusetzen ([STZ+99]). Auf der anderen Seite gilt dieser Vorteil nur so lange, wie die Daten in der Struktur und Reihenfolge, in der sie vorliegen, abgefragt werden. Wird eine andere Sicht auf die Daten verlangt, so *kann* es für ein SDBS notwendig sein, *alle* Dokumente zu parsen, um eine Anfrage zu beantworten.

Weitere Nachteile der Spezialdatenbanksysteme ergeben sich aus der Tatsache, dass sie im Vergleich zu beispielsweise RDBS noch sehr jung sind. Ein Teil der für Datenbankmanagementsysteme geforderten Funktionalitäten ([Cod82]) sind in SDBS nicht oder nur sehr rudimentär vorhanden, wie beispielsweise Benutzersichten, Datenschutz, Transaktionen oder Synchronisation. Aus diesem Grund besitzen SDBS momentan ein eingeschränktes Anwendungsfeld ([FK99], [KKRSR00]), auf dem sie aber durch die anderen vorgestellten Technologien kaum zu schlagen sind.

[3] Oft handelt es sich um Metadaten.

**Objektorientierte Datenbanksysteme** können sowohl für daten- als auch für dokumentorientierte Dokumente verwendet werden. So können beliebige XML-Dokumente zum Beispiel als DOM-Bäume gespeichert werden. Bei datenorientierten Dokumenten kann durch geeignete Klassen die Struktur der Dokumente adäquat nachgebildet werden.

Jedoch gelten die zuletzt aufgeführten Nachteile der Spezialdatenbanksysteme zu großen Teilen auch für objektorientierte Datenbanksysteme.

**Objekt-relationale Datenbanksysteme** unterstützen durch die objektorientierten Eigenschaften die Abbildung der Struktur vom XML-Dokumenten. Da sich objekt-relationale Datenbanksysteme in den meisten Fällen durch einen evolutionären Prozess aus relationalen Datenbanksystemen entwickelt haben, werden sie im Folgenden, soweit nicht anders erwähnt, diesen zugeordnet und nicht getrennt betrachtet.

**Relationale Datenbanksysteme** sind von Haus aus nicht besonders gut zur Speicherung von XML-Dokumenten geeignet. Die empfohlene Normalisierung der Struktur und die mengenwertige Speicherung in relationalen Systemen widerspricht den Eigenschaften von XML. Am einfachsten ist noch die Abbildung von datenorientierten XML-Dokumenten, jedoch können sehr komplexe Schemata entstehen. Anfragen und die Rekonstruktion der Dokumente resultieren oft in umfangreichen SQL-Anfragen mit vielen Verbundoperationen.

Trotz all dieser Nachteile sprechen viele Fakten für die Verwendung von (objekt-)relationalen Datenbanksystemen zur Speicherung vom XML. Relationale Datenbanksysteme repräsentieren eine etablierte, ausgereifte und wohlerforschte Technologie. Sie sind weit verbreitet und ermöglichen dadurch eine einfache Integration der bereits vorhandenen Datenbestände. Im Gegensatz zu Spezial- und objektorientierten DBS bieten sie (in den meisten Fällen) alle für DBS geforderten Funktionalitäten ([Cod82]). Im Folgenden wird die Verwendung eines (objekt-)relationalen Datenbanksystems detaillierter betrachtet.

## 3.1 Speicherung in einem (objekt-)relationalen DBS

XML-Dokumente können durch ein (objekt-)relationales Datenbanksystem auf vier verschiedenen Wegen gespeichert werden:

- Das komplette XML-Dokument wird in *einem* Attribut der Datenbank gespeichert.
- Der Nutzer bestimmt ein Mapping der Elemente/Attribute auf ein relationales Schema.
- Die XML-Dokumente werden als Graphen interpretiert und in geeigneten Schemata abgelegt.

- Die Struktur der XML-Dokumente wird analysiert und ein passendes Schema generiert.

Die erste Methode, welche einer Speicherung im Dateisystem oder der Verwendung eines Dokument oder Content Management Systemes entspricht, ist am einfachsten umzusetzen, erlaubt jedoch keine sinnvolle Unterstützung durch das DBMS (Anfragen, Indexstrukturen oder Zugriffskontrolle) und wird deshalb nicht weiter betrachtet.

Die anderen drei Methoden erlauben die Verwendung der durch das Datenbankmanagementsystem angebotenen Funktionalitäten.

Da die zweite Variante nur eine geringe oder gar keine Automatisierung ermöglicht, wird sie nicht weiter untersucht.

Die beiden letzten Methoden unterscheiden sich in dem verwendeten „Arbeitsmodell“, XML versus SQL.

Die durch die dritte Methode (Interpretation der Dokumente als Graph) generierten Schemata sind normalerweise sehr klein (teilweise nur eine Relation). Das relationale Schema enthält dadurch einerseits keinerlei Informationen über die Struktur der XML-Daten, andererseits können beliebige Dokumente gespeichert werden. Aus diesem Grund wird von einem Zugriff aus der XML-Welt (durch eine Anwendung oder die Verwendung einer XML-Anfragesprache mit entsprechender Transformation in SQL) auf die gespeicherten Daten ausgegangen. Das Datenbanksystem wird nur zur Speicherung und zur Ausführung der (in SQL übersetzten) Anfragen genutzt. Einige der durch das Datenbankmanagementsystem angebotenen Funktionalitäten, wie beispielsweise Zugriffskontrolle oder Mehrbenutzerbetrieb (abhängig von den Sperrmöglichkeiten des DBMS), sind aufgrund der minimalen Schemata nur sehr schwer oder gar nicht anzuwenden. Ein Vertreter dieses Ansatzes wird im Abschnitt 3.2 vorgestellt.

Die vierte Methode (Analyse der Struktur der XML-Dokumente) generiert Schemata, die (teilweise) der Struktur der XML-Daten entsprechen. Daraus resultiert, dass der Abbildungsprozess für jede neue Dokumentklasse durchgeführt werden muss. Anwendung findet dieser Ansatz besonders dann, wenn XML nur als Austauschformat benutzt wird, auf die Daten also primär mittels SQL zugegriffen werden soll. Durch die reichhaltigen (teilweise aber auch sehr umfangreichen) Schemata ist eine volle Unterstützung durch das DBMS möglich. Im Abschnitt 3.3 wird ein Vertreter dieser Methode vorgestellt, auch der später präsentierte eigene Ansatz ist hier einzuordnen.

Da die Konzepte von XML und relationalen Datenbanken für Strukturierung, Typisierung, Reihenfolge, Identifikation oder Referenzen teilweise stark differieren[4], ist das Finden einer geeigneten Abbildung nicht trivial.

Im Folgenden werden verschiedene Ansätze zum Speichern von XML-Dokumenten in (objekt-)relationalen Datenbanken vorgestellt und bewertet.

[4]Eine detaillierte Untersuchung der Gemeinsamkeiten und Unterschiede findet sich in [KKRSR00].

## 3.2 Speicherung in einem RDBS nach Florescu/-Kossmann

[FK99] beschreiben in ihrem Artikel die, ihren eigenen Worten nach, einfachsten und offensichtlichsten Ansätze, um XML-Daten in einer relationalen Datenbank zu speichern. Um die Datenbank-Schemata zu erzeugen, werden die XML-Daten nicht analysiert und möglicherweise vorhandene DTD- oder XML Schema-Informationen ignoriert. Weiterhin ist keine Interaktion mit dem Nutzer notwendig.

Die Eingabe für den Abbildungsprozess bildet eine Menge von XML-Dokumenten. Die Dokumente werden einzeln geparst und in Tabellen einer relationalen Datenbank gespeichert.

Ein XML-Dokument wird vereinfacht als geordneter gerichteter Graph interpretiert. Jedes Element wird durch einen Knoten im Baum dargestellt, welcher mit der *oid* des Elementes[5] bezeichnet ist. Die Hierarchiebeziehungen zwischen den Elementen werden durch die Kanten des Baumes, die mit dem Namen des Subelementes bezeichnet sind, repräsentiert. Die Ordnung der Subelemente wird mit in den Baum übernommen. Werte (Text oder Attributwerte) bilden die Blätter des Baumes.

Durch die vereinfachende Baumdarstellung gehen Informationen verloren. So werden Kommentare, Processing Instructions oder Entities ignoriert und es wird nicht zwischen Subelementen und Attributen unterschieden. Dadurch ist es nicht möglich, das Originaldokument aus der Datenbank wiederherzustellen.

Florescu und Kaufmann geben sechs verschiedene Möglichkeiten zum Speichern des Dokumentbaumes in einer relationalen Datenbank an, drei verschiedene Abbildungen der Kanten und zwei zur Speicherung der Werte, die sich beliebig miteinander kombinieren lassen.

Das Beispiel 3.1, welches Informationen über vier Personen enthält, wird im Folgenden genutzt, um die verschiedenen Abbildungen zu illustrieren.

**Beispiel 3.1.** - XML-Fragment

```
1  <person id="1" age="55">
2    <name>Peter</name>
3    <address>4711 Fruitdale Ave.</address>
4    <child>
5      <person id="3" age="22">
6        <name>John</name>
7        <address>5361 Columbia Ave.</address>
8        <hobby>swimming</hobby>
9        <hobby>cycling</hobby>
10     </person>
11   </child>
12   <child>
```

[5] Wenn ein Element keinen eindeutigen Identifier besitzt, so wird vom System eine *oid* generiert.

```
13      <person id="4" age="7">
14        <name>David</name>
15        <address>4711 Fruitdale Ave.</address>
16      </person>
17    </child>
18  </person>
19  <person id="2" age="38" child="4">
20    <name>Mary</name>
21    <address>4711 Fruitdale Ave.</address>
22    <hobby>painting</hobby>
23  </person>
```

### 3.2.1 Abbildung der Kanten

Zum Speichern der Kanten stellen [FK99] drei verschiedene Möglichkeiten vor: eine *Kanten*-Tabelle, das Aufspalten der Kanten-Tabelle in verschiedene *Binary*-Tabellen und eine *Universal*-Tabelle.

#### Der Kanten-Ansatz

Die einfachste Abbildung der Kanten stellt eine einzelne Tabelle dar, die alle Kanten enthält. Die Kanten-Tabelle besitzt die folgenden Attribute: die oids der beiden verbundenen Knoten (`source` und `target`), die Bezeichnung der Kante (`name`), eine Kennzeichnung, ob die Kante zu einem internen Knoten oder zu einem Wert führt (`flag`), und eine Ordnungsnummer (`ordinal`), um die Ordnung der Subelemente zu speichern. Die Kanten-Tabelle besitzt die folgende Struktur:

```
1 Edge (source, ordinal, name, flag, target)
```

Der Primärschlüssel wird über die Spalten `source` und `ordinal` gebildet. Abbildung 3.1 zeigt eine Kanten-Tabelle, die mit den Werten aus Beispiel 3.1 gefüllt ist. Zusätzlich wird auch eine Möglichkeit zum Speichern der Werte aufgezeigt, diese wird später erläutert. Die fett gedruckten Zahlen in der `target`-Spalte sind oids der Zielknoten, die kursiven Einträge referenzieren Werte.

#### Der Binary-Ansatz

Die zweite Möglichkeit, die Kanten in einer relationalen Datenbank abzulegen, besteht darin, alle Kanten mit der gleichen Bezeichnung in einer Tabelle zu speichern. Dieses Vorgehen entspricht einer horizontalen Partitionierung der Kanten-Tabelle über das `name`-Attribut. Es werden so viele Binary-Tabellen erstellt, wie verschiedene Subelemente und Attribute in dem XML-Dokument existieren. Die Binary-Tabelle besitzt die folgende Struktur:

```
1 Binary_name (source, ordinal, flag, target)
```

Edge

| *source* | *ordinal* | *name* | *flag* | *target* |
|---|---|---|---|---|
| 1 | 1 | age | int | *v1* |
| 1 | 2 | name | string | *v2* |
| 1 | 3 | address | string | *v3* |
| 1 | 4 | child | ref | **3** |
| 1 | 5 | child | ref | **4** |
| 2 | 1 | age | int | *v4* |
| ... | ... | ... | ... | ... |

$V_{int}$

| *vid* | *value* |
|---|---|
| v1 | 55 |
| v4 | 38 |
| v8 | 22 |
| v13 | 7 |

$V_{string}$

| *vid* | *value* |
|---|---|
| v2 | Peter |
| v3 | 4711 Fruitdale Ave. |
| v5 | Mary |
| v6 | 4711 Fruitdale Ave. |
| v6 | painting |
| ... | ... |
| v15 | 4711 Fruitdale Ave. |

Abbildung 3.1: Kanten-Tabelle mit separaten Werte-Tabellen

Der Primärschlüssel und die Bedeutung der Attribute entspricht der Kanten-Tabelle.

#### Eine Universal-Tabelle

Der dritte Ansatz generiert eine einzige Universal-Tabelle, welche alle Kanten enthält. Konzeptionell entspricht dieses Vorgehen einem full outer join aller Binary-Tabellen. Die Struktur der Universal-Tabelle ist wie folgt, wobei $n_1$, ..., $n_k$ die Bezeichnungen sind.

```
1 Universal (source, ordinal_n1, flag_n1, target_n1, ..., ordinal_nk, flag_nk, target_nk)
```

Abbildung 3.2 zeigt die Universal-Tabelle für das Beispiel 3.1. Man kann deutlich erkennen, dass die Universal-Tabelle viele null-Werte generiert und dass Redundanzen auftreten. Die Universal-Tabelle ist denormalisiert, mit all den damit verbundenen Vor- und Nachteilen.

### 3.2.2 Abbildung der Werte

[FK99] stellen zwei verschiedene Wege zum Speichern von Werten vor: *separate Tabellen* und das *Inlinen* der Werte.

| *source* | | $ordinal_{name}$ | $target_{name}$ | | $ordinal_{child}$ | $target_{child}$ | $ordinal_{hobby}$ | $target_{hobby}$ |
|---|---|---|---|---|---|---|---|---|
| 1 | | 2 | Peter | | 4 | **3** | null | null |
| 1 | | 2 | Peter | | 5 | **4** | null | null |
| 2 | | 2 | Mary | | 4 | **4** | 5 | *painting* |
| 3 | | 2 | John | | null | null | 4 | *swimming* |
| 3 | | 2 | John | | null | null | 5 | *cycling* |
| 4 | | 2 | David | | null | null | null | null |

Abbildung 3.2: Universal-Tabelle

### Separate Werte-Tabellen

Die erste Möglichkeit zum Speichern der Werte ist das Anlegen von separaten Werte-Tabellen für jeden Datentyp[6]. Die Struktur der Werte-Tabellen ist wie folgt, wobei der Datentyp der `value`-Spalte dem *Typ* der Tabelle entspricht:

```
1 V_Typ (vid, value)
```

Eine Anwendung der separaten Werte-Tabellen ist in Abbildung 3.1 zu sehen. Für das Beispiel 3.1 wurden zwei Werte-Tabellen generiert, eine für Integerwerte ($V_{int}$) und eine für Stringwerte ($V_{string}$). Die `vid`-Werte für die Werte-Tabellen werden durch den Umformungsprozess automatisch erzeugt.

In Abbildung 3.1 werden die separaten Werte-Tabellen mit dem Kanten-Ansatz kombiniert. Die `flag`-Spalte in der Kanten-Tabelle wird benötigt, damit der Kante die richtige Werte-Tabelle zugeordnet wird. Als Werte für die `flag`-Spalte dienen die *Typ*en der Werte-Tabellen oder „*ref*“, um eine Kante zwischen zwei internen Knoten zu kennzeichnen. In gleicher Weise können die separaten Werte-Tabellen auch mit dem Binary-Ansatz und der Universal-Tabelle kombiniert werden.

### Inlining

Die Alternative zu den separaten Werte-Tabellen stellt das Inlining dar. Dazu werden die Werte und Attribute mit in die Tabelle(n) für die Kanten übernommen.

Beim Kanten-Ansatz entspricht dies einem Outer Join der Kanten- und der Werte-Tabelle(n), analoges gilt für die anderen Ansätze. Für jeden Datentyp wird eine Spalte in die Tabelle übernommen, woraus eine große Anzahl von null-Werten resultieren.

In Abbildung 3.3 wurde der Binary-Ansatz mit dem Inlining kombiniert. Die `flag`-Spalte wird nicht mehr benötigt, es entstehen sehr viele null-Werte[7].

[6] *Wie* die unterschiedlichen Datentypen erkannt werden, wird in [FK99] nicht betrachtet. Eine Möglichkeit wäre die Analyse aller XML-Daten, eine andere die Verarbeitung von XML Schema-Informationen (soweit vorhanden).

[7] Es ist nicht nachvollziehbar, warum an dieser Stelle überflüssige Spalten generiert werden. Beim Binary-Ansatz wird für jeden möglichen Kantentyp eine Relation erstellt. Da eine Kante nur zwei

$B_{hobby}$

| *source* | *ordinal* | $value_{int}$ | $value_{string}$ | *target* |
|---|---|---|---|---|
| 2 | 5 | null | *painting* | null |
| 3 | 4 | null | *swimming* | null |
| 3 | 5 | null | *cycling* | null |

$B_{child}$

| *source* | *ordinal* | $value_{int}$ | $value_{string}$ | *target* |
|---|---|---|---|---|
| 1 | 4 | null | null | **3** |
| 1 | 5 | null | null | **4** |
| 2 | 4 | null | null | **4** |

Abbildung 3.3: Binary-Tabellen mit Inlining

### 3.2.3 Bewertung

Der Vorteil dieses Ansatzes ist, dass jedes beliebige XML-Dokument automatisch in einer relationalen Datenbank gespeichert werden kann. Zur Generierung das Schemas werden keine weiteren Informationen über die Struktur (zum Beispiel DTDs) oder Eingaben des Nutzers benötigt. Weiterhin kann durch die Wahl der Abbildung die Größe des generierten Schemas beeinflusst werden (eine Tabelle für jede Beziehung bei Binary versus eine einzige Tabelle bei Universal mit Inlining).

Auf der anderen Seite enthält das generierte Schema keinerlei Informationen über die Struktur der Daten. Dadurch und durch die fehlende Unterscheidung zwischen Elementen und Attributen ist nicht möglich, die XML-Dokumente wiederherzustellen. Vom Kanten-Ansatz mit Wertetabellen abgesehen, entstehen immer null-Werte oder Redundanzen. Zusätzlich enthält das Schema Informationen, die nicht den Daten aus den XML-Dokumenten zugeordnet werden können (zum Beispiel die Spalte „`target`“). Solche Metadaten sollten besser getrennt von den Daten gespeichert werden. Weiterhin sind die vorgestellten Abbildungen auf das Relationenmodell beschränkt.

## 3.3 Speicherung in einer relationalen Datenbank nach [STZ+99]

[STZ+99] schlagen einen anderen Weg zum Speichern von XML-Daten in relationalen Datenbanken vor. Sie zeigen, dass relationale Datenbanken genutzt werden können, um Anfragen auf XML-Daten auszuführen. Dazu wird basierend auf einer DTD ein relationales Schema generiert, XML-Daten, die der DTD entsprechen, in die Datenbank importiert, Anfragen auf die XML-Daten in SQL-Anfragen transformiert, auf der Datenbank ausgeführt und die Ergebnisse zurück nach XML konvertiert.

Knoten verbindet, ist eindeutig bestimmt, welchen Datentyp der Wert des Zielknotens (oder „*ref*“ für interne Knoten) hat. Aus diesem Grund ist nur eine Werte-Spalte notwendig, bzw. keine für Kanten zwischen interne Knoten.

Im Gegensatz zu Florescu/Kaufmann (siehe Abschnitt 3.2) wird zur Generierung des relationalen Schemas zwingend eine DTD[8] benötigt.

[STZ+99] identifizieren die folgenden Probleme bei der Generierung des relationalen Schemas:

- die Komplexität der Elementdefinition in der DTD
- der Konflikt zwischen der Zwei-Ebenen-Natur von relationalen Schemata (Tabelle/Entity - Attribut) und der Schachtelung von Elementen in XML
- die Behandlung von mehrwertigen Attributen
- Rekursion von Elementen

Zur Behandlung der letzten beiden Probleme wird die Standardtechnik des relationalen Datenbankentwurfes zur Umsetzung angewandt: Auslagerung in eine separate Relation und Fremdschlüsselbeziehungen.

Die DTD in Beispiel 3.2 wird im Folgenden zur Illustration genutzt[9].

**Beispiel 3.2.** - DTD

```
1  <!ELEMENT book (booktitle, author)>
2  <!ELEMENT booktitle (#PCDATA)>
3  <!ELEMENT article (title, author*, contactauthor?)>
4  <!ELEMENT title (#PCDATA)>
5  <!ELEMENT contactauthor EMPTY>
6  <!ATTLIST contactauthor authorID IDREF #IMPLIED>
7  <!ELEMENT monograph (title, author, editor)>
8  <!ELEMENT editor (monograph*)>
9  <!ATTLIST editor name CDATA #REQUIRED>
10 <!ELEMENT author (name, address)>
11 <!ATTLIST author authorID ID #REQUIRED>
12 <!ELEMENT name (#PCDATA)>
13 <!ELEMENT address ANY>
```

### 3.3.1 Vereinfachung der DTD

Die aus DTDs generierten relationalen Schemata können sehr umfangreich werden, um der Komplexität der DTDs zu entsprechen. Glücklicherweise ist es möglich, die DTDs so zu vereinfachen, dass die entstehenden relationalen Schemata deutlich verkleinert

[8] XML-Schema (siehe Abschnitt 2.4) oder Document Content Descriptors (DCDs, siehe [BFM]) würden ein detaillierteres relationales Schema erlauben, da Angaben bezüglich Datentypen, (kombinierte) Schlüssel oder Größen von Listen oder Sets vorhanden wären.

[9] Das Beispiel stammt aus [STZ+99] und wurde um einige für die weitere Betrachtung unnötige Elemente gekürzt.

werden. Durch die Vereinfachung gehen natürlich Informationen verloren, aber da es nicht notwendig ist, die DTD wieder aus dem relationalen Schema herzustellen, sondern nur alle XML-Dokumente, die zu den DTDs konform sind, in der Datenbank speichern zu können, ist diese Vereinfachung praktikabel.

Ein Großteil der Komplexität von DTDs steckt in der Spezifikation von Elementen (siehe Abschnitt 2.2.1). Elemente können in XML nahezu beliebig geschachtelt werden und DTDs müssen alle Möglichkeiten zur Definition dieser Schachtelungen und Wertigkeiten unterstützen. Als abschreckendes Beispiel kann die folgende Elementdefinition für ein Element „`a`“ gesehen werden (wobei „`b`“, „`c`“, „`e`“ und „`f`“ andere Elemente sind):

```
1 <!ELEMENT a ((b | c | e)?, (e? | (f?, (b, b)*))*)>
```

Da das Hauptanliegen von [STZ+99] jedoch in der Ausführung von Anfragen auf den Daten besteht, ist nur die relative Position sowie die Vater-Kind-Beziehung eines Elementes relevant. Aus diesem Grund gibt es verschiedene Transformationen, die eine DTD vereinfachen, ohne dass für die Ausführung einer Anfrage relevante Informationen verloren gehen. Die Transformationen sind eine Obermenge ähnlicher Transformationen, wie sie in [DFS99] vorgestellt werden.

```
1 (e1, e2)* → e1*, e2*
2 (e1, e2)? → e1?, e2?
3 (e1 | e2) → e1?, e2?
```

Abbildung 3.4: Transformationen zur Auflösung geschachtelter Definitionen

```
1 e1** → e1*
2 e1*? → e1*
3 e1?* → e1*
4 e1?? → e1?
```

Abbildung 3.5: Transformationen zur Eliminierung „überflüssiger“ Operatoren

```
1 ..., e1*, ..., e1*, ... → ..., e1*, ...
2 ..., e1*, ..., e1?, ... → ..., e1*, ...
3 ..., e1?, ..., e1*, ... → ..., e1*, ...
4 ..., e1?, ..., e1?, ... → ..., e1*, ...
5 ..., e1, ..., e1, ... → ..., e1*, ...
```

Abbildung 3.6: Transformationen zur Zusammenfassung gleicher Elemente

Die erste Art von Transformationen (siehe Abbildung 3.4) löst geschachtelte Definitionen auf. So sollen „`,`“ und „`|`“ nicht innerhalb anderer Operatoren auftreten.

Die zweite Art von Transformationen (siehe Abbildung 3.5) reduziert die Anzahl der unären Operatoren.

Die letzte Art von Transformationen (siehe Abbildung 3.6) fasst mehrmals auftretende gleiche Elemente zusammen. Zusätzlich werden alle „+“-Wertigkeiten (mindestens einmal, beliebig oft) durch „*“ (optional, beliebig oft) ersetzt. Das eingangs aufgeführte Beispiel kann durch diese Transformationsregeln auf die folgende Form vereinfacht werden:

```
1 <!ELEMENT a (b* , c? , e*, f*)>
```

Die Vereinfachung der DTDs erfolgt so, dass die Semantiken „einer oder viele“ und „null oder nicht null“ erhalten bleiben. Jedoch gehen bei der Transformation die Informationen über die relative Ordnung der Elemente verloren. [STZ+99] merken an, dass diese Informationen beim Import eines spezifischen XML-Dokumentes in die Datenbank festgehalten werden können.

### 3.3.2 Analyse der DTDs

Nachdem die DTDs durch die oben beschriebenen Transformationen vereinfacht wurden, werden sie analysiert, um die relationalen Schemata zu generieren. Zur Analyse der DTDs werden DTD-Graphen und Element-Graphen verwendet.

#### DTD-Graphen

Ein DTD-Graph repräsentiert die Struktur einer DTD. Die Knoten bilden die Elemente, Attribute und Operatoren der DTD ab, die Kanten die Beziehungen zwischen ihnen. Elemente erscheinen nur einmal im DTD-Graphen, die Attribute und Operatoren so oft, wie sie auch in der DTD auftreten.

Abbildung 3.7 zeigt den DTD-Graphen für die DTD aus Beispiel 3.2, die Attribute sind grau dargestellt.

#### Element-Graphen

Element-Graphen werden zur Bestimmung der zu generierenden Relationen für die Elemente aus dem DTD-Graphen erstellt.

Auf das Element im DTD-Graphen, für das der Element-Graph erzeugt werden soll, wird ein *deep first*-Algorithmus angewandt. Jeder Knoten wird markiert, wenn er beim Durchlauf das erste Mal passiert wird. Sind alle Nachfolgeknoten durchlaufen worden, so wird die Markierung wieder entfernt.

Wird beim Durchlauf im DTD-Graphen ein unmarkierter Knoten erreicht, so wird ein gleichnamiger Knoten im Element-Graphen erzeugt (und der Knoten markiert). Der neue Knoten wird durch eine reguläre Kante mit dem, dem DTD-Graphen entsprechenden, Vaterknoten verbunden.

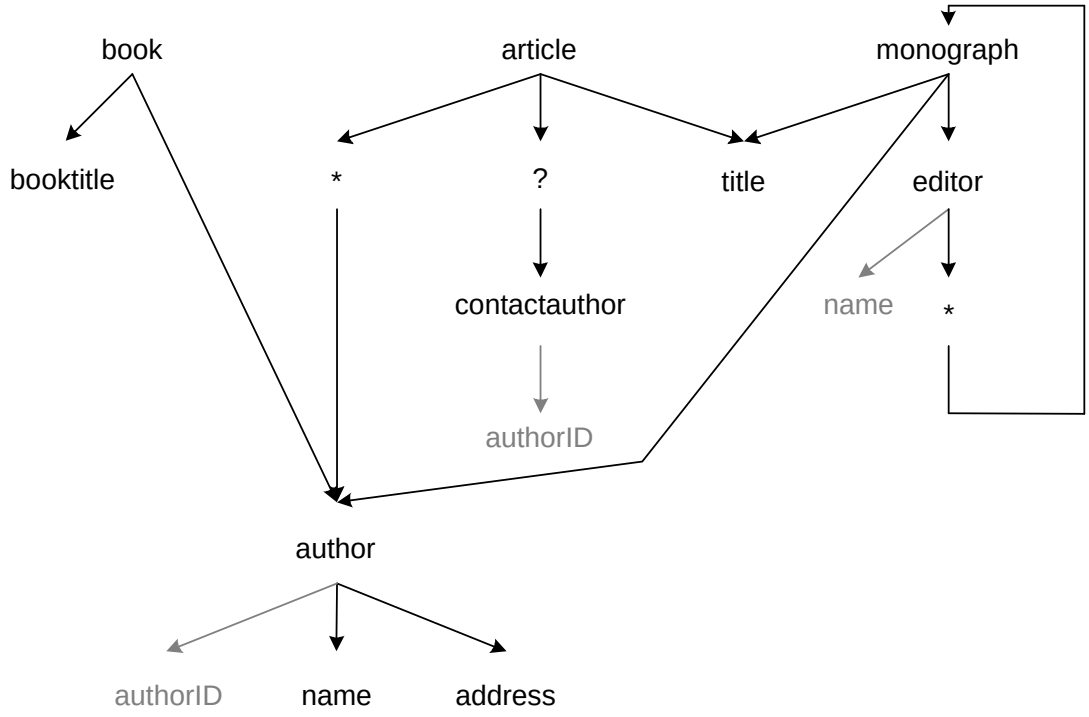


Abbildung 3.7: DTD-Graph zum Beispiel 3.2

Wird ein markierter Knoten erreicht, so wird keine reguläre Kante, sondern ein *Backpointer* erzeugt. Dieser verbindet den zuletzt erzeugten Knoten im Elemente-Graphen mit dem, dem markierten Knoten des DTD-Graphen entsprechenden, Knoten im Element-Graphen.

In Abbildung 3.8 ist der Element-Graph für das `editor`-Element zum DTD-Graphen aus Abbildung 3.7 zu sehen.

Nachdem die DTD analysiert und die Element-Graphen erstellt worden sind, kann das relationale Schema generiert werden. [STZ+99] stellen drei (aufeinander aufbauende) Techniken zum Erstellen des relationalen Schemas vor: *Basic Inlining*, *Shared Inlining* und *Hybrid Inlining*.

### 3.3.3 Basic Inlining

Beim Basic Inlining wird versucht, so viele Nachfolger eines Elementes wie möglich in die Relation mit aufzunehmen. Trotzdem muss für jedes Element eine Relation erstellt werden, da ein XML-Dokument jedes Element aus der DTD als *root*-Element haben kann.

Ausgehend von den Element-Graphen werden die Relationen wie folgt erzeugt. Für den *root*-Knoten des Graphen wird eine Relation gleichen Namens erzeugt. Alle Subknoten werden (mit zwei Ausnahmen) als Attribute in die Relation aufgenommen. Separate

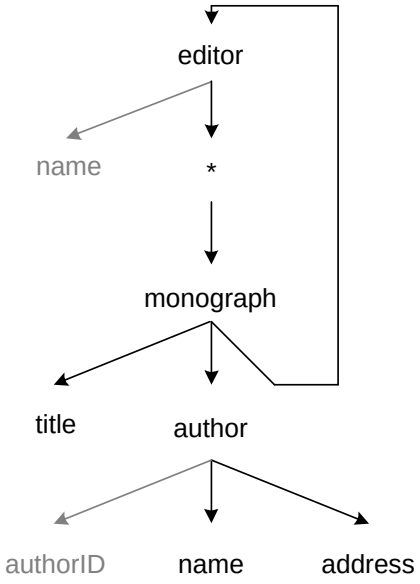


Abbildung 3.8: Element-Graph für das `editor`-Element zum DTD-Graphen aus Abbildung 3.7

Relationen werden für alle Knoten direkt nach einem „`*`“-Knoten (um Sets abzubilden) und für Knoten, in die ein *Backpointer* mündet (um Rekursion zu handeln), erzeugt.

Für jede Relation wird eine ID als Schlüssel generiert. Die Namen für die Attribute entsprechen dem Pfad vom *root*-Knoten aus mit einem Punkt („`.`“) als Trennzeichen. Alle Relationen, die für Subknoten erzeugt wurden, erhalten ein zusätzliches Attribut namens „`parentID`“ welches über einen Fremdschlüssel die Beziehung zum Vaterknoten herstellt. Alle Attribute (außer der ID und „`parentID`“[10]) sind vom Datentyp String.

Abbildung 3.9 zeigt das generierte relationale Schema für die DTD aus Beispiel 3.2.

### 3.3.4 Shared Inlining

Shared Inlining versucht die vielen Redundanzen des Basic Inlining zu vermeiden. Es werden die Elemente, die in mehreren Relationen auftreten, in neue Relationen ausgelagert und über Fremdschlüssel von mehreren Relationen genutzt.

Relationen werden beim Shared Inlining für alle Knoten des DTD-Graphen erzeugt, die einen Eingangsgrad von größer als eins haben (dies sind die Knoten, die beim Basic Inlining in mehreren Relationen auftreten). Knoten mit einem Eingangsgrad von eins werden als Attribute in die entsprechende Relation des Vaterknotens aufgenommen. Zusätzlich werden für alle Knoten mit einem Eingangsgrad von null Relationen erstellt. Knoten direkt nach einem „`*`“-Knoten werden wie beim Basic Inlining behandelt. Bei Rekursionen (Zyklen im Graphen), wo alle Knoten einen Eingangsgrad von eins haben

[10] Für die IDs kann ein entsprechend geeigneter Datentyp gewählt werden, [STZ+99] nutzen Integer.

```
book              (ID, book.booktitle, book.author.name, book.author.address,
                  book.author.authorID)
booktitle         (ID, booktitle)
article           (ID, article.contactauthor.authorID, article.title)
article.autor     (ID, article.author.parentID, article.author.name,
                  article.author.address, article.author.authorID)
contactauthor     (ID, contactauthor.authorID)
title             (ID, title)
monograph         (ID, monograph.parentID, monograph.title,
                  monograph.editor.name, monograph.author.name,
                  monograph.autor.address, monograph.author.authorID)
editor            (ID, editor.parentID, editor.name)
editor.monograph  (ID, editor.monograph.parentID, editor.monograph.title,
                  editor.monograph.author.name, editor.monograph.autor.address,
                  editor.monograph.author.authorID)
autor             (ID, author.name, author.address, author.authorID)
name              (ID, name)
address           (ID, address)
```

Abbildung 3.9: Generiertes Relationenschema für die DTD aus Beispiel 3.2 bei Anwendung der Basic Inlining Technik

(im Beispiel 3.2 „`monograph`“ und „`editor`“) wird genau ein (beliebiger) Knoten (sollten nicht schon andere Bedingungen zutreffen) zu einer separaten Relation.

Abbildung 3.10 zeigt das generierte relationale Schema für die DTD aus Beispiel 3.2.

```
book       (ID, book.booktitle.isRoot, book.booktitle)
article    (ID, article.contactauthor.isRoot,
           article.contactauthor.authorID)
monograph  (ID, monograph.parentID, monograph.parentCODE,
           monograph.editor.isRoot, monograph.editor.name)
title      (ID, title.parentID, title.parentCODE, title)
autor      (ID, author.parentID, author.parentCODE,
           author.name.isRoot, author.name, author.address.isRoot,
           author.address, author.authorID)
```

Abbildung 3.10: Generiertes Relationenschema für die DTD aus Beispiel 3.2 bei Anwendung der Shared Inlining Technik

Da nicht mehr für jedes Element eine Relation erstellt wird, ein XML-Dokument aber jedes Element als *root* haben kann, sind zusätzliche Metadaten notwendig. Diese werden bei [STZ+99] mit in die generierten Relationen aufgenommen. Für jeden Knoten, der in eine Relation als Attribut aufgenommen wird, wird ein zusätzliches Attribut namens „`isRoot`“ vom Typ Boolean generiert.

Weiterhin sind die Fremdschlüsselbeziehungen des Attributes „`parentID`“ nicht mehr

eindeutig. So kann das Element „author“ die drei Elemente „book“, „article“ und „monograph“ als Vater haben. Um zu kennzeichnen, zu welcher Relation das Attribut „parentID“ verweist, wird das Attribut „parentCODE“ genutzt. Natürlich können die nicht typisierten Fremdschlüsselbeziehungen in diesem Fall nicht mehr vom DBMS unterstützt werden.

### 3.3.5 Hybrid Inlining

[STZ+99] verknüpfen beim Hybrid Inlining die Vorteile (in Bezug auf eine gute Unterstützung von Anfragen auf XML-Daten[11], die durch ein RDBMS ausgeführt werden) des Basic und des Shared Inlining. Es entspricht dem Shared Inlining, nur dass auch die Knoten, die einen Eingangsgrad größer als eins haben, aber nicht an einer Rekursion beteiligt sind oder auf einen „*“-Knoten folgen, als Attribute in die Relation des entsprechenden Vaterknotens aufgenommen werden.

Das generierte relationale Schema für die DTD aus Beispiel 3.2 ist in Abbildung 3.11 zu sehen.

```
book       (ID, book.booktitle.isRoot, book.booktitle,
           book.autor.name, book.author.adress, book.author.authorID)
article    (ID, article.contactauthor.isRoot,
           article.contactauthor.authorID, article.title.isRoot,
           article.title)
monograph  (ID, monograph.parentID, monograph.parentCODE,
           monograph.title, monograph.editor.isRoot,
           monograph.editor.name, monograph.autor.name,
           monograph.author.adress, monograph.author.authorID)
autor      (ID, author.parentID, author.parentCODE,
           author.name.isRoot, author.name, author.address.isRoot,
           author.address, author.authorID)
```

Abbildung 3.11: Generiertes Relationenschema für die DTD aus Beispiel 3.2 bei Anwendung der Hybrid Inlining Technik

### 3.3.6 Bewertung

Durch die Einbeziehung der DTD spiegelt sich die Struktur der XML-Dokumente in dem erstellten Schema wider. Die vorgestellten Transformationen zur Vereinfachung der DTD können den Aufwand bei der Schemageneration drastisch reduzieren. Die Verwendung von Graphen zur Analyse der DTD ermöglicht die Anwendung von bewährten Algorithmen aus der Graphentheorie beispielsweise zur Erkennung von Zyklen.

[11] Die benutzten Metriken sind die durchschnittliche Anzahl von SQL-Anfragen und die darin enthaltenen Verbundoperationen, die benötigt werden, um einen Pfadausdruck einer gewissen Länge evaluieren zu können.

Ein Nachteil der vorgestellten Abbildungen beim Basic und Hybrid Inlining sind die Redundanzen auf Strukturebene (zum Beispiel „`title`“). Weiterhin werden beim Shared und Hybrid Inlining Metadaten, wie beispielsweise die Spalten „`*.isRoot`“, mit in die Relationen aufgenommen. Durch die Vereinfachung der DTD vor der Schemageneration können die XML-Dokumente nicht wieder in ihren Originalzustand zurücktransformiert werden. Auch dieser Ansatz ist auf das Relationenmodell beschränkt.

## 3.4 Weitere Ansätze

In [KM99] stellen die Autoren einen Algorithmus zur Generierung eines objekt-relationalen Schemas aus einer DTD vor. Dieses Schema kann durch Dekomposition auch auf relationale Datenbanken angewandt werden. Weiterhin betrachten sie die Verwendung eines *hybriden Datenbanksystems*[12], wodurch nur ein Teil der XML-Struktur (formale oder strukturierte Daten) auf Relationen/Attribute in der Datenbank abgebildet wird. Die informalen XML-Teile (z. B. Mixed Content, Text, ...) werden als XML-Attribute gespeichert. Die Entscheidung, welche Elemente und Attribute in der Datenbank modelliert werden, basiert auf einer Analyse der DTD, der bereits existierenden XML-Daten und der zu erwartenden Anfragen auf die Daten.

[KKRSR00] stellen eine (manuelle) regelbasierte Integration von XML-Daten in relationale DBSe vor. Dabei wird eine eigene Metastruktur in der Datenbank gespeichert, welche die Beziehungen zwischen der DTD (oder der analysierten Struktur eine Menge von XML-Dokumenten) und der Darstellung in der Datenbank verwaltet. Dadurch ist es möglich, auf der XML-Seite mehrere verschiedene DTDs mit den gleichen Relationen aus der Datenbank zu verknüpfen. Dies erlaubt es auch, voneinander unabhängig entwickelte Schemata auf beiden Seiten miteinander zu verknüpfen und einen Datenaustausch zu ermöglichen.

In [SYU99] und [YASU01] werden die XML-Dokumente als Baum interpretiert und die einzelnen Knoten in dem Knotentyp (es wird zwischen Element, Attribut oder Text unterschieden) entsprechenden Relationen abgelegt. Die Struktur wird durch die Zuordnung einer Dokument-ID, des Pfades zu jedem Knoten sowie der Angabe der Region, welche durch den Knoten umspannt wird, gespeichert. Das relationale Schema umfasst nur die vier Relationen *Element*, *Attribut*, *Text* und *Path*.

Ein Mapping zwischen semistrukturierten Daten und relationalem Schema basierend auf einer eigenen Anfragesprache (STORED) wird in [DFS99] vorgestellt. Der vorgestellte heuristische Algorithmus generiert ein relationales Schema und das dazugehörige Mapping. Interessant ist, dass neben dem relationalen Schema, welches der Struktur der XML-Daten entspricht, eine Speicherung der XML-Daten als Graph für XML-Fragmente verwendet wird, die nicht in das erstellte Schema passen. Dadurch sind Updates auf dem Datenbestand möglich.

[12] Ein hybrides Datenbanksystem kann ein (objekt-)relationales DBS mit erweiterter Funktionalität in Bezug auf XML sein. Attribute vom Typ XML, die komplette XML-Fragmente enthalten, können vom DBMS durch geeignete Methoden indiziert, durchsucht und geändert werden (z. B. [Por99]).

[LC00] untersuchen die Erweiterung existierender Transformationen von DTDs zu relationalen Schematas um semantische Bedingungen. Durch Analyse der DTDs werden die semantischen Bedingungen identifiziert und die entsprechenden Constraints für die Datenbank („`NOT NULL`“, „`CHECK`“, Primär- oder Fremdschlüssel, ...) generiert. Die Erweiterung kann für jeden Transformationsalgorithmus adaptiert werden.

Die bislang vorgestellten Ansätze beschäftigen sich mit der Speicherung von XML-Dokumenten in Datenbanksystemen. Aber auch der andere Weg, das Generieren von XML aus Daten, die in einer Datenbank gespeichert sind, wird untersucht. In [SSB+00] werden verschiedene Alternativen zur Ausgabe von relationalen Daten als XML untersucht.

In [KL02] wird ein Algorithmus zur automatischen Erzeugung von DTDs aus konzeptionellen Datanbankschemata repräsentiert. Es werden so viele Strukturinformationen wie möglich aus dem konzeptionellen Schema (erweitertes Entity-Relationship-Modell) in die DTD übertragen, um eine partielle Integritätsprüfung durch validierende XML-Parser zu ermöglichen.

## 3.5 Zusammenfassung

In diesem Kapitel wurden verschiedene Ansätze zur Speicherung von XML-Daten in (objekt-)relationalen Datenbanken vorgestellt.

Ein Großteil der existierenden Vorschläge ist auf zwei Probleme fokussiert: 1. eine gute Unterstützung von Anfragen aus der XML-Welt und 2. die Möglichkeit beliebige XML-Dokumente zu speichern ([FK99], [SYU99] oder [DFS99]). Daraus resultiert der Nachteil, dass die generierten Schemata sehr wenige oder gar keine semantischen Informationen enthalten (zum Beispiel eine Universal-Tabelle mit Inlining in [FK99]). Auch sind die Schemata in den meisten Fällen sehr schlecht gefüllt (sie enthalten viele null-Werte). Die Datenbank wird in diesem Fall nur zur reinen Speicherung genutzt, ein Zugriff auf die Daten ist in vielen Fällen nur über eine Anwendung möglich. Dieser Nachteil ist jedoch gleichzeit auch ein Vorteil, da oft in dem kompakten Schema ohne weitere Anpassungen beliebige XML-Dokumente gespeichert werden können. Weiterhin bieten diese Ansätze teilweise eine bessere Unterstützung von Anfragen, da weniger Relationen zur Beantwortung der Anfragen verknüpft werden müssen.

Die zweite Variante zur Speicherung von XML-Daten in einer Datenbank bezieht vorhandene Strukturinformationen aus der DTD in die Schemageneration mit ein ([STZ+99], [KM99] oder [KKRSR00]). Die generierten Schemata führen in vielen Fällen zu einer (teilweise starken) Fragmentierung der Daten. Um die daraus resultierenden Probleme bei der Unterstützung von Anfragen zu vermeiden, werden die generierten Schemata teilweise denormalisiert und enthalten Redundanzen (zumindest auf Strukturebene).

Eine interessante Alternative sind hybride Ansätze zur Speicherung von XML-Dokumenten. Die Teile der Dokumente, die wenig strukturiert sind, zu einer schwachen Füllung des Schemas führen oder selten in Anfragen enthalten sind, werden uninterpre-

tiert als XML-Fragmente gespeichert.

Bis auf wenige Ausnahmen (zum Beispiel [KKRSR00]) vermischen alle Ansätze zur Speicherung von XML-Dokumente die Daten mit den Metainformationen.

# Kapitel 4

# Generisches Integrationsmodell

Das Generische Integrationsmodell (*Generic Integration Model*, GIM, siehe [Sch98]) stammt aus dem Gebiet der Förderierten Datenbanken.

Ein förderiertes Datenbanksystem (FDBS, siehe [SL90, Con97]) verbindet (teil-)autonome und heterogene DBSe in der Form, dass globale Anwendungen Zugriff auf „alle“ Daten haben, die vorhandenen lokalen Anwendungen aber weiter genutzt werden können. Um dies zu erreichen, werden die verschiedenen lokalen, heterogenen Schemata in *ein* widerspruchsfreies homogenes Schema integriert, welches als Basis für anwendungsspezifische Sichten dienen kann. Da die lokalen DBSe ihre Autonomie (und damit die Heterogenität) weiter behalten, müssen die notwendigen Transformationen durch das FDBS durchgeführt werden.

Einzuordnen ist GIM in diesem Kontext als Werkzeug zur Schemaintegration im Entwurf von FDBen. Zu diesem Zweck werden die lokalen Schemata nach GIM transformiert, die vorhandenen Konflikte aufgelöst und ein objektorientiertes integriertes Schema abgeleitet.

## 4.1 Schemarepräsentation

Zur Repräsentation der Schemata wird ein semantisch armes Datenmodell verwendet, bestehend aus Klassen mit *Extensionen* und *Intensionen*.

Eine Extension repräsentiert die Menge aller *potenzieller* Objekte einer Klasse. Die Informationen über die extensionalen Überlappungen sind zwingend notwendig für die Integration der Vererbungshierarchien. Überlappende Extensionen werden in voneinander disjunkte Basisextensionen zerlegt.

Die Intension repräsentiert die Menge aller Attribute einer Klasse. Um die Intensionen abzuleiten, wird analysiert, welche Attribute zu welchen Basisextensionen gehören. Dabei ist es möglich, die Attribute zu clustern (mehrere Attribute, die immer gemeinsam auftreten, werden zu einer Gruppe zusammengefasst), um die Übersichtlichkeit zu erhöhen. Bei der Analyse der Attribute wird davon ausgegangen, dass Attribute gleichen

Namens die gleiche Semantik haben, mögliche Attributkonflikte (z.B. Homonyme oder Synonyme) müssen also schon entfernt sein.

Aus den gewonnenen extensionalen und intensionalen Informationen kann eine Matrix erstellt werden (im Folgenden als *GIM-Matrix* oder *normalisiertes Schema* bezeichnet), welche die Attribute den Basisextensionen zuordnet (siehe Abbildung 4.1). Ein Kreuz in einer Zelle bedeutet, dass für die möglichen Objekte der Basisextension (Spalte) der Attributwert (Zeile) bekannt ist.

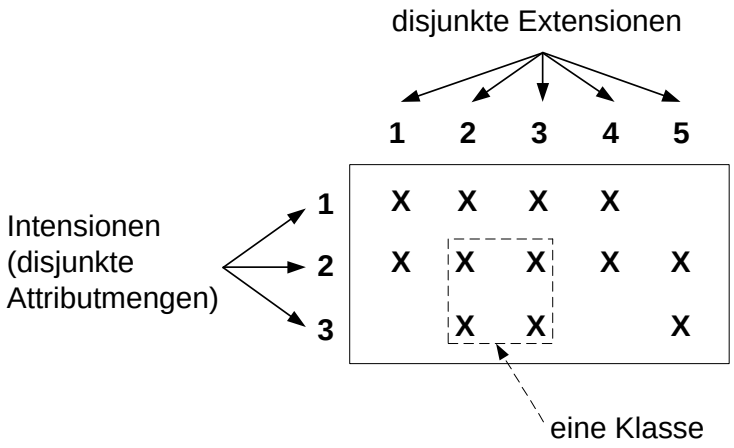


Abbildung 4.1: GIM-Matrix

Die GIM-Matrix wird für die weitere Analyse genutzt. Dabei ist die Ordnung innerhalb der Spalten bzw. Zeilen irrelevant, sie kann beliebig geändert werden. Durch einen Austausch wird nur die (graphische) Repräsentation verändert.

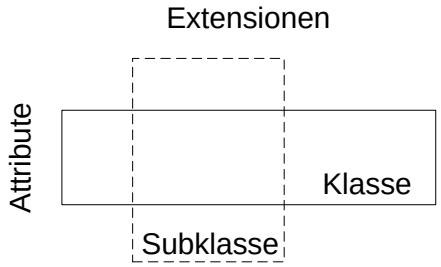


Abbildung 4.2: Subklassenbeziehungen durch überlappende Rechtecke

Aus der GIM-Matrix können zwei Informationen abgelesen werden. Zum einen repräsentiert ein durch Kreuze ausgefülltes Rechteck (in einer speziellen Anordnung der Spalten und Zeilen der GIM-Matrix) eine Klasse, also eine Menge von Basisextensionen, die die gleichen Attribute haben. Die zweite Information sind Subklassenbeziehungen. Überlappen sich die Rechtecke wie in Abbildung 4.2 dargestellt, befinden sie sich in einer Spezialisierungsbeziehung.

Das Problem ist, dass die durch Rechtecke gefundenen Klassen und Subklassenbeziehungen sehr stark von der gewählten Ordnung der Spalten und Zeilen abhängen. Außerdem müssen die maximal größtmöglichen Rechtecke gefunden werden, um das abgeleitete Schema zu minimieren. Aus diesen Gründen können aus dem normalisierten Schema durch verschiedene Algorithmen verschiedene integrierte Schemata abgeleitet werden.

## 4.2 Naiver Ansatz

Der naive Ansatz zur Generierung des optimalen integrierten Schemas ist der folgende:

1. Berechnung aller möglichen normalisierten Schemata durch Umsortierung und Ableitung der integrierten Schemata
2. Bewertung aller integrierten Schemata und Auswahl des besten

Obwohl dieser Algorithmus sehr einfach umzusetzen ist, hat er zwei Probleme, die den praktischen Einsatz erschweren oder gar verhindern. Das erste Problem ist die exponentielle Laufzeitkomplexität. Dadurch ist der Ansatz nur bis zu einer gewissen Schemagröße anwendbar. Das zweite Problem liegt in der Bewertung der generierten integrierten Schemata. Eine allgemeingültige Formel zur Bestimmung des subjektiv besten Schemas wird sich kaum finden lassen.

Da der naive Ansatz zur Bestimmung des optimalen integrierten Schemas nur für wenige Anwendungsfälle praktikabel ist, müssen andere Algorithmen betrachtet werden. Eine Alternative ist das „geschickte“ Sortieren der GIM-Matrix, um maximale Rechtecke zu erhalten. Dieser Ansatz generiert je nach Ausgangsmatrix verschiedene integrierte Schemata.

Der später vorgestellte Algorithmus generiert ein integriertes Schema und bietet zusätzliche Möglichkeiten, das entstandene Schema zu beeinflussen. Der Algorithmus basiert auf Begriffsverbänden, welche im Folgenden kurz vorgestellt werden.

## 4.3 Begriffsverbände

Begriffsverbände (*concept lattices*, siehe [Wil92]) stammen aus der formalen Begriffsanalyse, welche ein Teilgebiet der angewandten Mathematik ist.

Die Theorie der Begriffsverbände basiert auf der folgenden Formalisierung ([Duq87]): Ein formaler Kontext $(G, M, I)$ ist definiert durch

- $G$, der Menge der Entitäten im Kontext,
- $M$, der Menge der Eigenschaften im Kontext, und

- $I \subseteq G \times M$, welche eine binäre Relation zwischen beiden Mengen ausdrückt (eine Entität $g \in G$ hat die Eigenschaft $m \in M$ genau dann wenn $(g, m) \in I$).

Die Menge aller gemeinsamen Eigenschaften einer Entitätsmenge $A \subseteq G$ (*intent*) ist definiert durch:

$$A' := \{m \in M \mid \forall g \in A : (g, m) \in I\}.$$

Gleichermaßen ergibt sich die Menge aller Entitäten (*extent*), die alle in $B \subseteq M$ enthaltenen Eigenschaften haben:

$$B' := \{g \in G \mid \forall m \in B : (g, m) \in I\}.$$

Ein *Konzept* in $(G, M, I)$ ist ein Paar $(A, B) \in \mathcal{P}(G) \times \mathcal{P}(M)$ (Galois-Verbindung) mit $A' = B$ und $B' = A$, welches einem maximalen Rechteck in der binären Beziehung $I$ entspricht.

Die Menge aller Konzepte in $(G, M, I)$ sei:

$$L := \{(A, B) \in \mathcal{P}(G) \times \mathcal{P}(M) \mid A = B' \wedge B = A'\}$$

und $\leq$ die Ordnungsbeziehung in $L$ definiert durch:

$$(A_1, B_1) \leq (A_2, B_2) \Longleftrightarrow A_1 \subseteq A_2.$$

Der resultierende Verband kann notiert werden durch:

$$\mathcal{L} := (L, \leq, \wedge, \vee, (M', M), (G, G'))$$

mit den folgenden Verbandoperationen:

$$(A_1, B_1) \wedge (A_2, B_2) = (A_1 \cap A_2, (A_1 \cap A_2)')$$

und

$$(A_1, B_1) \vee (A_2, B_2) = ((B_1 \cap B_2)', B_1 \cap B_2).$$

Dieses formale Modell kann auf das Problem der Schemaintegration angewandt werden, wobei der Kontext der GIM-Matrix entspricht. Der Verbund kann als Vererbungshierarchie interpretiert werden:

- Entitäten aus $G$ entsprechen den Basisextensionen
- ein Konzept in $(G, M, I)$ ist eine Klasse mit Extensionen und Intensionen
- $\leq$ ist eine Spezialisierungsbeziehung zwischen zwei Klassen
- $\wedge$ ist eine Spezialisierung von zwei Klassen

- $\vee$ ist eine Generalisierung (Vereinigung zweier Extensionen) von zwei Klassen
- $(M', M)$ ist die tiefste Klasse in der Hierarchie (die Intension ist die Vereinigung aller Attribute, die Extension kann leer sein)
- $(G, G')$ ist die höchste Klasse in der Hierarchie (die Extension ist die Vereinigung aller Basisextensionen, Intension kann leer sein)

Basierend auf dieser Definition kann eine Vererbungshierarchie aufgebaut werden. Unglücklicherweise besitzt dieser Ansatz zwei Nachteile, eine Komplexität von $O(2^n)$ $(n = min(|G|, |M|))$ und die Tatsache, dass überflüssige Klassen (Konzepte), die weder eine eigene Extension noch eine Intension haben, generiert werden können[1]. Diese beiden Nachteile werden im folgenden Algorithmus adressiert.

## 4.4 Ein praktikabler Algorithmus

In diesem Abschnitt wird ein Algorithmus vorgestellt, welcher nur gültige Klassen (Klassen, die eine nicht leere Extension und/oder eine nicht leere Intension besitzen) in einer Laufzeitkomplexität von $O(n^3)$ generiert.

Die Mengen $Int_1$ und $Ext_1$ enthalten die *intents* (Zuordnung der Attribute zu den Basisextensionen) jeder Basisextension beziehungsweise *extents* (Zuordnung der Basisextensionen zu den Attributen) jedes einzelnen Attributes:

$$Int_1 := \{\{g\}' \mid g \in G\} \qquad Ext_1 := \{\{m\}' \mid m \in M\}.$$

Aus $Int_1$ und $Ext_1$ können zwei Mengen von Klassen hergeleitet werden:

$$Con_{1,I} := \{(I', I) \mid I \in Int_1\} \qquad Con_{1,E} := \{(E, E') \mid E \in Ext_1\}.$$

Anschließend werden die beiden Mengen vereinigt:

$$Con_1 := Con_{1,I} \cup Con_{1,E}.$$

Aus der entstandenen Menge von Klassen $K$ wird eine Matrix aufgebaut, welche die Spezialisierungsbeziehungen der Klassen darstellt. Eine „1“ in der Zelle $i, j$ der Matrix bedeutet, dass die Klasse $C_i$ eine Subklasse der Klasse $C_j$ ist. Existiert keine Subklassenbeziehung, wird die Zelle mit einer „0“ gefüllt. Um die Matrix zu berechnen, wird jede Klasse mit jeder anderen verglichen.

Durch die Berechnung der Matrix $K_1 = K - K \times K$ werden die transitiven Spezialisierungen entfernt. Die Klassenmenge $Con_1$ und die Matrix $K_1$ enthalten alle notwendigen

[1] In [Sch98] findet sich eine Betrachtung dieses Problemes, welche zeigt, dass solche Klassen weggelassen werden können. Dadurch werden allerdings die Verbandeigenschaften verletzt, die entstandene Vererbungshierarchie ist jedoch gültig.

Informationen, um das integrierte Schema aufzustellen. Das integrierte Schema ist eine Klassenhierarchie, welche die Beziehungen zwischen den Klassen sowie ihre Attribute enthält. Die Klassenmenge $Con_1$ gibt Auskunft, wie die Klassen auf die Basisextensionen abgebildet werden. Die Komplexität des Algortihmuses beträgt $O(n^3)$ ([Sch98]).

Das entstandene integrierte Schema kann in Bezug auf verschiedene Kriterien optimiert werden. Einige mögliche Transformationen werden im folgenden Abschnitt vorgestellt.

## 4.5 Transformation der Vererbungshierarchie

In diesem Abschnitt werden verschiedene Möglichkeiten vorgestellt, wie die entstandene Vererbungshierarchie beeinflusst werden kann.

### Entfernen von Klassen ohne eigene Extension

In dem entstandenen Schema können Klassen enthalten sein, welche mindestens eine Intension, aber keine eigene Extension besitzen (für diese Klassen gibt es keine eigenen Objekte, im allgemeinen als abstrakte Klassen bezeichnet). Die Extension solcher Klassen ergibt sich aus der Vereinigung der Subklassen. Diese Klassen können ohne Informationsverlust entfernt werden. Man sollte jedoch beachten, dass solche Klassen das Verständnis des Schemas erleichtern können.

### Entfernen von Klassen ohne eigene Intension

Parallel können Klassen, welche mindestens eine eigene Extension, aber keine Attribute besitzen, entfernt werden. Zu bedenken ist jedoch, ob in dem beabsichtigten Modellierungskonzept für die Implementation des Schemas (z.B. Objektorientierte Modellierung) das Rollenkonzept (ein Objekt kann mehreren spezialisierten Klassen zugeordnet werden, siehe [Bee93]) unterstützt wird. Ist dem nicht der Fall, so können diese Klassen nicht entfernt werden.

### Zusammenfassen von Klassen

Die Komplexität des entstandenen Schemas kann durch das Einführen von optionalen Attributwerten beliebig verkleinert werden. Durch die optionalen Werte können die Klassen vertikal und/oder horizontal zusammengefasst werden.

Beim vertikalen Zusammenfassen wird eine Klasse mit ihrer Superklasse vereinigt. Das Rechteck in der GIM-Matrix, welches die zusammengefasste Klasse repräsentiert, besteht aus der Vereinigung der Extensionen und Intensionen der zusammengefassten Klassen. Dieses Rechteck wird in der GIM-Matrix durch das Einführen von null-Werten aufgefüllt. Anschließend wird der vorgestellte Algorithmus auf die GIM-Matrix angewendet, das dann generierte integrierte Schema enthält die zusammengefasste Klasse.

Analog dazu erfolgt das horizontale Zusammenfassen von Klassen. Es können die Klassen zusammengefasst werden, die in einer direkten Spezialisierungbeziehung zu der gleiche Superklasse stehen. Wie beim vertikalen Zusammenfassen wird das Rechteck in der GIM-Matrix durch null-Werte aufgefüllt und der Algorithmus angewandt.

#### Aufteilen von Klassen

Die bisher vorgestellten Transformationen dienen zum Vereinfachen des Schemas. Dies ist jedoch nicht in jedem Fall gewünscht. In einigen Fällen kann eine Integration verschiedener Extensionen der Eingangsschemata in eine Extension des integrierten Schemas nicht gewünscht oder möglich sein, zum Beispiel durch nicht auflösbare Konflikte zwischen den einzelnen Extensionen. In diesem Fall kann der Integrationsprozess durch das Einführen von künstlichen Attributen[2] beeinflusst werden. Die künstlichen Attribute verhindern das Zusammenfassen der Klassen (in der GIM-Matrik müssen andere Rechtecke gebildet werden). Nachdem der Algorithmus angewendet wurde, können die künstlichen Attribute wieder entfernt werden.

#### Entfernen von Mehrfachvererbung

Das durch den vorgestellten Algorithmus generierte integrierte Schema kann Klassen enthalten, die mehrere Superklassen haben. Wird keine Mehrfachvererbung gewünscht, so kann diese durch das Duplizieren von Attributen verhindert werden.

Für jede Klasse $C$, welche mehrere Superklassen besitzt, wird festgelegt, welche Superklasse die einzige ($S$) sein soll. Die GIM-Matrix muss in der folgenden Weise modifiziert werden: Jede Zeile, die nicht ein Attribut der gewählten Superklasse $S$ (inklusive der geerbten) repräsentiert, wird verdoppelt. Alle Kreuze, welche zu Basisextensionen der gerade bearbeiteten Klasse $C$ gehören, werden in die duplizierte Zeile verschoben.

Nachdem der Algorithmus auf die modifizierte GIM-Matrix angewendet wurde, entsteht ein Schema, in dem die Klasse $C$ nur eine Superklasse $S$ besitzt. Die Attribute der ehemaligen Superklassen sind in die Klasse $C$ dupliziert wurden.

## 4.6 Zusammenfassung

Der vorgestellte Algorithmus erlaubt eine effiziente Herleitung von Vererbungshierarchien basierend auf zusammengefassten Hierarchien. Mögliche Anwendungen umfassen die Schemaintegration, -modifikation oder -evolution in verschiedenen Bereichen (zum Beispiel in der Datenbankintegration oder die Verfeinerung objektorientierter Schemata).

Das generierte Schema kann durch verschiedene Transformationen zielgerichtet modifiziert werden, während durch das darunterliegende formale Modell die Korrektheit des

[2]Für jede Basisextension, die nicht mit einer anderen zusammengefasst werden soll, wird ein künstliches Attribut gebildet. Dieses wird nur mit der Basisextension in Beziehung gesetzt.

Schemas sichergestellt wird. Die vorgestellten Transformationen basieren auf der Modifikation der GIM-Matrix. Sie können in beliebiger Reihenfolge kombiniert werden, so dass durch einen Zyklus der Form: 1. Schemageneration, 2. Modifikation der GIM-Matrix interaktiv das optimale integrierte Schema erstellt werden kann.

# Kapitel 5

# Schemageneration für XML-Daten mittels GIM

Nachdem in den vorangegangenen Kapiteln XML, verschiedene Methoden zum Speichern von XML in Datenbanken sowie das Generische Integrationsmodell vorgestellt wurden, wird in diesem Abschnitt eine Methode vorgestellt, um ein integriertes Schema zur Speicherung von XML-Daten zu generieren.

## 5.1 Ziele

Die zu erreichenden Ziele bei der Schemageneration für XML-Daten sind wie folgt:

**Integration möglichst vieler struktureller Informationen:** Im generierten Schema sollen alle verfügbaren strukturellen Informationen aus den XML-Daten enthalten sein.

**Keine Nutzerinteraktion notwendig:** Für die Schemageneration ist keine Interaktion mit dem Nutzer notwendig. Wenn nötig, ist dies durch vernünftige Standardbelegungen bei Entscheidungen zu realisieren. Eine interaktive Nutzung für erweiterte Funktionalitäten ist möglich.

**Unabhängigkeit von der Zieldatenbank:** Die Schemageneration muss unabhängig von der Zieldatenbank sein. Unterstützt werden objektorientierte, objektrelationale und relationale Datanbanksysteme.

**Strikte Trennung von Daten und Metadaten:** Die zur Verwaltung der Dokumente notwendigen Metadaten dürfen nicht in das generierte Schema integriert werden.

**Verständlichkeit des Schemas:** Das generierte Schema soll leicht verständlich sein. Für den Zugriff auf die Daten wird keine zusätzliche Anwendung benötigt. Im Idealfall ist das Schema selbsterklärend.

**Optimierung des generierten Schemas:** Es müssen Möglichkeiten angeboten werden, das generierte Schema auf einen konkreten Einsatzzweck hin zu optimieren (beispielsweise die Komplexität zu reduzieren).

## 5.2 Überblick

Abbildung 5.1 zeigt auf der rechten Seite den Ablauf für die Speicherung von XML-Daten in einer Datenbank. Basierend auf XML-Dokumenten oder verschiedenen Beschreibungsmechanismen für XML wird ein integriertes Schema generiert. Mit Hilfe dieses integrierten Schemas werden dann einerseits die notwendigen DDL-Befehle für die gewählte Zieldatenbanksprache und andererseits die Import- und Exportwerkzeuge erstellt.

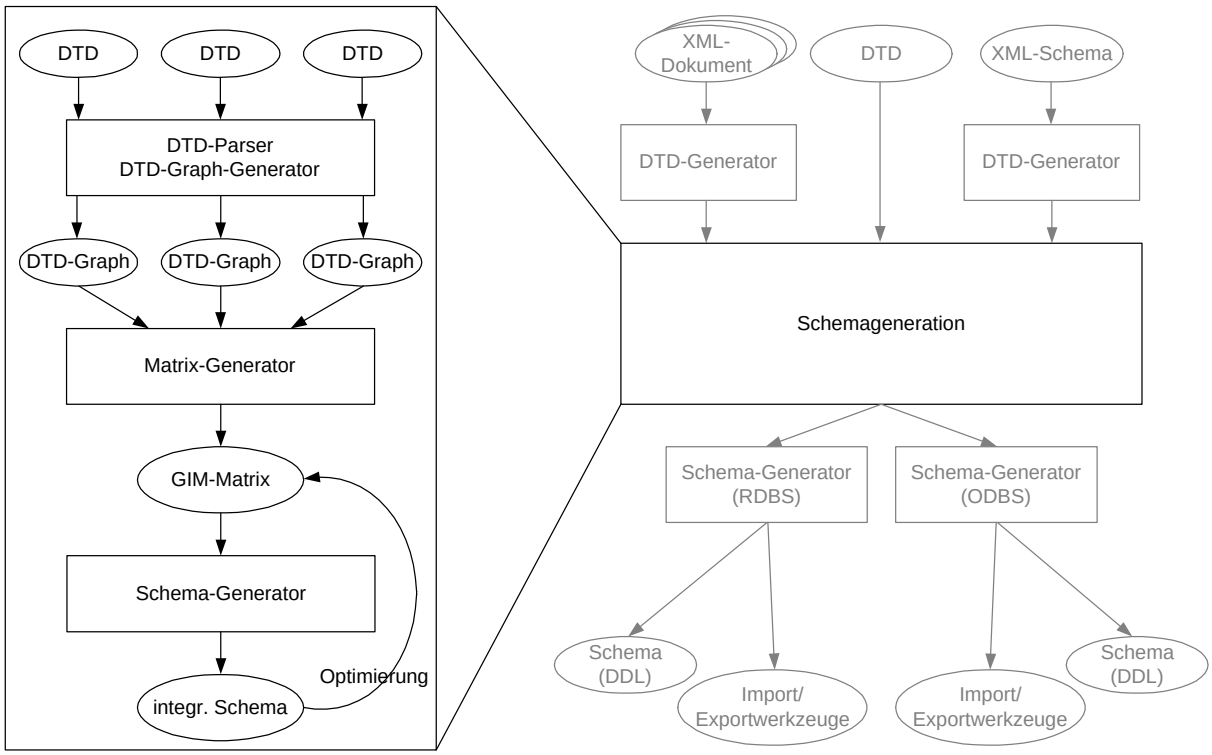


Abbildung 5.1: Ablauf der Schemageneration

Auf der linken Seite ist der Ablauf der Schemageneration dargestellt. Der Ausgangspunkt der Schemageneration ist eine oder mehrere DTDs. Die DTDs werden analysiert

und in DTD-Graphen überführt. Basierend auf den dadurch gewonnenen Informationen wird eine GIM-Matrix generiert. Der im vorigen Kapitel vorgestellte Algorithmus wird auf die Matrix angewandt, um ein integriertes Schema zu erhalten. Anschließend wird das Schema durch diverse Transformationen weiter optimiert und mit zusätzlichen Informationen aus der DTD, die während der Transformation verloren gegangen sind, angereichert und in einer datenbankunabhängigen Form gespeichert. Die einzelnen Schritte werden im Folgenden detaillierter beschrieben.

**Beispiel 5.1.** - DTD

```
1  <!ELEMENT purchase (customer, product, product+)>
2  <!ATTLIST purchase id ID #REQUIRED
3                     status (accepted | delivered) 'accepted'>
4  <!ELEMENT customer (name, address, deliveryaddress?)>
5  <!ATTLIST customer id ID #REQUIRED>
6  <!ELEMENT name (#PCDATA)*>
7  <!ELEMENT address (#PCDATA)*>
8  <!ELEMENT deliveryaddress (#PCDATA)*>
9  <!ELEMENT product (name, ((size, color?) | (size?, color)),
10                    amount, price, descr?)>
11 <!ATTLIST product id CDATA #REQUIRED>
12 <!ELEMENT size (#PCDATA)*>
13 <!ELEMENT color (#PCDATA)*>
14 <!ELEMENT amount (#PCDATA)*>
15 <!ELEMENT price (#PCDATA)*>
16 <!ELEMENT descr (#PCDATA | link | em)*>
17 <!ELEMENT link (#PCDATA)*>
18 <!ELEMENT em (#PCDATA)*>
```

Die DTD aus Beispiel 5.1 wird zur Illustration genutzt. Sie basiert auf dem Beispiel aus Kapitel 2 (Beispiele 2.1, 2.2 und 2.3) und wurde erweitert, um diverse Abbildungen beschreiben zu können. Die Attribute „`size`“ und „`color`“ des Elementes „`product`“ wurden zu Elementen umgewandelt, um sicherzustellen, dass mindestens eine Eigenschaft des Produktes angegeben wird. Das Attribut „`id`“ des Elementes „`product`“ enthält zwar die eindeutige ID des Produktes, kann aber nicht als Typ `ID` definiert werden, da ein Produkt in einem Dokument in verschiedenen (jeweils eindeutigen) Ausprägungen (Farbe oder Größe) mehrfach enthalten sein kann (ein Kunde kauft beispielsweise das gleiche Produkt in rot und blau). Diese Art der Eindeutigkeit kann in XML mittels einer DTD aufgrund der fehlenden kombinierten Schlüssel nicht abgebildet werden.

## 5.3 Analyse der DTD

Die Voraussetzung für die hier vorgestellte Schemageneration für XML-Dokumente mittels GIM ist eine (oder mehrere) DTD(s). Dies ist jedoch keine Einschränkung, da

für XML-Dokumente ohne DTD durch eine Analyse der Dokumente automatisch eine DTD erstellt werden kann[1]. Auch alternative Beschreibungsmechanismen für XML-Dokumente, wie XML Schema (siehe Abschnitt 2.4) oder Document Content Descriptors (siehe [BFM]), können zu DTDs transformiert werden.

Für die Schemageneration können eine oder mehrere DTDs als Ausgangsbasis dienen. Mehrere DTDs ergeben sich dann, wenn verschiedene Dokumentklassen in einem Schema integriert werden sollen. Im folgenden wird nur von einer DTD gesprochen, analoges gilt aber auch für mehrere DTDs. Die DTD wird mittels eines (modifizierten) XML-Parsers verarbeitet und in eine „vereinfachte“ Form überführt. Die Vereinfachungen betreffen das Inlinen von Parameter Entities oder externen DTDs (siehe Abschnitt 2.2.3) sowie das Entfernen von Kommentaren, Entities und Notationen. Das Ergebnis der Verarbeitung durch den Parser ist eine DTD, welche nur Element- und Attribut-Definitionen enthält.

Die vereinfachte DTD wird anschließend in einen DTD-Graphen überführt. Die Transformation basiert auf der in [STZ+99] (siehe Abschnitt 3.3.2) vorgestellten Transformation. Die Knoten des DTD-Graphen werden durch Elemente, Text („`PCDATA`“), Attribute und Operatoren gebildet, die Kanten repräsentieren die Beziehungen. Die Kanten sind gerichtet, um die Struktur der DTD abzubilden. Elemente erscheinen nur einmal im Graphen, Text, Attribute und Operatoren hingegen so oft, wie sie auch in der DTD auftreten.

In Erweiterung zu [STZ+99] können zwischen zwei Knoten mehrere Kanten existieren. Wird beispielsweise ein Element mehrfach in einem anderen aufgeführt („`<!ELEMENT x (a, a, a)>`“), so wird für jedes Auftreten eine Kante erzeugt (in diesem Fall würden drei Kanten von „`x`“ zu „`a`“ erzeugt werden). Diese Informationen werden später bei der Generierung der Matrix benötigt, um die Kardinalitäten für die Vater-Kind-Beziehungen zwischen den Elementen bestimmen zu können.

Zusätzlich werden alle Oder-Verknüpfungen („`|`“) als Operatorknoten in den Baum aufgenommen. Auch diese Informationen werden zur Bestimmung der Kardinalitäten für die Vater-Kind-Beziehungen benötigt. Um die Struktur der DTD adäquat abbilden zu können, wird zusätzlich zum Oder-Operator ein weiterer Operator benötigt. Der sogenannte Listen-Operator fasst alle Elemente oder Operatoren in der gleichen Gruppe auf der gleichen Ebene unterhalb eines Oder-Operators zusammen. Die genaue Verwendung lässt sich am einfachsten an einem Beispiel demonstrieren. Gegeben sei die folgende Elementdefinition[2]:

```
1  <!ELEMENT x ((a,((b,c?)|(c,b?))?)|
2               (b,((a,c?)|(c,a?))?)|
3               (c,((a,b?)|(b,a?))?))>
```

[1] Unter „http://www.hitsw.com/xmltools/“ bzw. „http://www.pault.com/pault/dtdgenerator/“ findet sich eine Onlineversion des DTD-Generators aus dem SAXON-Package („http://saxon.sourceforge.net/“).

[2] Diese Definition ist nicht konstruiert, sondern entspricht der Entitydefinition „`descTitleMetadata`“ aus SVG 1.0 („http://www.w3.org/TR/2001/REC-SVG-20010904/“, dort ist die Definition als Ganzes nochmals optional). Sie stellt sicher, dass mindestens ein „`a`“, „`b`“ oder „`c`“ angegeben werden muss. Die beiden anderen Elemente sind dann optional.

Abbildung 5.2 zeigt die jeweiligen DTD-Graphen mit (rechts) und ohne (links) Listen-Operator, dargestellt durch ein leeres Kästchen. Es ist zu erkennen, dass in der linken Abbildung (ohne Listen-Operator) die Struktur des Elementes „`x`“ nicht richtig abgebildet wird. Diese Diskrepanz entsteht dadurch, dass die *Gruppierungen* von Elementen oder Operatoren (durch die Klammerung) verloren gehen. Um diese Informationen zu erhalten, wird der Listen-Operator eingeführt. Er fasst alle Elemente oder Operatoren in einer Gruppe zusammen.

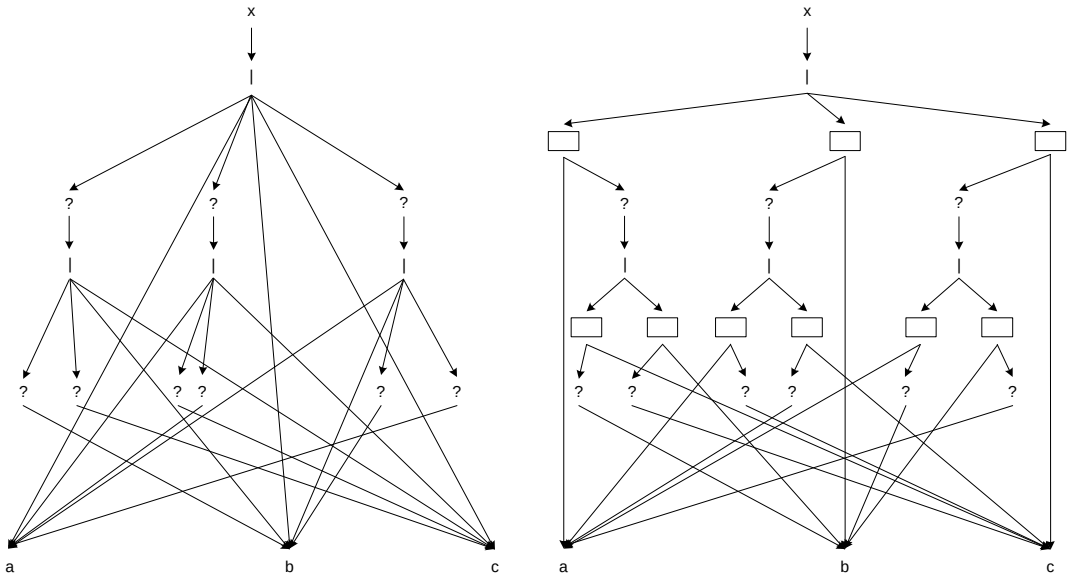


Abbildung 5.2: DTD-Graph mit (rechts) bzw. ohne (links) Listen-Operator

Der Oder-Operator unterscheidet sich von den anderen Operatoren in der Form, dass er nicht so oft, wie er in der DTD enthalten ist, im Graph erscheint. Alle „Oder“ in einer Gruppe werden auf einen Operator abgebildet. Für jeden Oder-Operator wird mindestens ein Listen-Operator erstellt. Werden durch das „Oder“ mehrere Gruppen verknüpft, so werden auch dementsprechend viele Listen-Operatoren generiert.

Der DTD-Graph für die DTD aus Beispiel 5.1 ist in Abbildung 5.3 zu sehen.

Der entstandene DTD-Graph enthält alle relevanten Informationen über die Struktur *einer* DTD. Werden mehrere DTDs für die Schemageneration verwendet, werden auch dementsprechend viele DTD-Graphen generiert. Die DTD-Graphen werden im Folgenden verarbeitet, um eine GIM-Matrix aufzubauen.

## 5.4 Generierung der GIM-Matrix

Die Generierung der GIM-Matrix erfolgt durch die Analyse der DTD-Graphen. Für die Elemente und Attribute werden die entsprechenden Basisextensionen (Spalten der Ma-

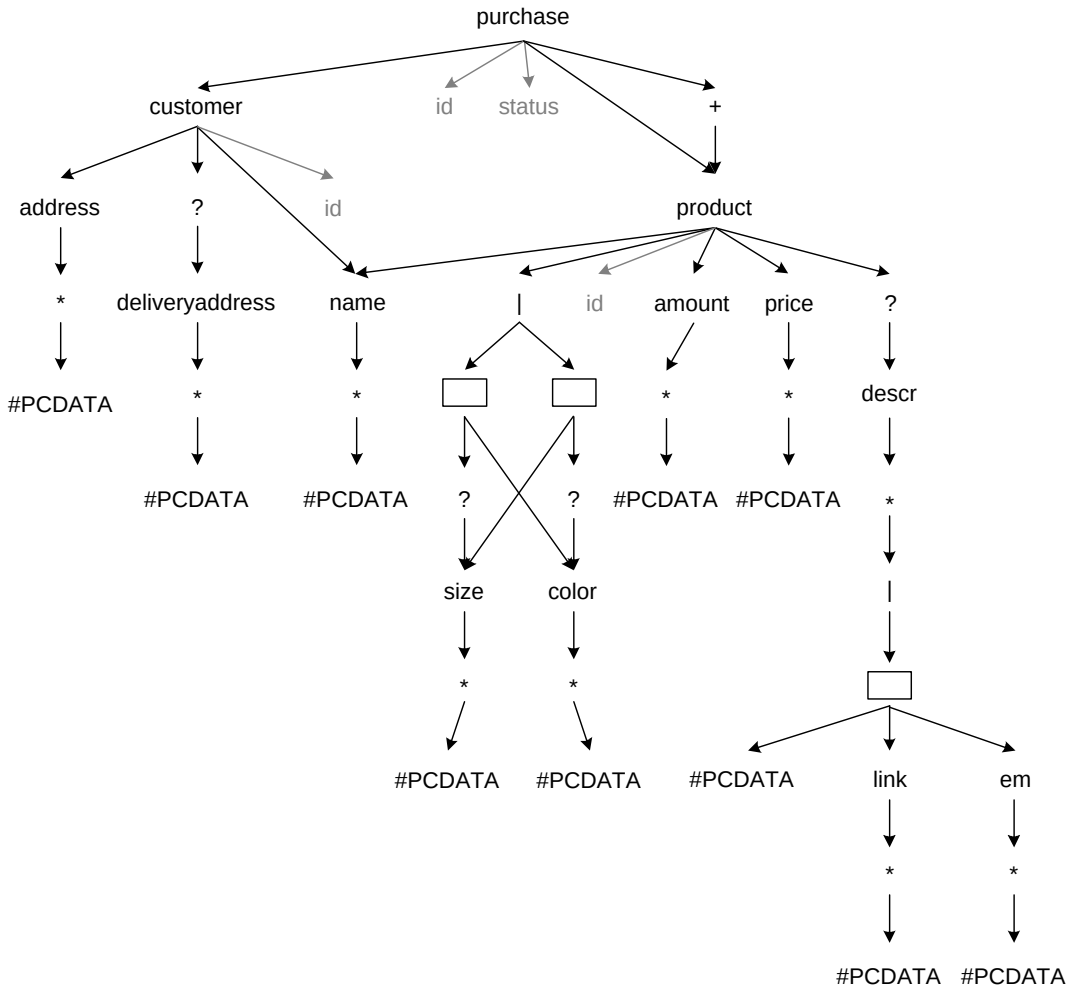


Abbildung 5.3: DTD-Graph für die DTD aus Beispiel 5.1

trix), Attribute (Zeilen in der Matrix, im Folgenden zur Unterscheidung von Attributen aus der DTD *GIM-Attribute* genannt) und die Beziehungen zwischen ihnen (Kreuze in der Matrix) erstellt.

Zuerst werden einige Überlegungen zu den auftretenden Konflikten vorgestellt. Anschließend wird ein Überblick gegeben, wie die einzelnen Komponenten des DTD-Graphen auf die Matrix abgebildet werden, und ein Algorithmus zur Generierung der Matrix repräsentiert. Abschließend werden verschiedene Alternativen zu der angegebenen Abbildung diskutiert.

### 5.4.1 Konfliktbehandlung

Schon bei der Abbildung einer DTD können Konflikte (mindestens auf Attributebene) auftreten. Werden mehrere DTDs integriert, so können nahezu alle Konflikte, welche aus der Schemaintegration beim Entwurf von FDBSen bekannt sind (z.B. in [Sch98]), beobachtet werden.

In dieser Arbeit werden nur die Konflikte betrachtet, welche Probleme bei der Abbildung auf die GIM-Matrix verursachen können. Bei allen anderen Konflikten wird davon ausgegangen, dass sie bereits aufgelöst wurden.

Da die Elemente und Attribute primär durch ihren Namen (und zum Teil durch den Typ) identifiziert werden, werden im Folgenden nur Namenskonflikte analysiert und mögliche Lösungen vorgestellt. Namenskonflikte[3] können schon bei einem DTD-Graphen auf Attributebene, bei mehreren DTD-Graphen auch auf Elementebene[4] auftreten.

#### Konflikte auf Elementebene

Konflikte auf Elementebene können nur auftreten, wenn mehrere DTD-Graphen integriert werden sollen. Zur Lösung dieser Konflikte gibt es drei verschiedene Möglichkeiten.

Die erste Möglichkeit besteht darin, gleichnamige Elemente durch das Einführen von (internen) Präfixen zu unterscheiden. Für jede DTD wird eine ID generiert (beispielsweise ein zugeordneter Namensraum), die als Präfix vor den Elementen geführt wird. Dadurch werden *alle* Elemente verschieden benannt und es existieren keine Konflikte mehr.

Die zweite Möglichkeit besteht darin, alle Elemente gleichen Namens als identisch zu betrachten. Die Attribute und Subelemente aller Elemente werden auf ein einziges Element abgebildet. Dies hat zur Folge, dass die Attribute und Subelemente, die nicht in allen Elementen der verschiedenen DTDs enthalten sind, als optional gekennzeichnet werden müssen.

[3]Ein Namenskonflikt besteht darin, dass zwei oder mehrere Elemente bzw. Attribute den gleichen Namen besitzen (Homonyme). Welcher Art beziehungsweise ob es sich im semantischen Sinne überhaupt um einen Konflikt handelt, ist an dieser Stelle irrelevant (abgesehen davon, dass für andere Konflikte, wie beispielsweise Synonyme, keine automatische Lösung gefunden werden kann).

[4]Innerhalb einer DTD kann es keine Konflikte auf Elementebene geben, da die Namen für Elemente in einer DTD eindeutig sein müssen.

Die dritte Variante ist eine Verfeinerung der zweiten und bezieht den Typ der Elemente (die Attribute und der Inhalt des Elementes) mit ein. Nur Elemente, die den gleichen Typ besitzen, werden zusammengefasst. Haben die Elemente unterschiedliche Typen, so werden sie mittels der ersten Variante verschieden benannt.

Zusätzlich sind Mischungen der Varianten denkbar, die jedoch eine Interaktion mit dem Benutzer erfordern. Durch den Benutzer wird festgelegt, welche Elemente zusammengefasst (Variante 2) und welche getrennt abgebildet werden sollen (Variante 1).

Da die erste Variante zu sehr komplexen Schemata führen kann, die dritte Variante eine detaillierte Analyse aller DTD-Graphen erfordert, wird im Folgenden (wenn notwendig) die zweite Variante angewandt.

### Konflikte auf Attributebene

Innerhalb eines Elementes sind die Attribute durch ihren Namen eindeutig gekennzeichnet. Werden allerdings mehrere Elemente betrachtet, so kann und wird es normalerweise zu Namenskonflikten zwischen den Attributen von verschiedenen Elementen kommen. Zur Lösung der Konflikte gibt es die folgenden Möglichkeiten:

1. Alle Attributnamen werden durch den Namen ihres Elementes (für die die Konflikte bereits behandelt wurden und die dadurch schon eindeutig sind) als Präfix erweitert, wodurch sie wieder eindeutig sind.

2. Alle Attribute gleichen Namens und Typs werden zusammengefasst.

Die erste Variante ist sehr einfach umzusetzten, bewirkt aber auch eine (wahrscheinlich unnötige) Vergrößerung des generierten Schemas, da keine Attribute und daraus folgend auch keine Elemente zusammengefasst werden können.

Die zweite Variante ist in Hinblick auf die Komplexität des generierten Schemas schon vielversprechender und wird daher im Folgenden auch angewandt. Das Zusammenfassen von Attributen gleichen Namens basiert auf ihrem Typ und den Beschränkungen durch den Standardwert. Neben dem Zusammenfassen von Attributen mit exakt dem gleichen Typ können auch Attribute unterschiedlichen Typs zusammengefasst werden. Die Abbildung 5.4 zeigt die möglichen Typ-Kombinationen.

Aus der Tabelle ist ersichtlich, dass alle „normalen“ einwertigen Attributtypen (`CDATA`, `ENTITY` und `NMTOKEN`) zusammengefasst werden können. Damit ist aus den folgenden Gründen kein (oder nur ein geringer) Informationsverlust verbunden:

- `ENTITY`-Attribute enthalten nur den Namen des Entities als Text und können daher mit Text-Attributen zusammengefasst werden[5].

[5]Hierbei wird davon ausgegangen, dass Entities nicht expandiert werden. Werden die Entities expandiert (sprich der jeweilige Inhalt hinter dem Namen wird an der entsprechenden Stelle eingefügt), so müssen Entity-Attribute getrennt behandelt werden. Problematisch ist in diesem Fall, dass Entities nicht nur Text sondern auch beliebige andere (Binär-)Daten enthalten können. Da ein Entity-Attribut nicht typisiert werden kann, muss, wenn die Entities expandiert werden, das Attribut beliebige Daten in unbestimmter Größe aufnehmen können.

| | CDATA | ENTITY | NMTOKEN | ID | IDREF | IDREFS | ENTITIES | NMTOKENS | Werteliste |
|---|---|---|---|---|---|---|---|---|---|
| CDATA | x | x | x | | | | | | |
| ENTITY | x | x | x | | | | | | |
| NMTOKEN | x | x | x | | | | | | |
| ID | | | | x | | | | | |
| IDREF | | | | | x | | | | |
| IDREFS | | | | | | x | | | |
| ENTITIES | | | | | | | x | x | |
| NMTOKENS | | | | | | | x | x | |
| Werteliste | | | | | | | | | x |

Abbildung 5.4: Typ-Kombinationen für das Zusammenfassen von Attributen

- `NMTOKEN`-Attribute sind ein Spezialfall der Text-Attribute. Da die zusätzlichen Bedingungen (nur ein Namenswert) jedoch nur schwer durch Integritätsbedingungen im generierten Schema zu unterstützen sind und der Informationsgewinn durch eine getrennte Behandlung der `NMTOKEN`-Attribute sehr gering ist, werden sie auch mit den Text-Attributen zusammengefasst.

Die Information, um welchen konkreten Attributtyp es sich bei einem Attribut des generierten Schemas handelt, geht durch diese Vereinfachungen verloren, kann jedoch für konkrete Werte aus der Zuordnung zu dem entsprechenden Element wieder rekonstruiert werden.

In gleicher Weise werden die mehrwertigen Attributtypen (`ENTITIES` und `NMTOKENS`) zusammengefasst. Sie werden deshalb getrennt betrachtet, weil der Inhalt der entsprechenden Attribute aus einer Menge von Werten besteht. Bei der Generierung der GIM-Matrix werden sie deshalb ausgelagert, um die einzelnen Werte getrennt speichern zu können.

Auch wenn der ID/IDREF-Mechanismus aus XML aufgrund der fehlenden Typisierung nur sehr schwer auf eine Datenbank abzubilden ist, werden die Attribute `ID`, `IDREF` oder `IDREFS` getrennt behandelt. Dadurch wird sichergestellt, dass in dem generierten Schema die jeweiligen Attribute eindeutig zugeordnet und typisiert werden können.

Eine weitere Ausnahme bilden Attribute mit einer fest vorgegebenen Werteliste. Sie werden nicht mit anderen Attributen zusammengefasst, da durch die vorgegebene Wertemenge zusätzliche Integritätsbedingungen für das generierte Schema gewonnen werden können.

Zusätzlich zum Typ der Attribute werden auch die Beschränkungen durch die Standardwerte bei der Entscheidung, welche Attribute zusammengefasst werden können, beachtet. Hierbei werden drei Möglichkeiten unterschieden, Pflichtattribute (`#REQUIRED`), optionale Attribute (`#IMPLIED` oder ein Standardwert) und Attribute mit festgelegtem Wert (`#FIXED`), da jeweils unterschiedliche Integritätsbedingungen abgeleitet werden können.

Die vorgestellten Bedingungen legen fest, welche Attribute bei auftretenden Konflikten zusammengefasst werden können. Existieren nach dem Zusammenfassen der Attribute noch Konflikte (Attribute gleichen Namens, die aufgrund unterschiedlichen Typs nicht zusammengefasst werden können), so werden die Attributnamen durch das Anfügen des Elementnamens als Präfix erweitert.

### 5.4.2 Abbildung der Konstrukte aus der DTD

In diesem Abschnitt wird die Abbildung der verschiedenen Konstrukte aus der DTD auf die Matrix vorgestellt. Die oben beschriebene Behandlung der Konflikte wird jeweils dann angewandt, wenn ein neues Element oder Attribut behandelt wird. Kann das Element oder das Attribut mit einem bereits existierenden zusammengefasst werden, wird keine neue Basisextension oder ein neues GIM-Attribut erzeugt, sondern die/das bereits existierende verwendet.

Da in dieser Arbeit im allgemeinen von datenorientierten Dokumenten ausgegangen wird, wird die relative Ordnung der Elemente bei der Schemageneration nicht betrachtet.

Die Operatoren werden nicht weiter behandelt, sie werden nur zur Bestimmung der Kardinalitäten für die Vater-Kind-Beziehungen benötigt.

#### Abbildung der Elemente

Alle Elemente werden auf Basisextensionen abgebildet und nicht auf GIM-Attribute[6]. Diese Art der Abbildung begründet sich darin, dass es in einer DTD keinen Mechanismus gibt, um zu kennzeichnen, welche Elemente als *root*-Elemente für XML-Dokumente erlaubt sind. Deshalb kann ein bezüglich der DTD gültiges Dokument mit jedem Element aus der DTD beginnen. Damit in dem generierten Schema alle möglichen Dokumente gespeichert werden können, müssen die Elemente auf Basisextensionen abgebildet werden. Nur dadurch ist sichergestellt, dass für jedes Element eine entsprechende Klasse generiert wird.

Die Namen der Basisextensionen ergeben sich aus den Namen der jeweiligen Elemente. So wird zum Beispiel das Element „`address`“ aus Beispiel 5.1 auf die gleichnamige Basisextension in der GIM-Matrix abgebildet (siehe Abbildung 5.7).

[6]Später wird eine Modifikation vorgestellt, welche mittels einer Interaktion mit dem Nutzer eine Abbildung von Elementen auf GIM-Attribute erlaubt.

### Abbildung von Text

`PCDATA` innerhalb des Elementeinhaltes wird auf *ein* einziges GIM-Attribut abgebildet.

Besteht der Elementeinhalt nur aus Text, so wird dieses GIM-Attribut direkt mit der Basisextension des jeweiligen Elementes in Beziehung gesetzt. Diese Form der Abbildung ist im Beispiel 5.1 unter anderem für den Inhalt der Elemente „`name`“, „`size`“ oder „`amount`“ zu finden.

Besteht das Element hingegen aus Mixed Content (Text und Elemente gemischt), so wird der Text (wie die Elemente) ausgelagert. Dazu wird das GIM-Attribut für Text mit einer neu erstellten Basisextension in Beziehung gesetzt. Diese Basisextension wird wie später beschrieben mit der Basisextension des Elementes verknüpft.

Im Beispiel wird der Text-Inhalt des Elementes „`descr`“ in dieser Form auf die GIM-Matrix abgebildet. Die neu erstellte Basisextension „`#PCDATA`“ (siehe Abbildung 5.7) wird mit dem GIM-Attribut für Text (als „`#PCDATA`“ bezeichnet) in Beziehung gesetzt. Zwischen den Basisextensionen „`#PCDATA`“ und der des Elementes „`descr`“ wird eine Referenz erstellt.

### Abbildung der Attribute

Bei der Abbildung der Attribute wird zwischen einwertigen und mehrwertigen unterschieden.

Einwertige Attribute (`CDATA`, `ID`, `IDREF`, `ENTITY` und `NMTOKEN`) werden auf GIM-Attribute abgebildet. Die Namen der GIM-Attribute entsprechen den Namen der Attribute. Die GIM-Attribute werden mit der Basisextension des jeweiligen Elementes in Beziehung gesetzt.

Im Beispiel 5.1 werden die Attribute „`id`“ der Elemente „`purchase`“ und „`customer`“ auf ein GIM-Attribut abgebildet und mit den Basisextensionen der Elemente in Beziehung gesetzt. Das Attribut „`id`“ des Elementes „`product`“ wird aufgrund des anderen Typs auf ein extra GIM-Attribut abgebildet (siehe Abbildung 5.7).

Mehrwertige Attribute (`IDREFS`, `ENTITIES` und `NMTOKENS`) werden ausgelagert. Dazu wird wie bei den einwertigen Attributen ein entsprechendes GIM-Attribut erstellt[7]. Dieses wird jedoch mit einer neu erstellten Basisextension in Beziehung gesetzt. Dadurch können die einzelnen Werte gesondert gespeichert werden. Die erstellte Basisextension wird mit der Basisextension des jeweiligen Elementes wie im folgenden Abschnitt beschrieben verknüpft.

[7]Dieses GIM-Attribut wird wie für ein entsprechendes einwertiges Attribut erstellt. Dadurch ist es möglich, dass Attribute gleichen Namens und „Typs“ (`ENTITY`/`NMTOKEN` und `ENTITIES`/`NMTOKENS` bzw. `IDREF` und `IDREFS`) auf das gleiche GIM-Attribut verweisen.

### Abbildung der Struktur

Um die Vater-Kind-Beziehungen zwischen den Elementen, Text und mehrwertigen Attributen abzubilden, werden *Referenzen* erstellt. Eine Referenz besteht aus zwei GIM-Attributen.

Das erste GIM-Attribut repräsentiert die Referenz des Vaters auf das Kind. Der Name wird durch das Aneinanderhängen der Namen des Vaters und des Kindes und den Suffix „`.ref`“ gebildet. Es wird mit der Basisextension des Vaters in Beziehung gesetzt. Für diese Referenz ist es durch Analyse des DTD-Graphen möglich, die Kardinalität zu bestimmen. Indem der Pfad im Graphen vom Vater zum Kind analysiert wird, ist es möglich, zu bestimmen, ob die Referenz optional ist (optional ist sie, wenn ein „`?`“, „`|`“[8] oder „`*`“ im Weg enthalten ist) und ob multiple Referenzen enthalten sein können (multiple Referenzen sind dann möglich, wenn der Weg ein „`+`“ oder ein „`*`“ enthält). Aufgrund diese Informationen wird die Kardinalität der Referenz bestimmt (0:1, 0:n, 1:1 oder 1:n).

So besitzt zum Beispiel die Referenz von „`product`“ auf „`descr`“ in Abbildung 5.3 eine Kardinalität von 0:1, da in dem Weg zwischen den Knoten nur ein „`?`“ enthalten ist. Die Referenz von „`product`“ auf „`price`“ hingegen besitzt eine Kardinalität von 1:1, da in dem Pfad kein weiterer Knoten enthalten ist.

Da im Graph mehrere (zum Teil identische) Wege zwischen zwei Elementen existieren können (z. B. zwischen „`purchase`“ und „`product`“ oder zwischen „`product`“ und „`size`“ bzw. „`color`“ in Abbildung 5.3) werden auch dementsprechend viele Referenzen in der Matrix erstellt. Diese werden nach der Generierung der Matrix wie später beschrieben zusammengefasst.

Das zweite GIM-Attribut, die Rückreferenz vom Kind auf den Vater, wird mit dem Namen des Vaters und dem Suffix „`.backref`“ bezeichnet und mit der Basisextension des Kindes in Beziehung gesetzt. Für die Rückreferenzen ist es nicht möglich, die Kardinalitäten zu bestimmen[9]. Aus diesem Grund werden alle Rückreferenzen zu einem Vater zu einem GIM-Attribut zusammengefasst.

Diese beiden GIM-Attribute können nach der Generierung des Schemas in einem

[8] Die Behandlung des „Oders“ als optional führt in dem Fall, dass in jedem Zweig des Oders das Kind-Element (nicht optional) enthalten ist, zu einer falschen minimalen Kardinalität. Wenn man die Elementdefinition

```
<!ELEMENT x ((a,b)|(a,c))>
```

betrachtet, so kann man erkennen, dass das Element „`a`“ nicht optional ist. Da diese Information jedoch nur durch eine detaillierte Analyse aller Pfade zu allen Kindern des aktuellen Vater-Elementes bestimmt werden kann, wird das „Oder“ in dieser Arbeit als optional behandelt.

[9] Genauer gesagt ist die Kardinalität für Beziehungen zwischen Elementen immer 0:n und zwischen Elementen und Attributen bzw. Text immer 1:n. Zwischen Elementen muss sie optional sein, da ein Element nicht in einem anderen geschachtelt sein muss. Bei Attributen oder Text muss mindestens ein Element angegeben sein, da sie nicht losgelöst auftreten können. Multiple Rückreferenzen müssen möglich sein, da das gleiche Element, Attribut oder der gleiche Text in mehreren Elemente enthalten sein kann (vorausgesetzt es ist möglich, in XML redundant gespeicherte identische Objekte bei der Speicherung in der Datenbank zu identifizieren und auf ein Objekt abzubilden).

Verfeinerungsschritt wieder in entsprechende Referenzen umgewandelt werden.

### 5.4.3 Generierung der Matrix

Um die Extensionen und die GIM-Attribute für die GIM-Matrix zu generieren, wird der Algorithmus aus Abbildung 5.5 angewendet. Basisextensionen werden verkürzt als „`Extension`“ bezeichnet.

Die jeweiligen Funktionen „`getOrCreate... (Node)`“ (nicht aufgeführt) implementieren die Konfliktbehandlung. Sie geben jeweils eine Basisextension oder ein GIM-Attribut zurück. Je nachdem, ob Elemente oder Attribute zusammengefasst werden können, werden neue Basisextensionen oder GIM-Attribute in der Matrix erstellt oder bereits existierende zurückgegeben.

Die Funktion „`createReference (Extension, Extension, Path)`“ (nicht aufgeführt, Verwendung zum Beispiel in Zeile 12 in Abbildung 5.5) erstellt die Referenzen zwischen Elementen, Text und Attributen (vgl. Abschnitt 29). Sie bekommt als Parameter den Pfad zwischen den Knoten übergeben, um die Kardinalität für die Referenz bestimmen zu können. Der Pfad wird intern an der Referenz gespeichert, um später doppelte Referenzen reduzieren zu können. Für die Referenzen auf Text in Mixed Content (siehe Zeile 34) und auf mehrwertige Attribute (siehe Zeile 26) werden spezielle vordefinierte Pfade verwendet, um sicherzustellen, dass die richtigen Kardinalitäten (0:n) erzeugt werden.

In ähnlicher Weise funktioniert die Funktion „`createReference (GIMAttribut, Extension)`“ (nicht aufgeführt, Verwendung zum Beispiel in Zeile 22) welche jedoch nur eine einfache Beziehung zwischen dem GIM-Attribut und der Basisextension in der Matrix erstellt.

Die Funktion „`handleNode (Node, ParentNode, Path)`“ implementiert die Behandlung eines Knotens des Graphen. Der Parameter „`Node`“ enthält den Knoten, der auf die GIM-Matrix abgebildet wird, „`ParentNode`“ den Knoten des übergeordneten Elementes (sofern eins existiert) und „`Path`“ den Pfad vom Knoten des Vater-Elementes zum aktuellen Knoten. Der aktuelle Knoten wird entsprechend seinem Typ behandelt.

Handelt es sich um einen Knoten, der ein Element repräsentiert (Zeilen 10 bis 18), so wird zuerst eine Basisextension für das Element erstellt (bzw. die bereits existierende Basisextension aus der Matrix gesucht). Wenn der „`ParentNode`“ nicht leer ist, so wird eine Referenz zwischen den Extensionen der beiden Knoten erstellt (um die Vater-Kind-Beziehung abzubilden). Anschließend wird überprüft, ob der Knoten bereits behandelt wurde. Ist dies nicht der Fall, so wird durch alle Knoten unterhalb des aktuellen traversiert (mit dem aktuellen Knoten als „`ParentNode`“ und einem leeren Pfad). Auf diese Weise wird sichergestellt, dass alle Pfade innerhalb des Graphen exakt einmal durchlaufen werden.

Bei Attributknoten (Zeilen 19 bis 28) wird ein GIM-Attribut erstellt (bzw. das entsprechende aus der Matrix gesucht). Handelt es sich um ein einwertiges Attribut, wird es direkt mit der Basisextension des „`ParentNode`“ (dem zugeordneten Element) in Be-

```
1  Matrix generate (DTDGraph) {
2    for each ElementNode in DTDGraph {
3      handleNode (ElementNode, null, emptyPath);
4    }
5    reduceReferences ();
6    return (Matrix);
7  }
8
9  handleNode (Node, ParentNode, Path) {
10   if (Node == ElementNode) {
11     Extension e = getOrCreateElementExtension (Node);
12     if (parent != null) createReference (e, getExtension (ParentNode), Path);
13     if (!nodeVisited (Node)) {
14       for each ClientNode in Node {
15         handleNode (ClientNode, Node, emptyPath);
16       }
17     }
18   }
19   if (Node == AttributeNode) {
20     GIMAttribut a = getOrCreateAttributeGIMAttribut (Node);
21     if (Node.isSingleValued ()) {
22       createReference (a, getExtension (ParentNode));
23     } else {
24       Extension e = getOrCreateAttributeExtension (Node);
25       createReference (a, e);
26       createReference (e, getExtension (ParentNode), attributePath);
27     }
28   }
29   if (Node == TextNode) {
30     GIMAttribut a = getTextGIMAttribut ();
31     if (ParentNode.isMixed ()) {
32       Extension e = getOrCreateTextExtension (Node);
33       createReference (a, e);
34       createReference (e, getExtension (ParentNode), textPath);
35     } else {
36       createReference (a, getExtension (ParentNode));
37     }
38   }
39   if (Node == OperatorNode) {
40     Path.add (Node);
41     for each ClientNode in Node {
42       handleNode (ClientNode, ParentNode, Path);
43     }
44   }
45 }
```

Abbildung 5.5: Matrixgeneration

ziehung gesetzt. Ist es ein mehrwertiges Attribut, so wird eine Basisextension erstellt, mit dem GIM-Attribut in Beziehung gesetzt und die Referenz vom „`ParentNode`“ auf die Basisextension erstellt.

In ähnlicher Weise werden die Textknoten (Zeilen 29 bis 38) behandelt.

Als letzte Möglichkeit für den Knotentyp bleiben die Operatorknoten (Zeilen 39 bis 44). Bei Operatorknoten wird der Pfad durch den aktuellen Knoten erweitert und anschließend durch alle Knoten unterhalb des aktuellen traversiert.

Da in den DTD-Graphen nicht alle Knoten miteinander verbunden sein müssen und damit auch kein (eindeutiger) Wurzelknoten existieren muss, muss die Behandlung der Knoten wie beschrieben für jeden Elementknoten im Graphen durchgeführt werden (siehe Zeile 3). Dadurch wird sichergestellt, dass jeder Knoten im Graphen einmal durchlaufen wird.

### Zusammenfassen der Referenzen

Da zwischen zwei Elementen mehrere Wege im Graphen existieren können, werden dementsprechend auch multiple Referenzen zwischen den gleichen Basisextensionen in der GIM-Matrix erstellt. Die Abbildung 5.6 zeigt einen Ausschnitt aus der GIM-Matrix für den DTD-Graphen aus Abbildung 5.3, wie sie mit dem bisher vorgestellten Algorithmus erstellt werden würde. So existierten beispielsweise zwei Referenzen von „`product`“ auf „`size`“.

Diese Referenzen können reduziert werden, wobei alle Referenzen gleichen Namens zusammengefasst werden. Im Beispiel wären dies unter anderem „`size.ref`“.

In einem ersten Schritt können alle Referenzen zusammengefasst werden, die exakt den gleichen Pfad besitzen. Dabei handelt es sich um Referenzen, welche für Elemente in der gleichen Gruppierung erstellt wurden (z.B. in „`<!ELEMENT x (a, a, a)>`“ die Referenzen auf „`a`“). Hierzu werden die minimalen und maximalen Kardinalitäten jeweils addiert und alle GIM-Attribute bis auf eines entfernt.

Im zweiten Schritt werden alle übrig gebliebenen Referenzen zusammengefasst, wobei die Berechnung der minimalen und maximalen Kardinalitäten vom Pfad abhängt. Es wird jeweils die Referenz mit dem längsten Pfad gesucht. Der letzte Operatorknoten wird aus dem Pfad entfernt. Anschließend wird überprüft, ob die Referenz mit einer anderen zusammengefasst werden kann (basierend darauf, ob sie den gleichen Pfad besitzen). Ist dies der Fall, so wird abhängig vom entfernten Operatorknoten die Kardinalität bestimmt. Handelt es sich um einen Oder-Knoten oder um einen Listen-Knoten, so ist die neue minimale Kardinalität das Minimum der minimalen Kardinalitäten der beiden Referenzen, die neue maximale dementsprechend das Maximum. Bei allen anderen Operatorknoten werden die Kardinalitäten wie im ersten Schritt addiert. Diese Abfolge wird solange wiederholt, bis nur noch eine Referenz existiert.

Im Beispiel werden die Referenzen „`color.ref`“, „`product.ref`“ und „`size.ref`“ zu einer reduziert. Bei „`color.ref`“ und „`size.ref`“ werden die jeweiligen beiden Referenzen über einen Oder-Operator zusammengeführt (vgl. Abbildung 5.3), daher ergibt sich eine

| | ... | color | customer | ... | product | purchase | size |
|---|---|---|---|---|---|---|---|
| ... | | | | | | | |
| product.backref | | x | | | | | x |
| purchase.backref | | | x | | x | | |
| ... | | | | | | | |
| product.color.ref (0 : 1) | | | | | x | | |
| product.color.ref (0 : 1) | | | | | x | | |
| ... | | | | | | | |
| product.size.ref (0 : 1) | | | | | x | | |
| product.size.ref (0 : 1) | | | | | x | | |
| ... | | | | | | | |
| purchase.product.ref (1 : 1) | | | | | | x | |
| purchase.product.ref (1 : n) | | | | | | x | |
| ... | | | | | | | |

Abbildung 5.6: Ausschnitt aus der GIM-Matrix für den DTD-Graphen aus Abbildung 5.3 vor der Reduzierung der Referenzen

Kardinalität von 0:1. Anders verhält es sich bei „`product.ref`“, dort ergibt sich eine Kardinalität von 2:n.

Nachdem die multiplen Referenzen entfernt wurden, ist die Generierung der Matrix abgeschlossen. Abbildung 5.7 zeigt die vollständige GIM-Matrix, die für den DTD-Graphen aus Abbildung 5.3 durch den vorgestellten Algorithmus generiert wird.

### 5.4.4 Mögliche Variationen

Neben der bereits beschriebenen Generierung der GIM-Matrix gibt es verschiedene Modifikationen, welche entweder die Komplexität der Matrix (und dadurch auch des generierten Schemas) reduzieren oder den Detaillierungsgrad der Abbildung erhöhen. Diese Variationen benötigen teilweise eine Interaktion mit dem Nutzer. Im Folgenden werden einige alternative Abbildungen der Konstrukte aus der DTD vorgestellt.

#### Einfache Elemente als Attribute

Die erste Variation besteht darin, einfache Elemente nicht auf Basisextensionen, sondern auf GIM-Attribute abzubilden. Einfache Elemente (*simple elements*) enthalten nur Text („`#PCDATA`“) und besitzen keine Attribute. Damit entsprechen sie prinzipiell Attributen des übergeordneten Elementes.

| | #PCDATA | address | amount | color | customer | deliveryadress | descr | em | link | name | price | product | purchase | size |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #PCDATA | x | x | x | x | | x | | x | x | x | x | | | x |
| customer.backref | | x | | | | x | | | | x | | | | |
| descr.backref | x | | | | | | | x | x | | | | | |
| product.backref | | | x | x | | | x | | | x | x | | | x |
| purchase.backref | | | | | x | | | | | | | x | | |
| customer.address.ref (1 : 1) | | | | | x | | | | | | | | | |
| customer.d-address.ref (0 : 1) | | | | | x | | | | | | | | | |
| customer.name.ref (1 : 1) | | | | | x | | | | | | | | | |
| descr.#PCDATA.ref (0 : n) | | | | | | | x | | | | | | | |
| descr.link.ref (0 : n) | | | | | | | x | | | | | | | |
| descr.em.ref (0 : n) | | | | | | | x | | | | | | | |
| product.amount.ref (1 : 1) | | | | | | | | | | | | x | | |
| product.color.ref (0 : 1) | | | | | | | | | | | | x | | |
| product.descr.ref (0 : 1) | | | | | | | | | | | | x | | |
| product.name.ref (1 : 1) | | | | | | | | | | | | x | | |
| product.price.ref (1 : 1) | | | | | | | | | | | | x | | |
| product.size.ref (0 : 1) | | | | | | | | | | | | x | | |
| purchase.customer.ref (1 : 1) | | | | | | | | | | | | | x | |
| purchase.product.ref (2 : n) | | | | | | | | | | | | | x | |
| id | | | | | | | | | | | | x | | |
| id (ID) | | | | | x | | | | | | | | x | |
| status (optional) | | | | | | | | | | | | | x | |

Abbildung 5.7: GIM-Matrix für den DTD-Graphen aus Abbildung 5.3

Diese Modifikation muss jedoch explizit durch den Nutzer gesteuert werden, da dadurch in dem generierten Schema nicht mehr jedes beliebige der DTD entsprechende Dokument gespeichert werden kann (vgl. Abschnitt 5.4.2). Vom Nutzer werden vor der Generierung der Matrix die einfachen Elemente, die in allen zu speichernden Dokumenten nur innerhalb anderer Elemente auftreten, markiert. Nach der Matrix-Generation können für diese Elemente, sofern alle Referenzen auf die Basisextension des Elementes eine maximale Kardinalität von eins besitzen[10] die Basisextensionen entfernt werden. Sie werden durch entsprechende GIM-Attribute ersetzt, welche mit den Basisextensionen, die eine Referenz auf die entfernte Basisextension besitzen, in Beziehung gesetzt werden. Besitzt eine Referenz auf die entfernte Basisextension eine minimale Kardinalität von null, so wird das erstellte GIM-Attribut als optional gekennzeichnet. Anschließend können die Referenzen und Rückreferenzen, welche keine Beziehungen mehr besitzen, aus der Matrix entfernt werden.

Im Beispiel 5.1 könnten (unter der Annahme, dass alle Dokumente ein `purchase`-Element als Wurzel besitzen) die Elemente „`name`“, „`address`“, „`deliveryaddress`“, „`size`“, „`color`“, „`amount`“ und „`price`“ auf GIM-Attribute abgebildet werden. Die Elemente „`link`“ und „`em`“ können nicht in dieser Form abgebildet werden, da die maximale Kardinalität der Referenzen auf die entsprechenden Basisextensionen größer als eins ist[11]. In Abbildung 5.8 ist die unter anderem in dieser Weise modifizierte GIM-Matrix zu sehen.

### Mixed Content nicht aufsplitten

Eine weitere Möglichkeit, die Generierung der GIM-Matrix zu beeinflussen, besteht darin, Mixed Content nicht aufzusplitten. Der gesamte Inhalt eines solchen Elementes wird auf ein GIM-Attribut abgebildet. Ein entsprechendes GIM-Attribut wird für jedes Element mit Mixed Content gebildet und mit der Basisextension des Elementes in Beziehung gesetzt. Die jeweiligen Kind-Elemente werden an dieser Stelle (im Algorithmus aus Abbildung 5.5 in den Zeilen 14 bis 16) nicht durchlaufen. Das GIM-Attribut wird entsprechend gekennzeichnet, so dass im generierten Schema für die daraus gebildete Eigenschaft die Information vorliegt, dass ein XML-Fragment darin gespeichert wird[12].

Im Beispiel 5.1 kann das Element „`descr`“ in dieser Weise vereinfacht werden. In Abbildung 5.8 wurde das Element „`descr`“ noch weiter vereinfacht. Da es keine Attribute besitzt und der Inhalt, wenn er nicht aufgesplittet wird, einfachem Text entspricht, handelt es sich um ein einfaches Element. Diese werden in diesem Beispiel auf GIM-Attribute abgebildet. Weiterhin verschwinden die Elemente „`link`“ und „`em`“ vollständig aus der

[10] Als zusätzliche Erweiterung wäre es möglich, größere Kardinalitäten zuzulassen und entsprechend viele GIM-Attribute zu bilden.

[11] Genauer gesagt können die Elemente „`link`“ und „`em`“ nie auf GIM-Attribute abgebildet werden, da die Referenzen auf die entsprechenden Basisextensionen eine unbestimmte maximale Kardinalität besitzen.

[12] Diese Information ist dann hilfreich, wenn das Zieldatenbanksystem eine besondere Unterstützung für Attribute vom Typ XML bieten.

Matrix, da keine Referenzen auf sie existieren (sie sind komplett in „`descr`“ enthalten).

### ID-Attribute zusammenfassen

Die dritte Möglichkeit, die generierte Matrix zu vereinfachen, besteht darin, ID/IDREF-Attribute nicht getrennt zu behandeln. Dies ist besonders dann möglich, wenn auf der Zieldatenbank keine geeignete Unterstützung für den ID/IDREF-Mechanismus aus XML gefunden werden kann[13]. Kommt es bei der Generierung der Matrix zu den bereits beschriebenen Konflikten, so werden Attribute vom Typ `ID` und `IDREF` mit denen vom Typ `CDATA`, `ENTITY` und `NMTOKEN` zusammengefasst, gleiches gilt für Attribute vom Typ `IDREFS`, welche mit `ENTITIES` und `NMTOKENS` zusammengefasst werden.

Im Beispiel 5.1 können bei Anwendung dieser Modifikation die Attribute „`id`“ der Elemente „`purchase`“, „`customer`“ und „`product`“ auf ein GIM-Attribut abgebildet werden. Die Umsetzung ist in Abbildung 5.8 zu sehen.

| | customer | product | purchase |
|---|---|---|---|
| purchase.backref | x | x | |
| address | x | | |
| amount | | x | |
| color (optional) | | x | |
| purchase.customer.ref (1 : 1) | | | x |
| deliveryaddress (optional) | x | | |
| descr (XML, optional) | | x | |
| name | x | x | |
| price | | x | |
| purchase.product.ref (2 : n) | | | x |
| size (optional) | | x | |
| id | x | x | x |
| status (optional) | | | x |

Abbildung 5.8: Vereinfachte GIM-Matrix für den DTD-Graphen aus Abbildung 5.3 (einfache Elemente sowie ID-Attribute zusammengefasst und Mixed Content nicht aufgesplittet)

### Uninterpretierte Speicherung beliebiger Elemente

Bietet das Zieldatenbanksystem eine besondere Unterstützung für Attribute vom Typ XML, so können die Teile der DTD, die wenig strukturiert sind, zu einer schwachen

[13] Zum Beispiel ist eine Umsetzung auf eine relationale Datenbank aufgrund der fehlenden Typisierung sehr schwierig. Einfacher ist es bei einer objektorientierten Datenbank, da das IDREF-Attribut dann beispielsweise einfach eine Referenz auf ein konkretes Objekt enthält.

Füllung des Schemas führen oder selten in Anfragen enthalten sind, uninterpretiert als XML-Fragmente gespeichert werden. Diese Variation entspricht dem Ansatz, Mixed Content nicht aufzusplitten, wird jedoch auf beliebige XML-Elemente angewandt. Vor der Schemageneration kann der Nutzer die Elemente, die nicht weiter interpretiert werden sollen, markieren. Der gesamte Inhalt eines jeden markierten Elementes wird dann auf ein entsprechendes GIM-Attribut (mit einer Kennzeichnung des Typs XML) abgebildet. Dieses Attribut wird mit der Basisextension des Elementes in Beziehung gesetzt. Die jeweiligen Kind-Elemente werden nicht weiter behandelt.

#### Typisierung der IDREF-Attribute

Die letzte vorgestellte Variation der Matrixgeneration vereinfacht die Matrix nicht, sondern erhöht den Detaillierungsgrad. Ist dem Nutzer bekannt, dass bestimmte IDREF(S)-Attribute immer auf einen Elementtyp verweisen, so können diese Informationen in die Matrixgeneration einfließen, um die IDREF(S)-Attribute zu typisieren. Vor der Matrixgeneration werden die IDREF(S)-Attribute markiert und das jeweilige Zielelement festgestellt. Während der Erstellung der Matrix werden diese Attribute getrennt behandelt und anstelle von einfachen GIM-Attributen Referenzen erstellt.

## 5.5 Generierung des integrierten Schemas

Nachdem die GIM-Matrix erstellt wurde, wird wie in Kapitel 4 beschrieben das integrierte Schema generiert.

Das für die Matrix aus Abbildung 5.7 generierte Schema ist in Abbildung 5.9 dargestellt. Die Klassen werden durch die Kästen repräsentiert. Die Basisextensionen der Matrix sind fett gedruckt, die GIM-Attribute in einfacher Schrift. Die schwarzen Pfeile zwischen den Klassen repräsentieren die Vererbungshierarchie. Der Pfeil zeigt jeweils von der Subklasse zur Oberklasse. Die grauen gestrichelten Pfeile verdeutlichen die Referenzen (auf eine gesonderte Darstellung der Rückreferenzen durch Pfeile wurde aus Übersichtlichkeitsgründen verzichtet).

## 5.6 Modifikation des Schemas

Da das generierte Schema in den seltensten Fällen bereits optimal ist, ist es ratsam, es weiter zu modifizieren. Die möglichen Transformationen des Schemas wurden bereits in Abschnitt 4.5 vorgestellt. Die Transformationen werden im Folgenden im Kontext der Schemageneration für XML-Daten bewertet.

Klassen ohne eigene Extensionen (in Abbildung 5.9 zum Beispiel die beiden oberen Klassen) können problemlos aus dem Schema entfernt werden. Dies ist sogar oft empfehlenswert, da durch den Algorithmus solche Klassen wann immer möglich generiert werden.

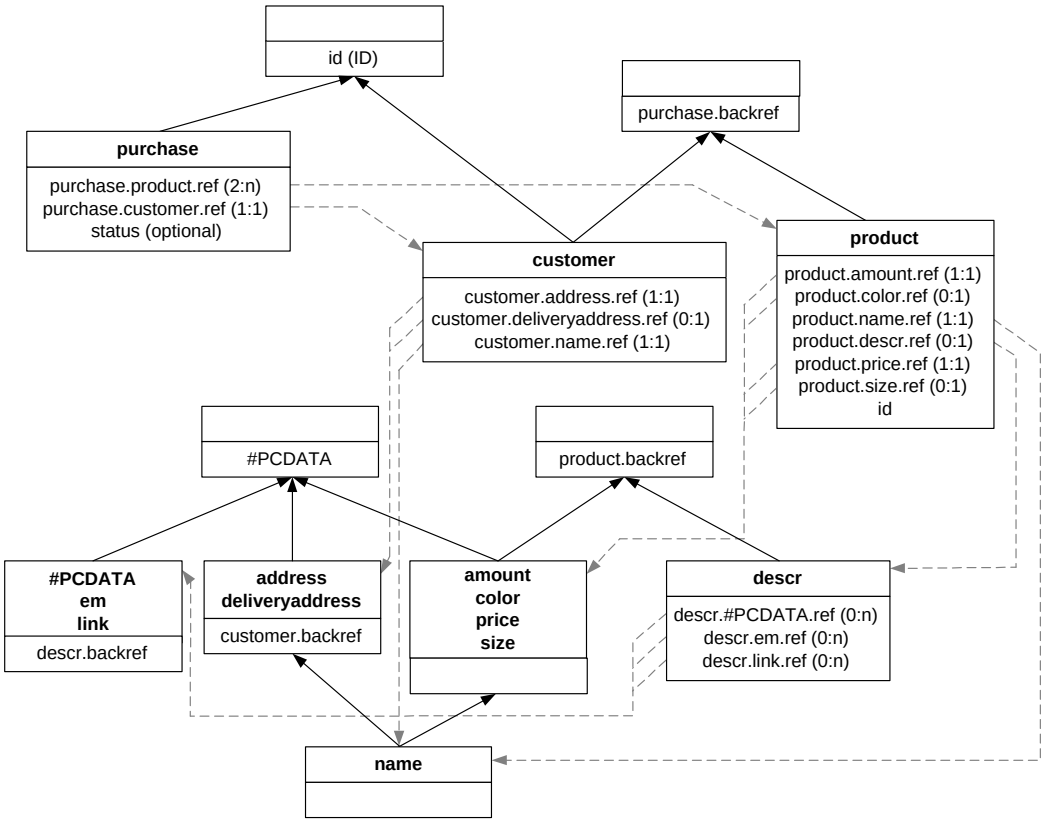


Abbildung 5.9: Generiertes Schema für die GIM-Matrix aus Abbildung 5.7

Des Entfernen von Klassen ohne eigene Intensionen (in Abbildung 5.9 die Klasse „name“) ist nicht problemlos möglich. Dadurch würde die eindeutige extensionale Zuordnung der Elemente aus der DTD auf die Klassen verloren gehen. Deshalb wird diese Transformation nicht unterstützt.

Die weiteren in Abschnitt 4.5 vorgestellten Transformationen können problemlos auf das generierte Schema angewandt werden.

Eine in diesem Kontext spezielle Modifikation des Schemas besteht in der Auflösung der Referenzen. Im Gegensatz zu den bisherigen Transformationen manipuliert diese zum Teil direkt das generierte Schema und nicht die GIM-Matrix. Deshalb muss das Auflösen der Referenzen als letzter Schritt auf das Schema angewandt werden.

Während der Generation der Matrix wurden Vorwärts- und Rückreferenzen erstellt. Abhängig von der gewählten Zieldatenbank wird oft nur ein Referenztyp benötigt[14].

[14]Für eine objektorientierte Datenbank werden die Vorwärtsreferenzen benötigt, welche ein „Besteht-

Um die Rückreferenzen zu entfernen, werden zuerst die Klassen, welche keine eigene Extension und nur Rückreferenzen als Eigenschaften[15] besitzen, wie oben beschrieben durch die Manipulation der GIM-Matrix entfernt. Dadurch wird sichergestellt, dass keine leeren Klassen (ohne eigene Extension und Eigenschaften) im Schema entstehen. Im zweiten Schritt werden direkt im generierten Schema die GIM-Attribute der Vorwärtsreferenzen durch Verweise auf die Klassen ersetzt und die der Rückreferenzen entfernt. Die Vorwärtsreferenzen können analog entfernt werden.

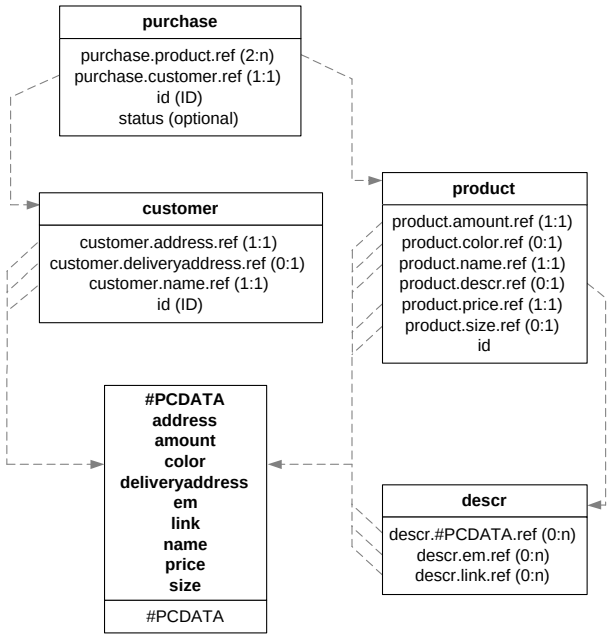


Abbildung 5.10: Modifiziertes Schema basieren auf dem Schema aus Abbildung 5.9

Abbildung 5.10 zeigt ein modifiziertes Schema. Im ersten Schritt wurden die drei Subklassen der Klasse „#PCDATA“ mit dieser und der Klasse „name“ zusammengefasst, da sie alle nur „#PCDATA“ als eigenständige Eigenschaft besitzen (die weiteren Unterscheidungen entstehen durch die Rückreferenzen). Zweitens wurde die oberste Klasse „id“ (die keine eigene Extension besitzt) entfernt. Anschließend wurden die Rückreferenzen entfernt, wodurch die Klassen „purchase.backref“ und „product.backref“ verschwinden.

Ein noch einfacheres Schema erhält man aus der alternativen Matrix aus Abbildung 5.8. Das Schema nach der Entfernung der beiden Klassen ohne Extensionen „id“

Aus“ modellieren. Anders wäre es bei einer relationalen Datenbank. Dort wird üblicherweise ein „Enthalten-In“ modelliert, so dass die Rückreferenzen benötigt werden.

[15] Als Eigenschaften werden die Attribute der Klassen bezeichnet, um sie von den Attributen aus der DTD unterscheiden zu können.

und „name“ und dem Entfernen der Rückreferenzen ist in Abbildung 5.11 zu sehen.

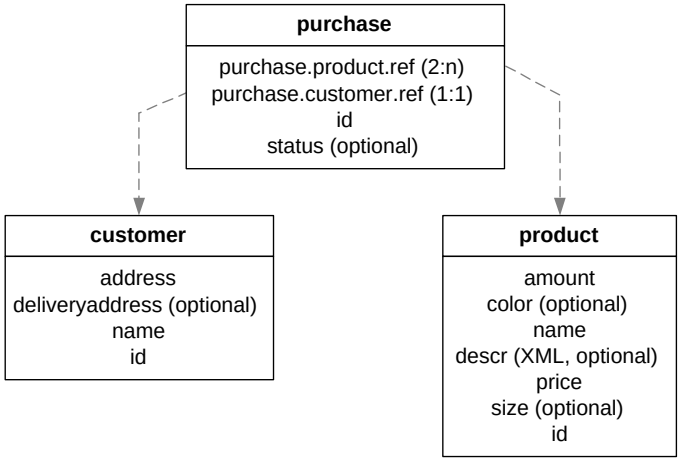


Abbildung 5.11: Modifiziertes Schema für die Matrix aus Abbildung 5.8

## 5.7 Speicherung des Schemas

Nachdem das Schema für die DTDs generiert und interaktiv optimiert wurde, kann es auf eine konkrete Datenbank umgesetzt werden. Diese Umsetzung wird in dieser Arbeit nicht betrachtet. Stattdessen wird im Folgenden eine datenbankunabhängige Speicherung des Schemas und des Mappings der DTDs auf das Schema vorgestellt. Das Mapping beschreibt, wie die Elemente und ihre Attribute auf die Klassen und ihre Eigenschaften abgebildet werden. Diese Zwischenrepräsentation kann später als Eingabe für die Generierung eines datenbankabhängigen Schemas und der notwendigen Import- und Export-Werkzeuge dienen.

Da sich diese Arbeit mit der Speicherung von XML-Daten beschäftigt, wurde auch für die Speicherung des Schemas XML als Datenformat gewählt. Eine ähnliche Form der Beschreibung von DTD-Schema-Mappings (wenn auch stärker auf eine konkrete Datenbank bezogen) findet sich in [BBB00, xml02].

Die Speicherung erfolgt in zwei Schritten. Zuerst wird das Schema gespeichert, anschließend für jede einzelne DTD das Mapping der Elemente und Attribute auf das Schema. Vor der Speicherung des Schemas werden automatisch die Rückreferenzen entfernt.

### 5.7.1 Speicherung des Schemas

Zur Speicherung des Schemas müssen drei verschiedene Informationen gespeichert werden, die Klassen, ihre Eigenschaften und die Vererbungshierarchie.

Für jede Klasse im generierten Schema wird ein Element namens „`class`“ generiert. Das Element besitzt ein Attribut „`name`“, welches als `ID` definiert ist. Der Name wird, soweit möglich, aus den zugeordneten Basisextensionen gebildet.

Die Vererbungshierarchie wird durch `superclass`-Elemente abgebildet. Sie werden für jede Subklassenbeziehung im Schema gebildet und jeweils unter dem Element der Subklasse angeordnet. Ein Attribut „`name`“ vom Typ `IDREF` verweist dabei auf das Element der jeweiligen Oberklasse.

In dem Element wird für jede Eigenschaft der Klasse ein eigenes Element gebildet. Unterschieden wird dabei zwischen vier verschiedenen Typen:

- einfache Eigenschaften („`property`“), welche aus den Attributen aus der DTD entstehen,
- Text („`text`“) zur Abbildung von `#PCDATA`,
- Text mit XML-Markup („`mixedcontent`“), welcher beispielsweise entsteht, wenn Mixed Content nicht aufgesplittet wird und
- Referenzen („`reference`“), welche zur Abbildung der Vater-Kind-Beziehungen gebildet werden.

Jedes dieser Elemente besitzt ein Attribut namens „`id`“ vom Typ `ID` zur eindeutigen Identifizierung. Die weiteren spezifischen Attribute sind in Abbildung 5.12 aufgeführt. Die vollständige DTD für die Speicherung des Schemas ist in Anhang B zu finden.

Das generierte `class`-Element für die Klasse „purchase“ aus dem Schema in Abbildung 5.9 ist in Beispiel 5.2 zu sehen. Für die Subklassenbeziehung wurde ein `superclass`-Element gebildet, welches auf die Oberklasse (als „id“ benannt) verweist. Für die Referenzen auf die Klassen „customer“ und „product“ wurden entsprechende Elemente erstellt. Die Eigenschaft „status“ ist optional und besitzt den Standardwert „accepted“.

**Beispiel 5.2.** - Generiertes `class`-Element für die Klasse „purchase“ aus dem Schema in Abbildung 5.9

```
1 <class name="purchase">
2     <superclass name="id"/>
3     <reference id="product.ref" min="2" max="n" toClass="product"/>
4     <reference id="customer.ref" min="1" max="1" toClass="customer"/>
5     <property id="status" attributes="status" optional="true"
6      default="accepted" values="accepted delivered"/>
7 </class>
```

Das vollständige generierte XML-Dokument, welches das Schema aus Abbildung 5.10 beschreibt, findet sich in Anhang A.

| Element | Attribut | Beschreibung |
|---|---|---|
| property | attributes | Auflistung der Attribute aus den DTDs, welche auf diese Eigenschaft abgebildet werden. Diese Information kann zur Namensgebung verwendet werden. |
| | optional | Ob null-Werte für diese Eigenschaft möglich sind. |
| | type | Ist der Typ für alle Attribute, welche auf diese Eigenschaft abgebildet werden, der gleiche, so wird dieser angegeben. Mögliche Werte sind `ID`, `IDREF`, `FIXED`, `LIST` und `text`. |
| | fixed | Wenn der Typ `FIXED` ist, dann wird hier der Wert, auf den die Eigenschaft fixiert ist, angegeben. |
| | default | Wenn der gleiche Standardwert für alle Attribute bestimmt werden kann, so wird dieser hier angegeben. |
| | values | Wenn der Typ `LIST` ist, dann werden hier die möglichen Werte angegeben. |
| text | optional | Ob null-Werte für diese Eigenschaft möglich sind. |
| mixedcontent | optional | Ob null-Werte für diese Eigenschaft möglich sind. |
| reference | min | Die minimale Kardinalität für diese Referenz. |
| | max | Die maximale Kardinalität für diese Referenz. |
| | toClass | Auf welche Klasse diese Referenz verweist. |

Abbildung 5.12: Attribute der Elemente zur Repräsentation der Eigenschaften der Klassen des generierten Schemas

### 5.7.2 Speicherung des Mappings

Für jede DTD, die in das Schema eingeflossen ist, wird beschrieben, wie die Elemente und Attribute auf die Klassen und Eigenschaften des Schemas abgebildet werden. Jede DTD wird hierbei durch einen zugeordneten Namensraum eindeutig identifiziert.

Unterhalb einer DTD werden die jeweiligen Elemente aufgeführt (jeweils ein Element namens „`element`“). Ein Element wird durch seinen Namen (Attribut „`name`“, innerhalb einer DTD eindeutig) bestimmt und kann eindeutig zu einer Klasse (Attribut „`toClass`“) zugeordnet werden.

Die Attribute werden durch „`attribute`“-Elemente abgebildet. Einfache Attribute werden direkt durch ein „`toProperty`“-Element einer Eigenschaft einer Klasse zugeordnet. Mehrwertige Attribute wurden während der Generierung des Schemas ausgelagert. Aus diesem Grund wird das Attribut durch ein „`toClass`“-Element auf eine Klasse abgebildet. Das zusätzliche „`toProperty`“-Element verweist in diesem Fall auf die Eigenschaft, welche zum Verknüpfen des Elementes aus der DTD mit den einzelnen Attributwerten genutzt wird.

Der Inhalt des Elementes kann auf zwei verschiedene Arten abgebildet werden. Wurde er während der Generation der GIM-Matrix nicht aufgesplittet, so wird er komplett

auf eine Eigenschaft abgebildet (gekennzeichnet durch ein Element „`toProperty`“). Wurde der Inhalt aufgesplittet, so wird für jedes Element des Inhaltes angegeben, welche Eigenschaft die Vater-Kind-Beziehung abbildet. Wie das Element selber auf das Schema gemappt wird, ist bei der Beschreibung des jeweiligen Elementes zu finden.

Schließlich muss noch das Mapping des Textinhaltes (`#PCDATA`, dargestellt durch ein Element „`textcontent`“) beschrieben werden. Auch hier gibt es zwei Möglichkeiten. Enthält das Element nur Text, so wird dieser auf eine Eigenschaft abgebildet (dargestellt durch ein Element „`toProperty`“). Enthält das Element jedoch Mixed Content, wurde der Textinhalt bei der Generation der GIM-Matrix ausgelagert. Um diese Abbildung zu beschreiben, wird wie bei den mehrwertigen Attributen die Klasse, auf die der Text abgebildet wird, und die Eigenschaft zum Verknüpfen des Elementes mit den Textfragmenten gekennzeichnet.

Das vollständige Mapping der DTD aus Beispiel 5.1 auf das Schema in Abbildung 5.10 ist in Anhang A ab Zeile 36 zu finden.

## 5.8 Implementation

Die in diesem Kapitel vorgestellte Schemageneration für XML-Daten wurde zur Validierung in der Programmiersprache Java implementiert[16].

Für eine DTD wird wie beschrieben der DTD-Graph gebildet. Es besteht die Möglichkeit, den generierten Graph anzuzeigen.

Bei der Konfliktbehandlung wird auf Elementebene die zweite Variante (alle Elemente gleichen Namens sind identisch) angewandt[17]. Auf Attributebene werden verschiedene Möglichkeiten angeboten:

- alle Attribute verschieden
- alle Attribute gleichen Namens, Typs und Standardwerts wie beschrieben zusammenfassen
- alle Attribute gleichen Namens und Typs wie beschrieben zusammenfassen
- alle Attribute gleichen Namens zusammenfassen

Eine Möglichkeit, die Konfliktbehandlung zu beeinflussen, besteht darin, die Elemente und Attribute vor der Generierung der Matrix manuell umzubenennen.

Die Matrixgeneration wird wie in Abschnitt 5.4.3 beschrieben durchgeführt. Die implementierten Modifikationen der Matrixgeneration sind beispielsweise das Zusammenfassen der ID-Attribute oder Mixed Content nicht aufzusplitten.

[16] Die Anwendung ist inclusive Quelltext unter „http://www.reidemeister.net/projects/xmlgim“ verfügbar.

[17] Abgesehen davon, dass bei der momentanen Implementation für eine DTD keine Konflikte auf Elementebene auftreten sollten. Sie können nur durch den Nutzer durch das manuelle Umbenennen eingeführt werden.

Die Schemageneration erfolgt mit dem in Abschnitt 4.4 vorgestellten Algorithmus. Das generierte Schema kann graphisch dargestellt werden. Die angebotenen Transformationen des Schemas umfassen das automatische Entfernen von allen Klassen ohne eigene Extension, das manuelle horizontale und vertikale Zusammenfassen von Klassen sowie das Entfernen von Mehrfachvererbung. Weiterhin wird eine Möglichkeit angeboten, durch das Einführen von null-Werten automatisch Klassen zusammenzufassen[18], um die Komplexität des Schemas zu reduzieren.

Das generierte (und wenn nötig modifizierte) Schema kann abschließend wie in Abschnitt 5.7 beschrieben gespeichert werden.

## 5.9 Zusammenfassung und Bewertung

In diesem Kapitel wurde ein Algorithmus zur Generierung eines integrierten Schemas für XML-Daten vorgestellt. Basierend auf einer DTD wird ein entsprechender Graph gebildet. Dieser Graph wird analysiert um eine GIM-Matrix aufzubauen. Mit Hilfe eines formal hinterlegten Algorithmus wird ein integriertes Schema erstellt. Durch verschiedene alternative Abbildungen der XML-Konstrukte und Transformationen zur Modifikation des Schemas ist es möglich, das Schema für einen konkreten Einsatzzweck zu optimieren.

Das Ergebnis der Schemageneration ist ein optimiertes Schema und eine Beschreibung, wie die XML-Konstrukte auf das Schema abgebildet werden.

### 5.9.1 Bewertung

In diesem Abschnitt wird eine Bewertung der vorgestellten Schemageneration in Bezug auf die eingangs definierten Ziele vorgenommen.

In das generierte Schema werden bei der vorgeschlagenen Standardabbildung fast alle vorhandenen Strukturinformationen übernommen. Verloren gehen einerseits alle Informationen, die nicht von der DTD in den DTD-Graphen übernommen werden, und andererseits zum Teil die Information, welchen Typ ein Attribut besitzt[19].

[18] Der verwendete Algorithmus wurde von Dr.-Ing. Ingo Schmitt entwickelt. Der Nutzer bestimmt ein maximales Verhältnis der optionalen Werte zu den normalen Werten in der GIM-Matrix. Der Algortihmus sucht jeweils die zwei Klassen, für die beim Zusammenfassen die wenigsten null-Werte in der GIM-Matrix benötigt werden. Wenn das Verhältnis der optionalen Werte zu den normalen Werten nach dem Zusammenfassen der beiden Klassen unter dem vom Nutzer vorgegeben Wert liegen würde, werden die beiden Klassen zusammengefasst und das Schema neu generiert. Anschließend werden erneut die beiden Klassen, für die beim Zusammenfassen die wenigsten null-Werte benötigt werden, gesucht. Dieser Zyklus wird solange wiederholt, bis das Verhältnis den vom Nutzer vorgegebenen Wert überschreiten würde.
Der Algorithmus wurde aus Performancegründen in der Form modifiziert, dass in einem Durchlauf alle Klassenpaare, für die beim Zusammenfassen die gleichen (minimalen) null-Werte in der GIM-Matrix benötigt werden, zusammengefasst werden.

[19] Diese Informationen können bei einem konkreten Import in eine Datenbank beispielsweise als Metadaten gespeichert werden.

Die Schemageneration ist ohne eine Interaktion mit dem Nutzer möglich. Ist allerdings eine alternative Abbildung oder eine Optimierung des Schemas gewünscht, so kann dies manuell gesteuert werden.

Das generierte Schema ist unabhängig von einem Datenbanksystem. Es kann für jede beliebige objektorientierte oder (objekt-)relationale Datenbank (wenn nötig mit einer entsprechenden Transformation) verwendet werden.

Im generierten Schema sind keine Metadaten enthalten[20]. Da die nötigen Metadaten stark von dem Zieldatenbanksystem und dem Einsatzzweck abhängen, wurden sie nicht weiter betrachtet.

Die durch den vorgestellten Algorithmus erstellten Schemata sind leicht verständlich. Durch die teilweise automatischen Transformationen, wie beispielsweise das Enfernen von Klassen ohne eigene Extension, kann leicht ein Schema generiert werden, das einem manuell erstellten entspricht.

Durch die verscheidenen alternativen Abbildungen der XML-Konstrukte auf die GIM-Matrix und die Transformationen für das generierte Schema kann das Schema für einen konkreten Einsatzzweck oder ein konkretes Zieldatenbanksystem optimiert werden.

### 5.9.2 Offene Probleme

Abschließend werden auszugsweise einige Probleme, die bei der Schemageneration nicht behandelt wurden, kurz diskutiert.

**Metadaten:** Um die Dokumente in einer Datenbank speichern zu können, sind gewisse Metadaten notwendig (mindestens der Pfads und der Dateiname und eine Zuordnung des *root*-Elementes), insbesondere dann, wenn eine (eindeutige) Rekonstruktion der XML-Dokumente gewünscht ist. Die Verwaltung der Metadaten wurde nicht betrachtet, da sie teilweise von der konkreten Zieldatenbank abhängt. Zu untersuchen wäre, ob die in [KKRSR00] vorgestellte Struktur adaptiert werden kann.

**Relative Ordnung:** Für gewisse XML-Dokumente ist die relative Ordnung der Elemente relevant. Auch die Speicherung dieser Informationen ist stark vom Zieldatenbanksystem abhängig und überlappt sich teilweise mit den anderen offenen Problemen. Denkbar wäre eine Verwaltung dieser Informationen als Metadaten oder durch eine sequenzielle Generierung der Schlüsselwerte entsprechend der Reihenfolge der Elemente.

**Entities:** Offen ist die Frage, ob und in welcher Form Entities gespeichert werden sollen. Bei internen Entities könnte man auf eine Speicherung verzichten und sie jeweils beim Auftreten in den Dokumenten expandieren (wodurch eine eindeutige Rekonstruktion der

[20] Man kann darüber diskutieren, ob die Zuordnungen der Elemente zu ihren Vaterelementen Metadaten sind. Da durch die Schachtelung für die Elemente ein Kontext definiert wird, wird die Vater-Kind-Beziehung in dieser Arbeit als relevante Information betrachtet.

Dokumente verhindert wird). Problematisch sind externe Enities, da sie auf beliebige Ressourcen verweisen können. Mögliche Varianten zur Speicherung wären beispielsweise

- die Speicherung des System- und des Public-Identifiers,
- die Speicherung als Verweise auf die externen Ressourcen[21] oder
- die Einbettung der externen Ressourcen (zum Beispiel als *Binary Large Object* in einer relationalen Datenbank).

**Schlüsselfindung / -generation:** Abhängig von der gewählten Zieldatenbank[22] kann es notwendig sein, Schlüssel für die Elemente zu bilden. Diese Schlüssel können entweder aus den Attributen und dem Inhalt des Elementes zusammengesetzt oder während des Imports in die Datenbank generiert werden.

**Identifikation von Duplikaten:** Ein interessantes Problem ist die Identifizierung von redundanten Fragmenten in XML-Dokumenten während des Datenbankimports. Wenn Duplikate identifiziert werden können und auf ein Objekt in der Datenbank abgebildet werden sollen, dann sind möglicherweise erweiterte Metadaten notwendig, um die Struktur der Dokumente adäquat abzubilden.

**Verweise in XML:** Zum Verknüpfen von XML-Dokumenten sind beispielsweise XLink und XPointer vorgesehen. Es wäre zu untersuchen, ob diese Verweise in einer Datenbank abgebildet werden können.

[21] Sofern das Zieldatenbanksystem eine entsprechende Unterstützung anbietet.

[22] Bei einer Objektdatenbank beispielsweise sind keine Schlüssel notwendig, da jedes Objekt bereits eine interne ID besitzt. Anders verhält es sich bei einer relationalen Datenbank, da hier explizite Schlüssel unter anderem zum Abbilden von Fremdschlüsselbeziehungen benötigt werden.

# Kapitel 6

# Zusammenfassung

Nach einer Einleitung, welche die Notwendigkeit der Integration von XML und Datenbanksystemen verdeutlicht, wird im zweiten Kapitel ein einführender Überblick über XML gegeben. Ausführlich werden der Aufbau von XML-Dokumenten und der Beschreibungsmechanismus DTD untersucht. Weiterhin erfolgt eine kurze Vorstellung von Erweiterungen und Technologien rund um XML.

Im dritten Kapitel werden einleitend die Möglichkeiten, XML-Dokumente in einer Datenbank zu speichern, untersucht. Dabei werden anfangs die verschiedenen Datenbanktechnologien einerseits auf ihre Eignung und andererseits auf ihre praktische Relevanz hin untersucht. Aufgrund ihrer Dominanz stehen in den weiteren Betrachtungen die (objekt-)relationalen Datenbanksysteme im Vordergrund. Für die beiden in der Literatur hauptsächlich untersuchten Ansätze zur Speicherung von XML-Daten (als Graph versus in einem der Struktur entsprechendem Schema) wird jeweils stellvertretend eine Arbeit ausführlich vorgestellt. Abschließend werden alternative Methoden oder Erweiterungen betrachtet.

Das vierte Kapitel gibt einen einführenden Überblick über das Generische Integrationsmodell. Durch die Anwendung des vorgestellten Algorithmus kann die Generierung eines integrierten Schemas für mehrere Eingangsschemata automatisiert werden. Zusätzlich werden verschiedene manuelle Transformationen des generierten Schemas präsentiert, welche eine zielgerichtete Optimierung des Schemas ermöglichen.

Der eigene Ansatz zur Generierung eines adäquaten Schemas zur Speicherung von XML-Dokumenten unter der Verwendung des Generischen Integrationsmodells wird im fünften Kapitel vorgestellt. Durch verschiedene alternative Abbildungen der XML-Konstrukte und die im vierten Kapitel eingeführten Transformationen kann das generierte Schema auf verschiedene Ziele hin optimiert werden. Abschließend wird eine datenbankunabhängige Speicherung des Schemas und der Beschreibung, wie die XML-Konstrukte auf dieses Schema abgebildet werden, präsentiert.

Während der Arbeit hat sich aufgrund der anfangs nicht absehbaren Komplexität des Themas der Fokus auf die Schemageneration für XML-Dokumente verlagert. Die ursprünglich geplante Entwicklung eines Import- und Exportwerkzeuges wurde

aus Zeitgründen nicht durchgeführt. In diesem Zusammenhang wird auf XML-DBMS ([BBB00, xml02]) verwiesen, welches einen Datenbankimport und -export von XML-Dokumenten für relationale Datenbanken basierend auf einer ähnlichen Beschreibung des Schemas und des Mappings wie in Abschnitt 5.7 vorgestellt erlaubt. Durch eine geeignete Transformation der eigenen Schemabeschreibung in die von XML-DBMS (beispielsweise mittels XSLT) sollte eine Nutzung von XML-DBMS für die konkrete Speicherung der XML-Dokumente in einer Datenbank möglich sein.

# Anhang A

# Gespeichertes Schema

Im Folgenden wird das komplette XML-Dokument zur Speicherung des Schemas aus Abbildung 5.10 und das Mapping der DTD aus Beispiel 5.1 auf dieses Schema aufgeführt. Die entsprechende DTD findet sich in Anhang B.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xmlgim>
3      <schema>
4          <class name="PCDATA_elements">
5              <text id="text" optional="false"/>
6          </class>
7          <class name="customer">
8              <reference id="address.ref" min="1" max="1" toClass="PCDATA_elements"/>
9              <reference id="deliveryaddress.ref" min="0" max="1"
10              toClass="PCDATA_elements"/>
11             <reference id="name.ref" min="1" max="1" toClass="PCDATA_elements"/>
12             <property attributes="id" id="customerID" optional="false" type="ID"/>
13         </class>
14         <class name="descr">
15             <reference id="PCDATA.ref" min="0" max="n" toClass="PCDATA_elements"/>
16             <reference id="em.ref" min="0" max="n" toClass="PCDATA_elements"/>
17             <reference id="link.ref" min="0" max="n" toClass="PCDATA_elements"/>
18         </class>
19         <class name="product">
20             <reference id="amount.ref" min="1" max="1" toClass="PCDATA_elements"/>
21             <reference id="color.ref" min="0" max="1" toClass="PCDATA_elements"/>
22             <reference id="descr.ref" min="0" max="1" toClass="descr"/>
23             <reference id="name.ref" min="1" max="1" toClass="PCDATA_elements"/>
24             <reference id="price.ref" min="1" max="1" toClass="PCDATA_elements"/>
25             <reference id="size.ref" min="0" max="1" toClass="PCDATA_elements"/>
26             <property attributes="id" id="productID" optional="false"/>
27         </class>
28         <class name="purchase">
```

```
29          <reference id="customer.ref1" min="1" max="1" toClass="customer"/>
30          <reference id="product.ref1" min="2" max="n" toClass="product"/>
31          <property attributes="id" id="purchaseID" optional="false" type="ID"/>
32          <property attributes="status" default="accepted" id="status"
33           optional="true" values="accepted delivered"/>
34      </class>
35  </schema>
36  <dtd namespace="www.reidemeister.net/projects/xmlgim/purchase">
37      <element name="name" toClass="PCDATA_elements">
38          <content>
39              <textcontent>
40                  <toProperty class="PCDATA_elements" property="text"/>
41              </textcontent>
42          </content>
43      </element>
44      <element name="em" toClass="PCDATA_elements">
45          <content>
46              <textcontent>
47                  <toProperty class="PCDATA_elements" property="text"/>
48              </textcontent>
49          </content>
50      </element>
51      <element name="purchase" toClass="purchase">
52          <attribute name="status">
53              <toProperty class="purchase" property="status"/>
54          </attribute>
55          <attribute name="id">
56              <toProperty class="purchase" property="purchaseID"/>
57          </attribute>
58          <content>
59              <element name="customer">
60                  <toProperty class="purchase" property="customer.ref1"/>
61              </element>
62              <element name="product">
63                  <toProperty class="purchase" property="product.ref1"/>
64              </element>
65          </content>
66      </element>
67      <element name="deliveryaddress" toClass="PCDATA_elements">
68          <content>
69              <textcontent>
70                  <toProperty class="PCDATA_elements" property="text"/>
71              </textcontent>
72          </content>
73      </element>
74      <element name="color" toClass="PCDATA_elements">
```

```
75              <content>
76                  <textcontent>
77                      <toProperty class="PCDATA_elements" property="text"/>
78                  </textcontent>
79              </content>
80          </element>
81          <element name="amount" toClass="PCDATA_elements">
82              <content>
83                  <textcontent>
84                      <toProperty class="PCDATA_elements" property="text"/>
85                  </textcontent>
86              </content>
87          </element>
88          <element name="price" toClass="PCDATA_elements">
89              <content>
90                  <textcontent>
91                      <toProperty class="PCDATA_elements" property="text"/>
92                  </textcontent>
93              </content>
94          </element>
95          <element name="descr" toClass="descr">
96              <content>
97                  <element name="em">
98                      <toProperty class="descr" property="em.ref"/>
99                  </element>
100                 <element name="link">
101                     <toProperty class="descr" property="link.ref"/>
102                 </element>
103                 <textcontent>
104                     <toClass class="PCDATA_elements"/>
105                     <toProperty class="descr" property="PCDATA.ref"/>
106                 </textcontent>
107             </content>
108         </element>
109         <element name="customer" toClass="customer">
110             <attribute name="id">
111                 <toProperty class="customer" property="customerID"/>
112             </attribute>
113             <content>
114                 <element name="address">
115                     <toProperty class="customer" property="address.ref"/>
116                 </element>
117                 <element name="deliveryaddress">
118                     <toProperty class="customer" property="deliveryaddress.ref"/>
119                 </element>
120                 <element name="name">
```

```
121                 <toProperty class="customer" property="name.ref"/>
122             </element>
123         </content>
124     </element>
125     <element name="link" toClass="PCDATA_elements">
126         <content>
127             <textcontent>
128                 <toProperty class="PCDATA_elements" property="text"/>
129             </textcontent>
130         </content>
131     </element>
132     <element name="address" toClass="PCDATA_elements">
133         <content>
134             <textcontent>
135                 <toProperty class="PCDATA_elements" property="text"/>
136             </textcontent>
137         </content>
138     </element>
139     <element name="product" toClass="product">
140         <attribute name="id">
141             <toProperty class="product" property="productID"/>
142         </attribute>
143         <content>
144             <element name="amount">
145                 <toProperty class="product" property="amount.ref"/>
146             </element>
147             <element name="color">
148                 <toProperty class="product" property="color.ref"/>
149             </element>
150             <element name="descr">
151                 <toProperty class="product" property="descr.ref"/>
152             </element>
153             <element name="name">
154                 <toProperty class="product" property="name.ref"/>
155             </element>
156             <element name="price">
157                 <toProperty class="product" property="price.ref"/>
158             </element>
159             <element name="size">
160                 <toProperty class="product" property="size.ref"/>
161             </element>
162         </content>
163     </element>
164     <element name="size" toClass="PCDATA_elements">
165         <content>
166             <textcontent>
```

```
167                     <toProperty class="PCDATA_elements" property="text"/>
168                 </textcontent>
169             </content>
170         </element>
171     </dtd>
172 </xmlgim>
```

# Anhang B

# DTD für die Speicherung des Schema und des Mappings

Die folgende DTD definiert eine Dokumentklasse zur Beschreibung des generierten Schemas und des Mappings von DTDs auf das Schema (vgl. Abschnitt 5.7). Ein Dokument, welches dieser DTD entspricht, ist in Anhang A aufgeführt.

```
1  <!--
2    Repräsentiert ein DTD-Schema-Mapping. Es können mehrere DTDs aufgeführt werden.
3  -->
4  <!ELEMENT xmlgim (schema, dtd+)>
5  <!ATTLIST xmlgim version CDATA "1.0">
6
7  <!--
8    Das Schema besteht aus einer Menge von Klassen.
9  -->
10 <!ELEMENT schema (class+)>
11
12 <!--
13   Eine Klasse wird eindeutig durch ihren Namen bestimmt. Sie kann mehrere
14   Superklassen und Eigenschaften haben. Es gibt vier mögliche Eigenschaften:
15     properties - generiert für Attribute
16     text - generiert für #PCDATA
17     mixedcontent - generiert für Mixed Content, wenn diese nicht aufgesplittet
18                    wird
19     reference - generiert um die Vater-Kind-Beziehungen der Elemente abzubilden
20   Jede Eigenschaft besitzt eine eindeutige id. Außer den Referenzen besitzt
21   jede Eigenschaft ein Attribut "optional", welches angibt, ob NULL-Werte
22   möglich sind.
23 -->
24 <!ELEMENT class (superclass*, (property | text | mixedcontent | reference)*)>
25 <!ATTLIST class name ID #REQUIRED>
26
```

```
<!--
  Eine Subklassenbeziehung im Schema wird durch ein superclass-Element darge-
  stellt. Es wird der Subklasse zugeordnet und verweist auf die Oberklasse.
-->
<!ELEMENT superclass EMPTY>
<!ATTLIST superclass name IDREF #REQUIRED>

<!--
  Eine einfache Eigenschaft wird für die Attribute der Elemente aus den DTDs
  gebildet. Das Attribut "attributes" enthält eine Liste der Attributnamen,
  die auf diese Eigenschaft abgebildet werden. Diese Information kann zur
  Namensgebung verwendet werden. Wenn alle Attribute den gleichen Typ besitzten,
  so wird dieser angegeben. Gleiches gilt für die Attribute "fixed", welches
  einen festgelegten Wert für diese Eigenschaft festlegt, "default", welches
  einen Standardwert enthält, und "values", welches eine Werteliste enthalten
  kann.
-->
<!ELEMENT property EMPTY>
<!ATTLIST property id         ID                                  #REQUIRED
                   attributes NMTOKENS                            #REQUIRED
                   optional   (true | false)                      #REQUIRED
                   type       (ID | IDREF | FIXED | LIST | text) "text"
                   fixed      CDATA                               #IMPLIED
                   default    CDATA                               #IMPLIED
                   values     NMTOKENS                            #IMPLIED>

<!--
  Jedes Auftreten von #PCDATA in den DTDs wird auf Text abgebildet. Die
  Eigenschaft der Klasse muss beliebig großen Text aufnehmen können.
-->
<!ELEMENT text EMPTY>
<!ATTLIST text id       ID              #REQUIRED
               optional (true | false) #REQUIRED>

<!--
  Mixed content entspricht prinzipiell Text, kann jedoch zusätzlich
  XML-Markup enthalten.
-->
<!ELEMENT mixedcontent EMPTY>
<!ATTLIST mixedcontent id       ID              #REQUIRED
                       optional (true | false) #REQUIRED>

<!--
  Referenzen werden zur Abbildung der Vater-Kind-Beziehungen in den DTDs
  verwendet. Sie verweisen auf eine Klasse. Die Kardinalitäten der Referenzen
  werden durch die Attribute "min" und "max" bestimmt.
```

```
73 -->
74 <!ELEMENT reference EMPTY>
75 <!ATTLIST reference id       ID    #REQUIRED
76                      min     CDATA #REQUIRED
77                      max     CDATA #REQUIRED
78                      toClass IDREF #REQUIRED>
79
80 <!--
81   Eine DTD besteht aus mehreren Elementen. Über einen zugeordneten Namensraum
82   wird die DTD eindeutig identifiziert.
83 -->
84 <!ELEMENT dtd (element+)>
85 <!ATTLIST dtd namespace ID #REQUIRED>
86
87 <!--
88   Elemente können an zwei Stellen erscheinen:
89    - als Kind von "dtd":
90      Das Element repräsentiert ein Element aus der DTD. Es kann Attribute
91      und Inhalt besitzten. Das Attribut "toClass" gibt die extensionale
92      Zuordnung an.
93    - als Kind von "content":
94      Innerhalb des Elementinhaltes wird beschrieben, wie die Vater-Kind-
95      Beziehung zwischen den Elementen abgebildet wird.
96   Jedes Element wird durch den Namen identifiziert.
97 -->
98 <!ELEMENT element ((attribute*, content?) | toProperty)>
99 <!ATTLIST element name    NMTOKEN #REQUIRED
100                  toClass IDREF   #IMPLIED>
101
102 <!--
103   Einwertige Attribute werden auf eine Eigenschaft abgebildet.
104   Mehrwertige Attribute werden auf eine Klasse ("toClass") abgebildet und die
105   Referenz zum Verknüpfen des Elementes mit den Attributwerten ("toProperty")
106   angegeben.
107 -->
108 <!ELEMENT attribute (toClass?, toProperty)>
109 <!ATTLIST attribute name NMTOKEN #REQUIRED>
110
111 <!--
112   Wenn der Elementinhalt nicht aufgesplittet wird, so wird er auf eine Eigen-
113   schaft abgebildet.
114   Andernfalls besteht der Inhalt aus einer Menge von Elementen und evt. Text
115   (#PCDATA). Bei Elementen wird angegeben, auf welche Referenz die Vater-Kind-
116   Beziehung abgebildet wird.
117 -->
118 <!ELEMENT content (toProperty | (element*, textcontent?))>
```

```
119
120 <!--
121   Text innerhalb von Elementen (#PCDATA) kann auf eine Klasse oder eine Eigen-
122   schaft abgebildet werden. Im ersten Fall wird die Klasse, auf die der Text ab-
123   gebildet wird ("toClass") und die Referenz zum Verknüpfen des Elementes mit
124   den einzelnen Textfragmenten ("toProperty") angegeben. Im zweiten Fall
125   wird direkt die Eigenschaft zum Aufnehmen des Textes ("toProperty") aufgeführt.
126 -->
127 <!ELEMENT textcontent (toClass?, toProperty)>
128
129 <!--
130   Beschreibt einen Verweis auf eine Eigenschaft. Die Eigenschaft ist durch die
131   Angabe der Klasse ("class") und der Eigenschaft ("property") eindeutig
132   identifiziert.
133 -->
134 <!ELEMENT toProperty EMPTY>
135 <!ATTLIST toProperty class    IDREF #REQUIRED
136                      property IDREF #REQUIRED>
137
138 <!--
139   Beschreibt einen Verweis auf eine Klasse. Die Eigenschaft ("property") wird
140   nur bei mehrwertigen Attributen verwendet.
141 -->
142 <!ELEMENT toClass EMPTY>
143 <!ATTLIST toClass class    IDREF #REQUIRED
144                   property IDREF #IMPLIED>
```

# Literaturverzeichnis

[AQM+97] Abiteboul, S.; Quass, D.; McHugh, J.; Widom, J.; Wiener, J.: The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, Band 1, Nr. 1, S. 68–88, 1997-04.

[BA96] Böhm, K.; Aberer, K.: HyperStorM - Administering Structured Documents Using Object-Oriented Database Technology. In *Proceedings of the ACM SIGMOD Conference*. Montreal, Kanada, 1996-06.

[BBB00] Bourret, R.; Bornhövd, C.; Buchmann, A. P.: A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Databases. In *2nd Int. Workshop on Advanced Issues of EC and Web-based Information Systems (WECWIS)*. San Jose, Kalifornien, 2000-06.

[BDHS96] Buneman, P.; Davidson, S.; Hillebrand, G.; Suciu, D.: A Query Language and Optimization Techniques for Unstructured Data. In *Proceedings of the ACM SIGMOD Conference*, S. 505–516. Montreal, Kanada, 1996-06.

[Bee93] Beeri, C.: Some Thought on the Future Evolution for Object-Oriented Database Concepts. In W.Stucky; Oberweis, A. (Hrsg.): *Proc. der GI Fachtagung 'Datenbanksysteme in Büro, Technik und Wissenschaft' (BTW'93)*, S. 18–23. Springer Verlag, Berlin, 1993.

[BFM] Bray, T.; Frankston, C.; Malhontra, A.: *Document Content Description for XML*. http://www.w3.org/TR/NOTE-dcd.

[BM98] Behme, H.; Mintert, S.: *XML in der Praxis - Professionelles Web-Publishing mit der Extensible Markup Language*. Addison Wesley, 1998.

[Bou00] Bourret, R.: *XML and Databases*. Technische Universität von Darmstadt, 2000-06. http://www.rpbourret.com/xml/XMLAndDatabases.htm.

[Bra98] Bradley, N.: *The XML Companion*. Addison Wesley, Harlow, England, 1998.

[CFP00] Cher, S.; Fraternali, P.; Paraboschi, S.: XML: Current Developements and Future Challenges for the Database Community. In *Proceedings of the*

*7th Int. Conference on Extending Database Technology (EDBT)*. Konstanz, 2000-03.

[CJS99] Cluet, S.; Jacqmin, S.; Simeon, J.: The new YATL: Design and Specifications. Technischer Bericht, INRIA, 1999.

[Cla97] Clark, J.: *Comparison of SGML and XML*, 1997-12-15. http://www.w3.org/TR/NOTE-sgml-xml-971215.

[Cod82] Codd, E. F.: Relational Database: A Practical Foundation for Productivity. *Communications of the ACM*, Band 25, Nr. 2, S. 109–117, 1982-02.

[Con97] Conrad, S.: *Förderierte Datenbanksysteme: Konzepte der Datenintegration*. Springer Verlag, Heidelberg/Berlin, 1997.

[CRF00] Chamberlin, D.; Robie, J.; Florescu, D.: Quilt: A XML query language for hetrogeneous data sources. In *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[DFLS] Deutsch, A.; Fernandez, M.; Levy, A.; Suciu, D.: *XML-QL: A Query Language for XML*. http://www.w3.org/TR/NOTE-xml-ql.

[DFS99] Deutsch, A.; Fernandez, M.; Suciu, D.: Storing Semistructured Data with STORED. In *Proceedings of the ACM SIGMOD Conference*, S. 431–442. Philadelphia, Pensylvania, 1999-05. http://citeseer.nj.nec.com/article/deutsch98storing.html.

[Duq87] Duquenne, V.: Conceptual implications between attributes and some properties of finite lattices. In Ganter, B.; Wille, R. (Hrsg.): *Beiträge zur Begriffsanalyse*, S. 213–239. B. I.-Wissenschaftsverlag, Mannheim, 1987.

[FK99] Florescu, D.; Kossmann, D.: Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, Band 22, Nr. 3, S. 27–34, 1999.

[FLM99] Florescu, D.; Levy, A.; Mendelzon, A.: Database Techniques for the World Wide Web: A Survey. *ACM SIGMOD Record*, Band 27, Nr. 3, 1999-09.

[GMW99] Goldman, R.; McHugh, J.; Widom, J.: From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *Proceedings of the 2nd Int. Workshop on the Web and Databases (WebDB)*. Philadelphia, Pensylvania, 1999-06.

[Hol01] Holman, G. K.: *Practical Transformation Using XSLT and XPath*. Crane Softwrights Ltd., Kars, Ontario, Kanada, 2001. http://www.cranesoftwrights.com/training/.

[ISO86] ISO 8879:1986: *Information Processing - Text and Office System - Standard Generalized Markup Language (SGML)*, 1986-10-15.

[KKRSR00] Kappel, G.; Kapsammer, E.; Rausch-Schott, S.; Retschitzegger, W.: X-Ray - Towards Integrating XML and Relational Database Systems. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, S. 339–353, 2000. http://citeseer.nj.nec.com/kappel00xray.html.

[KL02] Kleiner, C.; Lipeck, U. W.: Automatische Erzeugung von XML-DTDs aus konzeptionellen Datenbankschemata. *Datenbank-Spektrum*, Nr. 2, S. 14–22, 2002.

[KM00] Kanne, C.-C.; Moerkotte, G.: Efficient Storage of XML Data. In *ICDE*, S. 198, 2000. http://citeseer.nj.nec.com/kanne99efficient.html.

[KM99] Klettke, M.; Meyer, H.: Managing XML documents in object-relational databases, 1999. http://citeseer.nj.nec.com/article/klettke99managing.html.

[LC00] Lee, D.; Chu, W.: Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, S. 323–338, 2000. http://citeseer.nj.nec.com/lee00constraintspreserving.html.

[Obj99] Object Design, Inc.: *eXcelon - XML Data Server*, 1999. http://www.odi.com/excelon.

[Por99] Porst, B.: Untersuchungen zu Datentyperweiterungen für XML-Dokumente und ihre Anfragemethoden am Beispiel von DB2 und Informix. Master's thesis, Universität Rostock, 1999.

[RLS98] Robie, J.; Lapp, J.; Schach, D.: *XML Query Language (XQL)*, 1998. http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[Sch98] Schmitt, I.: *Schemaintegration für den Entwurf Föderierter Datenbanken*, Dissertationen zu Datenbanken und Informationssystemen, Band 43. infix-Verlag, Sankt Augustin, 1998.

[Se93] Shoens, K.; et al.: The Rufus system: Information organization for semi-structured data. In *Proceedings of the Int. Conference on VLDB*. Dublin, Ireland, 1993.

[SL90] Seth, A. P.; Larson, J. A.: Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases. *ACM Computing Surveys*, Band 22, Nr. 3, S. 183–236, 1990.

[SSB+00] Shanmugasundaram, J.; Shekita, E. J.; Barr, R.; Carey, M. J.; Lindsay, B. G.; Pirahesh, H.; Reinwald, B.: Efficiently Publishing Relational Data as XML Documents. In *The VLDB Journal*, S. 65–76, 2000. http://citeseer.nj.nec.com/shanmugasundaram00efficiently.html.

[STZ+99] Shanmugasundaram, J.; Tufte, K.; Zhang, C.; He, G.; DeWitt, D. J.; Naughton, J. F.: Relational Databases for Querying XML Documents: Limitations and Opportunities. In *The VLDB Journal*, S. 302–314, 1999. http://citeseer.nj.nec.com/295071.html.

[SW00] Schöning, H.; Wäsch, J.: Tamino - An Internet Database System. In *Proceedings of the 7th Int. Conference on Extending Database Technology (EDBT)*. Konstanz, 2000-03.

[SYU99] Shimura, T.; Yoshikawa, M.; Uemura, S.: Storage and Retrieval of XML Documents using Object-Relational Databases. In *Proc. of the 10th International Conference on Database and Expert Systems Applications (DEXA'99)*, S. 206–217. Springer-Verlag, 1999-08.

[TDCZ00] Tian, F.; DeWitt, D.; Chen, J.; Zhang, C.: The design and performance evaluation of alternative XML storage strategies, 2000. http://citeseer.nj.nec.com/tian00design.html.

[Wid99] Widom, J.: Data Management for XML - Research Directions. *IEEE Data Engineering Bulletin, Special Issue on XML*, Band 22, Nr. 3, 1999-09.

[Wil92] Wille, R.: Concept lattices and conceptual knowledge systems. *Computer & Mathematics with Applications*, Band 23, Nr. 6-9, S. 493–515, 1992.

[Wor01a] World Wide Web Consortium ($W_3C$): *Document Object Model (DOM)*, 2001. http://www.w3.org/DOM/.

[Wor01b] World Wide Web Consortium ($W_3C$): *XML Pointer Language (XPointer), Version 1.0 (Last Call Working Draft)*, 2001-01-08. http://www.w3.org/TR/2001/WD-xptr-20010108.

[Wor01c] World Wide Web Consortium ($W_3C$): *XML Schema Part 0: Primer*, 2001-05-02. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

[Wor01d] World Wide Web Consortium ($W_3C$): *XML Schema Part 1: Structures*, 2001-05-02. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/.

[Wor01e] World Wide Web Consortium ($W_3C$): *XML Schema Part 2: Datatypes*, 2001-05-02. http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.

[Wor01f] World Wide Web Consortium ($W_3C$): *XML Linking Language (XLink), Version 1.0*, 2001-06-27. http://www.w3.org/TR/2001/REC-xlink-20010627/.

[Wor01g] World Wide Web Consortium ($W_3C$): *The Extensible Stylesheet Language (XSL)*, 2001-11-16. http://www.w3.org/Style/XSL/.

[Wor98] World Wide Web Consortium ($W_3C$): *Extensible Markup Language (XML), Version 1.0*, 1998-02-10. http://www.w3c.org/TR/1998/REC-xml-19980210/.

[Wor99a] World Wide Web Consortium ($W_3C$): *Namespaces in XML*, 1999-01-14. http://www.w3.org/TR/1999/REC-xml-names-19990114/.

[Wor99b] World Wide Web Consortium ($W_3C$): *XML Path Language (XPath), Version 1.0*, 1999-11-16. http://www.w3.org/TR/1999/REC-xpath-19991116.

[Wor99c] World Wide Web Consortium ($W_3C$): *XSL Transformations (XSLT), Version 1.0*, 1999-11-16. http://www.w3.org/TR/1999/REC-xslt-19991116.

[xml02] *XML-DBMS - Middleware for Transferring Data between XML Documents and Relational Databases*, 2002-03-12. http://www.rpbourret.com/xmldbms/index.htm.

[YASU01] Yoshikawa, M.; Amagasa, T.; Shimura, T.; Uemura, S.: XRel: A Path-Based Approach to Storage and Retrieval of XML Documents using Relational Databases. *ACM Transactions on Internet Technology*, Band 1, Nr. 1, S. 110–141, 2001-08.

# Selbständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 24. März 2002

Jan Reidemeister