Änderungsoperationen in XML-Anfragesprachen

Diplomarbeit

Universität Rostock, Fachbereich Informatik

vorgelegt von: Birger Hänsel
geboren am: 03.07.1975 in Waren (Müritz)

Erstgutachter: Prof. Andreas Heuer
Zweitgutachter: Prof. Clemens Cap
Betreuer: Dr. Holger Meyer
Dr. Meike Klettke

Abgabedatum: 30.06.2002

Zusammenfassung

Mit dem Sprachvorschlag XML Query (kurz *XQuery*) des W3C gibt es einen hoffnungsvollen Kandidaten für eine Standard-XML-Anfragesprache. Jedoch wird die erste Version keine Möglichkeiten für Update-Operationen vorsehen. Heutige XML-Anwendungen realisieren Änderungen in der Regel mittels der DOM-API. Dies ist jedoch ein klassisches, prozedurales Vorgehen. Datenbankkonzepte, wie Transaktionen, Recovery und logische und physische Datenunabhängigkeit finden in diesen low-level-Schnittstellen keine Beachtung. Eine „echte“ Update-Sprache sollte dies aber sehr wohl beinhalten.

In dieser Diplomarbeit wurde zunächst ein Anforderungskatalog für eine XML-Update-Sprache aufgestellt. Als nächstes wurden existierende Ansätze untersucht und in Bezug auf diese Anforderungen verglichen. Weiterhin war es unverzichtbar Grundlagen für XML Änderungen zu definieren. Danach wurde ein Datenmodell und eine Menge von Grundoperationen für Änderungen aufgestellt, wobei auch auf potentielle Probleme hingewiesen wurde. Der nächste Schritt war die Integration der Änderungsoperationen in die Syntax der XML Query Language, die als Resultat die hier als *SUNFLWR* benannten Ausdrücke lieferte. Abschließend wurde gezeigt wie solche *SUNFLWR*-Ausdrücke auf ein relationales Datenbanksystem abgebildet werden können.

Abstract

The W3C is in process of developing a standard for a XML query language, called XQuery. However there will be no support for update operations in the first version. Today XML applications normally implement modifications in use of the DOM-API. Because this is a classic procedural method, database concepts like transactions, recovery as well as logical and physical data independence are not in observance at these low level interfaces. However a *real* Update Language should contain these concepts.

In this master thesis we started with defining requirements for a XML Update Language. Next we examined existing approaches and compared them with regard to this requirements. Furthermore it was indispensable to establish a basis of XML update technics. After that we determine a data model and a set of basic update operations whereby we indicate on potential problems. The next step was the integration of the update operations into the syntax of the XML Query Language which results in expressions called *SUNFLWR's*. Finally, we show how to map these *SUNFLWR* expressions to a relational database system.

CR-Klassifikation

E.1.	DATA STRUCTURES
E.2	DATA STORAGE REPRESENTATIONS
H.2	DATABASE MANAGEMENT
H.2.1	Logical Design
H.2.3	Languages
H.2.4	Systems

Key Words

XML, Query Language, Update Language, Data Model, Database

Inhaltsverzeichnis

1 Einleitung

Es sind nun schon mehr als fünf Jahre vergangen, seitdem das W3 Konsortium *www.w3c.org* den ersten Vorschlag für die Metasprache XML veröffentlicht hat. Inzwischen hat sich eine unüberschaubare Menge an Technologien und Anwendungen entwickelt, die auf der Extensible Markup Language basieren.
Als Einstiegspunkt soll eine kurze Übersicht zur Historie der Aktivitäten zu XML innerhalb des W3C gegeben werden. Die bisherigen Entwicklungen lassen sich in vier Phasen einteilen, deren zeitlicher Verlauf in der Abbildung 1 veranschaulicht wird.

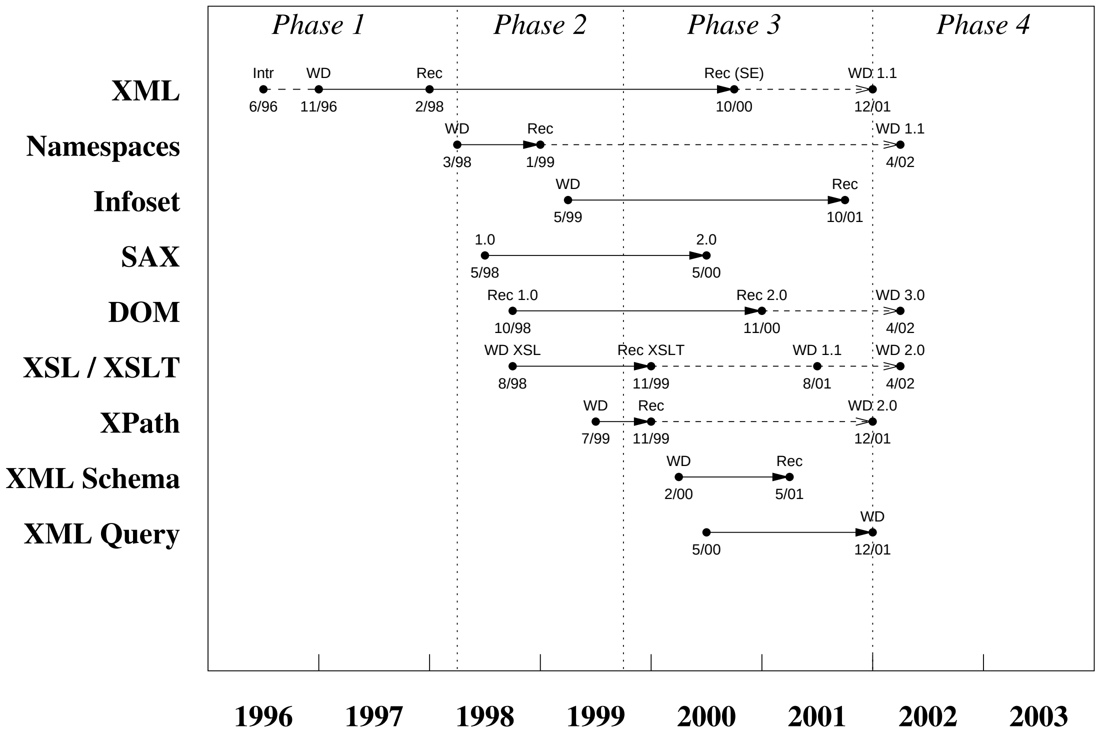

Abbildung 1: XML Zeitschiene

In der ersten Phase stand die Entwicklung des XML-Kerns zur Aufgabe, die durch die Empfehlung von XML 1.0 abgeschlossen wurde.
In Phase 2 wurden die ersten "Handwerkzeuge" für die Bearbeitung von XML vorgeschlagen, darunter die Schnittstellen *SAX* und *DOM* wie auch die Stylesheet-Sprache *XSL*. Für den XML-Kern waren hier die Einführung von Namensräumen und eines abstrakten Datenmodell für XML (Infoset) wichtige Grundlagen für die weitere Entwicklung.
Die erst vor kurzem abgeschlossene 3. Phase war eine regelrechte Fabrik für neue Arbeitsfelder und Arbeitspapiere. Hier entstanden die XML-Schema-Beschreibung und die Navigationssprache XPath 1.0. Als größte Herausforderung kann die Entwicklung der XML-Query Language (kurz XQuery) gesehen werden, die viele Konzepte und

Technologien in sich vereinen soll.
Durch die Masse an Neuentwicklungen ist das W3C zu dem Entschluss gekommen, die aktuelle Phase 4 zur Konsolidierung zu nutzen. Dazu sollen aktuelle Arbeiten abgeschlossen werden und vor allem die unterschiedlichen Technologien aufeinander angepasst werden, da zur Zeit noch viele Inkonsistenzen bestehen.

Dies ist auch der mutmaßliche Grund, warum Updates in der ersten Version der XQuery-Recommendation fehlen werden. Hier sollte erst die Stabilisierung des Anfrageanteils erfolgen, bevor man sich mit neuen Problemen auseinander setzen muss.

1.1 Ziel und Aufbau dieser Arbeit

Zum derzeitigen Stand existieren verschiedene Ansätze für XML-Anfragesprachen wie auch zugehörige Implementationen. XQuery wird daraus wahrscheinlich als *SQL4XML* hervorgehen, wie jedoch gesagt, ohne Update-Operationen. Anwendungen die trotzdem auf Änderungen angewiesen sind, müssen die entsprechenden Daten zusätzlich in relationalen Datenbanken halten, wo dann direkt Updates möglich sind oder sie bedienen sich einer prozeduralen Änderung über die DOM-API.
Eine vollständige XML-Anfragesprache[1] soll jedoch Updates enthalten und dazu Konzepte wie Transaktionen, Recovery sowie physische und logische Datenunabhängigkeit unterstützen.

Ziel dieser Arbeit soll der Entwurf einer XML-Update-Sprache als Erweiterung zu XML-Query Language sein.

Dazu wird in Abschnitt 2 ein Anforderungskatalog an eine solche Änderungssprache aufgestellt.
Eine Übersicht zu existierenden Ansätzen einer Update-Sprache wird in Abschnitt 3 vorgelegt.
Die eigentliche Konzeption erfolgt in Abschnitt 4, wobei hier erst Grundlagen von Änderungsoperationen diskutiert werden. Danach folgt die Festlegung von Semantik und Syntax dieser Sprache und zuletzt wird anhand einer Beispieldiskussion gezeigt, wie die Änderungen in einem relationalen System abgebildet werden können.

[1] Man beachte die Begriffserläuterungen im Anhang

2 Anforderungen an eine XML-Update-Sprache

2.1 Allgemeine Anforderungen

Im Fokus dieses Kapitels soll die Aufstellung und Untersuchung von Kriterien an eine XML-Update-Sprache stehen. Vorliegende Artikel zu diesem Thema ([Mar00], [Leh01]) sind leider oft sehr pragmatisch gehalten und können deshalb nicht als Informationsquelle herhalten. Als allgemeine Grundlage kann man jedoch auf die Anforderungen an (relationale) Datenbank-Anfragesprachen zurückgreifen. Eine entsprechende Kriteriensammlung wurde [HS00] entnommen und soll im Folgenden adaptiert werden.

- Ad-hoc-Formulierung, **Deskriptivität**, Mengenorientiertheit: Eine XML-Anfragesprache sollte auf Mengen von Dokumenten und deren Knoten[2] (Elemente, ...) operieren, anstelle der Navigation auf einzelnen Bestandteilen wie etwa in Programmiersprachen üblich. Dafür ist es auch geboten, dass die Anfrage nur das gewünschte Ergebnis beschreibt, ohne eine Berechnungsvorschrift anzugeben. Eine Voraussetzung hierzu ist die Abkopplung der Sprache von der Speicherungsstruktur der Daten auf eine höhere Ebene (Datenunabhängigkeit).

- **Abgeschlossenheit:** Jedes Ergebnis (und Teilergebnis) der Anfrage muss im Datenmodell darstellbar sein, also einem wohlgeformten XML-Dokument bzw. -Fragment entsprechen und somit als Eingabe wiederzuverwenden sein.

- **Adäquatheit:** Alle Konzepte des zugrundeliegenden Datenmodells werden auch durch die Anfragesprache ausgeschöpft. Hier wäre es also zum Beispiel nicht ausreichend, als Anfrageergebnis ausschließlich Sequenzen von XML-Knoten konstruieren zu können. Mittels Änderungsoperationen sollte es somit möglich sein, nicht nur einzelne Werte oder Knoten zu manipulieren, sondern komplexe Strukturen, wie z.B. vollständige XML-Fragmente.

- **Optimierbarkeit**, Effizienz: Die Deskriptivität und Abgeschlossenheit sind grundlegende Voraussetzungen für eine optimierbare Anfragesprache, für die jedoch eine formale Semantik definiert sein sollte, um z.B. im Fall von algebraischen Ausdrücken äquivalenzerhaltende Umformungen vornehmen zu können. Die Reihenfolge der Ausführung von Operationen sollte nicht durch die Anfrage vorgegeben sein, sondern bestmöglich vom System entschieden werden.
  Zur Entscheidungsfindung ist es notwendig, die Komplexität von Operationen bestimmen zu können, um so den Weg mit den geringsten (Laufzeit-)Kosten auszuwählen. Effiziente Implementierungen sollten Algorithmen vermeiden, die exponentiellen oder höheren Aufwand bedeuten.

[2] Einschränkung: Soweit die Ordnung innerhalb des Dokuments relevant ist, muss diese auch berücksichtigt werden.

- **Sicherheit:** Eine sichere Anfragesprache erfordert, dass jeder syntaktisch korrekte Anfrageausdruck ein endliches Ergebnis liefert, ohne dabei in eine Endlosschleife zu gelangen. Spezielle Konzepte wie Negation (z.B. beim Exists-Quantor), Rekursionen und nutzerdefinierte Funktionen müssen dazu eingeschränkt werden. Wenn dies nicht erwünscht oder möglich ist, wären auch zusätzliche Mechanismen auf tieferer Ebene denkbar, die solche Fälle erkennen und auflösen.

- **Orthogonalität:** Sprachkonstrukte sollen in ähnlichen Situationen gleichartig einsetzbar und möglichst uneingeschränkt kombinierbar sein. Beispielsweise ist es wünschenswert, Vergleichsoperatoren unabhängig vom Knotentyp einsetzen zu können.

- **Vollständigkeit**, Eingeschränktheit: Damit Eigenschaften wie Optimierbarkeit und Sicherheit verwirklicht werden können, muss die Mächtigkeit der Anfragesprache eingegrenzt werden. Diese Einschränkungen dürfen jedoch nicht zu Lasten der Vollständigkeit der Sprache gehen. Kann man mit ihr alle Anfragen stellen bzw. Änderungen erreichen, die auch auf dem basierenden formalen Modell möglich sind, so ist sie gleich mächtig und damit vollständig.

Es sei hier noch einmal ausdrücklich darauf hingewiesen, dass es sich bei diesen Kriterien um ein "Wunschszenario" handelt. Man kann also nicht davon ausgehen, dass eine XML-Update-Sprache *alle* der oben aufgezählten Anforderungen erfüllt. Vielmehr sollen damit Orientierungspunkte für den Entwurf einer solchen Sprache bereitgestellt werden.

2.2 XML-spezifische Anforderungen

Die Betrachtungen aus dem vorangegangen Abschnitt beziehen sich vorrangig auf Anfragen, sind aber auch für Änderungsoperationen auf diesen Daten gültig, die nach der hier vertretenen Auffassung integraler Bestandteil einer Anfragesprache sein sollten. Eine Update-Operation ist jedoch in der Lage, die Basisdokumente selbst zu verändern, während eine Anfrage nur temporäre Pseudoausprägungen dieser Daten darstellt.

Im Weiteren soll nun geklärt werden, welche Anforderungen an solche Änderungen im Umfeld von XML gestellt werden müssen und aus welchen Richtungen sich eine Änderungssprache entwickeln kann.

2.2.1 Datenmodell

Die gemeinsame Basis aller XML-Technologien stellt die Spezifikation des XML Information Set's (Infoset) [XInfo01] dar, in der die Kernabstraktionen jedes XML-

Dokuments definiert werden. Diese Beschreibung umfasst dabei weder syntaktische Details der Extensible Markup Language 1.0 [XML00SE] und damit auch keine Aussagen zur syntaktischen Wohlgeformtheit oder Gültigkeit von Dokumenten noch eine Forderung nach einem (bestimmten) Interface. Ein Information Set enthält 11 potentielle Arten sogenannter *Information Items* mit jeweils spezifischen Eigenschaften, deren Beziehungsmodell in der Abbildung 2 grafisch dargestellt wurde. [3] Ausgangspunkt ist

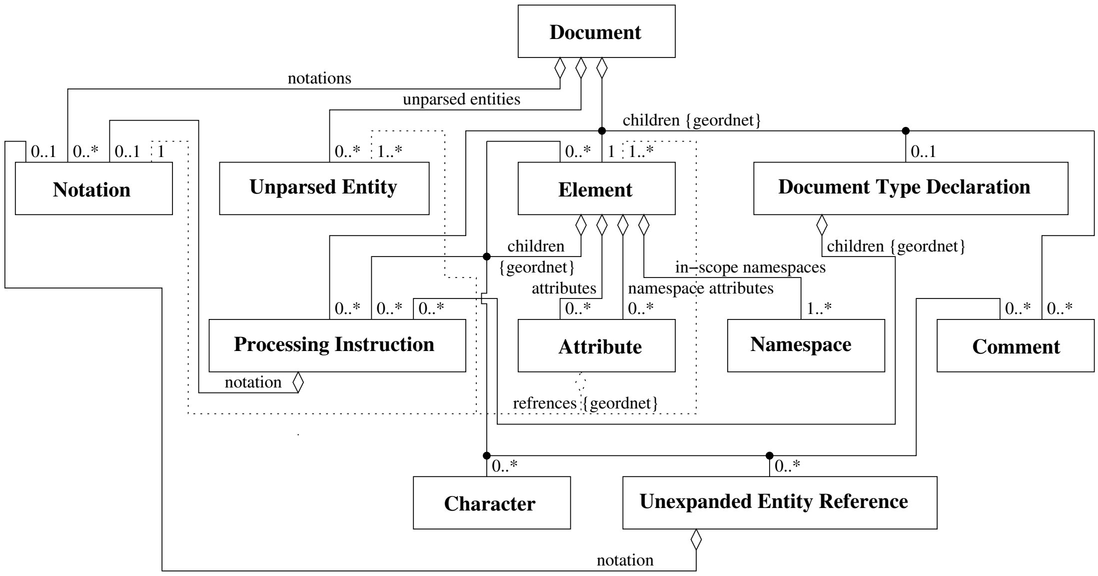

Abbildung 2: UML-Modell des XML-Infoset

dabei das Document-Information-Item, dass wie auch Element-Information-Items eine geordnete Liste von Kind-Information-Items `[children]` enthält. Diese bilden das Grundgerüst des Infoset's, das somit einem *geordneten Baum* von benannten Knoten (Items) gleich steht.
Während Sub-Items wie Processing Instructions, Characters (Zeichendaten) oder Elemente einer bestimmte Reihenfolge unterliegen, werden andere Items, wie Attribute oder Namespaces als ungeordnete Menge `[attribute]` eines übergeordneten Elements modelliert. Diese ungeordneten Items werden im Sinne des Infoset nicht als Teil des Inhaltsgraphen betrachtet.

Referenzen Das Infoset beinhaltet neben den "reinen" Baumstrukturen auch verschiedene Referenzmechanismen, wodurch das Modell auf einen *baumorientierten Graphen* ausgedehnt wird. An erster Stelle wäre hier die `ID/IDREF`-Funktionalität zu nennen, bei der Elemente durch Identifikatoren (Attributtyp `ID`) ausgezeichnet werden und damit innerhalb des Dokuments einen eindeutigen Zugriff (Attributtyp

[3] Aus Gründen der Übersichtlichkeit wurden Wertattribute ausgeblendet und nur Assoziationen modelliert.

`IDREF`) ermöglichen.
Da dieser Mechanismus auf bestimmten Attributtypen basiert die durch eine DTD festgelegt werden müssen, können `ID/IDREF`'s nur bei Vorhandensein eines solchen Schemas modelliert werden.

Eine andere Form der Referenzierung sind XML-Links (*XLink*, *XPointer*), die auch Zeiger über die Dokumentgrenzen hinaus ermöglichen, sie erweitern dabei unter anderem den internen `ID/IDREF`-Mechanismus. XML-Links sind jedoch nicht Bestandteil der Abstraktionen im XML-Infoset, sondern nur auf der Ebene der Extensible Markup Language zu finden. Außerdem kann hier nicht die referentielle Integrität gesichert werden, man sollte sie daher eher als Verweis im Sinne von href in HTML betrachten, wobei ihre Funktionalität allerdings deutlich umfangreicher ist.

Entities Einer gesonderten Betrachtung bedürfen Entities, deren "Arbeitsplatz" eigentlich auf tieferen Schichten zu finden ist. So bilden sie die Speichereinheiten von XML-Dokumenten auf der physischen Ebene (Abb. 3, entnommen [DSL01]), und können von diesen auch mehrfach referenziert werden. Allgemein kann ein

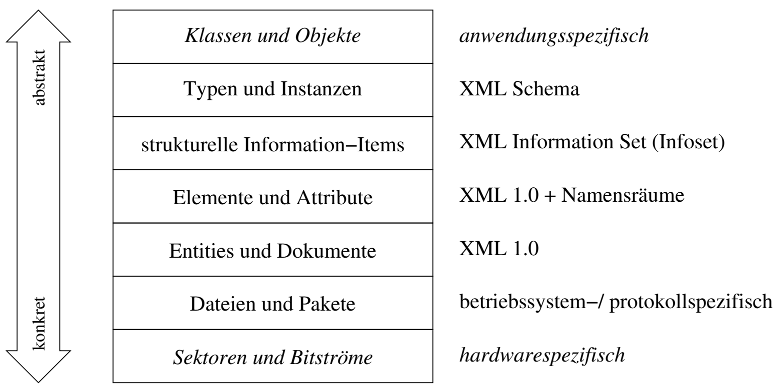

Abbildung 3: Schichtenmodell der XML-Technologie

XML-Dokument als Serialisierung von solchen Entities aufgefasst werden, das dann auf der logischen Ebene wieder ein Datenobjekt bildet. Im Infoset-Modell wird davon ausgegangen, dass diese Referenzen bereits aufgelöst und damit durch den korrespondierenden Inhalt ersetzt wurden.
Es sind jedoch Situationen denkbar, in denen ein XML-Prozessor keine Möglichkeit hat, diese Entities zu substituieren, beispielsweise wenn der Zugriff über externe URL's fehlschlägt. In solchen Fällen müssen diese Einheiten als `Unexpanded Entity Reference`'s im Infoset gekennzeichnet werden.
Eine ähnliche Behandlung erfahren `Unparsed Entity`'s, die von vornherein von der Analyse ausgeschlossen werden und als solche im Information Set erscheinen.

Jeder `Unparsed Entity` ist ein `Notation`-Item namentlich zugeordnet, um den betreffenden Inhalt und dessen Behandlung zu spezifizieren. Probleme, die gerade durch den Einsatz von externen Entities im Zusammenhang mit Update-Operationen auftreten, werden an anderer Stelle (4.2.2) diskutiert.

Die Betrachtungen für Referenzen als Schlüssel-Fremdschlüssel-Beziehungen werden im Weiteren auf `ID/IDREF` beschränkt.

Namespaces Neben der lokalen Namensgebung der Items innerhalb eines Dokuments ist das Konzept der *Namespaces* [XNS99] integraler Bestandteil der Infoset-Spezifikation geworden.
Ein XML-Namensraum ist eine Zusammenstellung von Namen, identifiziert durch einen URI-Verweis, und kann mit Element- oder Attribut-Items verknüpft werden. So ist eine global eindeutige Bezeichnung gewährleistet und damit der Austausch von XML-Daten über organisatorische Grenzen hinweg möglich.
Außerdem können Namenskonflikte aufgelöst werden, die innerhalb eines Dokuments auftreten, wenn es aus Teilen mit verschiedener Herkunft zusammengesetzt ist.

Alternative Modelle Die Spezifikation des Information Set's wird zwar als "modified tree" beschrieben, ist aber keine zwingende Anforderung für darauf arbeitende Schnittstellen, es auch als Baumstruktur zur Verfügung zu stellen, es könnte sich dabei auch um ereignis- oder anfragebasierte Interfaces handeln. Damit steht es Spezifikationen und insbesondere Sprachansätzen frei, dieses Modell für ihre Einsatzzwecke zu erweitern.
Beispielsweise werden in *XQuery* auf der obersten Hierarchieebene Sequenzen von Knoten oder ganzen Bäumen zugelassen. Zur Einhaltung der Konformität wird dann aber eine Beschreibung der Abbildung auf das Infoset gefordert.

2.2.2 Beachtung eines Schemas

Das Infoset fordert nicht zwingend die Validierung gegen ein explizites Schema. Implizit sind jedoch immer Schemainformationen in den XML-Daten enthalten, da das *Markup* die logische Struktur des Dokuments beschreibt und Attribute ihren logischen Strukturen zuweist. Aus einem XML-Dokument kann also immer ein Schema abgeleitet werden, wobei diese Ableitung nicht eindeutig sein muss.

Zur Zeit existieren zwei mögliche Schemaformen für XML-Dokumente, die vom W3C empfohlen werden:

Document Type Definition - DTD Diese Dokumenttypbeschreibung ist ein Überbleibsel aus den Wurzeln von XML, den Dokumentenverwaltungssystemen für die damals die *SGML* maßgeblich war. Die DTD war daher der ursprüngliche Vorschlag um Vorgaben an die Struktur eines XML-Dokuments aufzustellen.

Die Formulierung einer DTD ist einfach und knapp und daher für den händischen Gebrauch gut geeignet. Auf der anderen Seite sind die DTD-Konzepte limitiert (nur Zeichendaten-Typ, keine Kardinalitäten) und ihre Syntax liegt nicht in XML vor und bedarf deshalb zusätzlicher Werkzeuge zur Auswertung.
Substantielle Bestandteile sind Folgen von ungeordneten `ELEMENT`- und `ATTLIST`-Deklarationen sowie Entities und Notations als vordefinierte Speichereinheiten

XML-Schema Die XML-Schema Spezifikation [XSM01] ist der jüngere Vorschlag einer Schemabeschreibung und wurde vor allem durch die Sichtweise in Datenbanken beeinflusst.
Im Unterschied zu DTD's liegen XML-Schemas im XML-Format vor, dadurch werden Meta-Daten durch die gleichen Strukturen beschrieben wie Dokumentdaten, womit dann der Zugriff und die Auswertung des Schemas mit denselben Mitteln, die auch für Dokumente eingesetzt werden, möglich ist.

Die wichtigsten Konzepte werden in der folgende Übersicht dargestellt:

- Typsystem
  - Vielzahl vordefinierter Datentypen
  - Definition eigener Datentypen (einfach und komplex) möglich
  - Hierarchien von Typdefinitionen (`extension`, `restriction`)
  - lokale und globale Definitionen
- Erweiterte Inhaltsmodelle gegenüber DTD's
- Attributgruppen (Mehrfachverwendung von Attributen)
- Schlüsselreferenzen und Integritätsbedingungen (`key/keyref`, `unique`)

Dabei stehen alle Modellierungsmöglichkeiten von DTD's auch in XML-Schema zur Verfügung, weshalb eine verlustfreie Übersetzung einer DTD in ein XML-Schema möglich ist.

Vergleich: DTD und XML-Schema Insgesamt ist XML-Schema gegenüber DTD's die technisch "saubere" und umfassendere Lösung, zumal nur hiermit echte Typinformationen beschrieben werden können. Allerdings sind XML-Schema-Dokumente für den praktischen Einsatz viel zu aufgebläht und damit (für Menschen) schwer zu lesen und zu entwerfen.
Hier wird man meistens auf grafische Bearbeitungswerkzeuge zurückgreifen oder stattdessen eine DTD entwerfen und eine automatische Transformation in ein XML-Schema durchführen (z.B. Tools wie *XMLSpy*), das dann noch manuell nachbearbeitet werden kann.

Diese Verfahrensweise zeigt deutlich, dass DTD's auch mittelfristig ihre Daseinsberechtigung haben, da Einfachheit und Verständlichkeit entscheidende Faktoren für die Akzeptanz von neuen Technologien sind (vgl. SGML-XML-HTML).

Schema vs. schemalos Welche Vor- und Nachteile ergeben sich jetzt durch die Benutzung eines Schemas.

Durch ein Schema ist die ausdrückliche Bereitstellung der Strukturinformation, lesbar für Mensch und Maschine, möglich. Zum Beispiel weiß man manchmal nicht genau, ob ein String als Attribut-Wert oder Element-Inhalt auftaucht, anhand eines Schemas kann diese Information gewonnen werden. Durch diese Metadaten ist also die genaue Kenntnis über anzufragende oder zu ändernde Komponenten gegeben.
Als Vorteil daraus ist die Erzeugung eines Plans zur Auswertung einer Anfrage oder Änderung möglich, der dann zusätzlich noch optimiert werden kann. Die Ausführung der Operationen kann danach ohne Schemazugriff erfolgen.

Weiterhin können Mengen von XML-Dokumenten in gleiche Kategorien zusammengefasst werden und etwa in XML-Datenbanken gespeichert werden. Eine Anfrage/Änderung kann dann auf diesen Mengen von Dokumenten operieren und muss nicht prozedural über jedes einzelne Dokument iterieren.
Zu diesem Konzept zählt auch die Betrachtung der Schemaevolution. Die definierte Änderung des Schemas wird dazu in den zugehörigen Dokumenten nachgezogen. So kann die Konsistenz des gesamten Datenbestandes für die darauf arbeitenden Anwendungen gewahrt bleiben.

Ein letzter Vorteil ist die Ausnutzung eines Schemas zur Fehlererkennung und Validierung von XML-Daten, hiermit wird neben der Wohlgeformtheit die Konformität der Dokumente geprüft und damit insgesamt die Gültigkeit garantiert.

Schemalose Modelle, wie z.B. OEM aus Lorel, zeichnen sich vor allem durch ihre Einfachheit und Flexibilität aus. Als Nachteil ist hier jedoch keine zweistufige

Anfrageverarbeitung möglich, sondern es muss eine dynamische Ergebnisfindung stattfinden, die im Extremfall zur Untersuchung der gesamten Datenbank führen kann.

Im Endergebnis dieser Betrachtung muss also gesagt werden, dass die Argumente deutlich für die Verwendung eines Schemas sprechen. Nur für Anwendungsszenarien, bei denen Dokumente kaum Gemeinsamkeiten in ihren Strukturen haben, sollte eine schemalose Modellierung vorgezogen werden.

2.2.3 Ordnungserhaltung

Die Ordnungserhaltung in XML-Dokumenten stellt einen wichtigen Unterschied zu herkömmlichen Anfragensprachen dar, die in der Regel auf Mengenbasis und damit ohne Berücksichtigung der Ordnung arbeiten.
Wie schon im Datenmodell erläutert findet innerhalb des Dokuments eine unterschiedliche Behandlung der Ordnung statt, wobei also die Ordnung unter Attributen sowie Namespaces nicht relevant ist. Wird ein Dokument in eine Baumstruktur transformiert, darf eine Rücktransformation in ein Dokument keine Änderung in der Abfolge von Elemten, Text oder anderen Komponenten haben.
Dazu wird ein XML-Baum wie folgt geordnet.

- Der Dokument-Knoten ist immer der erste Knoten.
- Element-Knoten, PI-Knoten usw. treten in Reihenfolge ihrer Repräsentation in der XML auf.
- Element-Knoten stehen vor ihren Kindern.
- Namespaces und Attribute werden direkt nach ihren Element-Knoten angegeben.

Die relative Ordnung von Knoten in verschiedenen Dokumenten muss stabil sein, das heißt, wenn *ein* Knoten in einem Dokument A vor einem Knoten in einem Dokument B steht, so müssen *alle* Knoten in Dokument A vor den Knoten in Dokument B stehen. Dies ist wichtig, wenn man Sequenzen von Dokumenten oder Teilbäumen betrachtet.

Für Anwendungen die nicht auf die Ordnungserhaltung angewiesen sind, wäre es wünschenswert, wenn sich das System darauf einstellen kann und so eine Optimierung von Anfragen und Änderung erlaubt.
Für Anwendungen die nicht auf die Ordnungserhaltung angewiesen sind, ist es effizienter, auf einfachen Mengen zu arbeiten. Ein System sollte also beide Szenarien beherrschen und optimal umsetzen.
Ein relationales Datenbanksystem arbeitet zum Beispiel auf Mengen, dafür sind weniger Konzepte und Operationen notwendig, wodurch auch eine bessere Optimierbarkeit

ermöglicht wird.
Objektrelationale Datenbanken sind für die Ordnungserhaltung besser geeignet, da sie unter anderem Listentypen beinhalten. Es sind dann zusätzliche Operationen auf diesen Typen erforderlich, die aber im Vergleich zu relationalen Mengen-Operationen meistens eingeschränkt sind. Des Weiteren ist der Zugriff auf Listentypen komplexer und somit weniger effizient.

Wie in Abschnitt 4.2.2 zu sehen ist, erfordert die Beachtung der Ordnung eine spezielle Semantik beim Einfügen oder verschieben von Elementen, die eine genaue Positionierung des Ziels ermöglichen. Reine Mengenoperationen wie in relationalen Modellen würden hier nicht ausreichen.

2.2.4 Sprachkonzepte

Spracherweiterung/Eigenständige Sprache Ein Ansatz für eine Änderungssprache könnte darin bestehen, eine bestehende "reine" Anfragesprache um die Änderungssemantik zu erweitern. In dieser Vorgehensweise besteht der Vorteil darin, dann ein einheitliches, vollständiges System für den Zugriff und die Modifizierung von XML-Dokumenten zu Verfügung zu stellen.

Dagegen wäre auch die Definition einer eigenständigen Änderungssprache vorstellbar, die dann aber für die Selektion der zu manipulierenden Objekte wieder ein völlig neues Modell benötigt oder über spezifische Schnittstellen mit Fremdsystemen kommunizieren muss.

Sprachebene Die Wahl der Ebene eines Sprachansatzes bestimmt in entscheidendem Maße die Datenunabhängigkeit. Dazu ist es erforderlich, das Sprachmodell von der physischen Ebene, also den Speicherungsstrukturen abzutrennen.
Auf der logischen Ebene muss dieses Modell von Änderungen in den zugreifenden Anwendungen abgekoppelt werden.

2.2.5 Transaktionen

Durch das Einführen von Änderungsoperationen werden sogenannte *schreibende Zugriffe* auf die Datenmengen ermöglicht. Diese können im Mehrbenutzerbetrieb verschiedene Seiteneffekte hervorrufen (*lost update*, *dirty read*, ...), die bei ausschließlichen Lesezugriffen durch Anfragen nicht auftreten können. Um diese Probleme zu vermeiden, bedarf es einem Modell zur Erkennung und Auflösung von Konflikten: Die Transaktionsverwaltung.

Dazu werden Folgen von Operationen in Transaktion zusammengefasst, die nach den *ACID*-Eigenschaften verlaufen. Das sind:

- Atomarität (Atomicity), die besagt, dass eine Transaktion entweder ganz oder gar nicht ausgeführt wird. Bei einem Abbruch muss dann also wieder der Ausgangszustand vor Beginn der Transaktion vorliegen.
- Konsistenz (Consistency) muss für den Zustand vor und nach einer Transaktion gegeben sein.
- Isolation kapselt die Daten für gleichzeitig aktive Nutzer so voneinader ab, dass jeder die Sicht auf die Daten hat, als ob er allein arbeiten würde.
- Dauerhaftigkeit (Durability) sorgt dafür, dass ein Ergebnis der Transaktion persistent in der Datenbank gespeichert wird.

Konflikte können nun immer so vermieden werden, indem Transaktionen einfach hintereinander (seriell) ablaufen müssen. Da dies jedoch ineffizient und in praktischen Umgebungen oft ausgeschlossen ist (Warten auf die Beendigung einer Transaktion), müssen Transaktion verzahnt ablaufen können. Hier muss dann eine Transaktionsverwaltung zum Einsatz kommen um einen reibungslosen Ablauf zu gewährleisten.

In diesem Zusammenhang ist die Einführung von Sperren sinnvoll, durch die Bereiche oder Objekte für den exklusiven Zugriff einer Transaktion gekennzeichnet werden. Hierbei werden Lesesperren und Schreibsperren für die Vorbereitung auf einen Lese- oder Schreibzugriff unterschieden, die durch ein unlock wieder aufgehoben werden können. Ein Sperrprotokoll legt dabei unter Einhaltung einer Sperrdisziplin den Ablauf der Sperranforderungen und der Freigabe der Sperren fest. Ein Beispiel ist das Zwei-Phasen-Sperr-Protokoll (2PL) in in dem nach einer Sperrfreigabe keine weiteren Anforderungen mehr erfolgen dürfen. So teilt sich der Transaktionsverlauf in die beginnende Anforderungsphase und die abschließende Freigabephase.

Hierarchische Sperrprotokolle Für XML-Dokumente ist die Einbeziehung von Struktur-Informationen für die Sperrung unterschiedlicher Granularität notwendig. Dazu eignen sich hierarchische Sperrprotokolle die [Sch02] und [SH99] entnommen wurden.

Klassisch wird in relationalen Datenbanken die logische Granulariät betrachtet, also `Datenbank-Relation-Tupel-Attribut`, es wäre aber die physische Hierarchie denkbar `(Datenbank-Datei-Cluster-Seite)`.

Diese Ansatz ist auch oder gerade in Objekthierachien (ODB) und XML-Dokument-Strukturen (Dokument-Teilbaum-Knoten-Wert) anwendbar. Bei der Umsetzung ist zu beachten, dass die Hierarchietiefe in relationalen Systemen fest vorgeben ist, während sie in der XML-Struktur dynamisch variiert.

Multi-Granulartiy-Locking (MGL) Gesucht wird eine einfache Möglichkeit, ein (Teil-)Dokument zu sperren, ohne jeden einzelnen Knoten sperren zu müssen. Dazu soll ein einfacher Test ermitteln, ob eine Sperranforderung auf einem (Teil-)Dokument gewährt werden kann. Als Ziel soll ähnlich der Concurrency Control im flachen Modell vermieden werden, dass unverträgliche Sperren von verschiedenen Transaktionen gehalten werden.

Als Lösungsidee werden Intention Locks eingeführt, dass sind Warnungen vor Sperren in Blattrichtung. Weiterhin soll es wieder Shared Locks (auch Lesesperre) und eXclusive Locks (Schreibsperre) geben, die elementaren Sperren entsprechen.
Eine explizite Sperre (S bzw. X) auf einem umfassendem Objekt sperrt nun aber implizit alle enthaltenen Knoten, also das gesamte XML-Fragment. Ein Intention Lock (IS bzw. IX) auf einem umfassendem Objekt vereinfacht dabei den Test auf Gewährung einer Sperre.
Für den geläufigen Fall, dass eine Transaktion nur einige Knoten eines Teildokuments schreibt, während alle Knoten gelesen werden, wird eine kombinierte Lese/intentionale Schreibsperre (SIX) eingeführt.

Für alle hierarchischen Modelle wird dazu nach dem MGL-Protokoll verfahren:

1. Das Sperren verläuft $top \rightarrow down$ (von der Wurzel zu den Blättern), während die Freigabe $bottom \rightarrow up$ erfolgt
2. Das Lesen bzw. Schreiben eines Objektes a erfordert ein Sperren unter Beachtung der Sperrvertäglichkeitsmatrix (siehe Tabelle 1)
3. Alle Knoten auf dem Pfad von der Wurzel bis zum Zielobjekt werden mit intentionalen Sperren belegt (wieder Veträglichkeit beachten)
4. Sperren können verschärft werden ($S \rightarrow X, IS \rightarrow S$ und $IS \rightarrow IX$)

Das MGL-Protokoll garantiert, dass zwei Transaktionen nie unverträgliche Sperren besitzen. In Verbindung mit dem *2PL* oder dem *S2PL* kann so immer ein konfliktserialisierbarer Schedule geliefert werden.

DAG-Locking (Directed Acyclic Graph) Durch den Einsatz von Referenzen `ID/IDREF` wird aus dem XML-Baummodell ein Graph. Für solche azyklische

Verträglichkeit		Transaktion T_2 fordert Lock auf a an				
		S	X	IS	IX	SIX
T_1 hält Lock auf a	S	$+$	$-$	$+$	$-$	$-$
	X	$-$	$-$	$-$	$-$	$-$
	IS	$+$	$-$	$+$	$+$	$+$
	IX	$-$	$-$	$+$	$+$	$-$
	SIX	$-$	$-$	$+$	$-$	$-$

$-/+$ Sperren sind un-/verträglich

Tabelle 1: Kompatibilitätsmatrix für MGL

Graphen-Modelle eignet sich das *DAGL*. Dabei muss im Vergleich zu hierarchischen Modellen beachtet werden, das ein Knoten durch mehrere Wege erreichbar ist.
Das *DAGL* ist analog dem MGL-Protokoll mit einer Verfeinerung von Punkt 3:

3. (a) Bei einer Schreibsperre X müssen alle Knoten auf *jedem* möglichen Pfad durch eine IX-Sperre versehen werden
   (b) Für eine Lesesperre reicht die Markierung eines Pfades mit IS-Sperren

Zu weiteren Betrachtungen sei auf [Sch02] verwiesen.

Baumsperrverfahren Eine andere Form der hierarchischen Sperren ist das Baumprotokoll, bei dem keine implizite Sperrung eines Teilbaumes erfolgt sondern jeweils einzelne Knoten gesperrt werden. Dafür wird das einfache Lock/Unlock-Modell eingesetzt. Im Unterschied zu den vorherigen Protokollen wird hier nicht auf dem *2PL* aufgesetzt.
Ein Knoten kann nur dann von einer Transaktion gesperrt werden, wenn sein Vaterknoten bereits von ihr gesperrt wurde, die erste Sperre darf aber beliebig gesetzt werden. Sperren können zwar jederzeit wieder freigegeben werden, ein Knoten darf jedoch innerhalb einer Transaktion nicht noch einmal gesperrt werden.
Auch für das Baumprotokoll gilt, es liefert einen serialisierbaren Schedule, wenn alle Transaktionen sich an das Protokoll halten.

2.2.6 Transformationen / Änderungsoperationen

Transformationsprachen wie *XSLT* sollten nicht mit einer echten Änderungssprache gleichgesetzt werden.
Eine Transformation dient im Umfeld dokumentenzentrierter Anwendungen zur Überführung eines Dokuments von einem Schema in ein anderes. Dabei werden neue

Dokumentinstanzen generiert, die Basisdokumente bleiben jedoch unverändert.

Eine Änderungssprache kann hingegen inkrementelle Änderungen an Dokumenten vornehmen, die in einer XML-Datenbank gespeichert sind. Dies ist ein datenzentriertes Vorgehen.

2.2.7 Weitere Konzepte

Neben den oben vorgestellten Kriterien existieren weitere Konzepte, die hier nicht näher betrachtet werden sollen:

- Sichtenmodell und Änderungen auf Sichten
- hierarchische Rechtevergabe
- Backup und Recovery
- Verteilte Datenquellen (bisher wurde immer *eine* Speichereinheit angenommen)

3 Existierende Ansätze

3.1 Das Document Object Model

Der bestgewachsene Wald nützt Holzfäller und Schreiner nichts, wenn sie keine Werkzeuge zum Fällen, Entlauben, Entasten und Zersägen des Baums und zur Bearbeitung des Holzes haben. XML-Prozessoren sind entsprechende Werkzeuge, die XML-Dokumente verarbeiten und den Applikationen die Daten "mundgerecht" servieren. Die *Simple API for XML (SAX)* und das *Document Object Model (DOM)*[DOM00] sind abstrakte Programmierschnittstellen und stellen praktisch den Standard für die Implementierung von XML-Parsern dar. Beide gehen dabei sehr unterschiedliche Wege, während *SAX* aus einer Menge von Streaming-Schnittstellen besteht und die Informationen eines XML-Dokuments in eine lineare Sequenz bekannter Methodenaufrufe zerlegt, wird in *DOM* ein Satz von Schnittstellen bereitgestellt, die das XML-Dokument in einen hierarchischen Baum aus generischen Objekten (Knoten) abbilden.
Neben dem Parsen von Dokumenten können die Modelle von SAX und DOM auch zur Generierung von XML-Dokumenten benutzt werden.

Untersucht man die beiden Schnittstellen auf ihre Möglichkeiten zur Manipulation von XML-Daten, so muss man klar sagen, dass *DOM* dafür geradezu prädestiniert ist, da es spezielle Eigenschaften und Methoden zur Navigation und Änderungen im Modell bereithält. *SAX* ist dagegen zustandslos, wodurch kontextabhängige Zugriffe auf der Implementationsebene nachgebildet werden müssen. Auch die Änderung von Daten scheint innerhalb des *SAX*-Modells nicht möglich, ein Vorschlag für einen Ersatzmechanismus wird in Abbschnitt 3.1.4 skizziert.

3.1.1 Das Datenmodell

Man kann DOM als Projektion des XML-Infoset's ansehen, wobei die Infoset-Items als Knoten eines Graphen in Baumstruktur repräsentiert werden. Zu jedem Knoten wird eine Schnittstelle inklusive Syntax und Semantik zur Verfügung gestellt und die Beziehungen zwischen den verschiedenen Knotentypen spezifiziert. DOM legt dabei aber nicht fest, wie die zu Grunde liegende Speicherungsstruktur aussehen soll und welche Algorithmen einzusetzen sind.

Eine zentrale Rolle in diesem Objektmodell spielt das generische *Node*-Interface, von dem alle konkreten Knotenarten abgeleitet werden. Neben den einfachen Eigenschaften eines Knotens wie Name, Wert oder Knotentyp werden durch die *Node*-Schnittstelle auch dynamische Eigenschaften zur Nachbildung der *[parent]/[children]*-Beziehung des Infoset's und Methoden zur Manipulation dieser

Strukturen angeboten.

Auch die unterschiedliche Behandlung der Ordnung zwischen bestimmten Knoten wird in DOM berücksichtigt. Geordnete Knoten werden in einer *NodeList* aufgereiht und können über ihren Index zurückgewonnen werden, ungeordnete Knoten wie Attribute werden in einer *NamedNodeMap* gesammelt, ein Zugriff erfolgt hier über den Namen.

Vergleicht man die Knotentypen von DOM mit den Items des Infoset's, kann man im Wesentlichen eine 1:1-Abbildungen feststellen. Allerdings wird im Infoset grundsätzlich davon ausgegangen, dass alle Entities durch ihren referenzierten Inhalt ersetzt und Inhalte normalisiert wurden, in DOM werden auch Dokumente in ihrem *Rohzustand* unterstützt.

3.1.2 Navigation

Die schrittweise Navigation durch die Baumstruktur kann durch die dynamischen Nur-Lese-Attribute der *Node*-Schnittstelle bewerkstelligt werden. Die Abbildung 4 zeigt dazu alle vom Knoten `Chrischi` ausgehenden "Familien-Knoten", also Eltern, Geschwister und Kinder sowie das zugehörige Dokument. Auf diesem Weg ist jeder

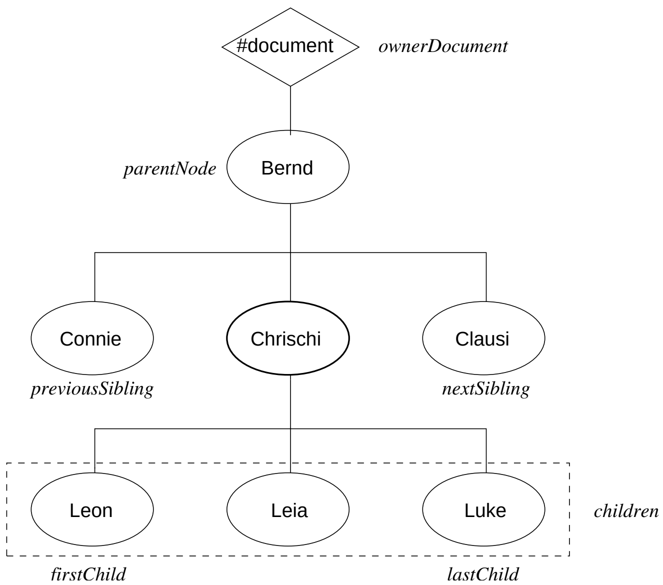

Abbildung 4: Navigation entlang der Baumstruktur

Knoten des Dokuments, beginnend von einer beliebigen Position, erreichbar.
Ausklammern muss man dabei Attribute, die in DOM nicht als Teil des Dokumentenbaums betrachtet werden, obwohl `Attr` vom Interface `node` abgeleitet wird. Sie

stellen vielmehr *Eigenschaften* von Elementen dar und können so entweder als einfache NameNodeMap-Eigenschaft `attributes` (nur lesend) oder über spezifische Methoden der Element-Schnittstelle (s. Tabelle 2) erreicht werden.

Um das Durchlaufen der DOM-Hierarchie zu vereinfachen, wurde das optional zu unterstützende *DOM-Traversal* entwickelt. Hierin werden zwei verschiedene Mechanismen definiert:

- Der `NodeIterator` iteriert durch eine Liste von Knoten (`nextNode`, `previousNode`) ähnlich der *NodeList*-Schnittstelle.
- Die `TreeWalker`-Schnittstelle verfügt zusätzlich über die Funktionalität zur Bewegung in der Baumhierarchie wie die *Node*-Schnittstelle.

Den eigentlichen Vorteil erreicht man in *DOM-Traversal* durch den Einsatz eines *NodeFilter's*. Im einfachsten Fall legt man durch Flags fest, welche Knotentypen bei einem Durchlauf einbezogen werden sollen. Zusätzlich ist aber auch die Definition eigener Filter möglich, deren Implementation der Methode `acceptNode` entscheidet, wie mit dem aktuellen Knoten zu verfahren ist. Dadurch werden natürlich auch sehr komplexe Auswahlkriterien möglich.

In *DOM Level 3* wird, wie schon in einigen DOM-Implementationen geschehen, die Einbettung von XPath-Ausdrücken möglich sein. Dazu ist es allerdings notwendig, die Unterschiede zwischen dem XPath-Model und DOM zu überbrücken. Dies geschieht durch ein universelles *Result-Set* für XPath-Ergebnisse, die gleichzeitig wieder als Eingabe verwendbar sind. Bei der Auswertung einer Expression muss mitgeteilt werden, in welchen Typ das Ergebnis konvertiert werden soll, ansonsten wird ein ungeordnetes Node-Set angenommen.
Trotz Problemen bei der Interaktion zwischen beiden Technologien ist XPath eine sehr wertvolle Erweiterung zu DOM, die einen hohen Programmieraufwand für die Selektion von Dokumentteilen vermeiden kann. Die einzige Möglichkeit zur Auswahl einer Menge von Elementen bestand zuvor aus den Methoden `getElementByTagName` und `getElementById`.

3.1.3 Manipulation

Wie schon oben erwähnt, beinhaltet *DOM* verschiedene Mechanismen zur elementaren Änderung von Strukturen und Inhalten. Für die Generierung neuer Knoten stellt die *Document*-Schnittstelle als zentrale Factory Konstruktoren für jeden Knotentyp zur Verfügung. Die Erzeugung eines neuen Element-Knotens mit dem Namen Lothar würde so aussehen:

```
elementLothar = document.createElement("Lothar")
```

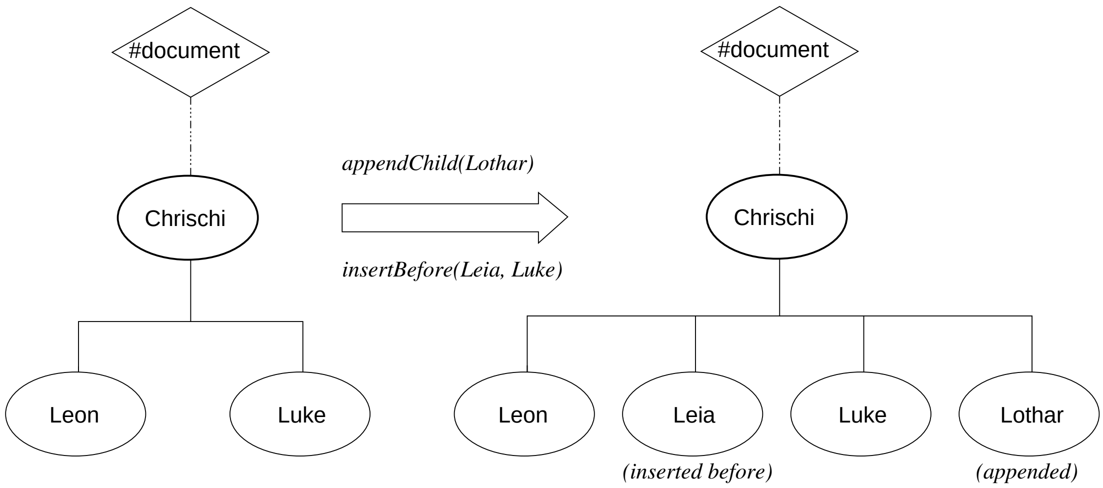

Abbildung 5: Anhängen und Einfügen von Knoten

Ein Knoten existiert also immer im Kontext seines Dokuments und darf auch nur einmal auftreten. Möchte man einen Knoten aus einem anderen Dokument integrieren, so muss er durch `document.importNode` als Kopie in den neuen Dokumentenkontext eingebunden werden. Eventuell muss hierbei auch eine Adaption an das neue Schema (DTD) erfolgen.

Das Node-Interface hält neben den Mitteln zur Navigation auch Methoden zur Manipulation von Knoten bereit, wobei der Wirkungsbereich immer die *children*-Eigenschaft Bezugsknoten ist. Die Methoden lassen sich in drei Gruppen einteilen:

- Das **Einfügen von Knoten** umfasst die Methode `appendChild` zum Anhängen von Knoten an das Ende der *children*-Liste sowie `insertBefore`, um das neue Kind vor einen spezifizierten Referenzknoten einzufügen. Abbildung 5 visualisiert die Wirkungsweise beider Operationen. In diesem Beispiel werden zwei neu erzeugte Elemente in das Dokument aufgenommen. Falls es sich um Elemente handeln würde, die bereits an einer Stelle der Hierarchie existiert hätten, so würde sich die Semantik der Methoden in ein *Verschieben* ändern. Das heißt, implizit würden die Elemente vor dem Einfügen aus ihrer Position gelöst werden, was aus der Identität jedes Knotens folgt.
  Um die Kopie eines Knotens zu ermöglichen, wäre vorher eine Duplizierung durch die Methode `cloneNode` erforderlich.

- Zum **Entfernen von Knoten** aus der Hierarchie dient die Methode `removeChild` (Abb. 6).

- Das **Ersetzen von Knoten** kann durch `replaceChild` vorgenommen werden (Abb. 6), hierbei gilt auch wieder die mögliche *Move*-Semantik oder die *COPY*-

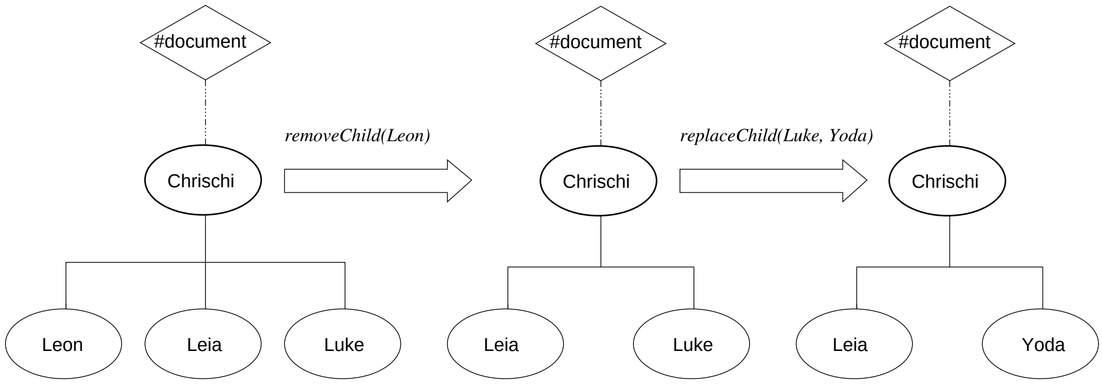

Abbildung 6: Löschen und Ersetzen von Knoten

Möglichkeit.

Es fällt auf, dass ein Umbenennen von Knoten in DOM nicht vorgesehen ist, dazu müsste man einen gleichartigen Knoten mit dem neuen Namen erzeugen, alle Eigenschaften und Beziehungen des Originalknotens übernehmen, diesen entfernen und zuletzt den neuen Knoten einfügen. In DOM3 wird es eine *renameNode*-Methode geben, die diesen Vorgang, wenn nötig, automatisiert.

Bisher wurden nur Operationen auf Einzelknoten betrachtet, zum Einfügen/Verschieben ganzer Teilbäume kann man die *DocumentFragment*-Schnittstelle benutzen. Dazu werden die entsprechenden Knoten in dieser Art Container gesammelt und schließlich wird das *DokumentFragment* als Parameter der Operation angegeben. Diese erkennt die Hülle und kümmert sich bei der Ausführung nur um die enthaltenen Knoten.
Die Methoden *importNode* und *cloneNode* verfügen außerdem über einen `deep`-Parameter, über den gesteuert wird, ob nur der Knoten selbst oder der gesamte Teilbaum kopiert wird.

Wie schon bei der Navigation angesprochen, existiert eine Sonderbehandlung von Attributen. Das gilt auch für die Manipulation, die über spezielle Methoden der Element-Schnittstelle erfolgen muss. Diese erstrecken sich über die Abfrage, das Setzen (Einfügen/Ändern) und das Löschen von Attribut-*Werten* wie auch von "echten" Attribut-*Knoten*, eine Übersicht dazu zeigt Tabelle 2[4].

Für die Anfrage und Änderung von Textinhalten werden ebenfalls zusätzliche Methoden angeboten. Diese werden in der allgemeinen Schnittstelle *CharacterData*

[4]Es existieren weiterhin alle Methoden aus DOM1, die keine Namespaces (NS) beherrschen. Eine gemischte Verwendung sollte jedoch ausgeschlossen werden

Funktion	Wert	Knoten
Abfrage	getAttributeNS	getAttributeNodeNS
Setzen	setAttributeNS	setAttributeNodeNS
Löschen	removeAttributeNS	removeAttributeNode

Tabelle 2: Zugriff und Manipulation von Attributen

definiert, von der die Knotentypen *Text* und *Comment* abgeleitet sind. Anstelle des Zugriffs auf den gesamten Knoteninhalt, kann hier eine Teilmenge über *substringData* und die Angabe von Offset und Anzahl der Zeichen extrahiert werden. Auch die allgemeinen Modifikatoren eines Knotens werden teilweise um die Parameter `offset` und `count` ergänzt, womit dann Änderungen an einer bestimmten Position und in einem festgelegten Umfang durchgeführt werden können.

Wichtig ist darauf zu achten, dass der Inhalt von Entities nicht verändert werden kann. Wenn eine solche Modifikation notwendig ist, müssen alle zugehörigen Entity-References durch einen Klon der Entity ersetzt und die gewünschten Änderungen auf jeden Klon angewendet werden.

3.1.4 XML-Manipulationen mittels SAX

Wie schon in der Einleitung dieses Abschnitts erläutert, ist *SAX* eher für die Verarbeitung und nicht für die Änderung von Daten vorgesehen. Hier soll jedoch kurz eine Idee vorgestellt werden, mit der einfache Update-Operationen auf XML-Dokumenten unter Verwendung von SAX möglich sind.
Die Abbildung 7 zeigt dazu einen Systementwurf, in dem von einem XML-Repositorium ausgegangen wird, das die Fähigkeiten zum Laden und Speichern von XML-Dokumenten besitzt und ihre Identifzierung (`oid`) sicherstellt.

Die zentrale Funktion zur Dokumentbearbeitung hat dabei ein kombinierter Parser/Generator, dessen Komponenten durch Pipelining verbunden sind.

Die Änderung eines Dokuments hat dabei folgenden Ablauf:

- Durch einen User oder eine Applikation wird eine Update-Anweisung an den Parser/Generator geschickt, die neben der Art des Updates (Insert, Delete, Update), einen Lokalisierungsausdruck zur Bestimmung des Zieles der Änderung (Dokument+Pfad) und eventuell den einzufügenden Inhalt enthält.

- Die SAX-Implementation fordert daraufhin aus der Dokumentenkollektion das

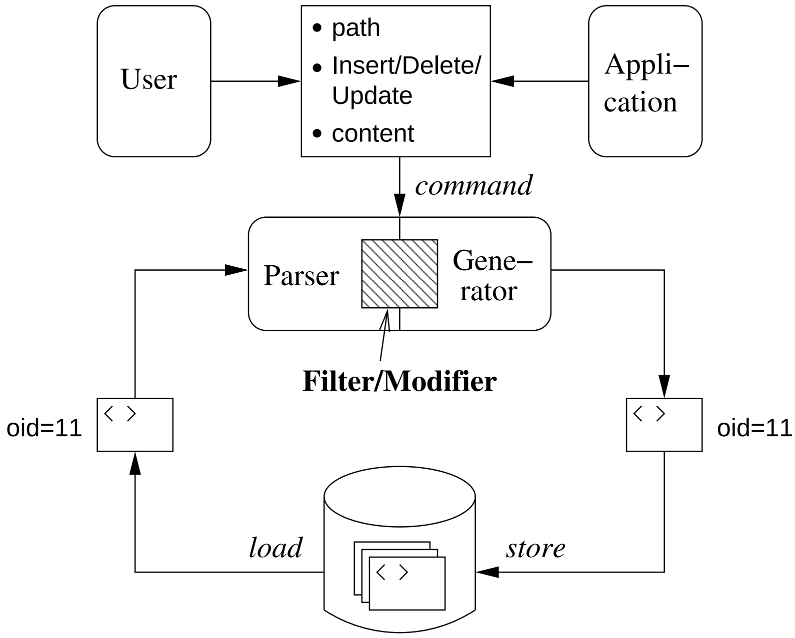

Abbildung 7: Systemvorschlag SAX-Datenmanipulation

entsprechende Dokument an, das innerhalb des Storage-Systems gesperrt werden sollte.

- Für den Bearbeitungsvorgang lassen sich zwei Zustände unterscheiden, die den zwischengeschalteten Filter/Modifier steuern, dazu wird ein lokaler Kontext für die aktuelle Position (Pfad) innerhalb der virtuellen Hierarchie bereitgestellt.

- Solange die Zielposition nicht erreicht ist, werden die erkannten Methoden des Parsers 1:1 an den Generator weitergeleitet und dort serialisiert.

- Wenn der passende Pfad erreicht wird, tritt der Modifier in Kraft, der die gewünschte Änderung wie folgt umsetzt:

  – *Insert:* Es werden zusätzliche Generator-Aufrufe mit den einzufügenden Werten erzeugt.

  – *Delete:* Durch einfaches Auslassen von Generator-Aufrufen kommen die zu löschenden Strukturen nicht mehr im Ergebnis vor.

  – *Update:* Hier werden entweder die Art der Generator-Methoden oder die Inhalte modifiziert, um die Ausgabe zu beeinflussen.

- Das ausgegebene Dokument wird wieder an das Storage-System zurückgeschickt, welches das Dokument anhand der mitgeführten `oid` erkennen sollte und eine Ersetzung des Originaldokuments vornimmt. Im Anschluss kann die Sperre wieder aufgehoben werden.

Die eigentliche Arbeit hat dabei der *Filter/Modifier* zu leisten, die je nach Art der Änderung auch sehr komplex aussehen kann. Für die Darstellung der Möglichkeit zur Datenmanipulation durch *SAX* sollte diese Übersicht jedoch genügen.
Eine Optimierung des Systems könnte dann erfolgen, wenn die unterliegende XML-Datenbank auch die Extraktion und Speicherung von Dokument-Fragmenten erlaubt. Damit könnten auch Transaktionsmechanismen angewandt werden, die in Abschnitt 2.2.5 vorgestellt wurden.

3.2 XML:DB - XUpdate

Die XML Update Sprache *XUpdate* ist ein Entwurf der *XML:DB Initiative for XML Databases*. Die *XML:DB* ist eine von der Industrie initiierte Unternehmung zur Entwicklung und Spezifizierung von Technologien des Datenmanagements in XML-Datenbanken. Des Weiteren sollen zu diesen Spezifikationen Referenzimplementierungen zur Verfügung gestellt werden wie auch eine Plattform für den Wissensaustausch zwischen den Entwicklern sowie den Anwendern. Neben XUpdate zählen auch die *XML Database API* und die *Simple XML Manipulation Language - SiXDML* zu den Projekten der *XML:DB*, auf die noch am Ende dieses Abschnitts eingegangen wird.

3.2.1 Grundlagen

Zum Zeitpunkt dieser Diplomarbeit wurden zwei Papiere innerhalb des *XUpdate*-Projektes mit dem Status *Working Draft* veröffentlicht, in [Mar00] werden Ziel und Anforderungen für eine XML-Update-Sprache abgesteckt, während die Spezifikation [LM00] Syntax und Semantik des Sprachvorschlags *XUpdate* beschreibt.
Die Anforderungen wurden sehr kurz gehalten und beziehen sich einerseits auf allgemeine Prinzipien des Entwurfs wie auch auf die zu unterstützenden Mechanismen zur Anfrage oder Änderungen von XML-Inhalten.
Die Spezifikation XUpdate ist vor allem dadurch geprägt, dass eine Änderung in Form eines wohlgeformten XML-Dokument ausgedrückt wird. Dies stellt insofern einen Vorteil dar, da die Werkzeuge zum Parsen und Erzeugen von XML-Dokumenten auch für die Verarbeitung von XUpdate-Ausdrücken verwendet werden können, auf der anderen Seite wirkt sich die XML-Syntax nachteilig auf die Komplexität und Lesbarkeit der Sprache aus.
Semantisch wurde der Sprachentwurf durch die W3C-Standards XPath (Version 1.0) und XSLT beeinflusst. Zur Auszeichnung der Sprachkonstrukte wird der Namensraum-URI `http://www.xmldb.org/xupdate` und gewöhnlich die ID *xmlns:xupdate* verwendet, so dass die Syntax sehr stark durch die typischen `<xupdate:...>`-Tags geprägt ist.

Das Datenmodell ergibt sich aus der Integration von XPath-Expressions, die im Wesentlichen auf das Infoset abgebildet werden können. Faktisch wurden in dieses Modell (Baum von Knoten) folgenden Knotentypen eingearbeitet: Elemente, Attribute, Text, Processing-Instructions und Kommentare. Dagegen fehlen Namensraumknoten und vor allem Dokument-Knoten (in XPath Wurzelknoten). In den *XML Update Language Requirements* wird zwar gefordert, dass die Sprache auch auf Mengen von Dokumenten operieren soll, die eigentliche Sprachbeschreibung lässt eine entsprechende Unterstützung jedoch vermissen. Weiterhin finden hier Referenzen und das Vorhandensein eines Schemas keine Beachtung.

3.2.2 Operationen

Eine Änderung kann aus einer oder mehrerer Update-Anweisungen bestehen und wird zur Gruppierung in ein *xupdate:modifikations*-Element eingeschlossen. Jede Update-Anweisung wird wiederum durch ein XML-Element repräsentiert. Gemein ist diesen Elementen das *select*-Attribut, über das die *Selektion* von Knoten in Form von Pfad-Ausdrücken aus XPath erzielt werden kann. Das Ergebnis des *select*-Ausdrucks ist nach der hier verwendeten XPath Version 1.0 ein *node-set*, also eine Menge von Knoten, die dann als Kontext in die nachfolgenden Operationen eingeht.
Des Weiteren beinhaltet XUpdate einen Mechanismus zur Variablenbindung von Objekten, die durch einen Pfad-Ausdruck berechnet werden können. Im nachstehenden Ausdruck wird die Variable *town* gebunden:

```
<xupdate:variable name="town"
    select="/adresses/address[0]/town"/>
```

und kann über den *value-of*-Ausdruck wieder ausgewertet werden:

```
<xupdate:value-of select="$town"/>
```

Über *value-of* können im Übrigen auch direkte Anfragen mittels XPath-Ausdrücken umgesetzt werden. Ein komplexeres Beispiel für den Variableneinsatz wird am Abschnittsende gezeigt.

Für die Erzeugung neuer Knoten werden (im Gegensatz zu XQuery) explizite Konstruktoren für jeden Knotentyp (`xupdate:element`, `xupdate:text`, usw.) bereitgestellt. Dazu wird der Name durch ein Attribut `name` festgelegt, während die Werte/Inhalte zwischen den Konstruktor-Tags angegeben werden. Der XUpdate-Konstruktor

```
<xupdate:element name="address">
  <xupdate:attribute name="id">2</xupdate:attribte>
  <town>San Francisco</town>
</xupdate:element>
```

würde folgendes Element samt Attribut und Inhalt erzeugen

```
<address id="2">
  <town>San Francisco</town>
</address>
```

Die eigentlichen Update-Operationen ermöglichen das Einfügen, Löschen, Ändern und Umbenennen von Knoten jeder Art. Hier soll nun eine Übersicht dazu gegeben werden.

- **Insert**
  Das hier verwendete Datenmodell nimmt generell eine Ordnung sämtlicher Knoten eines Dokuments an. Für das Einfügen von Knoten vor oder hinter jedes Kontext-Knoten des XPath-Ausdrucks, werden die Anweisungen *insert-before* bzw. *insert-after* benutzt. Dies macht allerdings für Attribut-Knoten keinen Sinn, die als ungeordnete Menge betrachtet werden.

- **Append**
  Während *Insert* eine *sibling*-Beziehung des Kontext-Knoten ergänzt, wird durch *Append* dessen *[children]*-Eigenschaft erweitert. So können also neue Tochter-Knoten angehängt werden, wobei optional durch ein *child*-Attribut die Position (Integer) des anzufügenden Knotens festgelegt wird. Ohne Angabe der Position wird der neue Knoten zum letzten Kind des ausgewählten Kontext-Knotens.

- **Update**
  Durch *Update* kann der Inhalt eines vorhandenen Knoten geändert werden.

  ```
  <xupdate:update select="/addresses/address[@id=1]
    /name/first">Jonathan</xupdate:update>
  ```

- **Remove**
  Diese Operation dient dem Löschen der selektierten Knotenmenge.

- **Rename**
  Das Umbenennen von Knoten ist nur für Elemente und Attribute möglich, dazu wird der neue Name als Element-Inhalt der *Rename*-Anweisung spezifiziert.

  ```
  <xupdate:rename
    select=\anf{/addresses}>Adressen</xupdate:update>
  ```

- **If**
  Eine bedingte Ausführung von Operationen wurde in *XUpdate* vorgesehen, ist aber in der Beschreibung noch offen geblieben.

Zur Darstellung der Kombinationsmöglichkeiten der Operationen soll folgendes Beispiel dienen, in dem die Kopie eines Knotens realisiert wird. Eine umfassende Beispielsammlung ist unter [Sta01] zu finden.

```
<xupdate:modifications version="1.0"
       xmlns:xupdate="http://www.xmldb.org/xupdate">

   <xupdate:variable name="state"
       select="/addresses/address[@id = 1]/state"/>
   <xupdate:insert-after
```

```
      select="/addresses/address[@id = 1]/country">
     <xupdate:value-of select="$state"/>
  </xupdate:insert-after>
</xupdate:modifications>
```

3.2.3 Die Projekte XAPI und SiXDML

XAPI Zu den frühesten Projekten der *XML:DB*-Gruppe gehört die *XML:DB Database API*, in dem es zur Aufgabe gemacht wurde, einen allgemeinen Mechanismus für den Zugriff auf XML-Datenbanken zu definieren. Das Ergebnis ist ein Interface ähnlich *DOM* oder *SAX*, dass die Implementierung von Anwendungen zur Speicherung, Rückgewinnung sowie Änderung und Anfrage von Daten in XML-Datenbanken erlaubt. Als zusätzlicher Mechanismus sollte auch die Möglichkeit von Transaktionen vorgesehen werden.
Die Grundstruktur eines solchen Systems soll anhand eines Beispiel (Abb. 8) veranschaulicht werden soll. Die Hülle des Gefüges stellt die *Database* dar, in die eine Hier-

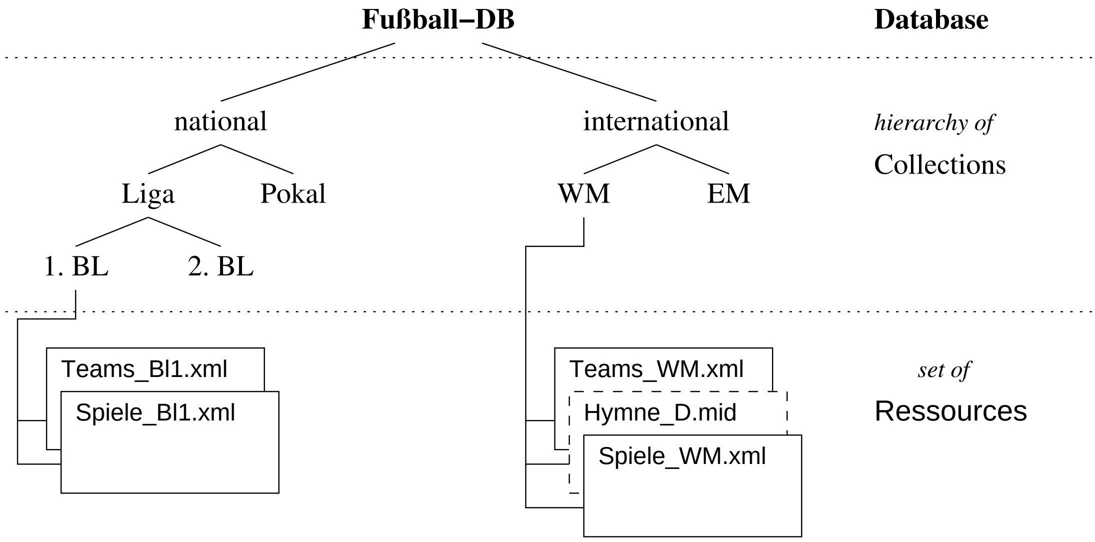

Abbildung 8: Beispieldatenbank für XAPI

archie von *Collections* aufgenommen werden kann.
Jede *Collection* stellt einen Sammelbehälter für eine Anzahl von *Resources* zur Verfügung, wobei es sich entweder um binäre oder XML Ressourcen handeln kann. Bei der *XMLResource* handelt es sich um den interessanteren Typ, da sie den Zugriff auf interpretierbare XML-Dokumente erlaubt, das kann in Form von Text wie auch durch *DOM*- oder *SAX*-API's erfolgen.
Zusätzlich kann eine *Collection* verschiedene *Services* anbieten, darunter zählen der elementare *CollectionManagemantService*, wie auch die Umsetzung von XPath- und XUpdate-Anfragen bis hin zum *TransactionService*.

Die Einbettung einer XUpdate-Anfrage durch einen *XUpdateQueryService* zeigt folgendes vereinfachtes Beispiel, das die Änderung gegen alle Dokumente der *WM_collection* ausführt:

```
String xupdate = "
<xupdate:modifications version="1.0"
    xmlns:xupdate="http://www.xmldb.org/xupdate">
   <xupdate:update select="//player[.='Miroslav Klose']">
      "Miroslav 'Miro' Klose"
   </xupdate:update>
</xupdate:modifications>";

XUpdateQueryService service =
   (XUpdateQueryService) WM_collection.getService(
      "XUpdateQueryService", "1.0");

service.update(xupdate);
```

SiXDML Seit dem 18.Februar 2002 ist die *SiXDML* (Simple XML Data Manipulation Language) ein Projekt der *XML:DB*-Initiative. Dieser Sprachvorschlag [Oba02] wurde von Dare Obasanjo entwickelt und vereinigt dabei *XUpdate* und die Fähigkeiten der *XAPI* auf der Sprachebene.
Genauer gesagt, wird das Datenmodell von *XUpdate* um die Komponenten *Collection*, *Document*, *Index* und *Schema* erweitert. Die *SiXDML* wurde vor allem durch SQL und die Datenmodellierungssprache DL/I aus dem hierarchischen Datenbanksystem IMS (IBM) inspiriert. Im Gegensatz zu *XUpdate* wurde, aufgrund der auch schon in Abschnitt 3.2.1 angedeuteten Nachteile, keine XML-Syntax verwendet.
Zu der erweiterten Funktionalität zählt die Erzeugung und Modifizierung von *Collections*, die den gleichnamigen Strukturen in der *XAPI* entsprechen. Zusätzlich kann über eine *Constrain*-Klausel die Validierung von Dokumenten einer Kollektion gegen ein Schema eingestellt werden.
Angelehnt an SQL wurde ein *Index*-Mechanismus in die Sprache aufgenommen, um den Zugriff von Anfragen auf Strukturen oder Werte zu optimieren.

```
CREATE INDEX val-index OF TYPE VALUE_INDEX
  WITH KEY=/name, ELEMENT=/document/person
  ON COLLECTION bookstore;
```

Durch den obigen Ausdruck wird eine Indexunterstützung für alle Dokumente der Kollektion *bookstore* angelegt, bei der die Namenswerte aller Elemente mit dem Pfad */document/person* indiziert werden.
Neben den vorgenannten Klauseln, enthält *SiXDML* die syntaktisch abgewandelten

Operationen aus *XUpdate*. Das für XAPI gezeigte Beispiel einer Änderung hätte hier folgendes Aussehen:[5]

```
REPLACE //player[.='Miroslav Klose']:text() WITH
%<%Miroslav 'Miro' Klose%>%
IN COLLECTION Fussball-DB/international/WM;
```

Außerdem existiert eine *SELECT*-Klausel, die eine XPath-Anfrage an ein Dokument oder eine Kollektion ausführt. Das Ergebnis kann optional durch XSLT transformiert oder anderweitig nachbearbeitet werden.

Neben der Sprachbeschreibung wurde von Dare Obasanjo auch ein zugehöriges, auf *XAPI* basierendes Interface und eine Implementation auf dem *eXcelon*-System umgesetzt.

[5]Die `%<%`,`%>%` sind hier XML-Tag-Kennzeichen, sondern Begrenzer für XML-Code innerhalb eines SiXDML-Ausdrucks

3.3 Logikbasierender Ansatz

Formalismen für Datenmanipulationssprachen im XML-Umfeld sind eine wichtige Voraussetzung für die Entwicklung und Standardisierung einer XML-Updatesprache. Zur Zeit sind diese theoretischen Grundlagen noch unterentwickelt. Zu diesem Schluss kommen auch die Autoren Liu und Hsu in ihren Artikeln [LH01] und [LH02]. Darin wird auch die Notwendigkeit verdeutlicht, neben dem Retrieval von XML-Dokumenten auch die Änderung sowohl von Inhalt als auch Struktur zu unterstützen. Sie stellen dafür einen Ansatz für eine XML Anfrage- und Änderungssprache basierend auf einem logischen Modell vor, der hier im weiteren erläutert werden soll.

3.3.1 Einführung

Gebräuchliche Formalismen für die Beschreibung von Anfragesprachen in relationalen Modellen sind der algebraische Formalismus, auch Relationenalgebra genannt, und der logikbasierende Formalismus, der im allgemeinen auch als Relationenkalkül bezeichnet wird.

In der Relationenalgebra werden Anfragen durch Anwendung spezieller Operatoren (Projektion π, Selektion σ, Verbund $\bowtie$, $\ldots$) auf den Relationen ausgedrückt. Durch die Schachtelung dieser Operationen erhält dieser Formalismus einen operationalen Charakter.

Einen deskriptiveren Ansatz stellt das Relationenkalkül dar, dem eine Form der Prädikatenlogik 1. Ordnung zugrunde liegt. Hierbei entsprechen die Datenbankinhalte den Belegungen von Prädikaten, und Anfragen den daraus abgeleiteten Prädikaten. Der beschreibende Charakter des Relationenkalküls wird dadurch geprägt, dass bei der Definition der abgeleiteten Prädikate nur angegeben wird, was zum Ergebnis gehören soll, und nicht wie es berechnet wird.

Allgemein hat eine Anfrage des Relationenkalküls folgende Form:

$$\{f(\bar{x}) \mid p(\bar{x})\}$$

mit

$$\bar{x} = \{x_1 : D_1, \ldots, x_n : D_n\}$$

Dabei ist:

- $\bar{x}$ eine Menge von freien Variablen und D_i der Wertebereich eines Datentyps oder eine Kollektion von Datenbank-Objekten an die x_i gebunden ist.

- f die Ergebnisfunktion über $\bar{x}$, die im einfachsten Fall eine Variablenangabe, also die Identitäsfunktion, aber auch ein Tupelkonstruktor sein kann.
- p das Selektionsprädikat über $\bar{x}$, dessen Formel aus atomaren Formeln und prädikatenlogischen Operatoren $(\vee, \wedge, \exists, \dots)$ gebildet werden kann. Atomare Formeln bestehen aus Termen zusammen mit den Prädikaten der Datentypen $(\leq, \geq, \dots)$ und speziellen Datenbankprädikaten, die den Bezug zur aktuellen Datenbank herstellen.

Die Berechnung des Ergebnisses läuft in zwei Schritten ab. Zuerst werden alle Belegungen der freien Variablen in $\bar{x}$ bestimmt, für die das Selektionsprädikat p wahr wird. Darauf wird die Ergebnisfunktion f auf die Werte der ermittelten Belegungen angewendet und so die Ergebnismenge gebildet.

Relationale Anfragesprachen beinhalten meist einen der obigen Formalismen, SQL sogar beide. Die vorherrschenden Relationenkalküle sind das Bereichs- und das Tupelkalkül, auf die hier jedoch nicht näher eingegangen werden soll.

Intuitiv könnte man nun versuchen, diese relationalen Vorlagen auf das Gebiet von XML zu übertragen. Eine direkte Anwendung der Formalismen von relationalen Anfragesprachen für XML-Anfragen ist jedoch aufgrund der unterschiedlichen Datenmodelle (Relational vs. Baum) nicht möglich.
Das W3C hat in [XQFS01] begonnen, einen algebraischen Formalismus für die vorgestellte XQuery-Anfragesprache aufzustellen. Zur Zeit basiert dieser jedoch noch auf einem spezifischem Datenmodell, der in der weiteren Entwicklung dem XQuery-Datenmodell [XQDM01] angepasst werden soll.
Der hier vorgestellten Ansatz, genannt Pfad-Prädikaten-Kalkül beruht direkt auf dem XML Dokumenten-Modell sowie dem XPath-Standard. Er kann als verallgemeinerte Form des Relationenkalküls angesehen werden und zeichnet sich durch folgende Eigenschaften aus:

- Atomare Formeln sind *Element-* und *Pfad-Prädikate*.
- Direkte Nutzung des XML-Baummodells, anstelle der indirekten Abbildung auf spezifische Datenbankmodelle, dadurch Vermeidung von potentiellen Risiken durch Inkonsistenzen mit dem XML Infoset Modell.
- Elegante Integration von Änderungen von Struktur und Inhalt in XML Anfragen auf einer höheren Ebene in knapper, deklarativer Art und Weise.
- Unterstützung von XML-Schema: Datentypen für MM-Dokumente (räumliche, zeitliche und visuelle Datentypen und Anfrage-Konstrukte können in eine XML Datenmanipulationssprache für MM XML Dokumente einbezogen werden. (Beispiele für MPEG-7 und MPEG-21 in [LH02] aufgezeigt).

In den folgenden Abschnitten soll nun der Sprachvorschlag mit seinen einzelnen Grundbausteinen dargestellt werden.

3.3.2 Pfad-Prädikaten-Kalkül

Als elementares Konzept wurde das Pfad-Prädikaten-Kalkül in die Multimedia Dokument Query Language *MMDOC-QL* eingebettet, bei dem Formeln eingeschränkte Formen der PL1 sind. Für das Aufstellen von logischen Aussagen über Elementen in einem XML-Dokumentenbaum wurden zwei grundlegende Prädikate entworfen, das *Element-Prädikat* und das *Pfad-Prädikat*.
Es wird nun die Rekursive Definition aller zulässigen Formeln vorgenommen:

Allgemeine Formeln im *Pfad-Prädikaten-Kalkül* haben folgende Form:

$$P(x_1, x_2, \ldots, x_n, c_1, c_2, \ldots, c_m, t_1, t_2, \ldots, t_p, a_1, \ldots, a_q, d_1, \ldots, d_r)$$

oder kurz

$$P(\bar{x}, \bar{c}, \bar{t}, \bar{a}, \bar{d})$$

wobei

- $\bar{x}$, $\bar{c}$, $\bar{t}$, $\bar{a}$, $\bar{d}$ freie Variablen mit folgender Bedeutung sind:
  $\bar{x}$ - Attribute [6]
  $\bar{c}$ - Inhalte (element <u>c</u>ontent)
  $\bar{t}$ - Namen (element <u>t</u>ag names)
  $\bar{a}$ - Positionen (element <u>a</u>dresses)
  $\bar{d}$ - Typen (element <u>d</u>atatypes)
- Wie gewohnt werden Variablen als "frei" bezeichnet, wenn sie nicht durch einen Quantor ($\forall$ bzw. $\exists$) gebunden sind (ansonsten eben "gebunden").
- Anfragen sind in diesem Formalismus gleich bedeutend mit der Bestimmung aller Belegungen der freien Variablen, für die $P(\bar{x}, \bar{c}, \bar{t}, \bar{a}, \bar{d})$ den Wert ""wahr"" annimmt.

Atomare Formeln sind wie folgt definiert:

1. E ist ein *Element-Prädikat* $E(x_1, x_2, \ldots, x_n, c, t, a)$, wobei $x_1, x_2, \ldots, x_n, c, t, a$ Variablen oder Konstanten sind.
   <u>Semantik:</u> Das Element mit dem Namen t an der Position a mit den Attributen $x_1, x_2, \ldots, x_n$ enthält den Inhalt c.

[6] Genauer Attribut-Werte

2. *mm-operator* ist ein Multimedia-Operator *mm-operator*$(x_1, x_2, x_3, \ldots, x_m)$, wobei $x_1, x_2, x_3, \ldots, x_m$ Konstanten, Positions-Variablen oder Typ-Variablen sind.
   Der *mm-operator* ermöglicht die Auswertung von Beziehungen zwischen Multimedia-Objekten (basierend auf XML-Schema), z.B. `OVERLAP(element1: RegionLocaterType, element2: RegionLocaterType)` für Berechnung des Schnitts zweier Regionen.

3. $x \circ y$, wobei $\circ$ ein arithmetischer Vergleichsoperator ($\leq$, $\geq$, $\neq$ usw.) ist und x, y sind entweder Konstanten, Attribut-Variablen oder Typ-Variablen.

4. *TYPEP* ist ein Typ-Prädikat *TYPEP*(x) wobei x entweder eine Konstante, Typ-Variable oder Positions-Variable ist. Es wird benutzt um eine prädikatenlogische Aussage über den Datentyp eines Elements zu formulieren. (Semantik: ??Holger??)

Logische Formeln werden rekursiv aus den atomaren gebildet:

1. *Boole'sche Formel*: Wenn P_1 und P_2 wohlgeformte Formeln des Pfad-Prädikaten-Kalküls sind, dann sind auch $(P_1 \wedge P_2)$, $(P_1 \vee P_2)$ und $\neg P_1$ wohlgeformte Formeln. (Semantik wie gewohnt.)

2. *Pfad-Prädikate*: Wenn P_1 und P_2 wohlgeformte Formeln mit wenigstens einem Element-Prädikat sind, dann ist auch $(P_1 \diamond P_2)$ eine wohlgeformte Formel.
   $\diamond$ ist dabei ein Achs-Operator angelehnt an XPath. Beispiele dafür sind Eltern-Kind-Beziehung wie `INSIDE`, `CONTAINING` und Vorgänger/Nachfolger-Beziehungen (`BEFORE`, `AFTER`).
   Ein Pfad-Prädikat könnte durch mehrfache Verschachtelung so aussehen:

```
(<bibref> INSIDE (<gcapaper> CONTAINING
  (<fname> CONTAINING "Peiya") AND
  (<surname> CONTAINING "Liu")))
```

3. *Quantifizierte Formel*: Wenn P eine Formel ist, so sind auch $\forall x P$ und $\exists x P$ Formeln. Die Semantik entspricht der allgemein bekannten aus der Prädikatenlogik, wobei also freie Variablen an die Quantoren gebunden werden.

Um in diesem Pfad-Prädikaten-Kalkül *syntaktisch sichere Anfragen* zu gewährleisten, müssen die Domänen der Variablen endlich sein, d.h. es existiert eine endliche Anzahl an Attributen, Namen, Typen usw. Der folgende Sprachvorschlag beschränkt Variablen zusätzlich mittels regulärer Ausdrücke an bestimmte Muster.

3.3.3 Die XML Dokumenten Manipulations Sprache MMDOC-QL

Das zuvor beschriebene Pfad-Prädikaten-Kalkül wurde zur Beschreibung von XML Anfragen und Änderung in die Sprache *MMDOC-QL* eingebettet. Dazu werden vier Klauseln (*GENERATE*, *INSERT*, *DELETE*, *UPDATE*) eingeführt, um prädikatenlogische Aussagen in Form der oben beschriebenen Element- und Pfad-Prädikaten zu beschreiben. Jede Anfrage/Änderung beginnt mit einer dieser vier Klauseln gefolgt von einem *PATTERN-FROM-CONTEXT*-Block, wobei Variablen durch ein `%` gekennzeichnet werden (z.B. `%objectid`).
Mittels der *PATTERN*-Klausel können Bedingungen an die Domänen der Variablen durch reguläre Ausdrücke beschrieben werden.
Die *FROM*-Klausel dient der Festlegung der zu verarbeitenden Dokumente und wird paarweise mit der *CONTEXT*-Klausel verwendet, wobei auch mehrfache Dokumentenquellen möglich sind.
Die eigentliche Beschreibung prädikatenlogischer Aussagen über den XML Elementen erfolgt in der *CONTEXT*-Klausel, wofür Formeln des Pfad-Prädikaten-Kalküls benutzt werden. Ein Element-Prädikat $E(x_1, x_2, \dots, x_n, c, t, a)$ wird dazu in folgende Syntax überführt:

```
(<%t> WITH attr1=%x1,...,attrn=$xn AT %a CONTAINING %c)
```

Die vier Basis-Klauseln können in zwei Gruppen eingeteilt werden, Anfragen werden durch `GENERATE` ermöglicht während Änderungen mit *INSERT*, *DELETE* und *UPDATE* umgesetzt werden. Die Anwendung dieser Klauseln soll nun mit Hilfe von Beispielen verdeutlicht werden.

GENERATE Diese Anfrageoperations-Klausel fungiert als Ergebnis-Konstruktor, in dem der auszugebende XML-Code einfach durch Variablen ergänzt werden kann. Folgendes Beispiel zeigt die Anwendung der verschiedenen Klauseln wie auch den Einsatz von MultiMedia-Operatoren (MEMBERP, OVERLAP, TRAJECTORY). Über die PATTERN-Klausel wird in diesem Fall die Domäne der Variable `%objectid` auf Strings begrenzt, die dem Muster {MR"gefolgt von zwei Ziffern} entsprechen.

```
GENERATE: <LIST>
            <Videoobject>%objectid</videoobject>
            <ShowUpTime>%t<ShowUpTime>
          </LIST>
PATTERN: {"MR"[0-9][0-9]/%objectid}
         {<region> ... </region>/%focusarea}
FROM:    mpeg7video.xml
```

```
CONTEXT  ((&lt;Segment&gt; WITH xsi:type="MovingRegionType",
           id=%objectid AT %movingregion)
         CONTAINING
         (<SpatioTemporalLocator> DIRECTLY CONTAINING
            (<MediaTime> AT %x)))
         AND MEMBERP(%t,%x)
         AND OVERLAP(TRAJECTORY(%movingregion,%t),
                        %focusarea))
```

INSERT Zum Einfügen von XML-Komponenten verwendet *INSERT* die gleiche Form der Ergebniskonstruktion wie *GENERATE*, wobei im unteren Beispiel der orthogonale Einsatz des Pfad-Prädikaten-Kalküls für das Einfügen von Hypertext-Links veranschaulicht wird:

```
INSERT:  (<a> WITH href="#"%refloc)
         DIRECTLY CONTAINING (<bibref> WITH refloc=%refloc)
PATTERN: {*[0-9][0-9]/%refloc};
FROM:    mygcapaper.xml
CONTEXT: {(<bibref> with refloc=%refloc)
          AND (<bibitem> with id=%refloc)};
```

DELETE Diese Spezifikation für das Entfernen von Dokumentteilen entspricht logisch der *INSERT*-Klausel. Im nächsten Beispiel werden alle `href`-Links mit der Position `%a` gelöscht, die durch die *CONTEXT*-Klausel gebunden wurden.

```
DELETE:  (<a> WITH href="#"%refloc)
         DIRECTLY CONTAINING (AT %a)
PATTERN: {*[0-9][0-9]/%refloc};
FROM:    mygcapaper.xml
CONTEXT: {(<bibref> with refloc=%refloc AT %a)
          AND (<bibitem> with id=%refloc)};
```

UPDATE Die *UPDATE*-Klausel ist eigentlich eine spezifische Kombination von INSERT und DELETE. Hier wird ähnlich der SQL-Notation eine Sub-Klausel `SET` benutzt, um die Zuweisungen für die Ziel-Elemente festzulegen. Die Änderungen einer e-Mail-Adresse könnte so aussehen:

```
UPDATE:  ( {<email> DIRECTLY CONTAINING %e-mail):
         SET (%e-mail = "peiya.liu@scr.siemens.com"))
PATTERN: {};
FROM:    mygcapaper.xml
CONTEXT: { TRUE };
```

3.4 XQuery

3.4.1 Konzepte

Die XML Anfragesprache XQuery soll in diesem Rahmen zur Untersuchung existierender Ansätze für Änderungssprachen zum Vergleich der Konzepte benutzt werden. Dazu wird hier keine umfassende Beschreibung von XQuery erfolgen, die bei Bedarf in [KM02] oder [Ros01] nachgelesen werden kann, sondern es sollen nur die Kernkonzepte aufgezählt werden.
XQuery [XQ01] ist eine funktionale Sprache, in der eine Anfrage durch einen Ausdruck repräsentiert wird. Diese Ausdrücke können dabei völlig orthogonal geschachtelt werden. Die Basis solcher "Expressions" ist die Syntax von XPath 1.0, die um zusätzliche Komponenten erweitert wurde. Ein komplexerer Pfad-Ausdruck, zur Suche aller Figuren, die durch `<figref>`-Elemente im Kapitel "Frogs" des Dokuments "zoo.xml" referenziert werden, hat folgendes Aussehen:

```
document("zoo.xml")/chapter[title = "Frogs"]//figref/
  @refid->fig/caption
```

Solche Ausdrücke können durch sogenannte *FLWR-Expressions* gebunden werden:

```
FOR var IN expr
LET var IN expr
WHERE expr
RETURN expr
```

`FOR` und `LET` generieren dabei eine Liste von Tupeln aus gebundenen Ausdrücken, wobei die Dokumentenordnung erhalten bleibt.
Durch die `WHERE`-Klausel können Tupel durch die Anwendung eines Prädikats ausgefiltert werden.
`RETURN` wird für jedes "überlebende" Tuple ausgeführt, und generiert dabei eine geordnete Liste von Ergebnisbestandteilen. Für die Ergebnisbeschreibung können *Element-Konstruktoren* verwendet werden, die aus einem Start- und End-Tag bestehen und sowohl normalen XML-Inhalt als auch Ausdrücke enthalten können.

Weitere Konzepte werden nun aufgezählt:

- Projektion, Iteration, Selektion
- Verbund
- Restrukturierung
- Gruppierung, Aggregation, Sortierung

- Datentypen (einfach, komplex, benutzerdefiniert) und Sequenzen
- Quantifizierte Ausdrücke
- Funktionen (built-in, user defined)
- Knotenidentität und Referenzen

3.4.2 Arbeitsgebiete

Durch die XML Query Working Group wurde eine Reihe von Workings Drafts verabschiedet, zu denen hier eine kurze Übersicht gegeben wird.

XML Query 1.0 and XPath 2.0 Data Model Das Datenmodell zu XQuery [XQDM01] legt fest, welche Informationen für eine Anfrage oder einen einen XPath Prozessor verfügbar sind.
Es enthält:

- Das Infoset mit diesen Erweiterungen:
  - Unterstützung für XML Schema Datentypen (PSVI)
  - Unterstützung von Dokumentenkollektionen
  - Unterstützung von Referenzen
- Ein konstruiertes Baummodell mit benannten Knoten und Knotenidentität
- Die Abbildung des Infoset auf das XQuery Datenmodell

XML Query 1.0 Formal Semantics Dieses Dokument [XQFS01] stellt den Versuch einer Formalisierung von XQuery dar, der ursprünglich als Algebra bezeichnet wurde. Sie dient der Definition der XQuery Semantik und zur Unterstützung der Anfrageoptimierung.
Es werden unterschieden:

- Die statische Semantik, als Repräsentation von Typableitungen (XQuery/FS Ausdrücke $\longrightarrow$ Typen)
- Die dynamische, auch operationale Semantik, als Regeln zur Wertableitung (XQuery/FS Ausdrücke $\longrightarrow$ Werte des XQuery Data Models)

XQueryX Die XML Query Language ist selbst nicht in XML Form gehalten, kann aber auf die XML-Syntax abgebildet werden, als *XQueryX*.

Dies ist für folgende Szenarien vorteilhaft:

- Wiederverwendung eines Parsers zur Sprachverarbeitung
- Anfragen auf Anfragen
- Erzeugung von Anfragen
- Einbettung von Anfragen in XML Dokumenten

3.5 Weitere Ansätze

Neben den hier vorgestellten Modellen, gibt es noch weitere Ansätze für die Einbindung einer Updatesprache in das XML-Umfeld, die jedoch durch bestimmte Eigenschaften die Umsetzung sehr erschweren oder gar unmöglich machen.

3.5.1 Objektorientierte Datenmodelle

Für die Nutzung von Objektorientierten Datenbanken spricht die Möglichkeit Strukturhierarchien zu modellieren, wie auch teilweise die Unterstützung von Transaktionen und eine verhältnismäßig (im Vergleich zu XML) lange Entwicklungs- und Evaluierungszeit. Einige XML-System-Implementationen wie eXcelon basieren auch auf entsprechenden Datenbanksystemen. (ObjectStore)
Dagegen spricht allerdings, dass das XML-Datenmodell vollständig auf das objektorientierte abgebildet werden muss, und so auch mit Inkonsistenzen zu rechnen ist. Zum Beispiel wird in OODM keine Ordnung von Objekten unterstützt und meist strenge Klassen- und Typhierarchien gefordert, während die Schemaunterstützung in XML relativ "weich" behandelt wird.
Eine Übersicht zu Modellen, Systeme und Anfragesprachen für Objektorientierte Datenbanksysteme findet sich in [Heu97].

3.5.2 Hierarchische Modelle

Hierarchische Modelle existieren schon seit Jahrzehnten und haben den großen Vorteil, XML Daten so gut wie direkt aufnehmen zu können, da sie selbst durch Baumstrukturen aufgebaut werden.
Der Vertreter dieser Modelle ist das IMS-System von IBM, das eine Datendefinitionssprache (DDL) wie auch eine Datenmanipulationssprache (DL/I) enthält. Leider entspricht die DL/I weniger einer Änderungs- sondern eher einer Programmiersprache. Ähnlich *DOM* erfolgt hier eine Navigation durch die Datensatz-Hierarchie und die anschließende Änderung durch `insert`, `delete` oder `replace`. Zusätzlich wurde auch eine Art Sperrmechanismus vorgesehen, der Probleme bei Änderungen in Multi-User-Umgebungen ausschließen soll.

3.5.3 Lorel

Zuletzt soll das *LORE*-Projekt der Stanford University (`http://www-db.stanford.edu/lore`) erwähnt werden, in dem die datenbankorientierte Anfragesprache *Lorel* als Erweiterung der *OQL* entstanden ist.
Ausgehend von *OEM*, einem graphenorientierten Modell für semistrukturierte Daten, ist die Abbildung auf das XML-Datenmodell erfolgt. Neben der Anfragefunktionalität wurde hier eine Update-Sprache integriert, zur der folgende Operationen gehören:

- Erzeugung von Objekten

  $new\_oem(val\text{-}type, value) \rightarrow object$

- Zuweisung von Namen zu Objekten

  ```
  name <name> := <expression>
  ```

- Updates auf Objekten

  ```
  update P := <expression>
  from <from-clause>
  where <where-clause>
  ```

Zusätzlich ist auch ein *Bulk loading* von komplexen Objekten aus Dateien vorgesehen, durch das auch existierende Objekte durch neue Anteile ergänzt werden können.

3.6 Vergleich

Abschließend soll hier ein kurzer Vergleich der betrachteten Ansätze gezogen werden, der in der Tabelle 3 festgehalten wurde.

Kriterium	DOM	XUpdate	MMDOC-QL	XQuery
Deskriptiv	-	+	+	+
Adäquat	+	-	-	+
Optimierbar	-	+	+	+
Orthogonal	-	-	+	+
Datenmodell	Infoset	eingeschränkt	Infoset	erw. Infoset
Schemas	-	-	-	+
Sprache	Interface/PL	(XML,XPath)+	eigene	XPath+
Transaktionen	-	-	-	-

Tabelle 3: Vergleich der Sprachansätze

Das Document Object Modell kann als Interface natürlich schlecht mit den "echten" Sprachansätzen mithalten, stellt aber einen direkten Bezug zum Infoset dar und ermöglicht vielfältige Manipulationen der Baumstrukturen.
Die Einschätzung von XUpdate fällt etwas schwer, da in der Beschreibung nur die Syntax und kaum die Semantik betrachtet wird. Dazu war es erforderlich, sich durch die zugehörige Newsgroup zu "wühlen". Es ist aber deutlich geworden, dass XUpdate einigen Einschränkungen unterliegt und Ungenauigkeiten enthält.
Die MMDOC-QL basiert auf einem formalen Modell und kann damit die allgemeinen Anforderungen an Anfragesprachen erfüllen. In Anbetracht von Änderungen wird dieses Modell jedoch nicht konsequent weitergeführt, sondern diese nur durch Beispiele angedeutet.
XQuery stellt einen sehr vielseitigen und gut durchdachten Sprachansatz dar, dessen Syntax auch für "Neulinge" leicht zu verstehen sein sollte. Für die vollständige Durchdringung aller Details muss man sich allerdings viel Zeit nehmen. Da hier zur Zeit noch keine Änderungsoperationen eingearbeitet wurden, muss man XQuery in diesem Vergleich gesondert betrachten.

Auffällig ist, dass in allen Ansätzen das Konzept der Transaktion fehlt, und eine Validierung von Änderungen gegen ein Schema unbeachtet bleibt.
Einige Implementationen stellen diese Funktionalität schon zur Verfügung, womit deutlich wird, dass hier der typische Fehler in der Softwareentwicklung gemacht wird, erst die Umsetzung vorzunehmen und im Anschluss eine allgemeine Konzeption zu beschreiben.

4 Konzeption einer XML-Updatesprache

4.1 Grundlagen zu Änderungsoperationen

In diesem Abschnitt sollen Grundlagen für die Definition von XML-Änderungsoperationen geschaffen werden. Dazu wird ausgehend von den bekannten Formalismen und Grundprinzipien des Relationenmodells der Versuch einer Übertragung dieser Konzepte auf das XML-Datenmodell gestartet.

4.1.1 Relationale Änderungsoperationen

Zur Formalisierung kann man jeden Zustand einer Datenbank allgemein als Kollektion von Mengen verstehen, also etwa einer Sammlung von Schemas und zugehörigen Relationen, die wiederum aus Mengen von Tupeln bestehen. Dann entspricht eine Änderung der Datenbank semantisch dem Übergang eines Datenbankzustands in einen neuen. Um eine solche Zustandsänderung zu erreichen, sind nun zwei Varianten denkbar:

1. Die vollständige Ersetzung der Datenmengen durch andere.
2. Die Verwendung von *lokalen* Änderungen als Änderungsmodell.

Während die erste Form eher theoretische Bedeutung hat, sind lokale Änderungen in realen Umgebungen besser geeignet, folglich wird hier nur Punkt 2 ausgeführt.
Da hier also von Mengen als Zustände von Datenbanken gesprochen wird, wären die allgemeinen Mengenoperationen mögliche lokale Änderungen dieses Modell. Dazu gehören

- Das Einfügen von Elementen in die Mengen (*insert*).
- Das Entfernen von Elementen aus den Mengen (*delete*).

Zu einer Datenbank gehören neben Schemas und Instanzen auch eine Menge von Integritätsbedingungen, die bestimmte Forderungen an die Tupelmengen stellen. Da nach der Semantik von Anfragen die Verarbeitung von Datenmengen betrachtet wird, soll auch für Änderungen das gleichzeitige Einfügen oder Löschen von Mengen zugrundegelegt werden. Dieses Vorgehen beinhaltet auch den Vorteil, dass so inkonsistente Zustände vermieden werden, die bei der Einzelanwendung von Änderungen entstehen können.
Hier muss allerdings hinzugefügt werden, dass bisherige relationale Sprachen nur eine Art von Änderungen auf nur einer Relation erlauben. Zum atomaren Löschen und Einfügen eines Elements wird eine 3. Operation eingeführt:

- Das Ersetzen von Elementen in Mengen (*replace*).

Eine Sequenz zur Kombination der drei Arten von Änderungsoperationen wird in einer *Transaktion* zusammengefasst. Diese wird dann wieder als atomarer Zustandsübergang angesehen, wobei auftretende Inkonsistenzen innerhalb der Transaktion unbeachtet bleiben, und nur die Zustände vor und nach der Transaktion den Integritätsbedingungen folgen müssen.

Formalismus Zur Präzisierung der eben vorgestellten Betrachtungen soll eine Formalisierung der drei Änderungsoperationen erfolgen, die wie auch die obigen Grundlagen [HS00] entnommen wurde.

Gegeben sei:
Das Datenbankschema $\mathcal{S} := \{S, \Gamma\}$ mit Relationen $S := \{\mathcal{R}_1, \ldots, \mathcal{R}_p\}$ und Integritätsbedingungen Γ.
Eine gültige Datenbank $d(S) := \{r_1(\mathcal{R}_1), \ldots, r_i(\mathcal{R}_i), \ldots, r_p(\mathcal{R}_p)\}$ und zwei Tupel t und t' über $\mathcal{R}_i$.

Eine *Update-Operation* ist Abbildung $u : \mathbf{DAT}(\mathcal{S}) \longrightarrow \mathbf{DAT}(\mathcal{S})$ als spezielle Notation des Zustandsübergangs im Relationenmodell.

Die drei *Update-Operationen* werden wie folgt definiert:

- $u(d) := \texttt{insert}\ t\ \texttt{into}\ r_i(\mathcal{R}_i)$

$$d \longmapsto \begin{cases} d' := \{r_1, \ldots, r_i \bigcup \{t\}, \ldots, r_p\} & \text{falls } d' \in \mathbf{DAT}(\mathcal{S}) \\ d & \text{sonst} \end{cases}$$

- $u(d) := \texttt{delete}\ t\ \texttt{from}\ r_i(\mathcal{R}_i)$

$$d \longmapsto \begin{cases} d' := \{r_1, \ldots, r_i - \{t\}, \ldots, r_p\} & \text{falls } d' \in \mathbf{DAT}(\mathcal{S}) \\ d & \text{sonst} \end{cases}$$

- $u(d) := \texttt{replace}\ t \rightarrow t'\ \texttt{in}\ r_i(\mathcal{R}_i)$

$$d \longmapsto \begin{cases} d' := \{r_1, \ldots, (r_i - \{t\}) \bigcup \{t'\}, \ldots, r_p\} & \text{falls } d' \in \mathbf{DAT}(\mathcal{S}) \\ d & \text{sonst} \end{cases}$$

Eine *Transaktion* ist dann eine Folge von *Update-Operationen*, eine Abbildung von $\mathbf{DAT}(\mathcal{S})$ in $\mathbf{DAT}(\mathcal{S})$.

4.1.2 Änderungen im XML-Datenmodell

In Analogie zum Relationenmodell könnte man den Zustand von XML-Datenbanken als Kollektion von Mengen betrachten, wobei die innere Struktur vorerst unberücksichtigt bleiben soll. Eine Änderung dieses Zustands könnte entsprechend durch die vollständige Substitution der bestehenden Datenmengen durch andere beschrieben werden. Im übertragenen Sinne machen das auch einige bestehende XML-Verwaltungssysteme, indem sie XML-Dokumente nur im Ganzen einlesen oder wieder entfernen können. Während diese Vorgehensweise für bestimmte Einsatzzwecke wie der Dokumentenverarbeitung ausreichend ist, stellt sie Anwendungen mit datenzentrierter Sichtweise vor große Probleme. Dazu kommt, dass Konzepte wie Nebenläufigkeit (Transaktionen) und Optimierbarkeit dann nicht mehr umsetzbar wären.

Für datenzentrierte Anwendungen sollten also inkrementelle Änderungen der Datenbestände möglich sein, wobei an dieser Stelle wieder ein Rückschluss zum Relationenmodell gezogen werden kann. Vergleichbar dazu würde das Einfügen und Entfernen von Elementen aus den Datenmengen als Grundoperationen für jede mögliche lokale Änderung genügen.

Zu beachten sind hier jedoch die unterschiedlichen Datenmodelle. Relationale Änderungsoperationen nehmen ein Tupel als Basiseinheit der einzelnen Datenmengen (Relationen) an, wie auch ein festes Datenbankschema mit zugehörigen Integritätsbedingungen. Das XML-Datenmodell basiert nun auf geordneten Bäumen von Knoten mit dynamischer Tiefe und ist im allgemeinen Fall schemalos. Zur Definition der Änderungsoperationen reichen deshalb einfache Mengenoperationen nicht mehr aus. Im Änderungsmodell müssten zumindest die flachen Mengen durch eine Schachtelung der Tabellen und einen *List-Konstruktor* zur Modellierung der Ordnung erweitert werden. Dafür sind dann jedoch wieder spezifische Konstruktoren und Operationen erforderlich.

Eine diesbezügliche Konzeption erfolgt in Abschnitt 4.2 in normalsprachlicher Form, während ein formaler Ansatz in Abschnitt 3.3 vorgestellt wurde, der sich jedoch hauptsächlich mit den Anfrageformalismen für XML-Dokumente beschäftigt und Änderungen nur in einer knappen Übersicht dargestellt werden.

Ein weiterer Unterschied besteht darin, dass bei relationalen Änderungsoperationen nur die Änderung der *Basisrelationen*, nicht aber der zugehörigen Schemata erfolgt. Semistrukturierte Daten wie XML beinhalten jedoch implizit Struktur- und Schemainformationen (nach [Abi97]). Es muss also die Änderung von Daten (Werten) und Strukturen durch eine XML-Update-Sprache möglich sein. Dazu sollen mögliche XML-Baummanipulationen in Werte-, Struktur- und Schemaänderung eingeteilt werden.

Änderung des Schemas Es ist klar, dass Schemaänderungen nur betrachtet werden können, wenn auch ein explizites Schema (DTD, XML Schema) zu einer Dokumentenkollektion vorhanden ist. Für eine DTD, die aus `element`- und `attribute`-Deklarationen aufgebaut ist, besteht eine Schemaänderungen aus dem Zufügen oder Löschen solcher Deklarationen oder der Modifizierung der Inhalte. Eine Änderung des Schemas zieht aber immer die Anpassung (Transformation) der zugehörigen Dokumente nach sich. Diese Form der Änderungen sollen nicht Gegenstand dieser Arbeit sein, stattdessen wird auf [Zei01] verwiesen.
Es sei noch angemerkt, dass das Einfügen eines *Document-Type*-Knotens an ein Dokument, also der Wechsel von einem schemalosen zu einem DTD-validierenden Dokument, wie auch die Änderung von *Namespace*-Attributen ebenfalls in diese Kategorie einsortiert werden.

Änderung von Werten Hierzu gehören im Allgemeinen das Einfügen, Löschen und Ändern von Textknoten, Attributinhalten wie auch das Umbenennen von Elementen oder Attributen. Fälle die hier nicht eingeordnet werden, können werden im nächsten Punkt erläutert.

Änderung von Strukturen Wie schon oben erwähnt, besteht ein XML-Dokument neben Daten aus Strukturen, die sich in einer Baumhierarchie darstellen lassen. Auch wenn ein XML-Dokument keinem expliziten Schema zugeordnet ist, lässt sich für jedes Dokument implizit ein Schema ableiten, welches das sogenannte Inhaltsmodell darstellt. Hierauf werden also die strukturellen Beziehungen der XML-Daten abgebildet.
Wenn sich durch eine Änderungsoperation das Inhaltsmodell des Dokuments ändert, so spricht man von einer strukturellen Änderung. Dazu können das Einfügen/Löschen von Knoten verschiedener Typen wie auch das Umbenennen zählen.

Beispiel:

```
<team>                       implizite DTD:
<player>Kahn</player>  <!ELEMENT team (player|coach)*>
<player>Klose</player> <!ELEMENT player (#PCDATA)>
<coach>Völler</coach>  <!ELEMENT coach (#PCDATA)>
<coach>Skibbe</coach>
</team>
```

Würde man das `<coach>`-Element mit dem Inhalt "Völler" in `<player>` umbenennen, entspräche dies einer wertmäßigen Änderung, da das zugehörige

Inhaltsmodell konstant bleibt. Dies wäre auch für das Einfügen eines neuen `<coach>` oder `<player>` der Fall.
Eine Umbenennung des `<player>` "Kahn" in `<keeper>` führt dagegen zur Erweiterung der impliziten DTD

```
<!ELEMENT team (keeper|player|coach)*>
<!ELEMENT keeper (#PCDATA)> ...
```

und stellt damit eine strukturelle Änderung dar.

Es ist offensichtlich, dass eine Änderung von Strukturen bei Vorhandensein eines expliziten Schemas Konflikte zwischen diesem Schema und dem impliziten Inhaltsmodell nachsichzieht. Dazu wären zwei Auflösungsstrategien sinnvoll:

1. Strukturelle Änderung werden bei schemavalidierenden Dokumenten generell zurückgewiesen.
2. Das explizite Schema wird generalisiert, so dass es wieder für die gesamte Dokumentenkollektion gültig ist.

Ausgenommen von dieser Betrachtungsweise sind strukturelle Änderungen, für deren geändertes Inhaltsmodell eine Verminderung der Informationskapazität stattfindet. Das geänderte Dokument ist dann immer noch gegen das allgemeinere Schema gültig.
Würde man zum Beispiel das implizite Schema des letzten Beispiels als gegebene DTD für das zugehörige Dokument annehmen, so lege durch das Löschen des `<keepers>` "Kahn" sicherlich eine strukturelle Änderung vor. Die zugehörige DTD bliebe jedoch weiterhin gültig.

4.2 Logisches Update-Modell

Nachdem nun die Grundlagen zu Änderungsoperationen für XML gelegt wurden, steht in diesem Abschnitt die Spezifizierung eines Update-Modells auf der logischen Ebene im Vordergrund. Im ersten Schritt wird ein Datenmodell aufgestellt und als zweites dann die darauf aufsetzenden Basisoperationen beschrieben. Ausgangspunkt für diese Schritte war der Artikel [TIHW01] von Igor Tatarinov.

4.2.1 Datenmodell

Da das Ziel dieser Arbeit eine Erweiterung des Sprachentwurfs zu XQuery ist, liegt es nahe, dass zugehörige *XQuery 1.0 and XPath 2.0 Data Model* [XQDM01] als Basis zu verwenden. Um die Spezifikation möglichst einfach zu halten, wurde erst eine Vereinfachung des Modells vorgenommen und notwendige Konzepte für die Integration von Update-Operationen ergänzt.
Kurz gesagt, ist ein XML Dokument in diesem Datenmodell ein Baum mit benannten Knoten und Referenzen. Bis auf Attribute wird eine Ordnung innerhalb des Dokuments angenommen. Das betrifft auch *Sequenzen*, die im XQuery-Datenmodell einer flachen Liste mit 0 oder mehr Einträgen entsprechen. Sequenzen können dabei auch verschiedene Kombinationen von Inhalten Aufnehmen. Eine Liste mit nur einem Eintrag ist dabei äquivalent zu dem Eintrag selbst.
Eine Behandlung von Sequenzen als Dokument-Kollektion wurde in dieser Arbeit vorerst nicht vorgesehen. Weiterhin werden folgende Knotentypen nicht betrachtet: `Namespaces`, `Processing Instructions`, `Comments`. Als letzte Einschränkungen findet keine Auswertung von Typinformationen in diesem Datenmodell statt.

Folgende Arten von XML-Inhalten, genauer Knotentypen, werden hier unterschieden:

- Das `document` ist der Wurzelknoten des Baums und enthält genau ein Element, das sogenannte *document element*. Es dient hier der Identifizierung eines Dokuments und kann nicht verändert oder verschoben werden. Die Änderung des document element's ist jedoch möglich.
- Ein `element` ist ein Tupel aus (Name, Menge von Attributen, Menge von Referenzen, Liste von Kindern). Kinder können dabei wieder Elemente oder `text` sein.
- Ein `attribute` ist ein Paar aus (Name, Wert), wobei der Wert ein einfacher String ist. Ein Attribut existiert im Dokument nur innerhalb seines übergeordneten Elements.
- `IDREFS` In diesem Modell wird eine explizite Unterscheidung zwischen normalen Attributen und Referenz-Attributen vorgenommen, da Referenzen strukturelle Informationen im Gegensatz zu Attributwerten darstellen. Hierzu werden

`IDREFS` eingeführt, `IDREFS` ist eine benannte geordnete Liste von `ID`'s, die jeweils auf das zugehörige Element verweisen. Hier ist zur Integritätserhaltung darauf zu achten, das eine referenzierte `ID` auch tatsächlich existiert, ansonsten wird das Dokument als ungültig betrachtet.
Zur Vereinfachung wird nicht zwischen `IDREF` und `IDREFS` unterschieden, wobei dann analog zu Sequenzen ein `IDREF` eine Liste mit nur einem Eintrag ist. Zur Dereferenzierung wird wie in XQuery der ->-Operator benutzt.
Es gilt hier zu beachten, dass `ID/IDREFS` nur durch ein Schema erkannt bzw. festgelegt werden können. Daher werden `IDREFS` hier explizit im Datenmodell aufgenommen und für `ID`'s werden vereinfacht alle Attribute mit dem Namen "ID" als ID-Typen betrachtet.

- Ein `text`-Knoten enthält nur einen String und kann nur innerhalb eines Elements auftreten.

Zur Generierung dieser Knoten werden auf logischer Ebene explizite Konstruktoren für jeden Knotentyp definiert:

- $createDoc(name, docElement)$: Erzeugt einen benannten `document`-Knoten (URL oder andere Kennzeichnung) mit einem document-element.

- $createElem(name, attributes, references, children)$: Erzeugt einen benannten `element`-Knoten. Die Mengen attribute, references bzw. die Liste children dürfen dabei leer sein.

- $createAttr(name, value)$: Erzeugt ein benanntes `attribute` mit dem angegebenen Wert. Wenn der Name gleich "ID" ist, muss `value` den XML-Namenskonventionen entsprechen.

- $createRefs(name, targetIDs*)$: Erzeugt eine benannte Liste `IDREFS`, mit 0 oder mehr Einträgen von Element-`ID`'s.

- $createText(content)$: Erzeugt einen `text`-Knoten mit dem spezifizierten Inhalt.

Bei der Erzeugung wird jedem Knoten eine Identität vergeben, die zur internen Beziehungsmodellierung (`[parent]/[child]`) verwendet wird, dabei jedoch nicht mit dem `ID/IDREF`-Mechanismus zu verwechseln ist.

4.2.2 Basisoperationen

Für die Definition der Basisoperationen gelten wieder gleichen Voraussetzungen wie für das Datenmodell, die Operationen sollen als Erweiterung der Semantik von XQuery aufgestellt werden. Dazu wird davon ausgegangen, das eine Operation zur

Berechnung von Pfad-Ausdrücken (XPath 2.0) existiert, die Variablen an Objekte aus XML-Dokumenten bindet, im Speziellen auch Sequenzen von Objekten. Diese Operation bestimmt das "Ziel" ($target$) der nun folgenden Update-Basisoperationen, und wird auf dieser logischen Ebene implizit als Kontext angenommen.
Mit dieser Menge von Änderungen soll die Veränderung von sowohl einfachen Werten (skalare Typen und Knotenblätter) wie auch von komplexen Inhalten (Strukturen, Teilbäume) möglich sein. Von einer speziellen Typisierung in Form von XML-Schema mit abgeleiteten Untertypen wird an dieser Stelle jedoch abgesehen.
Jede Operation wird mit spezifischen Parametern versehen, wobei $child$ bzw. $source$ wiederum einer Bindung entsprechen. $name$ hat der XML-Namenskonvention und $content$ dem Inhaltsmodell des übergeordneten Knoten zu genügen. $content$ kann dabei einfacher XML-Inhalt inklusive einfacher Werte, wie auch berechneter Inhalt sein. Zur Auswertung solcher Inhalte gelten die in XQuery vorgestellten Abbildungsregeln. Ein Objekt kann in den folgenden Definitionen ein einzelner Knoten, wie auch ein ganzer Teilbaum sein.

Als *Update* wird nun eine *Sequenz* von folgenden Grundoperationen definiert:

- $Insert(content)$
  - *Semantik*: *Anfügen* von neuem Inhalt.
  - *Ziel*: Alle Knotentypen außer `document`
  - $\{InsertBefore|InsertAfter\}(child, content)$ für geordnetes Modell. Fügt Inhalt vor oder nach dem $child$-Knoten ein. Falls $content$ ein Attribute ist, entspricht die Semantik dem einfachen $Insert$.
  - *Copy*-Funktionalität, für berechnete $content$-Ausdrücke mit Knoten.
- $Delete(child)$
  - *Semantik*: Wenn $child$ ein Nachkomme des $target$-Objekts ist, wird es entfernt.
  - *Ziel*: `{element,IDREFS}`
  - *Parameter*: Falls $child$ ein `IDREF` einer `IDREFS`-Liste ist, wird nur dieser Eintrag entfernt, die Restliste bleibt bestehen.
    Das Löschen des *document-elements* ist nicht möglich.
- $Rename(child, name)$
  - *Semantik*: Wenn $child$ ein Nachkomme des $target$-Objekts ist, wird ihm ein neuer Name vergeben.
  - *Ziel*: `{document,element}`

  - *Parameter* Für $child$ können außer `text` alle Knotentypen verwendet werden. Auch einzelne Einträge einer `IDREFS` können nicht umbenannt werden, dazu ist der Name für die ganze Liste zu ändern.

- $Replace(child, content)$
  - *Semantik*: Atomares Ersetzen eines Objektes, entspricht einem $insert\{Before\}$ und anschließendem $delete$ von $child$.
  - *Ziel*: `{document,element,IDREFS}`
  - *Parameter*: Analog Delete und Insert mit der Abwandlung, dass ein *document-elements* auch ersetzt werden kann.

- $Move(source)$ bzw.
  $MoveBefore|MoveAfter(child, source)$
  - *Semantik*: Atomares Verschieben eines Objektes. (analog $Insert$ für ungeordnetes bzw. geordnetes Modell und anschließender Löschung von $source$).
  - *Ziel*: code{element,IDREFS}
  - *Parameter*: Alle Knotentypen außer `document-element`

- $SubUpdate(xpathOP, predicate, updateOP+)$
  - *Semantik*: Nestung von Updates, durch Auslösen einer neuen Pfad-Operation ($xpathOP$), die wieder eine Bindung von Variablen im aktuellen Kontext vornimmt und durch $predicate$ selektiert werden kann. Diese Bindungen werden als neuer Sub-Kontext der geschachtelten Update-Sequenz ($updateOP+$) bereitgestellt.
  - *Ziel*: Aktueller Kontext.
  - *Erläuterungen*: Für die Sub-Updateoperation erfolgt ein rekursiver Aufruf für jede Bindung. Dieses Konzept ermöglicht die gleichzeitige Änderung von XML-Daten auf unterschiedlichen Hierarchieebenen in einer Update-Anweisung.
  - *Restriktionen*: Es müssen erst alle Bindungen einer $SubUpdate$ berechnet werden, bevor mit irgendwelchen Updates begonnen werden kann. Sonst käme es eventuell zu einer Veränderung in der Auswertungsreihenfolge, z.B. durch Einfügen oder Löschen von Bindungsknoten. Dies kann sogar zu einer Endlos-Schleife führen, falls immer wieder neue Bindungen eingefügt würden.
    Die selbe Vorgehensweise gilt auch für die Auswertung von $content$, die auch vor der Bindung erfolgen muss.

> Wenn in einer Update-Sequenz eine Bindung gelöscht wurde, dürfen nachfolgende Operationen der Sequenz die darauf Bezug nehmen würden, nicht mehr ausgeführt werden.

Wie gezeigt, können *Sequenzen* von Operationen in einer Anweisung bzw. $SubUpdate$ festgelegt werden. Das sorgt einerseits für eine effizientere Ausführung dieser Sequenzen und stützt andererseits das Transaktionskonzept durch mögliche Teiltransaktionen auf unterschiedlichen Granularitäten.
Für diese Menge an Basisoperationen lässt sich wieder eine Kategorisierung in Werteänderungen und Strukturänderungen vornehmen, wobei Schemaänderungen hier generell nicht betrachtet werden. Da alle Änderungen auch durch ein minimales Set aus $insert$ und $delete$ beschrieben werden können, wird dazu auf den Abschnitt 4.1.2 verwiesen.

Es folgt nun eine Problembetrachtung zu den vorgestellten Basisänderungen, die eine besondere Behandlung durch das Umfeld erfordern.

1. Ein *Dokument-Knoten* darf nur Kontext einer Operation sein, nie das zu ändernde Objekt selbst. Das zeigt sich in den *Ziel* und *Parameter*-Beschreibungen der Operationen. Er enthält immer genau ein *document-element*, dass deshalb nicht entfernt werden darf, aber Modifikationen unterliegen kann.

2. Die *Kopie eines Elements* innerhalb desselben Dokuments ist nicht möglich, wenn es ein `ID`-Attribut enthält, da der `ID`-Wert nur einmal im Dokument vorkommen darf. Die Kopie eines `ID`-Attributs selbst ist dann natürlich auch verboten.
   Lösung: Durch ein kombiniertes $insert/replace(id)$ erfolgt die gleichzeitige Korrektur des doppelten `ID`-Werts in einer Teiltransaktion. Eine andere Möglichkeit wäre, das Element nicht 1:1 zu übernehmen, sondern über einen berechneten Element-Inhalt vor dem Einfügen anzupassen.
   Das Kopieren von `IDREFS` stellt dagegen kein Problem dar.

3. Das *Einfügen von Attributen* an ein Element mit einem Attribut-Namen, den ein schon vorhandenes Attribut trägt, muss *zurückgewiesen* werden.
   Im Unterschied dazu führt das Einfügen von `IDREFS` mit existierenden Namen zum Anhängen der Einräge an die vorhandene Liste.

4. Das *Löschen von Elementen* mit einer `ID` oder des `ID`-Attributs selbst kann in *dangling references* resultieren. Dies muss auch auch bei der Löschung ganzer Teilbäume für die enthaltenen Knoten überprüft werden.
   Lösung:

   (a) Die Löschung von Elementen oder `ID`'s zurückweisen, wenn noch Referenzen auf diese Knoten gehalten werde. (sehr einschränkend)

(b) Alle eingehenden Referenz-Kanten zurückverfolgen und zugehörige `IDREFS`-Einträge kaskadierend löschen (besser, aber aufwendig)

Was hier für die Löschung gilt, ist natürlich im gleichen Maße für $relace$ und $move$ gültig.

5. *Mehrdeutige Updates* entstehen dadurch, dass durch Referenzierung mehrfache Pfade zu einem Element existieren (Graph).
Möchte man zum Beispiel die von `<player>` "Klose" geschossenen Tore um 3 erhöhen und würde über den Pfad-Ausdruck `Player[.="Klose"]/Goals` eine Ergebnismenge von 2 identischen Knoten erhalten, so würde die anschließende Update-Operation insgesamt zu einer Erhöhung von 6 Toren führen. Solche Updates sind nicht deterministisch.
Lösung: Wie in der Semantik der Basisoperationen schon zu erkennen ist, darf sich eine Updateoperation nur auf die $children$ des aktuellen Kontext-Knoten auswirken. Diese Restriktion führt immer zur eindeutigen Identifizierung des zu modifizierenden Objekts.

6. *Betrachtung der unterliegenden Ebene:*
Für das hier beschriebene Modell wird immer angenommen, dass das Dokument vollständig geparst wurde und das Einsetzen aller Entities und Default-Attribute erfolgt ist. Wenn nun Teile des Dokuments verändert wurden, muss sich ein Mechanismus mit dem Nachvollziehen der Änderung auf den tieferen Ebenen beschäftigen.

   (a) Probleme entstehen bei aufgelösten *Entity-Referenzen*, wenn eine Entity im Dokument mehrfach referenziert war und somit auch mehrfach aufgelöst wurde. Es wird nun eine Ausprägung dieser Entity verändert und stellt damit beim Zurückschreiben einen Konflikt mit den nicht veränderten Instanzen dar. *Lösung* Man könnte hier, wie in DOM (Abschnitt 3.1.3) beschrieben, vorgehen, dass heißt, für die geänderten Entity-Referenzen ist je ein Klon der Entity anzulegen, auf dem die entsprechende Änderungen umgesetzt werden können. Diese Klone werden dann integriert und neu referenziert.

   (b) Die Löschung von *Default-Attributen* könnte folgende Auswirkungen haben.

      - Die Löschung wird unwirksam, da Attribute gemäß DTD wieder auf ihren Standard-Wert gesetzt werden. (inkonsitent)
      - Änderung der zugehörigen DTD (keine `#DEFAULT` mehr). Dann muss der DEFAULT-Wert in die restlichen Dokumente der DTD-Kollektion explizit einfügt werden. (konsistent, aber aufwendig)
      - Zurückweisen der Löschung von DEFAULT-Attributen. (entspricht am ehesten der DEFAULT-Semantik)

(c) `#fixed`-Attribute dürfen generell nicht verändert werden. Daher müssen hier Updateoperationen immer zurückgewiesen werden.

4.3 SUNFLWR's - Syntaktische Erweiterung von XQuery

Zur Sprachdefinition gehören neben logischen Konzepten auch eine Syntax für die Sprachkonstrukte. Dazu sollen die beschrieben Update-Basisoperationen als Erweiterung in die XQuery-Syntax abgebildet werden. Die daraus entstandenen Ausdrücke sollen hier auf den Namen *SUNFLWR* (Simple Update Notation for XQuery-FLWR-Expressions) getauft werden.

Eine *SUNFLWR*-Anweisung hat allgemein folgendes Aussehen:

```
FOR $i IN xpath-expr1, ...
LET $j := xpath-expr2, ...
WHERE predicate, ...
UPDATE $i { updateOp {, updateOp}* }
```

Der *FLW*-Block wurde direkt aus XQuery übernommen und wird auch mit der selben Semantik versehen. `UPDATE` ersetzt die bisherige `RETURN`-Klausel und kann Sequenzen von Updateoperationen aufnehmen. Der Kontext oder das Ziel dieser Operationen ist hier durch die gebundene Variable `$i` gekennzeichnet.

Eine Updateoperation `updateOp` kann nun folgende Syntax haben:

```
INSERT content [BEFORE | AFTER $child] |
DELETE $child |
RENAME $child TO name |
REPLACE $child WITH content |
MOVE $source [BEFORE | AFTER $child] |

FOR $l IN xpath-subexpr1, ...
LET $m := xpath-subexpr2, ...
WHERE predicate, ...
UPDATE $l { updateOp {, updateOp}* }
```

Die Semantik der Operationen entspricht der in Abschnitt 4.2.2 beschriebenen und bedarf hier keiner gesonderten Erläuterung. Es sei nur darauf hingewiesen, dass `content` und `name` neben konstantem Inhalt auch berechneten Ausdrücken entsprechen können.
Die Nestung des Update ermöglicht zusätzliche innere Bindungen durch eine FLW-Klausel wie auch die Auswertung der Bindungen des äußeren Blocks. Auch hier sind wieder mehrfache Update-Operationen in einer Abfolge möglich. Man sollte sich an dieser Stelle klarmachen, dass der `LET`-Ausdruck die Bindung einer Sequenz ermöglicht, während mit `FOR` über die einzelnen Objekte einer Sequenz iteriert wird.

Im logischen Modell wurden explizite Konstruktoren für die verschiedenen Knotentypen definiert. Hier soll aufgezeigt werden, welche Mittel aus XQuery zur Generierung von Knoten verwendet werden können oder ob zusätzliche Konstruktoren für die Sprachebene erforderlich sind.

Element Für Elemente ist in XQuery ein Standard-Konstruktor vorgesehen der neben literalem XML auch berechnete Ausdrücke aufnehmen kann, z.B.

```
<keeper> $p/name </keeper>
```

Weiterhin existiert ein expliziter *berechneter* Konstruktor der Form:

```
"element" (Name | Ausdruck) { ExprSequence? }
```

wie zum Beispiel

```
element book { Es war einmal... }
```

Attribut Für Attribute steht nur ein *berechneter* Konstruktor zur Verfügung. Beispiel:

```
attribute isbn { 3-8266-0513-6 }
```

Ansonsten können Attribute nur innerhalb eines Elements angegeben werden.

Dokument Die Erzeugung von Dokumenten ist auf der Sprachebene nicht notwendig, da sie nicht den Änderungsoperationen unterliegen.

Text Text kann als literales XML, wie für Elemente beschrieben, angegeben werden. Einfache Strings werden gewöhnlich in Anführungszeichen eingeschlossen: `"string"`

IDREFS Da `IDREFS` in XQuery nicht vorkommen, gibt es folglich keinen zugehörigen Konstruktor. Deshalb wird hier einfach der Konstruktor aus dem logischen Modell als Funktion übernommen, z.B.

```
createRefs(verweis, "no007")
```

Weiterhin ist es wichtig zu zeigen, wie die Bindung von verschieden Knotentypen in *SUNFLWR*-Ausdrücken erfolgen kann. Für die Verwendung in Updates werden dabei keine Inhalte sondern Knotenreferenzen gebunden, wenn allerdings der Inhalt eines Ausdrucks auszuwerten ist, wird hier sein *string-value* berechnet.

Übersicht zur Selektion von Knotentypen zur Bindung an Variablen:

Knotentyp	Knotentest	Beispiel
Element	Standard	`/player[1]`
Attribut	`@name`	`/book/@goals`
Dokument	`/`	`document("wm.xml")`
Text	`text()`	`/book/text()[1]`
`IDREFS`	`ref(label,target)`	`/book/ref(verweis, "part1")`

Hinweis: Die `ref`-Funktion wurde hier eingeführt, um eine Variable an eine bestimmte Referenz innerhalb einer `IDREFS`-Liste zu binden. Für `target` kann auch ein `"*"` gesetzt werden, um alle Referenzen zu betrachten.

4.4 Sprachabbildung auf ein Speicherungsmodell

4.4.1 Wahl eines Speicherungsmodells

In diesem Abschnitt soll eine mögliche Umsetzung der vorgestellten Update-Sprache auf ein konkretes Speicherungsmodell gezeigt werden. Dazu soll der Ansatz zu Speicherung von XML-Dokumenten in (objekt-)relationalen Datenbanken aus [SYU99] verwendet werden. Dieser Ansatz wurde auch für die Arbeit [Ros01] verwendet und dort vorgestellt, weshalb hier nur eine kurze Einführung zu den Kernkonzepten erfolgt.

Grundidee des Ansatzes ist die Verwendung eines generischen Datenbankschemas, das jegliche XML-Dokumente aufnehmen kann. Dafür wird aber davon ausgegangen, dass eine Gültigkeitsprüfung gegen eine DTD durch einen validierenden Parser beim Einfügen der Dokumente in die Datenbank erfolgt. In dem Speicherungsmodell selbst findet keine Beachtung von Schemas statt, deshalb können hier auch keine `ID/IDREFS` behandelt werden und es findet keine Validierung von Änderungsoperationen statt.
Zur Speicherung wird die Baumstruktur eines XML-Dokuments gemäß der Knotentypen in die zugehörigen Relationen gespeichert. Einbezogen werden dabei Elemente, Attribute und Textknoten, die zusätzlich noch um einen einfachen Pfad-Ausdruck erweitert werden.
z.B. `/books/book/@style`

Zusätzlich werden die Knoten nach ihrem Auftreten im Dokument nummeriert. Dazu werden sogenannte *Regionen* in der Form `(start_pos, end_pos)` eingeführt. Textknoten werden dabei mit ganzzahligen Positionen versehen, wobei jedes einzelne Wort gezählt wird. Die sonstigen Knoten werden durch Realzahlen indexiert, wobei der Ganzzahlanteil die Nummer des vorhergehenden Wortes ist und der Nachkommaanteil die laufende Nummerierung in der aktuellen Elementensequenz ist.
Um mehrfache Elemente in ihrem Auftreten zu kennzeichnen, legt man für Elemente zusätzlich einen Ordnungsindex an, sowohl in positive (vorwärts) wie auch in negative (rückwärts) Richtung.
Diese Regionen können in einem ADT abgelegt werden und dieser durch UDF's zur Prüfung von Inklusionen und Ordnung ergänzt werden.

Im Anhang wird hierzu ein Beispieldokument (Abschnitt A.5.1) und seine Baumstruktur samt Nummerierung (Abb. 11) gezeigt, die auch [SYU99] entnommen wurden.

Das ER-Schema des generischen Modells wird in Abbildung 9 dargestellt. Hier wurden neben den in [SYU99] betrachteten Knotentypen zusätzlich `document`-Knoten aufgenommen. Zudem wurden die Realzahlen der Elementpositionen hier in Ganzzahlen für Wordindex (wi) und Elementindex (ei) umgesetzt.

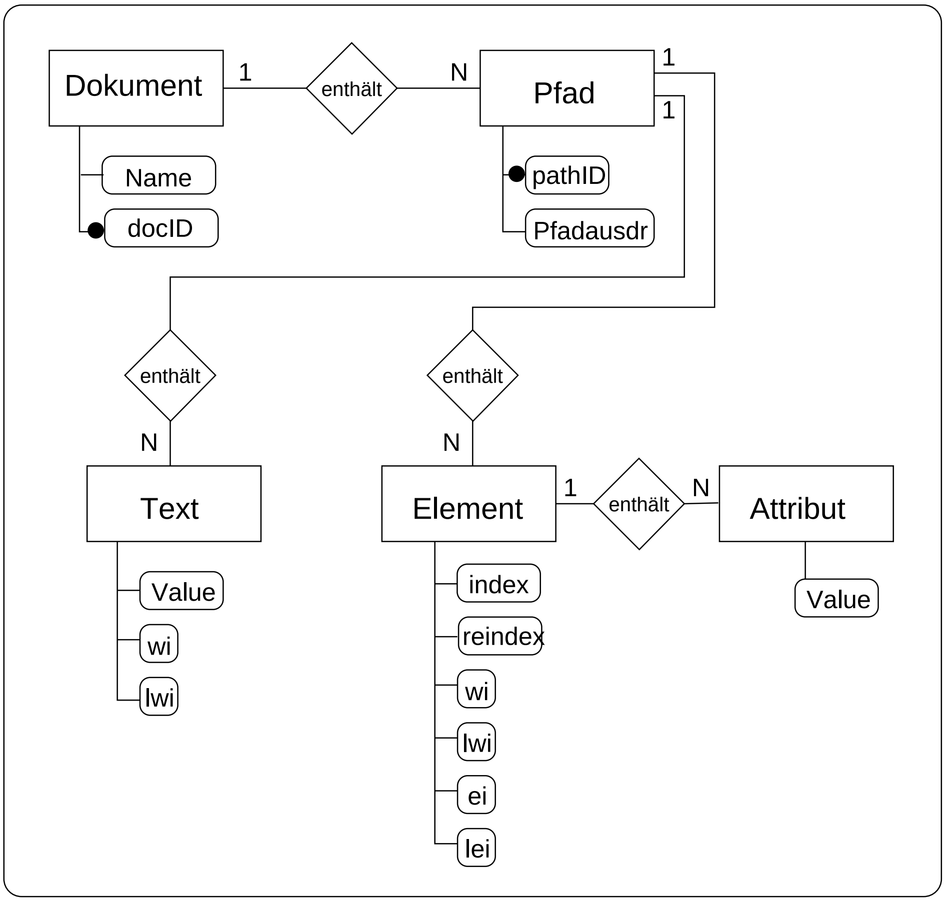

Abbildung 9: Speichersystem

Die Abbildung des ER–Diagramms auf das relationale Datenmodell resultiert in dem folgendem Datenbankschema:

```
documents(docID,docName)
path(docID,pathID,pathexp)
element(docID,pathID,index,reindex,wi,ei,lwi,lei)
text(docID,pathID,value,wi,lwi)
attribute(docID,pathID,attvalue,wi,ei,lwi,lei)
```

Die zugehörigen Relationen des Beispiels sind im Anhang im Abschnitt A.5.2 zu finden.

4.4.2 Auswertungsstrategie

In der Arbeit von Guido Rost [Ros01] wurde beschrieben, wie ein vollständiges System zur Speicherung und Anfrage von XML-Dokumenten, basierend auf dem obigen Speicherungsmodell, aufgebaut ist und arbeitet. Hier soll nun angedeutet werden, wie eine Erweiterung dieses Systems um eine Update-Funktionalität aussehen könnte. Dazu werden die Module der Anfragebearbeitung auch für die Umsetzung von Änderungen vorausgesetzt.

Die Auswertung einer Änderungsanweisung würde folgenden Ablauf nehmen.

1. Lexikalische und Syntaktische Analyse mit einem Syntaxbaum von (Teil-) Ausdrücken als Ergebnis.
2. Auswertung der *SUNFLWR*-Ausdrücke generiert einen Tupelbaum mit gebundenen Variablen.
3. Auswertung der $content$-Ausdrücke ähnlich der *RETURN*-Klausel von XQuery.
4. Ausführung der Update-Operationen in entsprechender Reihenfolge (innere vor äußere Updates, Sequenzordnung).

Für die nun folgende Betrachtung der Sprachabbildung wird vorausgesetzt, dass einzufügende Inhalte bereits berechnet wurden und die `docID` und `pathID` für zu ändernden Knoten vorliegen. Außerdem wird die Syntax zur Verdeutlichung der zu Grunde liegenden Techniken stark vereinfacht.
Wie schon in Abschnitt 4.1.1 beschrieben, kann in SQL nur eine Relation mit einer Update-Anweisung geändert und nur eine Änderungsart eingesetzt werden. Falls mehrere SQL-Anweisungen für ein XML-Update notwendig werden, sollten diese in einer Transaktion gekapselt werden, um inkonsistente Zustände zu vermeiden.

4.4.3 Einfache Abbildungen

Wertmäßige Änderung Zu den trivialen Abbildungen gehört das Einfügen, Löschen oder Ändern von Attributwerten bzw. Textinhalten.

Durch den folgenden Ausdruck wird dem `title`-Element Text angefügt. Da dieses schon einen Textknoten enthält, kann der neue Text diesem Inhalt einfach hinzugefügt werden.

SUN

```
UPDATE /books/book/title { INSERT ' for fun' }
```

SQL

```
UPDATE text
SET value = append(value, ' for fun')
    lwi   = lwi + 2
WHERE (docID = 1) and (pathID = 4) and (wi = 1)
```

Die Änderung auf der Datenbank-Ebene resultiert in einem Anfügen des Strings an den Inhalt des bestehenden Textknoten durch eine `append`-Funktion, die zur Konkatenation von Text systemspezifisch sein kann. Durch `lwi = lwi + 2` wird die Endposition des veränderten Textknotens um 2 Wörter erhöht.
Anschließend ist die Anpassung des Wordindex für alle nachfolgenden Textbausteine, Elemente und Attribute des Dokuments notwendig. Der folgende SQL-Code nimmt diese Anpassung für die Text-Relation vor:

SQL
```
UPDATE text
SET wi   =  wi + 2
    lwi = lwi + 2
WHERE (docID = 1) and (wi >= 3)
```

Diese Korrektur wäre bei der Änderung von Attribut-Werten nicht nötig, da sich hierbei keine Verschiebungen in den Positionen ergeben (Attribut-Inhalte werden nicht indexiert).

Umbenennung Da Namen von Elementen und Attributen in diesem Speichermodell nicht Bestandteile des DB-Schemas sind, sondern zentral in der *path*-Relation gespeichert werden, gestaltet sich die Umbenennung dieser Knoten relativ simpel. Hinzu kommt, dass die `pathID` als Identifier unverändert bleibt, und so keine weiteren Anpassungen nötig sind.

Im folgenden Beispiel werden alle `author`-Elemente zu "writer" umbenannt.

SUN
```
UPDATE /books/book { RENAME author TO 'writer' }
```

SQL
```
UPDATE path SET pathexp = substitute(pathexp,
  '/books/book/author', '/books/book/writer')
WHERE pathexp LIKE '/books/book/author%'
```

Durch diese SQL-Anweisung wird der Name der `author`-Elemente und damit deren Pfad und die Pfade ihrer Abkömmlinge an den neuen Namen angepasst. Die systemspezifische Funktion `substitute` dient hierbei zur Ersetzung von Teil-Strings.

4.4.4 Komplexe Abbildungen

Strukturelle Änderung verändern zwar nicht das Schema des generischen Speichermodells, erfordern aber einen höheren Aufwand durch komplexe Anpassungen der Indexe in allen Relationen.

Einfügen/Löschen von Text-Knoten Diese Operation stellt eine einfache Art der Strukturänderung dar, da hierzu keine Pfade angepasst werden müssen.

Durch die folgende Operation wird ein neuer Textknoten an das Element `editor` vor das Kind-Element `family` mit dem Textknoteninhalt `'Bob'` eingefügt.

SUN
```
UPDATE /books/book/editor { INSERT 'mit Text' BEFORE
family[.='Bob']}
```

SQL
```
INSERT text (docID, pathID, value, wi, lwi)
VALUES (1, 5, 'mit Text', 4 , 5)
```

Die zugehörige SQL-Abbildung ist eine einfache `INSERT`-Anweisung. Vorher sollten die Wordindexe (wi, lwi) der nachfolgenden Knoten analog der Vorgehensweise bei der Wertänderung angepasst werden (Erhöhung um 2).

Das nächste Beispiel zeigt die Löschung des Textknotens aus dem `summary`-Element.

SUN
```
UPDATE /books/book/summary { DELETE text()[1] }
```

SQL
```
DELETE text
WHERE (docID = 1) and (pathID = 11) and (wi = 10)
```

Auf der Datenbankebene reicht hier eine `DELETE`-Anweisung. Die Wordindex-Anpassung wäre hier nach dem Löschen sinnvoll.

Einfügen/Löschen von Attribut-Knoten Da Attribute benannt sind, wird beim Einfügen und Löschen eine Änderung in der Pfad-Relation erforderlich.

Durch den folgenden Ausdruck wird dem Element `editor` ein neues Attribut `origin` mit dem Wert `'Fresno'` angefügt.

SUN
```
UPDATE /books/book/editor { INSERT attribute origin
{Fresno} }
```

SQL
```
INSERT path
VALUES ('/books/books/title/@origin', 13)

INSERT attribute (docID, pathID, attvalue,
                  wi, ei, lwi, lei)
VALUES (1, 13, 'Fresno', 3, 2, 3, 2)
```

Für die Anpassung der Speicherungsstruktur wird zuerst ein Pfad mit dem neuen Namen angelegt. Im Anschluss daran kann das eigentliche Einfügen in die `attribute`-Tabelle erfolgen.

Im zweiten Beispiel wird das `style`-Attribut aus dem `book`-Element entfernt.

SUN
```
UPDATE /books/book/ { DELETE @style }
```

SQL
```
DELETE attributes
WHERE (docID = 1) AND (pathID = 3)

DELETE path
WHERE (pathID = 3)
```

Die SQL-Änderungen laufen hierbei umgekehrt zum Einfügen von Attributen ab. Für das Löschen und Einfügen von Attributen gilt, das keine Anpassungen von Word- oder Elementindexen notwendig sind.

Einfügen/Löschen von Element-Knoten Das Einfügen und Löschen von Element-Knoten stellt eine komplexe Abbildung dar, da jede Relation von Änderungen betroffen wird und neben der Neuberechnung von Word- und Elementindexen auch die Überprüfung der Word-Occurrence (index, reindex) erfolgen muss.

Ein verhältnismäßig simples Beispiel, stellt das Einfügen eines leeren Geschwister-Knotens `family` nach dem ersten `given`-Element in das `author`-Element dar.

SUN

```
UPDATE /books/book/author { INSERT element family {}
AFTER given[1] }
```

SQL

```
UPDATE element
SET reindex = reindex - 1
WHERE (docID = 1) AND (pathID = 9) AND (index < 1)

UPDATE element
SET index = index + 1
WHERE (docID = 1) AND (pathID = 9) AND (reindex > -2)

INSERT element (docID, pathID, index, reindex,
                wi, ei, lwi, lei)
VALUES (1, 9, 1, -2, 7, 2, 7, 3)
```

Durch die ersten beiden SQL-Updates werden die Ordnungsindexe der Geschwister-Elemente von `family` auf den neuen "Bruder" eingestellt. Zusätzlich wäre es noch erforderlich, die Elementindexe für *element*- und *attribute*-Relationen vorher anzupassen. Dann kann das eigentliche Einfügen des neuen Elements erfolgen. Ein Update der Pfad-Relation war in diesem Fall nicht notwendig, da ja schon die gleich lautenden Namen der Geschwisterknoten eingetragen waren.
Beim Einfügen von Elementen mit Inhalt (Text) muss natürlich auch der Wordindex angepasst werden. Dazu ist dann auch das Einfügen des zugehörigen Textknotens anzuschließen.

Bei einem `INSERT` von komplexen Elementen, also einem neuem Teilbaum, kann man entweder *top-down* das Einfügen jedes einzelnen Knoten des Teilbaums vornehmen oder optimiert den "Platz" des neuen Teilbaums vorausberechnen und damit die entsprechenden Positionsanpassungen einmalig durchführen. Danach kann dann einfach die Sequenz an gesammelten SQL-Update-Anweisungen angestoßen werden.

Die nun folgende Operation löscht das `title`-Element samt Text-Knoten aus dem `book`-Element heraus.

SUN

```
UPDATE /books/book/ { DELETE title }
```

SQL

```
DELETE text
```

```
WHERE (docID = 1) and (pathID = 4)

DELETE element
WHERE (docID = 1) and (pathID = 4)

DELETE path
WHERE (pathID = 4)
...
```

Die zugehörige SQL-Abbildung ist eine Folge von `DELETE`-Operationen auf die `text`-, `element`- und `path`-Relation. Danach muss die Neuberechnung von Word- und Elementindexen angeschlossen werden.

Für den allgemeinen Fall gibt es analog zum Einfügen von Elementen wieder zwei Wege, um einen Teilbaum aus dem Dokument zu löschen, die Einzellöschung jedes Knotens *bottom-up* oder die Vorausberechnung der Anpassungen und ein anschließendes Bulk-Delete.

Ersetzen und Verschieben von Knoten Dies sind Operationen, die durch ein Einfügen und Löschen von Knoten nachgestellt werden können. Dazu lassen sich also oben beschriebene Abbildungen benutzten.
Für ein Ersetzen lassen sich dann Optimierungen vornehmen, falls die Inhaltsmodelle des zu ersetzenden und neuen Knotens gleich sind. Beispielsweise würde in dem `/books`-Dokument die Ersetzung des `editor`-Elements durch ein neues `autor`-Element mit `family`- und `given`-Kindern zu einer einfachen Umbenennung und Werteänderung mit anschließender Indexanpassung führen.

4.4.5 Optimierungen

Einige effizienzsteigernde Maßnahmen wurden schon in den vorgestellten Beispielabbildungen angesprochen. Zusätzlich muss man bei Sequenzen von Updates, wie sie ja bei *SUNFLWR*'s möglich sind, überprüfen, ob die Reihenfolge von Operationen nicht zur Optimierung verändert werden kann und ob dann bestimmte Updates auch wegfallen können.
Auf tieferen Ebenen sollte man durch Trigger oder Ausnutzung systemspezifischer Mechanismen wie *stored procedures* die sich immer wieder nachziehenden Anpassungen des Nummerierungsschema automatisieren. Dazu könnte eventuell auch eine andere Form der Indexierung beitragen.

5 Schlussbetrachtung

5.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit wurden verschiedene Aspekte im Umfeld von XML und Updates untersucht. Dazu wurde ein Anforderungskatalog erarbeitet, im dem sowohl klassische Kriterien an Anfragesprachen wie auch XML-spezifische Konzepte untersucht wurden.
Bei der Recherche nach existierenden Ansätzen zu einer XML-Änderungssprache stößt man schnell auf verschiedene Implementierungen. Artikel die eine umfassendere und tiefergehende Untersuchung auf höheren Ebenen vornehmen, sind zur Zeit leider noch absolute "Mangelware". Das zeigt sich auch in der Qualität der in Abschnitt 3 verglichenen Arbeiten, die an verschiedenen Stellen zu einer oberflächlichen Betrachtung neigen.
Deshalb wurden für die im Abschnitt 4 vorgestellte Konzeption zuerst Grundlagen zu Änderungen in XML-Dokumenten geschaffen. Auf der logischen Ebene stand die Beschreibung des zu Grunde liegenden Datenmodells und die darauf agierende Menge an Änderungsbasisoperationen im Vordergrund. Daran wurde eine Problemdiskussion angeschlossen, bevor die Übertragung der Grundoperationen in die XML-Syntax durchgeführt wurde. Den Abschluss bildet die Skizzierung einer Umsetzung auf ein relationales Datenbanksystem.

5.2 Ausblick

Im Verlauf dieser Arbeit wurden an verschiedenen Stellen Restriktionen in Kauf genommen und Problembereiche unberührt gelassen. Dies betrifft besonders folgende Punkte, die durch eine nachfolgende Arbeit berücksichtigt werden könnten:

- Formalisierung von Datenmodell und Operationen (für eine mögliche Optimierung)
- Einführung eines XML-Sichtenmodells und darin Betrachtung von Effekten durch Änderungsoperationen.
- Untersuchung der Schemavalidierung von atomaren Änderungen, ohne dabei vollständige Dokumente gegen ein Schema zu prüfen.
- Gemeinsames System aus Schemaänderungen und Instanzenänderungen (Schemaevolution)
- Vollständige Umsetzung der *SUNFLWR*-Ausdrücke in die bestehende XQuery-Implementierung
- Entwurf und Integration eines Transaktionssystems

- Untersuchung von möglichen Backup- und Recovery-Strategien speziell für XML-Repositories

A Anhang

A.1 Begriffsfestlegungen

Eine *Anfragesprache* soll in dieser Arbeit allgemein für Anfrage- und Änderungssprache verstanden werden.

Eine *Anweisung (statement)* ist eine komplexe Kombination von einzelnen Operationen.

Eine *Menge* oder *Satz* ist eine ungeordnete Zusammenstellung von Objekten.

Eine *Liste* oder *Sequenz* ist in einer Reihenfolge geordnete Aufzählung von Objekten.

Ein *Baum* ist eine Datenstruktur, bestehend aus einer Wurzel (root), Knoten und Endknoten (Blätter), die durch Kanten miteinander verbunden sind. Ein Knoten kann dabei immer nur direkter Nachkomme (child, Kind, Tochterknoten) *eines* Vaterknoten (parent) sein. Benachbarte Knoten auf einer Ebene nennt man Geschwister (siblings), wenn sie vom selben Vaterknoten abstammen.

A.2 XML Information Set

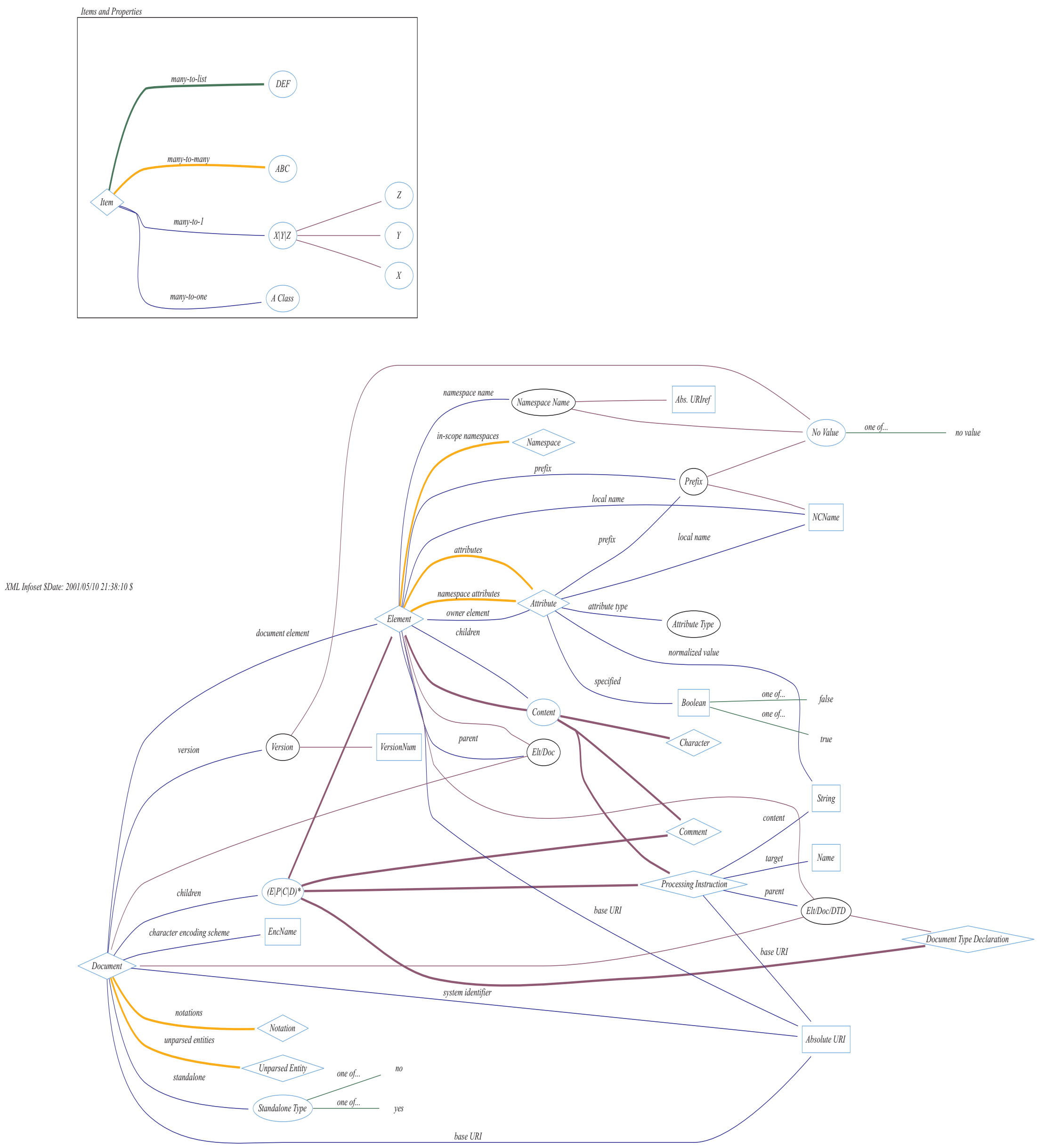

Abbildung 10: Infoset als UML-Modell im .SVG Format

A.3 Semistrukturierte Daten

Abiteboul: Eigenschaften von semistrukturierten Daten [Abi97]

1. Struktur der Daten ist unregelmäßig (inhomogen)
2. Schema ist implizit in den Daten enthalten
3. Struktur der Daten ist unvollständig
4. Schema ist flexibel (nicht konforme Update-OPs werden bei DB abgelehnt)
5. Schema ist relativ groß (Anfrage/Änderungen auf Daten und Schema notwendig)
6. Schema unterliegt häufigen Änderungen (folgt aus Punkt 1,2,4 und 5)
7. Unterschied zwischen Struktur und Daten ist unscharf

A.4 Implementationen

- Tamino (SoftwareAG) und Quip
  `http://www.softwareag.com/developer/quip/`
- Microsoft .NET
  `http://xqueryservices.com/`
- eXcelon XIS
  `http://www.xmlquickstart.com/`
- XML:DB Lexus, XHive, SiXDML
  `http://www.xmldb.org/xupdate/`

A.5 Beispieldokument und Datenbank

A.5.1 Beispieldokument books

```
<books>
<book style="textbook">
<title>Designing XML applications</title>
<editor>
  <family>Bob</family> <given>Kraft</given>
</editor>
<author>
  <family>Nick</family> <given>Marcus</given>
  <family>Bob</family> <given>Pant</given>
</author>
<summary>
This book is the guide to design <keyword>XML</keyword>
applications.
</summary>
</book>
</books>
```

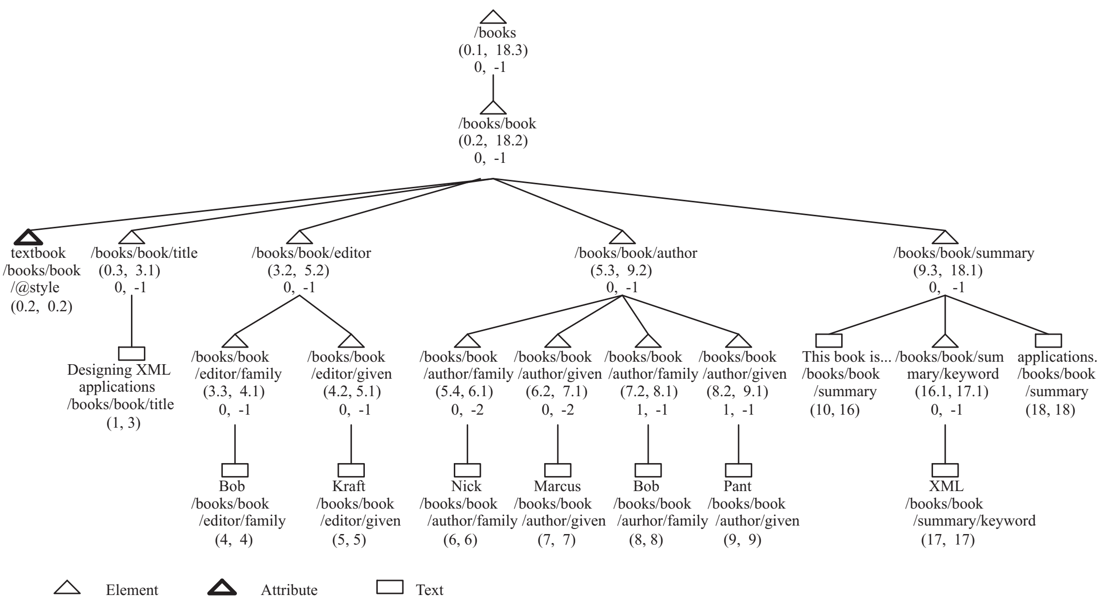

Abbildung 11: Baumstruktur und Nummerierung des books-Dokument

A.5.2 Beispielrelationen

path

pathexp	pathID
/books	1
/books/book	2
/books/book/style	3
/books/book/title	4
/books/book/editor	5
/books/book/editor/family	6
/books/book/editor/given	7
/books/book/author	8
/books/book/author/family	9
/books/book/author/given	10
/books/book/summary	11
/books/book/summary/keyword	12

text

docID	pathID	value	pos
1	4	Designing XML applications	1, 3
1	6	Bob	4, 4
1	7	Kraft	5, 5
1	9	Nick	6, 6
1	10	Marcus	7, 7
1	9	Bob	8, 8
1	10	Pant	9, 9
1	11	This book is the guide to design	10, 16
1	12	XML	17, 17
1	11	applications.	18, 18

attribute

docID	pathID	attvalue	pos
1	3	textbook	0.2, 0.2

element

docID	pathID	index	reindex	pos
1	1	0	-1	0.1, 18.3
1	2	0	-1	0.2, 18.2
1	4	0	-1	0.3, 3.1
1	5	0	-1	3.2, 5.2
1	6	0	-1	3.3, 4.1
1	7	0	-1	4.2, 5.1
1	8	0	-1	5.3, 9.2
1	9	0	-1	5.4, 6.1
1	10	0	-1	6.2, 7.1
1	9	1	-2	7.2, 8.1
1	10	1	-2	8.2, 9.1
1	11	0	-1	9.3, 18.1
1	12	0	-1	16.1, 17.1

Literatur

[Abi97] Serge Abiteboul. *Querying Semi-Structured Data* In: Proceedings of the International Conference on Database Theory. 1997.

[Beh00] Ralf Behrens. *Ein XML-basiertes Modell zur Modellierung und Strukturierung von Informationen im Kontext eines Medienarchivs für die Lehre.* Inauguraldissertation, Lübeck November 2000. `http://www.ifis.mu-luebeck.de/public/Diss/dissrb.ps.gz`

[DOM00] *Document Object Model Level 2 Core.* W3C Recommendation. 13 November 2000. `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/`

[DSL01] Don Box, Aaron Skonnard, John Lam. *Essential XML - XML für die Softwareentwicklung.* Addison-Wesley, München 2001.

[Heu97] Andreas Heuer. *Objektorientierte Datenbanken - Konzepte, Modelle, Standards und Systeme.* 2., aktualisierte Auflage, Addison-Wesley, Bonn 1997.

[HS00] Andreas Heuer, Gunter Saake. *Datenbanken: Konzepte und Sprachen.* MITP-Verlag, Bonn 2000.

[KM02] Meike Klettke, Holger Meyer. *XML & Datenbanken - Konzepte, Sprachen und Systeme*. dpunkt.verlag, Heidelberg (Oktober 2002;-)

[Leh01] Patrick Lehti. *Design and Implementation of a Data Manipulation Processor for an XML Query Language.* August 2001. `http://www.lehti.de/beruf/diplomarbeit.pdf`

[LH01] Peiya Liu, Liang H. Hsu. *A Logic Approach to XML Document Update Query Specifications* XML Europe 2001, Berlin 2001. `http://www.gca.org/papers/xmleurope2001/papers/html/s13-2.html`

[LH02] Peiya Liu, Liang H. Hsu. *Queries of Digital Content Descriptions in MPEG-7 and MPEG-21 XML documents* XML Europe 2002, Barcelona 2002. `http://www.idealliance.org/papers/xmle02/dx_xmle02/papers/03-02-01/03-02-01.html`

[LM00] Andreas Laux, Lars Martin. *XUpdate - XML Update Language.* Working Draft. 14.09.2000. `http://www.xmldb.org/xupdate/xupdate-wd.html`

[Mar00] Lars Martin. *Requirements for XML Update Language.* Working Draft. 24.11.2000. `http://www.xmldb.org/xupdate/xupdate-req.html`

[May00] Wolfgang May. *An Overview Of XML & Friends* Preliminary report. Freiburg, März 2001. `http://www.informatik.uni-freiburg.de/~may/Lectures/xml-overview.ps`

[May01a] Wolfgang May. *XPathLog: A Declarative, Native XML Data Manipulation Language* Grenoble. 16.07.2001 `http://www.informatik.uni-freiburg.de/~may/Publics/01/ideas01talk.ps`

[May01b] Wolfgang May. *A Rule-Base Querying and Updating Language for XML* DBPL Workshop Frascati. 09.09.2001 `http://www.informatik.uni-freiburg.de/~may/Publics/01/dbpl01talk.ps`

[Oba02] Dare Obasnjo. *A Proposal For A Simple XML Data Manipulation Language* 2002. `http://www.25hoursaday.com/sixdml`

[Ros01] Guido Rost. *Implementierung von XQuery auf einem Objekt-relationalem XML-Speicher.* Diplomarbeit, Universität Rostock. 6. Dezember 2001.

[Sch02] Heiko Schuldt. *Transaktionsverwaltung in modernen Informationssystemen* Skripte zur Vorlesung WS 2001/2002. ETH Zürich. `http://www-dbs.inf.ethz.ch/~timi/WS_01_02/folien/V-Moderne-IS.pdf`

[SH99] Gunter Saake, Andreas Heuer. *Datenbanken: Implementierungstechniken.* MITP-Verlag, Bonn 1999.

[Sta01] Kimbro Staken. *XML:DB XUpdate Use Cases* 20.11.2001. `http://www.xmldatabases.org/projects/XUpdate-UseCases/`

[SYU99] Takeyuki Shimura, Masatoshi Yoshikawa, Shunsuke Uemur. *Storage and Retrieval of XML Documents using Object-Relational Databases* DEXA 1999.

[TIHW01] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, Daniel S. Weld. *Updating XML* ACM SIGMOD '01, May 2001.

[XNS99] *Namespaces in XML.* W3C Recommandation. 14 Januar 1999. `http://www.w3.org/TR/1999/REC-xml-names-19990114`

[XML00SE] *Extensible Markup Language (XML) 1.0 (Second Edition).* W3C Recommendation. 6 October 2000. `http://www.w3.org/TR/2000/REC-xml-20001006`

[XInfo01] *XML Information Set.* W3C Recommendation. 24 October 2001. `http://www.w3.org/TR/2001/REC-xml-infoset-20011024`

[XQ01] *XQuery 1.0: An XML Query Language* W3C Working Draft. 20 December 2001. `http://www.w3.org/TR/2001/WD-xquery-20011220`

[XQReq01] *XML Query Requirements.* W3C Working Draft. 15 February 2001. `http://www.w3.org/TR/2001/WD-xmlquery-req-20010215`

[XQDM01] *XQuery 1.0 and XPath 2.0 Data Model.* W3C Working Draft. 20 December 2001. `http://www.w3.org/TR/2001/WD-query-datamodel-20011220`

[XQFS01] *XQuery 1.0 Formal Semantics.* W3C Working Draft. 07 June 2001. `http://www.w3.org/TR/2001/WD-query-semantics-20010607`

[XQX01] *XML Syntax for XQuery 1.0 (XQueryX).* W3C Working Draft. 07 June 2001. `http://www.w3.org/TR/2001/WD-xqueryx-20010607`

[XSM01] *XML Schema Part 0: Primer.* W3C Recommendation. 2 May 2001. `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`

[Zei01] Andre Zeitz. *Evolution von XML-Dokumenten.* Studienarbeit, Universität Rostock, 2001.

Abbildungsverzeichnis

Tabellenverzeichnis

Thesen

1. Änderungsoperationen sollen ein integraler Bestandteil einer vollständigen (XML-) Anfragesprache sein.
2. Updates auf XML-Dokumenten sind nicht äquivalent zu XML-Transformationen, da diese eher eine Umstrukturierung im Sinne von XML-Anfragen darstellen und dabei nur neue Instanzen erzeugen können, aber nicht die Änderung der Basisdokumente ermöglichen.
3. Das Infoset reduziert ein XML-Dokument auf seine Kerninformationen und bietet damit die Möglichkeit, Dokumente hinsichtlich ihrer Informationskapazität zu vergleichen, was mit einem Bit-Vergleich ausgeschlossen wäre.
4. Die Entwicklung von XML-Schema wird nicht zu einer vollständigen Ablösung von DTD's führen, da XML-Schema für die manuelle Schemabeschreibung zu komplex und umständlich ist.
5. Die Verwendung eines Schemas sollte einer schemalosen Modellierung vorgezogen werden.
6. Hierarchische Sperrprotokolle, die bisher bei relationalen Systemen eingesetzt wurden, eignen sich auch für XML-Datenstrukturen.
7. Die meisten Ansätze einer XML-Änderungssprache werden *bottom-up* entwickelt, also erst die Implementation und dann der Versuch einer allgemeineren Beschreibung.
8. Die Manipulation von Daten ist auf Systemebene auch mit *SAX* möglich.
9. Die Spezifikation der Sprache *XML:DB-XUpdate* kann nur als Syntaxbeschreibung verstanden werden, tiefer gehende Konzepte werden hier nicht betrachtet.
10. Die Erweiterung der *MMDOC-QL* um Änderungsoperationen erfolgt nicht auf gleicher Ebene wie die Definition des Pfad-Prädikaten-Kalküls und ist damit unbrauchbar.
11. Auch wenn der FLWR-Ausdruck aus XQuery eine Implementation der Ergebnisauswertung ähnlich von Schleifen in Programmiersprachen nahelegt, ist sie so nicht zwingend. Die Bindung der Variablen durch FLWR-Ausdrücke bewirkt nur die Generierung einer Liste von Bindungen, bei der zwar die Dokumentenordnung beachtet werden muss, die Auswertungsstrategie jedoch frei bleibt.
12. Die Abbildung von strukturellen XML-Änderungsoperationen ist auf generische Speichermodelle einfacher umzusetzen als auf schemaabhängige Datenbankstrukturen.

Abbildung 12: SUNFLWR’s can heal the world ;-)

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 30.06.2002