Diplomarbeit

Entwicklung und Implementierung einer Sprache zur Evolution von XML-Schemata

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik

vorgelegt von	:	Christian Will
geboren am	:	08.05.1979 in Neubrandenburg
Gutachter	:	Prof. Dr. Andreas Heuer
	:	Prof. Dr. Peter Forbrig
Betreuerin	:	Dr. Meike Klettke
Abgabedatum	:	05.09.2006

Selbständigkeitserklärung

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

Neubrandenburg, den 3. September 2006

Zusammenfassung

Die Schemaevolution für das relationale und objektorientierte Datenmodell ist weitestgehend untersucht. Die aktuelle Forschung konzentriert sich auf XML und die dazugehörenden Schemasprachen.

Im Rahmen dieser Arbeit wird eine Sprache zur Beschreibung der Evolution von XML-Schema vorgestellt. Die Sprache unterstützt dabei alle Konzepte des zugrunde liegenden Datenmodells. Es wird gezeigt, wie XML-Updatenanweisung zur Anpassung der Instanzdokumente anhand des Schemas und der Schemaänderung generiert werden können. Die Instanzdokumente selber sollen dabei möglichst unbeachtet bleiben.

Abstract

The process of schema evolution for the relational and object-oriented data model is well-known and studied. Current focuses on research are XML and corresponding schema languages.

This paper introduces a language for the evolution process of XML-Schema. The language supports all concepts of the data model. It also shows a way how to adapt the instances by generating XML-Updates depending on the schema and the schema change. The instance documents should be untouched, if possible.

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
I.7	DOCUMENT AND TEXT PROCESSING
I.7.1	Document and Text Editing

Schlüsselwörter

XML, XML-Schema, Datenmodell, Schema Evolution, XQuery, Anfragesprache, Änderungssprache, Dokumentanpassung

Keywords

XML, XML-Schema, Data Model, Schema Evolution, XQuery, Query Language, Update Language, Document Adaption

Inhaltsverzeichnis

1 Einführung und Motivation **1**

2 Grundlagen **5**
2.1 Extensible Markup Language (XML) 5
2.2 Das Datenmodell von XML-Schema 6
2.2.1 Schemakomponenten 7
2.2.2 Datentypen 10
2.2.3 Namensräume 10
2.2.4 Validierungsprozess 11

3 Schemaevolution **13**
3.1 Klassifikation 13
3.2 Anpassen der Instanzen 17
3.2.1 Allgemein 17
3.2.2 XML-Updatesprache 18
3.2.3 Auffinden der Knoten 19
3.2.4 Problem der Standardwerte 19
3.3 Regeln der Schemaevolution 19
3.3.1 Ändern des Schemas 20
3.3.2 Ändern der Attributmenge 21
3.3.3 Ändern der Elementgruppe 28
3.3.4 Ändern des Inhalts der Elemente 34
3.3.5 Ändern des einfachen Datentyps 35
3.3.6 Ändern der Typhierarchie 38
3.3.7 Ändern von Beziehungen 39
3.3.8 Ändern von Identitätsdefinitionen 40

4 Sprachvorschlag **43**
4.1 Einführung 43
4.2 Navigation 44
4.2.1 Pfadausdrücke 44

4.2.2 Schritte . . . 45
4.2.3 Achsen . . . 45
4.2.4 Knotentests . . . 51
4.2.5 Prädikate . . . 53
4.2.6 Abkürzungen . . . 53
4.3 Operationen . . . 55
4.3.1 Insert . . . 55
4.3.2 Delete . . . 57
4.3.3 Set . . . 57
4.3.4 Rename . . . 58
4.3.5 Move . . . 59
4.4 Eigenschaften der Schemaknoten . . . 59
4.4.1 Schemawurzel . . . 59
4.4.2 Attribut-Knoten . . . 60
4.4.3 Element-Knoten . . . 62
4.4.4 Elementgruppen-Knoten . . . 63
4.4.5 Typdefinitions-Knoten . . . 64
4.4.6 Identitätsbeschränkungs-Definitions-Knoten . . . 66
4.4.7 Wildcard- und Attribut-Wildcard-Knoten . . . 67
4.4.8 Elementgruppen-Definitions-Knoten . . . 68
4.4.9 Attributgruppen-Definitions-Knoten . . . 69
4.4.10 Notation-Definitions-Knoten . . . 69
4.4.11 Anmerkungs-Knoten . . . 70
4.5 Konstruktoren . . . 71
4.5.1 Element-Knoten . . . 71
4.5.2 Attribut-Knoten . . . 72
4.5.3 Typdefinitions-Knoten . . . 72
4.5.4 Elementgruppen-Knoten . . . 72
4.5.5 Identitätsbeschränkungs-Definitions-Knoten . . . 73
4.5.6 Wildcard-Knoten . . . 73
4.5.7 Attribut-Wildcard-Knoten . . . 73
4.5.8 Elementgruppen-Definitions-Knoten . . . 74
4.5.9 Attributgruppen-Definitions-Knoten . . . 74
4.5.10 Notations-Knoten . . . 74
4.5.11 Anmerkungs-Knoten . . . 74
4.6 Funktionen . . . 74

5 Implementierungsdetails **79**
5.1 Aufgabe . . . 79
5.2 Verwendete Technologien . . . 79
5.3 Architektur . . . 80

5.4 Beispiel 82
5.5 Ergebnisse 86

6 Verwandte Arbeiten 87

7 Schlussbetrachtung 91
7.1 Zusammenfassung 91
7.2 Ausblick und weitere Ideen 92

A Notation der graphischen Symbole 93

B Grammatik 95
B.1 XQuery mit Erweiterung 95

Verzeichnis der Abkürzungen 99

Verzeichnis der Abbildungen 101

Literaturverzeichnis 107

Kapitel 1

Einführung und Motivation

Mit dem Erfolg des Internets hat sich HTML[1] zu dem am meist verwendeten Austauschformat der Welt entwickelt. HTML ist eine mit SGML[2], einer Meta-Auszeichnungssprache, definierten Sprache. Sie dient vor allem der Darstellung von Informationen, die über das Internet versendet werden. Im Verlauf der Zeit erwies sich HTML jedoch als unzureichend für den Austausch von Informationen. HTML beschreibt zwar deren Darstellung, jedoch weder Struktur und Inhalt.

Daraufhin entwickelte das W3C die Sprache XML[3]. XML ist eine Untermenge von SGML und selbst eine Meta-Auszeichnungssprache. Mit XML war es nun möglich, beliebige Strukturen zu definieren. Um weiterhin die Darstellung der Informationen zu ermöglichen, wurde XSLT entwickelt. XSLT beschreibt die Umformung eines XML-Dokumentes in ein anderes XML- oder HTML-Format.

Prinzipiell kommt XML ohne eine Schemasprache aus, da die Struktur implizit durch das Dokument definiert wird. Um jedoch sicherstellen zu können, dass ein Dokument einer gewissen Struktur entspricht, wurde in der ersten Version von XML die Schemasprache DTD aus dem SGML-Umfeld übernommen. DTD entsprach jedoch nicht den derzeitigen Anforderungen. Sie ist streng hierarchisch, besitzt nur wenige statische Datentypen und einen rudimentären Referenzierungsmechanismus. Zudem gibt es keine Möglichkeit der Wiederverwendung oder Trennung von Komponenten.

Das W3C begann demzufolge mit der Entwicklung von XML-Schema, wovon im Mai 2001 die erste Empfehlung veröffentlicht wurde. XML-Schema behob die meisten Mängel von DTD. Die strenge Hierarchie wurde aufgebrochen. Eine strikte Trennung zwischen Deklaration und Typdefinition wurde

[1] Hypertext Markup Language
[2] Standard Generalized Markup Language
[3] Extensible Markup Language

eingeführt. Es gibt eine große Menge vordefinierter Typen und die Möglichkeit, neue Typen durch Erweiterung und Einschränkung existierender zu bilden. Der Referenzierungsmechanismus wurde mit einem aus Datenbanken ähnlichen Mechanismus zum Definieren von Schlüsselbeziehungen erweitert. Durch den reichhaltigen Umfang und der erreichten Akzeptanz hat die Integration des Schemas in den Prozess der Softwareentwicklung begonnen. Es werden Tools zur Schemaeditierung, -konvertierung und -visualisierung entwickelt. Es entstehen Methoden für Design Pattern (Entwurfsmuster) und Leitfaden zur Schemaerstellung. Es gibt Programmierschnittstellen wie z.B. die XML-Schema-API, die den Zugriff auf die Schemainformationen ermöglicht. XML selber enthält keinen Programmcode, es gibt jedoch Tools die Code auf Basis des Schemas generieren.

Durch den Softwareentwicklungsprozess und den immer weiter wachsenden Datenaustausch unterschiedlichster Teilnehmer unterliegt ein Schema einem ständigen Evolutionsprozess. Diese Arbeit widmet sich dem Thema der Schemaevolution. Es wird eine Sprache entworfen, mit der auf der Basis des Datenmodells von XML-Schema Strukturänderungen beschrieben werden können. Außerdem wird untersucht, welche Änderungen welche Auswirkungen auf die existierenden Instanzen haben und wenn möglich wie XML-Updateanweisungen generiert werden können.

Ziel und Aufbau der Arbeit

Der Aufbau der Arbeit stellt sich wie folgt dar:

- **Kapitel 1: Einführung und Motivation**
  Dieses Kapitel gibt eine kurze Einführung und dient vor allem der Motivation für die Thematik.

- **Kapitel 2: Grundlagen**
  Das zweite Kapitel vermittelt die Grundlagen und Konzepte von XML-Schema, die notwendig sind, um die folgenden Kapitel zu verstehen.

- **Kapitel 3: Schema Evolution**
  Das dritte Kapitel widmet sich der Schema Evolution. Es wird eine Klassifizierung möglicher Änderungen erstellt. Für jede Änderung wird gezeigt, welche Auswirkung sie auf das Schema und deren Instanzen hat. Es werden weiterhin Updateanweisungen generiert, welche die Instanzen anpassen.

- **Kapitel 4: Sprachvorschlag**
  Dieses vierte Kapitel enthält den Sprachvorschlag, der speziell für die

Evolution von XML-Schemata entworfen wurde.

- **Kapitel 5: Implementierungsdetails**
  Das fünfte Kapitel beschreibt, wie die prototypische Implementierung umgesetzt wurde.

- **Kapitel 6: Verwandte Arbeiten**
  Das sechste Kapitel gibt einen Überblick über verwandte Arbeiten.

- **Kapitel 7: Schlussbetrachtung und Ausblick**
  Das letzte Kapitel dient der Zusammenfassung. Es wird außerdem ein Ausblick gegeben und weitere Ideen zusammengetragen.

Kapitel 2

Grundlagen

In diesem Kapitel wird das Basiswissen zusammengetragen, was für das Verständnis der darauf folgenden Kapitel wichtig ist. Dazu gehört ein kurzer Einblick in das XML-Format und das Datenmodell von XML-Schema.

2.1 Extensible Markup Language (XML)

Die *Extensible Markup Language* ist eine Empfehlung des W3C[1] vom Februar 1998. Es handelt sich dabei um eine Teilmenge der aus dem Jahre 1986 von der ISO standardisierten Sprache SGML[2]. XML ist eine Meta-Auszeichnungssprache und erlaubt das Speichern semistrukturierter Daten.

Semistrukturierte Daten besitzen daten- und dokumentzentrierte Anteile. Der datenzentrierte Anteil besitzt eine reguläre und feste Struktur und lässt sich sehr effektiv mit dem Relationalen Datenmodell darstellen. Hingegen ist der dokumentzentrierte Anteil meist ohne oder mit irregulärer Struktur und unterliegt häufigen Änderungen. Dies spiegelt sich auch auf die Komplexität des Schemas nieder. Während der datenzentrierte Anteil mit wenigen Schemainformationen auskommt, ist der dokumentzentrierte Anteil oft sehr umfangreich.

Ein XML-Dokument besteht hauptsächlich aus Elementen und Attributen. Elemente sind die Auszeichnung (engl. Markup) und bestehen aus jeweils einem Start- und End-Tag. Elemente können Zeichenketten und andere Elemente als Inhalt haben. Ein Attribut ist immer an ein Element gebunden und kann ausschließlich Zeichenketten als Inhalt besitzen. Ein XML-Dokument besitzt genau ein Wurzelelement, wodurch die Abbildung des Dokumentes einem Baum entspricht.

[1] World Wide Web Consortium

[2] Standard Generalized Markup Language

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
  <book year="2007">
    <title>XML Everywhere</title>
    <author>Michael K.</>
    <content>
      <headline>XML...</headline>
      The generalized <cursiv>markup</cursiv>...
    </content>
  </book>
</books>
```

Abbildung 2.1: Ein Beispiel für ein XML Dokument

Die Abbildung 2.1 zeigt ein Beispiel eines XML-Dokumentes. Das Wurzelelement *books* enthält ein Unterelement *book*, welches ein Attribut *year* und weitere Unterelemente enthält.

Neben Elementen und Attributen gibt es noch Kommentare, PIs[3], Entities und CDATA-Sektionen. Sie spielen jedoch für die Arbeit keine Rolle und daher wird bei Interesse auf die W3C-Empfehlung [BPSM+04] verwiesen.

2.2 Das Datenmodell von XML-Schema

Dieser Abschnitt gibt einen Überblick über die der Arbeit zugrunde liegenden Schemasprache XML-Schema. Sie ist eine Entwicklung des W3C und liegt in der zweiten Version vor, die seit Oktober 2004 den Status einer Empfehlung besitzt. Die Empfehlung ist in drei Dokumente unterteilt: Einführung [FW04], Strukturen [TBMM04] und Datentypen [BM04]. Nach einer Einführung werden die Schemakomponenten, Datentypen, das Prinzip der Namensräume und der Validierungsprozess beschrieben.

Einführung

Es wird zwischen zwei Grundtypen von Schemasprachen unterschieden. Der erste erstellt eine komplette kontextfreie Grammatik für die Dokumente und erzeugt einen Top-Down Parser zur Verarbeitung. Der zweite Typ erlaubt es dem Benutzer gewisse Regeln zu definieren, die nur gewünschte Teile des Dokumentes betreffen. Grammatik basierte Sprachen sind unter anderem Relax NG[CM01], DTD[BPSM+04] und das in dieser Arbeit verwendete XML-Schema[FW04]. Ein Beispiel einer regelbasierten Sprache ist Schematron[SCH04].

[3] Processing Instruction

DTD[4], was als Vorgänger von XML-Schema bekannt ist, besitzt eine strenge Hierarchie. Alle Elemente werden global definiert und es gibt keine Möglichkeit zwei Elemente gleichen Namens und unterschiedliches Typs zu haben. Mit XML-Schema ist es möglich diese Hierarchie aufzubrechen. Eine strikte Trennung von Deklaration und Definitionen wurde eingeführt. Definitionen und Deklarationen können einen globalen oder lokalen Gültigkeitsbereich besitzen. Durch die Einführung von Namensräumen können auch Elemente verschiedener Schemata unterschieden werden.

Der aus SGML bekannte ID/IDREF-Mechanismuss wurde um einen stärkeren Schlüsselmechanismus erweitert.

Das HTML Format kann man leicht erweitern, da unbekannte Tags vom Parser ignoriert werden. Um dies auch für XML zu ermöglichen wurde ein Mechanismus zur Definition fremden Inhalts eingeführt.

2.2.1 Schemakomponenten

Das Datenmodell von XML-Schema besteht aus 13 Komponenten, die in drei Gruppen unterteilt werden.

- **Primäre Komponenten -** Die Primären Komponenten dienen der Definition von Typen, sowie der Deklaration von Attributen und Elementen.
  - **Einfache Typendefinition:** Zu den Einfachen Typdefinitionen zählen alle atomaren Datentypen, Listen und Vereinigungen einfacher Typen. Einfache Typdefinitionen können keine Struktur, wie Attribute oder Unterelemente, besitzen.
  - **Komplexe Typendefinition:** Komplexe Typendefinitionen erlauben das zuordnen von Attributen und Unterelementen. Dabei wird zwischen vier Inhaltsmodellen unterschieden.
    - **Leerer Inhalt:** Das Element besitzt keinerlei Inhalt, weder Elemente, Attribute oder Zeichen.
    - **Einfacher Inhalt:** Als Basis wird eine einfache Typdefinition verwendet, die zusätzlich mit einer Menge von Attributen erweitert werden kann.
    - **Komplexer Inhalt:** Das Element enthält keine Zeicheninformationen. Attribute und Unterelemente sind erlaubt.

[4]Document Type Definition

  * **Gemischter Inhalt:** Das Element darf Zeichen, Attribute und Unterelemente besitzen. Die Angabe und damit Validierung der Zeichenkette mit einer Einfachen Typdefinition, wie beim Einfachen Inhalt, ist jedoch nicht möglich.

- **Attribut-Deklaration:** Eine Attribut-Deklaration besitzt einen Namen und weist auf eine Einfache Typdefinition. Mit Hilfe einer Attribut-Verwendung wird sie entweder einer Komplexen Typdefinition oder einer Attributgruppe zugewiesen.

- **Element-Deklaration:** Eine Element-Deklaration besitzt einen Namen und weist auf eine Einfache- oder Komplexe Typdefinition. Global definierte nicht abstrakte Element-Deklarationen können als Wurzelelement der Instanz verwendet werden.

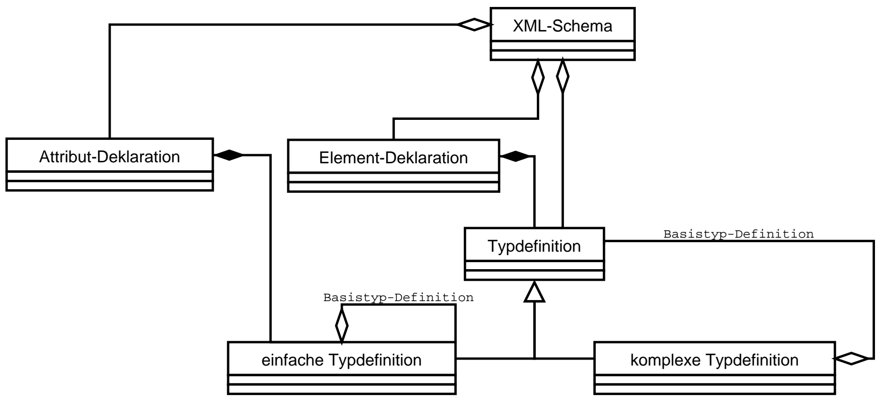


Abbildung 2.2: Darstellung der Primären Komponenten

• **Hilfskomponenten -** Die Hilfskomponenten definieren die Beziehungen der Primären Komponenten. Zusätzlich können Anmerkungen eingefügt und fremder Inhalt definiert werden.

  - **Attribut-Verwendung:** Eine Attribut-Verwendung bestimmt die Beziehung zwischen einer Komplexen Typdefinition und einer Attribut-Deklaration. Die Attribut-Verwendung bestimmt, ob das Attribut optional oder erforderlich ist. Auch die Definition von default oder festen Vorgabewerten ist möglich.

  - **Partikel:** Ein Partikel bestimmt das Vorkommen einer Element-Informationseinheit (Element oder Elementgruppe) in einer Elementgruppe.

- **Elementgruppe:** Eine Elementgruppe ist entweder eine Sequenz, Auswahl oder Menge von Element-Informationseinheiten. Eine Element-Informationseinheit ist entweder eine Element-Deklaration, Elementgruppen-Definition und selbst Elementgruppe.
- **Wildcard:** Mit Hilfe einer Wildcard kann fremder Inhalt eingefügt werden. Es gibt jeweils ein Wildcard zur Definition von Elementen und Attributen. Der fremde Inhalt kann jegendlich durch eine Liste von Namensräumen eingeschränkt werden. Weiterhin gibt es die Möglichkeit den Validierungslevel zu bestimmen, was im Abschnitt Validierungsprozess genauer erläutert wird.
- **Anmerkung:** Beinahe jede Komponente kann mit Anmerkungen erweitert werden. Es wird zwischen Anmerkungen für den Benutzer und für die Applikation unterschieden. Auf den Validierunsprozess haben sie jedoch keinen Einfluss.

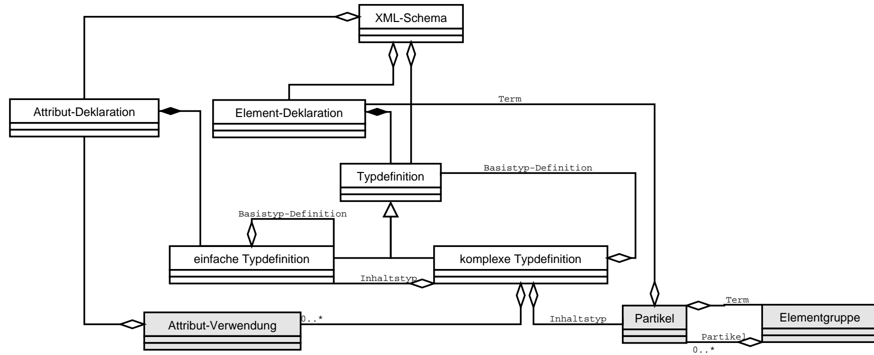


Abbildung 2.3: Darstellung der Hilfskomponenten

- **Sekundäre Komponenten -** Die Sekundären Komponenten dienen unter anderem der Gruppierung bestehender Primärer Komponenten und der Definition von Schlüsselbeziehungen.
  - **Attribut- und Elementgruppen-Definitionen:** Attribut- und Elementgruppen-Definitionen bieten eine weitere Möglichkeit der Wiederverwendung, indem sie Elemente bzw. Attribute zu einer Gruppe zusammenzufassen. Diese Gruppe kann dann in jede beliebige Komplexe Typdefinition einzufügt werden.
  - **Identitätsbeschränkungs-Definition:** Mit Identitätsbeschränkungs-Definitionen können aus dem Relationalen Datenmodell bekannte Schlüssel-/Fremdschlüsselbedingungen definiert werden.

  - **Notations-Deklaration:** Eine Notation kann nur in der Schemawurzel definiert werden. Sie stellt zusätzliche Informationen über das Schema dem Benutzer oder einer Applikation zur Verfügung. Die Notations-Deklaration ist nicht an dem Validierungsprozess beteiligt.

2.2.2 Datentypen

XML-Schema besitzt ein ausgereiftes Datenmodell mit 45 vordefinierten Typdefinitionen. Es gibt einen Vererbungsmechanismus, der es erlaubt durch Erweiterung, Einschränkung und Neudefinition Typendefinitionen zu bilden. Es wird zwischen Komplexen- und Einfachen Datentypen unterschieden. Einfache Datentypen besitzen keine Struktur. Mit Hilfe von so genannten Fassetten wird die Menge der erlaubten Zeichenketten eingeschränkt.

Komplexen Datentypen werden in vier Inhaltsgruppen unterteilt:

- **Leer -** Das Element besitzt keinen Inhalt.
- **Einfach -** Das Element besitzt Zeichenketten und möglicherweise Attribute. Die Zeichenketten werden gegenüber einem Datentyp validiert.
- **Element -** Das Element besitzt Unterelemente oder Attribute. Zeichenketten sind nicht erlaubt.
- **Gemischt -** Entspricht dem Komplexen Inhalt, wobei zusätzlich Zeichenketten erlaubt sind. Die Zeichenketten werden aber nicht gegen einen Datentyp validiert.

Die Abbildung 2.4 zeigt einen Überblick der in XML-Schema vordefinierten Datentypen. Für eine detaillierte Beschreibung wird auf das W3C Dokument [BM04] verwiesen.

2.2.3 Namensräume

Der Namensraum ist ein Begriff der Informatik und beschreibt allgemein die Zuordnung eines Objektes.

Im Umfeld von XML besitzt jedes Schemadokument einen Zielnamensraum. Der Zielnamensraum ist eine URI oder wenn keine angegeben wurde *##local*. Jedes globale Objekt in einem XML-Schema wird eindeutig mit einem QName[5], welcher aus Namensraum und Namen des Objektes besteht,

[5] Qualifizierter Namen

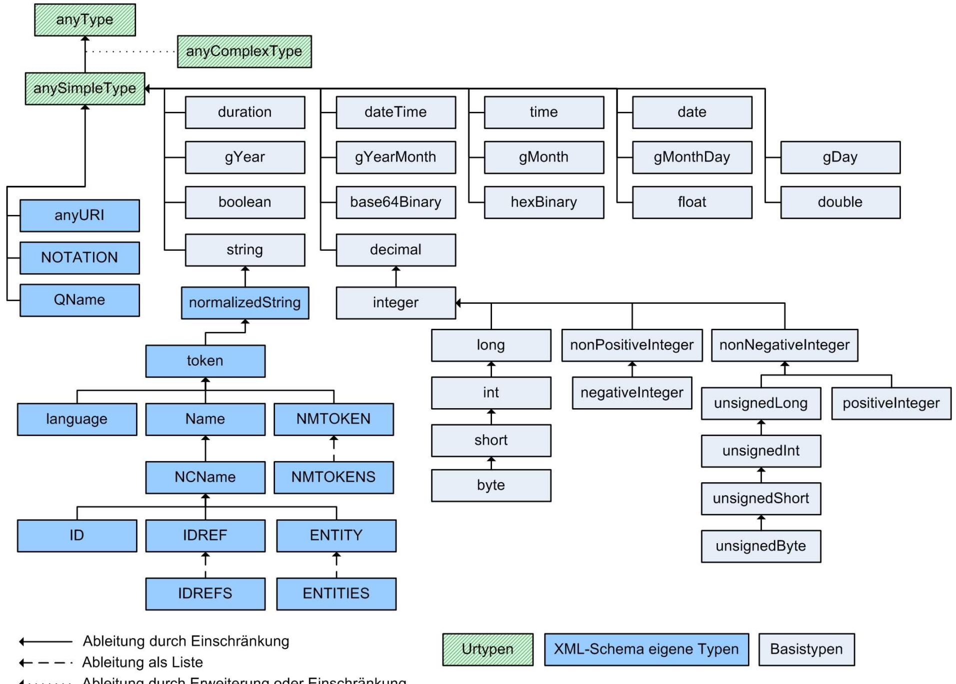


Abbildung 2.4: Überblick der Datentypen

identifiziert. Lokale Objekte, die Bestandteil anderer Objekte sind, können nicht referenziert werden.

Das Konzept der Namensräume kann zur Modularisierung und Versionierung verwendet werden. Zudem können global definierte Objekte gleichen Namens mit unterschiedlichen Typen verwendet werden.

2.2.4 Validierungsprozess

Die Validierung der XML-Instanzen unterteilt sich in vier Abschnitte: Wohlgeformtheit, Struktur, Inhalt und Integrität. Abhängig von der Implementation laufen sie gleichzeitig oder hintereinander ab.

Die Wohlgeformtheit ist in der XML-Spezifikation [BPSM+04] definiert und wird für jedes XML-Dokument, unabhängig von einem Schema, gefordert. Sie sichert unter anderem zu, dass alle Elemente einen Start- und End-Tag besitzen, keine Attribute gleichen Namens in einem Element vorkommen, es maximal ein Wurzelelement gibt und dass keine nicht erlaubten Zeichen vorkommen.

Die Überprüfung der Struktur ist Teil der Schemavalidierung. Dabei werden Namen, Position und Vorkommen von Elementen und Attributen überprüft. Die Verarbeitung nimmt die meiste Zeit in Anspruch und wird mit

einem generierten Top-Down Parser durchgeführt.

Während der Validierung der Struktur wird der Inhalt der Elemente und Attribute erst normalisiert und danach anhand im Schema vorgegebener Datentypen überprüft.

Integritätsbedingungen lassen sich mit XML-Schema in Form von Schlüssel/Fremdschlüsselbeziehungen definieren. Die Überprüfung ist erst möglich, wenn die vorherigen drei Abschnitte abgeschlossen sind. Eine Schwäche von XML-Schema ist es, dass es keine Möglichkeit gibt, Integritätsbedingungen über eine Menge von Dokumenten zu definieren.

XML-Schema unterscheidet zwischen drei Validierungsstufen: strikt (engl. strict), locker (engl. lax) und gar nicht (engl. skip). In einem Schema kann die Validierungsstufe nur für fremden Inhalt, der mit Wildcards definiert wird, gewählt werden. Bei einer strikten Validierung muss jedes Element eine Deklaration im Schema besitzen. Es wird eine komplette Validierung der Struktur, des Datentyps und der Integrität durchgeführt. Bei einer lockeren Validierung werden nur Elemente geprüft, wenn eine Deklaration im Schema existiert. Ist keine Validierung gefordert, wird nur die Wohlgeformtheit geprüft.

Kapitel 3

Schemaevolution

Der Schwerpunkt dieses Kapitels ist die Schemaevolution. Für das Datenmodell von XML-Schema wird eine Klassifizierung möglicher Änderungen erstellt. Für jede dieser Änderungen wird untersucht, welche Auswirkungen dies auf das Schema und deren Instanzen hat. Da nicht jede Änderung sinnvoll ist, werden bestimmte Einschränkungen für die Implementierung bestimmt. Um die Gültigkeit der Instanzen zu erhalten, wird gezeigt wie diese mit Hilfe einer XML-Updatesprache angepasst werden können.

3.1 Klassifikation

Dieser Abschnitt stellt eine Klassifizierung der möglichen Schemaänderungen des XML-Schema Datenmodells vor.

OODB

Die Schemaevolution für das OODM ist bereits sehr gut erforscht. Banerjee stellte in der Arbeit [BKKK87] eine Klassifizierung der Schemaevolutonsschritte für das Datenmodel ORION vor. Die Einteilung geschieht in drei Kategorien. Weiterhin werden diese Kategorien so weit verfeinert bis eine Menge von Schemaevolutionsschritten entsteht. Für eine detaillierte Beschreibung wird auf die Arbeit von Banerjee verwiesen.

1. Ändern des Inhalts der Klasse
    - Ändern der Attributmenge
        - Einfügen, Entfernen und Umbenennen eines Attributes
        - Ändern des Wertebereichs
        - Ändern des Vorgabewertes

    - Ändern der Vererbung
    - Ändern des gemeinsamer Attribute
      * Einfügen, Entfernen und Umbenennen eines gemeinsamen Attributes
  - Ändern der Methoden
    - Einfügen, Entfernen und Umbenennen einer Methode
    - Ändern des Programmcodes
    - Ändern des Vorgabewertes
    - Ändern der Vererbung

2. Ändern der Klassenhierarchie
   - Definieren einer Klasse als Oberklasse einer anderen Klasse
   - Entfernen einer Klasse aus der Hierarchie
   - Ändern der Ordnung der Hierarchie

3. Ändern der Klassenmenge
   - Hinzufügen einer neuen Klasse
   - Entfernen einer Klasse
   - Umbenennen einer Klasse

XML-Schema

Nach dem Vorbild von Banerjee's Arbeit wird in diesem Abschnitt eine Klassifizierung möglicher Schemaevolutionsschritte für das Datenmodell von XML-Schema [TBMM04] erstellt. Diese Klassifizierung bildet die Basis für die darauf folgenden Abschnitte.

Die Klassifikation ist in fünf Gruppen unterteilt. Die erste Gruppe enthält alle Operationen, die auf die Schemawurzel anwendbar sind. Dazu gehört das Hinzufügen und Entfernen von Elementen. Die zweite Gruppe enthält alle Operationen, die auf den Inhalt einer Typdefinition angewandt werden können. Alle Operationen, welche die Typhierarchie betreffen, befinden sich in der dritten Gruppe. XML-Schema besitzt den aus SGML bekannten ID/IDREF-Mechanismus und eine Möglichkeit zur Definition von Schlüssel/Fremdschlüsselbeziehungen. Die vierte Gruppe enthält alle Operationen zur Definitionen von Schlüsselbeziehungen. Die letzte und fünfte Gruppe enthält alle Operationen für den ID/IDREF-Mechanismus.

1. Ändern des Schemas

- Hinzufügen und Entfernen von Typdefinitionen
- Hinzufügen und Entfernen von Attributdeklarationen
- Hinzufügen und Entfernen von Elementdeklarationen
- Hinzufügen und Entfernen von Attributgruppen-Definitionen
- Hinzufügen und Entfernen von Elementgruppen-Definitionen
- Hinzufügen und Entfernen von Notationen
- Hinzufügen und Entfernen von Anmerkungen

2. Ändern des Inhalts einer Typdefinition

   (a) Ändern der Attributmenge
      - Hinzufügen eines neuen Attributes
      - Löschen eines Attributes
      - Umbenennen eines Attributes
      - Ändern einer Attributeinheit
        - Ändern des Attributverwendung
        - Ändern der Wertebereichsbeschränkung
        - Ändern der Typdefinition
      - Fremde Attribute
        - Hinzufügen einer Attribut-Wildcard
        - Entfernen einer Attribut-Wildcard
        - Ändern der Attribut-Wildcard

   (b) Ändern der Elementgruppe (Komplexer Inhalt)
      - Ändern des Typs (Sequenz, Alternative, Menge)
      - Ändern des Minimalen und Maximalen Vorkommens
      - Hinzufügen eines neuen Elementes
      - Entfernen eines Elementes
      - Umbenennen eines Elementes
      - Hinzufügen einer Elementgruppe
      - Entfernen einer Elementgruppe
      - Fremde Elemente
        - Hinzufügen einer Wildcard
        - Entfernen einer Wildcard

   (c) Ändern des Inhalts der Elemente
      - Ändern der zugewiesenen Typdefinition

- Ändern des Minimalen und Maximalen Vorkommens
- Ändern der Nullwertfähigkeit

(d) Ändern des einfachen Datentyps (Einfacher Inhalt)

- Ändern des Typs (Atomar, Liste, Vereinigung)
- Grundlegende Fassetten
- Einschränkende Fassetten

(e) Umbenennen der Typdefinition

(f) Ändern des Ziel-Namensraumes

(g) Festlegen des Inhaltstyps

(h) Festlegen der Eigenschaft Abgeschlossenheit

3. Ändern der Typhierarchie
   - Einfache Typdefinition zu einer Vereinigung hinzufügen
   - Einfache Typdefinition aus einer Vereinigung entfernen
   - Komplexe Typdefinition in Hierarchie als Erweiterung einfügen
   - Komplexe Typdefinition in Hierarchie als Einschränkung einfügen
   - Entfernen einer Typdefinition aus der Hierarchie
4. Ändern von Beziehungen
   - Hinzufügen eines Primärschlüssels (key)
   - Entfernen eines Primärschlüssels
   - Hinzufügen eines optionalen Primärschlüssels (unique)
   - Entfernen eines optionalen Primärschlüssels
   - Hinzufügen eines Fremdschlüssels (keyref)
   - Entfernen eines Fremdschlüssels
5. Ändern von Identitätsdefinitionen
   - Hinzufügen einer Identität (ID)
   - Entfernen der Identität
   - Hinzufügen einer Identitätsreferenz (IDREF)
   - Entfernen einer Identitätsreferenz

3.2 Anpassen der Instanzen

Teil der Schemaevolution ist auch das Anpassen der zugehörigen Instanzen. Dieser Abschnitt gibt einen Überblick über allgemeine Regeln und Probleme.

3.2.1 Allgemein

Das Ändern von Komponenten die keine Instanzen besitzen ist immer möglich, solange das Schema ein gültiges Schema bleibt. Daher wird im folgenden Abschnitt immer davon ausgegangen, dass eine Instanz existiert.

Die Änderung von Komponenten mit Instanzen ist nicht immer möglich. Für jede Schemaänderung wird eine Einstufung vorgenommen.

1. **Niemals -** Beinhaltet alle Schemaänderung, die keine Anpassung der Instanzen nach sich ziehen.
2. **Möglichweise -** Alle Anweisungen bei denen erst nachgeschaut werden muss, ob eine Änderung nötig ist.
3. **Immer -** Beinhaltet alle Anweisungen die mich Sicherheit eine Anpassung der Instanz nach sich zieht.
4. **möglicherweise oder immer Benutzer nötig -** Einige Anpassungen der Instanzen sind nicht automatisch möglich, da der erforderliche Inhalt nicht automatisch generiert werden kann.

Architektur der Anwendung

Die Möglichkeit der Instanzanpassung kann auch durch die Architektur der Anwendung beeinflusst werden.

1. Schema und Instanzen können beliebig geändert werden.
2. Ein Teil der Instanzen kann nicht verändert werden.
   (a) Alle Schemaänderungen sind mit Hilfe von Versionierung möglich.
   (b) Schema muss kompatibel zum alten Schema bleiben.

In der Abbildung 3.1 sind drei Beispiele dargestellt. Das erste Beispiel enthält nur eine Anwendung. Dabei ist der Zugriff auf das Schema und die Instanzen nicht eingeschränkt. Das zweite Beispiel besitzt zwei Anwendungen. Die zweite Anwendung kann nicht verändert werden und erzeugt immer

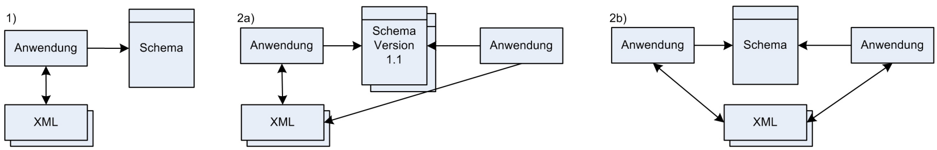


Abbildung 3.1: Abbildung der Architekturen

XML-Instanzen nach dem alten Schema. Mit einer Versionierung kann hier zwischen den XML-Instanzen unterschieden werden. Liest die erste Anwendung Daten des alten Schemas müssen sie umgewandelt werden. Im dritten Beispiel kommt hinzu, dass die zweite Anwendung nicht nur Daten erzeugt, sondern auch Instanzen liest. In diesem Fall müssen alle Schemaänderungen kompatibel zum alten Schema bleiben.

3.2.2 XML-Updatesprache

Die für die Anpassung der Instanzen ausgewählte XML-Updatesprache muss bestimmte Vorrausetzungen erfüllen. Die Updateanweisung soll möglichst ohne vorherige Betrachtung der Instanzen generiert werden können. Die durch XML-Schema mögliche Rekursion von Elementen muss auch mit Hilfe der Updatesprache formulierbar sein.

Alle Updatesprachen basieren auf XPath für die Selektion der Knotenmenge. Die Ausdrucksstärke entspricht jedoch nicht der Logik 1. Stufe.

Beispiel: Ein Element A besitzt ein Kindelement B. Das Element B referenziert als Kindelement wiederum auf das Element A. Die Instanz kann demnach ein beliebig tiefer Baum mit Elementen der Serie A/B sein. Wird dem Element A oder B z.B. ein Attribut hinzugefügt, würden unendlich viele XPath-Anweisungen (A/B, A/B/A/B, A/B/A/B/A/B, ...) entstehen. Eine Möglichkeit, dies auszudrücken ist eine Sprache mit der Ausdruckskraft der Logik 1. Stufe. Ein Beispiel hierfür ist Conditional XPath[Mar04].

Zur Auswahl stehen: DOM, XUpdate, XQuery, XSLT, LOREL und XML-GL. Aus den betrachteten Sprachen sind nur XQuery und XSLT in der Lage die Rekursion darzustellen. Die Wahl fällt daher auf XQuery, da sie als Anfrage- und Updatesprache entwickelt wurde. XSLT ist eine Transformationssprache, die ein Dokument in ein neues umwandelt. Das Formulieren von Updateanweisung ist sehr umständlich und auch im Nachhinein nicht menschenlesbar.

3.2.3 Auffinden der Knoten

Ausgehend von einem Schemaknoten gibt es zwei Möglichkeiten, die Instanzen zu selektieren, die im Folge der Schemaevolution angepasst werden müssen.

Die erste Möglichkeit ist sehr leicht umsetzbar, ist aber auch sehr zeitaufwendig. Während ein XML-Parser eine Instanz analysiert, durchläuft er gleichzeitig den Schemagraphen und überprüft die Gültigkeit der Instanz. Dieser Vorgang kann verwendet werden, um alle Instanzknoten, die den zu verändernden Schemaknoten referenzieren, zu selektieren. Bei großen Datenmengen ist diese Methode jedoch ungeeignet. Falls Teile der Daten nicht erreichbar ist sie gar nicht anwendbar.

Die zweite Möglichkeit betrachtet die Instanzdokumente nicht und durchläuft das Schema rückwärts. Dabei wird eine Menge von XPath-Ausdrücken generiert, welche die möglichen Instanzknoten selektiert. Anhand der Informationen aus der Schemaevolutionsanweisung und der generierten XPath-Ausdrücke werden XQuery-Update-Anweisungen generiert. Die Anweisung können dann an die Server mit den Instanzdokumenten versandt werden. Eine Versionierung der Instanzdokumente ist somit auch möglich. Ein Nachteil ist, dass nicht bekannt ist, welche Teile des Schemas unbenutzt sind.

3.2.4 Problem der Standardwerte

Wird ein Schema um eine notwendige Komponente erweitert, müssen die Instanzdokumente möglicherweise um die geforderten Knoten erweitert werden. Problematisch ist jedoch, den neuen Knoten mit Inhalt zu füllen. Ist ein Standardwert oder konstanter Wert definiert, wird dieser eingesetzt. Ist das Element als nullbar (nil) definiert, wird das Element auf den leeren Inhalt gesetzt. Ist jedoch kein Vorgabewert definiert, kann die Schemaevolution nicht durchgeführt werden. Einen zufällig generierten Wert einzufügen, ist sicher nicht sinnvoll. Existiert daher für das neue Element kein definierter Vorgabewert und darf der Inhalt nicht den Nullwert besitzen, wird die Schemaevolution zurückgesetzt.

3.3 Regeln der Schemaevolution

In diesem Abschnitt wird für jede im vorherigen Abschnitt klassifizierte Änderung untersucht, welche Auswirkung sie auf das Schema und den zugehörigen Instanzen hat. Ist eine Instanzanpassung notwendig, wird gezeigt wie eine XML-Updateanweisung generiert werden kann. Weiterhin werden

gewisse Regeln aufgestellt, welche die möglichen Schemaänderungen sinnvoll einschränken.

3.3.1 Ändern des Schemas

Änderung des Schemas umfasst alle Operationen, die Bestandteile aus der Schemawurzel entfernen oder einfügen. Im Vergleich zu relationalen Systemen, wäre dies z.B. das Einfügen oder Löschen einer neuen Relation oder eines Typs.

Hinzufügen neuer Elemente

Das Hinzufügen neuer Elemente (darunter Typdefinitionen, Attribut-, Elementdeklaration, Attribut-, Elementgruppen-Definitionen, Notationen und Anmerkungen) kann die Gültigkeit der Menge der Instanzen nicht beeinflussen. Daher muss ausschließlich die Integrität des Schemas sichergestellt werden. Elemente gleichen Namens und Typs dürfen nicht im Schema vorkommen.

Löschen von Elementen

Notationen und Anmerkungen dienen ausschließlich der Beschreibung des Schemas und können beliebig entfernt werden.

Nicht referenzierte Elemente (Typdefinitionen, Attributdeklarationen, Attributgruppen- und Elementgruppen-Definitionen), die nicht von einer anderen Schemakomponente oder in der Instanz verwendet werden, können ebenfalls ohne einen Evolutionsschritt entfernt werden.

Ein Problem stellt das Löschen von Elementen dar, die noch referenziert werden. Durch das Entfernen entstehen fehlerhafte Referenzen, die z.B. auch durch das Löschen der referenzierenden Komponente gelöst werden kann. Um kaskadierende Löschoperationen zu vermeiden, fordern wir jedoch das vorherige Löschen der Referenz vor dem Löschen des Elementes.

Jede globale, nicht abstrakte Elementdeklaration, die im Schema definiert wurde, kann als Wurzelelement in der XML-Instanz verwendet werden. Falls XML-Instanzen existieren, in denen die Elementdeklaration als Wurzelelement verwendet wurde, darf diese nicht gelöscht werden. Eine XML-Instanz besitzt genau ein Wurzelelement. Um sicherzustellen, dass die Elementdeklaration nicht verwendet, müssen alle Instanzen überprüft werden.

3.3.2 Ändern der Attributmenge

Attribute und Elemente sind die zwei im XML-Standard definierten Informationsträger. Seit XML-Schema und der Einführung Einfacher- und Komplexer Typdefinition gibt es für alles, was mit Attributen darstellbar ist, eine äquivalente Methode für Elemente. Eine Attribut-Deklaration besitzt immer eine Einfache Typdefinition, welche auch von Element-Deklaration verwendet werden kann. Attribute können im Gegensatz zu Elementen keine Ordnung besitzen und lassen sich nicht weiter strukturieren.

Attribut-Deklarationen können lokal oder global definiert werden. Lokale Attribut-Deklarationen treten innerhalb von Typ- oder Attributgruppen-Definitionen auf. Globale Attribute sind in der Wurzel des Schemas definiert und werden innerhalb einer Typ- oder Attributgruppen-Definitionen referenziert. Nur Globale Attribute können referenziert und somit wieder verwendet werden. Lokale Definitionen erleichtern oft die Lesbarkeit und erlauben es, Attribute mit gleichen Namen und Namensraum unterschiedliche Typen zuzuweisen. Der ID/IDREF Mechanismus ist seit XML-Schema nicht nur für Attribute, sondern auch auf Elemente anwendbar.

Neben der Attribut-Deklaration gibt es eine neue Komponente zur Definition fremder Attribute. Die Komponente heißt Attribut-Wildcard. Die Menge der erlaubten fremden Attribute wird dabei einzig durch die Angabe einer Menge aus Namensräumen begrenzt.

Dieser Abschnitt umfasst alle Änderungen, die auf eine Attributmenge angewendet werden können. Eine Attributmenge ist immer an eine Element-Deklaration gebunden. Es können Attribute hinzugefügt und entfernt werden. Weiterhin kann die Attribut-Verwendung sowie die verwendete Typdefinition geändert werden.

Hinzufügen eines Attributes

Beim Hinzufügen eines neuen Attributes wird zwischen fünf Fällen, die sich durch die Attributverwendung und dem Vorgabewert ergeben, unterschieden.

- Zwingend
  Wird ein als zwingend definiertes Attribut hinzugefügt, ist **immer** eine Anpassung der Instanzen erforderlich. Die der Typdefinition zugehörigen Instanzelemente werden um das neue Attribut erweitert. Dabei wird unterschieden zwischen:
  - Ohne fixierten Vorgabewert
    Gibt die Attribut-Deklaration keinen fixierten Vorgabewert vor,

muss bei der Schemaevolution der Inhalt der neuen Attribute vorgegeben werden. Die Definition eines Default-Wertes für zwingende Attribute ist nach dem Schema-Standard nicht möglich.

Den Vorgabewert anhand des Wertebereiches der Typdefinition auszuwählen ist denkbar. Das Problem, einen Wert auszuwählen, wird im Kapitel 3 Abschnitt 2 näher erläutert.

Die XML-Updateanweisung erzeugt die Instanzen des neuen Attributes. Der Inhalt wird dabei bei der Schemaevolutions-Anweisung vorgegeben.

```
Schema: insert attribute $name use='required' {$value}
               into $target

Instanz: for $e in xsd:instance($target) return
         do insert attribute $name{$value} into $e
```

– Mit fixierten Vorgabewert
  Wird der Wertebereich des Attributes auf einen fixierten Vorgabewert eingeschränkt, entfällt die Angabe des Attributinhaltes.

  Die XML-Updateanweisung erzeugt die Instanzen des neuen Attributes. Die neuen Attribute bekommen als Inhalt den fixierten Vorgabewert zugewiesen.

```
Schema: insert attribute $name use='required'
               fixed=$fixedvalue into $target

Instanz: for $e in xsd:instance($target) return
         do insert attribute $name{$fixedvalue} into $e
```

- Optional
  Das Einfügen optionaler Attribute erfordert **niemals** eine Anpassung. Es existieren keine Instanzen vor dem Einfügen, die angepasst werden müssten. Und nach der Schemaänderung ist die Existenz des Attributes nicht zwingend.

  Eine XML-Updateanweisung muss nicht generiert werden.

```
Schema: insert attribute $name into $target
        insert attribute $name default=$defaultvalue
               into $target
        insert attribute $name fixed=$fixedvalue
               into $target
```

```
Instanz: -
```

- Verboten
  Eine Typdefinition erbt alle Attribute ihrer Basistypdefinition. Ist die Typdefinition eine Einschränkung, kann mit einer Attribut-Deklaration, deren Verwendung verboten ist, die Vererbung eines optionalen Attributes aufgehoben werden. Falls bereits Instanzen des Attributes existieren, müssen diese entfernt werden. Da das Attribut des Basistyps immer optional ist, existieren **möglicherweise** Instanzen.

  Die XML-Updateanweisung entfernt alle möglichen Vorkommen des Attributes.

```
Schema: insert attribute $name use='prohibited' into $target

Instanz: for $e in xsd:instance($target/@$name) return
         do delete $e
```

Ausnahme Attribut-Wildcards Enthält die Typdefinition ein Attribut-Wildcard, muss geprüft werden, ob die Existenz des neuen Attributes bereits durch das Wildcard erlaubt war. Falls ja, sind möglicherweise Attribute, die der neuen Attribut-Deklaration entsprechen, in der Instanz. In dieser Situation wird die Schemaevolution durchgeführt, als wäre eine existierende Attribut-Deklaration geändert worden.

Hinweis Was in einigen Implementationen oft missachtet wurde, ist dass Attribut-Deklarationen, deren Verwendung verboten ist, keinen Einfluss auf bestehende Wildcard-Definitionen haben.

Löschen eines Attributes

- Zwingend / Optional
  Wird ein zwingendes oder optionales Attribut gelöscht, **müssen** auch die Instanzen des Attributes entfernt werden. Eine Ausnahme ist, wenn ein Attribut-Wildcard die Existenz des Attributes zusätzlich erlaubt. In diesem Fall werden die Instanzen nicht verändert. Attribut-Deklarationen, welche die Vererbung des zu löschenden Attributes einschränken, werden auch aus dem Schema entfernt.

  Die XML-Updateanweisung entfernt alle möglichen Vorkommen des Attributes.

```
Schema: delete $target

Instanz: for $e in xsd:instance($target) return
         do delete $e
```

- Verboten
  Wird ein verbotenes Attribut entfernt, ist **niemals** eine Instanzanpassung notwendig. Das vorher durch die Attribut-Deklaration unterdrückte Attribut des Basistyps muss optional sein.

  Eine XML-Updateanweisung muss nicht generiert werden.

```
Schema: delete $target

Instanz: -
```

Umbenennen einer Attribut-Deklaration

- Optional / Zwingend / Verboten
  Wird der Name eines Attributes geändert, müssen alle Instanzen und zugehörigen Attribut-Deklarationen umbenannt werden. Zugehörige Attribut-Deklarationen sind jene, die das Attribut definieren oder die Vererbung einschränken.

  Die XML-Updateanweisung passt die Namen der Instanzen an.

```
Schema: rename $target as $newname;

Instanz: for $e in xsd:instance($target) return
         do rename $e as $newname
```

Änderung der Attributverwendung

Die Attributverwendung definiert die Existenz des Attributes als optional, zwingend oder verboten. Zusätzlich kann ein Default- oder Fixierter-Wert bestimmt werden. Die Abbildung 3.2 zeigte alle Zustandsübergänge, die **niemals** ein Anpassung der XML-Instanzen nach sich ziehen.

Bei allen in Abbildung 3.3 dargestellten Übergängen ist mindestens eine Überprüfung der Instanzen notwendig. Existieren keine Instanzen der optionalen Attribute, ist keine Änderung beim Übergang zu *verboten* notwendig. Existiert hingegen für jede Möglichkeit eine Instanz, ist der Übergang zu *notwendig* ohne Änderung möglich. Parallel ist die Änderung von *optional* mit festem Wert und *notwendig* mit festem Wert. Der Übergang von *verboten* zu

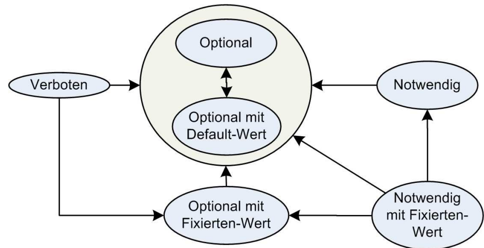


Abbildung 3.2: Zustandsänderungen, die keinen Einfluss auf die Instanzen haben

notwendig ist nicht möglich, da das im Basistypen definierten Attribut nur optional sein darf.

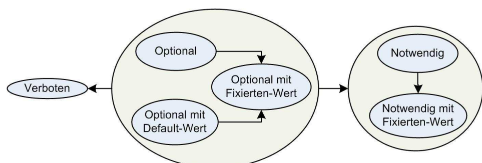


Abbildung 3.3: Zustandsänderungen, die Einfluss auf die Instanzen haben

- Zwingend
  Wird ein optionales Attribut zu einem zwingenden Attribut, müssen möglicherweise fehlende Instanzen eingefügt werden. Der Inhalt der neuen Attribute wird mit der Schemaanweisung angegeben.

  Die XML-Updateanweisung fügt fehlende Attribute ein.

  ```
  Schema: set use='required' {$value} of $target

  Instanz: for $e in xsd:instance($target) return
           if not($e) then
           do insert attribute $name{$value} into $e/..
  ```

- Notwendig mit Fixiertem Vorgabewert
  Wird zusätzlich der Wertebereich auf einen Wert eingeschränkt, wird der Inhalt aller existierenden Instanzen angepasst und fehlende Instanzen eingefügt.

  Die XML-Updateanweisung ersetzt alle Attribute mit dem fixierten Vorgabewert und fügt fehlende Attribute ein.

```
Schema: set use='required' fixed=$fixedvalue of $target

Instanz: for $e in xsd:instance($target) return
         if ($e) then do replace value of $e with $fixedvalue
         else do insert attribute $name{$value} into $e/..
```

- Verboten
  Wird die Attributverwendung für die Typdefinition verboten, werden alle existierenden Instanzen entfernt.

```
Schema: set use='prohibited' of $target

Instanz: for $e in xsd:instance($target) return
         do delete $e
```

- Optional mit Fixiertem Vorgabewert
  Wird der Wertebereich eines optionalen Attributes auf einen Wert eingeschränkt, wird der Inhalt aller existierenden Instanzen auf dem fixierten Vorgabewert gesetzt.

```
Schema: set use='optional' fixed=$fixedvalue of $target

Instanz: for $e in xsd:instance($target) return
         do replace value of $e with $fixedvalue
```

Änderung der Wertebereichsbeschränkung

- Fixierter-Wert
  Wird der Fixierte-Wert geändert, wird der Inhalt aller existierenden Instanzen auf den neuen Wert gesetzt.

```
Schema: set fixed=$fixedvalue of $target;

Instanz: for $e in xsd:instance($target) return
         do replace value of $e with $fixedvalue
```

- Default-Wert
  Der Default-Wert kann beliebig geändert werden. Eine Anpassung der Instanzen ist **niemals** erforderlich, da der Default-Wert nur für neue Attribute von Bedeutung ist.

Änderung der Typdefinition

- Einschränkung
  Ist die neue Typdefinition eine Einschränkung einer bestehenden Typdefinition, ist **möglicherweise** eine Anpassung der Instanzen notwendig.

  Die XML-Updateanweisung überprüft, ob der Inhalt im Wertebereich der neuen Typdefinition liegt. Falls nein, wird der Inhalt mit dem in der Schemaanweisung angegebenen Wert ersetzt.

  ```
  Schema: set type=$newtype {$value} of $target;

  Instanz: for $e in xsd:instance($target) return
           if not(upd:revalidate($e)) then
           do replace value of $e with $value
  ```

- Erweiterung
  Ist die neue Typdefinition eine Erweiterung in Form einer Liste oder Vereinigung ist **niemals** eine Anpassung nötig, da der alte Wertebereich Teilmenge des neuen ist.

Hinzufügen einer Attribut-Wildcard

Eine Typdefinition darf maximal eine Attribut-Wildcard besitzen. Die Existenz der durch Attribut-Wildcards definierten Attribute ist immer optional. Daher ist eine Instanzanpassung nicht notwendig.

```
Schema: insert anyAttribute namespace='##any'
        into /book/typedefinition();

Instanz: -
```

Entfernen einer Attribut-Wildcard

Das Entfernen einer Attribut-Wildcard ist eine Einschränkung der erlaubten Attributmenge. Da nicht bekannt ist, welche Attribute in der Instanz vorhanden sind, werden alle Attribute überprüft und nicht durch Attribut-Deklarationen erlaubte Attribute entfernt.

```
Schema: delete /book/typedefinition()/wildcard();

Instanz: for $e in xsd:instance($target)/@* return
         if not($attr1) and not($attr2) and not(...) then
         do delete $e
```

Ändern der Attribut-Wildcard

Mögliche Änderungen sind Erweitern oder Einschränken der erlaubten Namensräume und das Ändern der Validierungsvorschrift.

Wird die Menge der erlaubten Namensräume erweitert, ist keine Anpassung der Instanzen notwendig, da die Informationskapazität erweitert wird. Bei einer Verringerung der erlaubten Namensräume müssen alle Attribute des Elementes mit dem entfernten Namensraum überprüft werden. Werden sie nicht durch andere Attribut-Deklarationen erlaubt, müssen sie entfernt werden.

Die Validierungsvorschrift ist entweder deaktiviert (skip), locker (lax) oder strikt (strict). Wird die Validierung deaktiviert, wird der Inhalt der fremden Attribute nicht mehr geprüft. Die Instanzen bleiben daher in jedem Fall gültig. Wird die Validierungvorschrift von strikt auf locker gesetzt, ist ebenfalls keine Evolution nötig. Wenn die Validierungvorschrift erhöht wird, ist der Inhalt möglich nicht gegenüber der Deklaration gültig oder es wurde keine Deklaration gefunden. Die Schemaevolution wird in diesem Fall abgebrochen.

3.3.3 Ändern der Elementgruppe

Jedes Element besitzt eine Menge aus Attributen, welche keine Ordnung besitzt und deren Inhalt keine Duplikate erlaubt. Elemente, Wildcards und selbst Elementgruppen werden in eine Art Container definiert, welcher als Elementgruppe bezeichnet wird. Dabei wird zwischen den Typen Sequenz (sequence), Auswahl (choice) und Menge (set) unterschieden. Eine Sequenz besitzt eine Ordnung und das Minimale und Maximale Vorkommen der Elemente ist beliebig wählbar. Bei einer Auswahl wird eines der Element gewählt, wobei das Minimale und Maximale Vorkommen des Elementes beliebig ist. Eine Menge ist äquivalent zu der Menge von Attributen. Es wird keine Ordnung für die Elemente vorgegeben und Duplikate sind nicht erlaubt.

Dieser Abschnitt umfasst alle Änderungen, die auf eine Elementgruppe angewendet werden können. Dazu gehört das Umformen einer Elementgruppe in einen anderen Typ. Das Ändern des Minimalen und Maximalen Vorkommens. Sowie das Hinzufügen und Entfernen des Inhalts (Elementdeklarationen, Elementgruppen und Wildcards) einer Elementgruppe.

Ändern des Typs

Bei der Änderung des Typs von Sequenz zu Auswahl und umgekehrt entsteht immer ein gültiges Schema. Eine Änderung von einer Sequenz oder Alterna-

tive zu einer Menge ist nur möglich, wenn sie die einzige Elementgruppe der Typdefinition ist. Nach [TBMM04] darf eine Menge weder Teil einer Sequenz oder Alternative sein, noch eine enthalten. Eine Menge kann immer zu einer Sequenz oder Auswahl umgeformt werden.

Abbildung 3.4

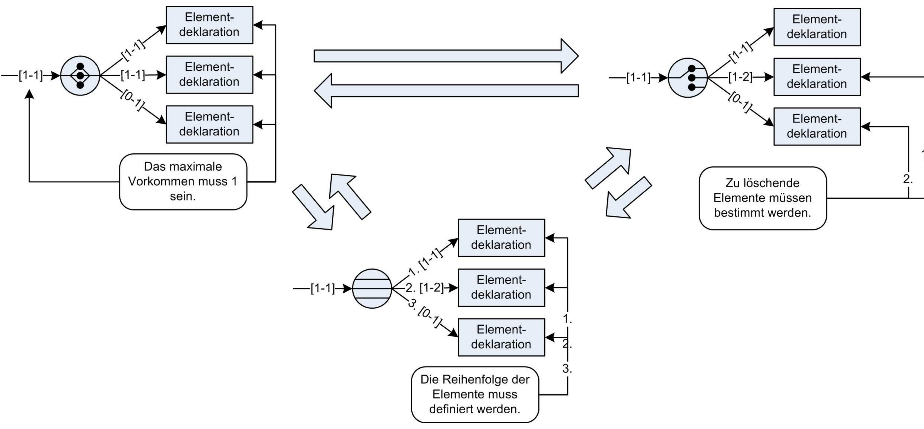


Abbildung 3.4: Übergänge der verschiedenen Elementgruppen

- Sequenz (maxOccurs=1) zu Menge
  Um eine Sequenz in eine Menge umzuwandeln, müssen bestimmten Bedingungen erfüllt sein. Das maximale Vorkommen der Elementgruppe in der Typdefinition muss eins sein. Da in der Menge jedes Element maximal einmal vorkommen darf, muss dies bereits für die Bestandteile der Sequenz gelten.

  Falls diese Bedingungen erfüllt sind, kann eine Sequenz in eine Menge umgewandelt werden. Ein Anpassung der Instanzen ist **niemals** notwendig. Die Information über die Anordnung der Elemente geht verloren.

- Sequenz / Menge zu Auswahl
  Bei der Umwandlung einer Sequenz oder Menge zu einer Auswahl reduziert sich die Anzahl der in der Instanz erlaubten Elemente auf eins. Sind in der Sequenz oder Menge mindesten zwei Element als zwingend definiert ist **immer** eine Anpassung der Instanzen notwendig.

  Mit der Schemaevolutions-Anweisung wird eine Liste von Elementen übergeben. Entspricht eine Instanz nicht dem Schema, wird ein Element nach dem anderen der Reihe nach entfernt, bis die Instanz gültig ist. Wird kein gültiger Status erreicht, wird die gesamte Schemaevolution rückgängig gemacht.

```
set category='choice' {$first $second ...} of $target;

for $e in xsd:instance($target) return
if not(upd:revalidate($e)) then
{
  do delete $target/$first
  if not(upd:revalidate($e)) then
  {
    do delete $target/$second
    if not(upd:revalidate($e)) then
    ...
    xsd:rollback
  }
}
```

- Menge zu Sequenz
  Eine Menge kann immer zu einer Sequenz umgewandelt werden. Mit der Schemaevolution muss jedoch die Ordnung der Elemente definiert werden.

  Die XML-Updateanweisung sortiert die Elemente nach der mit der Schemanweisung bestimmten Reihenfolge. Wird keine Reihenfolge angegeben, könnte sie auch per Alphabet oder Zufall bestimmt werden.

```
set category='sequence' {$first $second} of $target;

for $e in xsd:instance($target) return
{
  for $f in xsd:instance($second) return
  do move $f as first $e
  for $f in xsd:instance($first) return
  do move $f as first $e
}
```

- Auswahl zu Menge
  Ist mehr als ein Element in der Auswahl definiert, muss für jedes Element, das nicht in dem Instanzdokument vorkommt, eine Instanz generiert werden. Dies setzt voraus, dass alle fehlenden Elemente entweder optional oder nullbar sind bzw. einen Vorgabe-Wert besitzen. Kann der Inhalt der Menge nicht automatisch generiert werden, wird die Schemaevolution zurückgesetzt.

```
set category='all' of $target;

for $e in $target/child() return
{
  if (xsd:nillable($e)) then
  do insert <xsd:name($e) xs:nill='true'/>
  else if not(xsd:default($e)) and
          not(xsd:optional($e)) then xsd:rollback
}
```

- Auswahl zu Sequenz
  Die Umwandlung einer Auswahl zu einer Sequenz verhält sich ähnlich wie die einer Auswahl zu einer Menge. Zusätzlich muss nur die Reihenfolge der Elemente bestimmt werden.

```
set category='sequence' {$first $second} of $target;

for $e in xsd:instance($target) return
{
  for $f in xsd:instance($second) return
  do move $f as first $e
  for $f in xsd:instance($first) return
  do move $f as first $e
}
```

Minimales Vorkommen

Verringern Das Minimale Vorkommen einer Elementgruppe kann ohne Instanzanpassung verringert werden.

Erhöhen Wird das Minimale Vorkommen erhöht, müssen die Instanzen überprüft werden, ob genügend Instanzen existieren. Ist die Anzahl der benötigten Instanzen geringer, wird der Inhalt generiert. Falls die Generierung nicht automatisch machbar ist, wird die Schemaevolution abgebrochen.

Maximales Vorkommen

Verringern Wird das Maximale Vorkommen verringert, muss geprüft werden, ob nicht zu viele Instanzen der Elementgruppe existieren. Falls dies der Fall ist, wird die Schemaevolution abgebrochen.

Erhöhen Das Maximale Vorkommen einer Elementgruppe kann ohne Instanzanpassung erhöht werden.

Hinzufügen eines neuen Elementes

- Auswahl
  Wird ein Element in eine nicht leere Auswahl eingefügt, ist **keine** Anpassung der Instanzen erforderlich. Handelt es sich um eine leere Auswahl und sind Element und Auswahl zwingend, **muss** die Instanz mit dem neuen Element erweitert werden.

- Sequenz / Menge
  Die Positionsangabe *after* und *before* kann nur für Elemente verwendet werden, die Teil einer Sequenz sind.

```
Schema: insert element $name {$value}
          (as first into | as last into | into) $target

Instanz: for $e in xsd:instance($target) return
         do insert <$name>$value</$name>
          (as first into | as last into | into) $e

Schema: insert element $name {$value}
               (after | before) $target

Instanz: for $e in xsd:instance($target) return
         do insert <$name>$value</$name> (after | before) $e
```

Entfernen eines Elementes

- Sequenz / Auswahl / Menge
  Wird ein Element gelöscht, **müssen** alle Instanzen des Elementes entfernt werden. Eine Ausnahme ist, wenn ein Wildcard die Existenz des Elementes zusätzlich erlaubt. In diesem Fall werden die Instanzen nicht verändert.

  Ist das Element zwingend und Teil einer Sequenz oder Menge, sind **immer** Instanzen des Elementes vorhanden. Ist das Element optional oder Teil einer Auswahl, sind **möglicherweise** Instanzen vorhanden.

  Entsteht durch das Löschen eine leere Elementgruppe, wird diese auch entfernt.

Die XML-Updateanweisung entfernt alle möglichen Vorkommen des Elementes.

```
Schema: delete $target

Instanz: for $e in xsd:instance($target) return
         do delete $e
```

Umbenennen eines Elementes

- Optional / Zwingend
  Wird der Name eines Elements geändert, müssen alle Instanzen umbenannt werden.

  Die XML-Updateanweisung passt die Namen der Instanzen an.

```
Schema: rename $target as $newname;

Instanz: for $e in xsd:instance($target) return
         do rename $e as $newname
```

Hinzufügen einer Elementgruppe

Das Hinzufügen einer leeren Elementgruppe hat keine Auswirkungen auf die Gültigkeit der Instanzen.

Entfernen einer Elementgruppe

Wird eine Elementgruppe entfernt, wird auch der gesamte Inhalt dieser entfernt. Die XML-Updateanweisung entfernt alle mit der Elementgruppe definierten Instanzen.

```
Schema: delete $target;

Instanz: for $e in xsd:instance($target/*) return
         do delete $e
```

Hinzufügen einer Wildcard

Ein Wildcard definiert den Inhalt fremder Elemente. Ist das Minimale Vorkommen der Wildcard null ist keine Anpassung der Instanzen notwendig. Wenn das Minimale Vorkommen größer oder gleich eins ist, wird das mit der Updateanweisung angegebene Element eingefügt. Falls kein Element angegebenen wurde, wird die Schemaevolution abgebrochen.

```
Schema: insert any minOccurs=1 { $element } into $target;

Instanz: for $e in xsd:instance($target) return
         do insert $element into $e
```

Entfernen einer Wildcard

Wird eine Wildcard entfernt, werden alle nicht durch das Schema erlaubten Elemente aus der Instanz entfernt.

```
Schema: delete $target

Instanz: for $e in xsd:instance($target) return
         do delete $e
```

3.3.4 Ändern des Inhalts der Elemente

Ändern der zugewiesenen Typdefinition

Beim Zuweisen einer Typdefinition zu einer Element-Deklaration wird zwischen zwei Fällen unterschieden. Ist die neue Typdefinition eine Erweiterung der Vorherigen, vergrößert sich der Wertebereich und der alte Wertebereich ist eine Teilmenge des Neuen. Die Instanzen brauchen in diesem Fall nicht betrachtet werden. Falls die neue Typdefinition eine Einschränkung der Vorherigen ist, werden alle Instanzen überprüft, ob sie in dem neuen eingeschränkten Wertebereich liegen. Falls dies bei mindestens einer Instanz fehlschlägt wird die Schemaevolution abgebrochen und der Benutzer wird aufgefordert die Werte, die nicht in dem neuen Wertebereich liegen, zu entfernen.

Ändern des Minimalen und Maximalen Vorkommens

Wird das Minimale Vorkommen verringert oder das Maximale Vorkommen erhöht ist **keine** Anpassung der Instanzen erforderlich.

Wird das Maximale Vorkommen verringert, werden die Instanzen überprüft, ob nicht zu viele Elemente vorkommen. Ist dies der Fall wird die Schemaevolution abgebrochen.

Wird das Minimale Vorkommen erhöht, muss geprüft werden ob genügend Elemente in der Instanz existieren. Falls nein wird geprüft, ob anhand des Schemas der Inhalt automatisch generiert werden kann. Kann der Inhalt nicht generiert werden, wird die Schemaevolution abgebrochen.

Ändern der Nullwertfähigkeit

Der Nullwert[1] ist ein bestimmter Wert, der zu keinem Wertebereich gehören kann. XML-Schema bietet die Möglichkeit die Verwendung von Nullwerten für Elemente in der Instanz zu bestimmen. Er wird verwendet um ausdrücken zu können, dass der Wert unbekannt oder nicht anwendbar ist. Besitzt ein Element den Wert Null hat es keinen Inhalt, weder Zeichen noch zwingende Kindelemente.

Zustand Null wird erlaubt Wird durch eine Schemaänderung der Zustand Null erlaubt, ist keine Anpassung der Instanzen notwendig. Die Änderung stellt eine Kapazitätserweiterung, da ein weiterer Zustand erlaubt wird.

Zustand Null wird verboten Wird der Zustand Null verboten, handelt es sich um eine Kapazitätsverringerung. Alle existierenden Elemente der Instanz, welche den Zustand Null besitzen, müssen umgewandelt werden.

Folgende Möglichkeiten der Evolution gibt es: Hat ein Element den Zustand Null und ist die Element-Deklaration optional wird das Element aus der Instanz entfernt. Hierbei kann es aber zu einem Informationsverlust kommen. Besitzt zum Beispiel ein Buch 2 Autoren und war der Name des ersten Autors bei der Eingabe unbekannt, wird der 2. Autor durch das löschen des 1. Autors mit dem Nullwert zum 1. Autor.

Eine weitere Möglichkeit ist die Generierung des Inhaltes mit Hilfe des Standartwertes, falls einer für die Element-Deklaration und deren zwingenden Kindelementen definiert wurde. War das Gehalt einer Person z.B. vorher unbekannt, wird dadurch ein vorgegebner Standardwert eingefügt. Falls Nullwerte existieren, die keinen Vorgabewert besitzen, kann die Schemaänderung immer noch abgebrochen werden.

Die Implementierung dieser Arbeit führt die Schemaänderung nur durch, wenn keine Instanzen mit Nullwerten existieren.

```
Schema: set nillable='true' of $target
Instanz: -
```

3.3.5 Ändern des einfachen Datentyps

Ein einfacher Datentyp ist entweder eine Einfache Typdefinition oder eine Komplexe Typdefinition mit einfachem Inhalt. Ein einfacher Datentyp kann durch Einschränkung, Listenbildung oder Vereinigung gebildet werden.

[1] nil

Ändern des Typs (Atomar, Liste, Vereinigung)

Wird ein atomarer Datentyp zu einer Liste oder Vereinigung umgewandelt, wird der Wertebereich erweitert. Eine Anpassung der Instanzen ist deshalb nicht notwendig.

Wird eine Liste oder eine Vereinigung zu einem atomaren Typen umgewandelt, verringert sich der Wertebereich. Ist der Ausgangstyp eine Liste, müssen alle Instanzen auf einen Wert gekürzt werden. Ist der Ausgangstyp eine Vereinigung, muss der Wertebereich aller Instanzen überprüft werden. Liegt ein Wert einer Instanz außerhalb des neuen Wertebereich, wird in dieser Implementierung die Schemaevolution abgebrochen.

Die Umwandlung einer Liste zu einer Vereinigung ist gleich der Kombination der Umwandlung der Liste zu einem atomaren Typen und danach zu einer Vereinigung. Der Wertebereich wird eingeschränkt und erweitert, alle Instanzwerte sind zu überprüfen. Äquivalent ist die Umwandlung einer Vereinigung zu einer Liste.

Grundlegende Fassetten

Grundlegende Fassetten dienen der semantischen Charakterisierung eines Wertes aus dem Wertebereich. Es handelt sich dabei um abstrakte Eigenschaften, die entweder gar nicht oder nur implizit durch das definieren einschränkender Fassetten verändern lassen.

Einschränkende Fassetten

Mit einschränkenden Fassetten kann der Wertebereich einer Einfache Typdefinition beschränkt werden.

- **Length -** Bestimmt die Anzahl der Längeneinheiten.
- **MinLength -** Bestimmt die Anzahl der minimalen Längeneinheiten.
- **MaxLength -** Bestimmt die Anzahl der maximalen Längeneinheiten.
- **Enumeration -** Beschränkt den Wertebereich auf eines bestimmte Menge definierter Werte.
- **Whitespace -** Bestimmt den Ablauf der Normalisierung besonderer Zeichen, wie z.B. Zeilenvorschub und Leerzeichen.
- **Pattern -** Beschränkt den Wertebereich mit Hilfe eines regulären
- **MinInclusive -** Bestimmt die Untergrenze des Wertebereichs.

- **MaxInclusive -** Bestimmt die Obergrenze des Wertebereichs.
- **MinExclusive -** Bestimmt die Untergrenze einer exklusiven Schranke.
- **MaxExclusive -** Bestimmt die Obergrenze einer exklusiven Schranke.
- **TotalDigits -** Bestimmt die maximale Anzahl von Dezimalstellen.
- **FractionDigits -** Bestimmt die Anzahl der Nachkommastellen.

In dieser Situation gilt auch der allgemeine Fall. Da der Definition von Fassetten immer eine Einschränkung des Wertebereichs vorgenommen wird, müssen alle Instanzwerte überprüft werden. Falls Instanzen sich nicht im neuen Wertebereich befinden, muss der Benutzer dies anpassen.

Umbenennen der Typdefinition

Der Bezeichner einer Typdefinition hat keine Bedeutung für den Validierungsprozess.

Ändern des Ziel-Namensraumes

Der Namensraum in der sich eine Typdefinition befindet hat keine Auswirkung auf die Validierung. Wird der Namensraum geändert, müssen alle Referenzen auf die Typdefinition des Schemas angepasst werden.

Festlegen des Inhaltstyps

Der Inhaltstyp einer Komplexen Typdefinition ist entweder leer, einfach, komplex oder gemischt. Dabei ergibt sich der leere, einfache und komplexe Inhaltstyp implizit durch die Typdefinition. Der gemischte Inhalt wird mit der Eigenschaft *mixed* erlaubt oder verboten. Ist *mixed* auf *true* gesetzt, können Elemente neben Unterelementen auch Zeichenketten besitzen. Die Zeichenketten werden in diesem Fall aber nicht validiert, da dies nach [TBMM04] nicht möglich ist.

Somit reduziert sich die Betrachtung auf die Eigenschaft *mixed*. Wird gemischter Inhalt erlaubt, ist keine Instanzanpassung notwendig, da der Wertebereich erweitert wird. Wird der gemischte Inhalt verboten, werden alle Zeichenketten der Elemente aus den Instanzen entfernt.

Festlegen der Eigenschaft Abgeschlossenheit

Die weitere Ableitung einer Typdefinition kann mit der Eigenschaft Abgeschlossenheit eingeschränkt werden. Dazu gibt es abhängig vom Typ, ob komplex oder einfach, eine Menge von Auswahlmöglichkeiten. Eine Einfache Typdefinition kann vor einer Extension, Restriktion (restriction), Liste (list) und Vereinigung (union) abgeschlossen werden. Für eine Komplexe Typdefinitionen kann eine weitere Extension und Restriktion verboten werden.

Das Setzen der Einschränkung wird nur erlaubt, wenn keine existierenden Untertypen unzulässig werden. Dies kann allein durch die Betrachtung des Schemas entschieden werden. Eine Anpassung der Instanzen ist niemals notwendig.

3.3.6 Ändern der Typhierarchie

Einfache Typdefinition zu einer Vereinigung hinzufügen

Eine Einfache Typdefinition kann unabhängig davon, ob sie selbst vom Typ atomar, Liste oder Vereinigung ist, zu einer Vereinigung hinzugefügt werden. Eine Evolution der Instanzen ist niemals notwendig, da durch das Hinzufügen sich die Kapazität um die hinzugefügte Typdefinition erweitert.

Einfache Typdefinition aus einer Vereinigung entfernen

Wird eine Typdefinition aus einer Vereinigung entfernt, verkleinert sich der Wertebereich. Alle Instanzen müssen überprüft werden, ob sich in dem eingeschränkten Wertebereich liegen. Falls dies der Fall ist, wird die Schemaänderung durchgeführt.

Komplexe Typdefinition in Hierarchie als Erweiterung einfügen

Ein Untertyp einer komplexen Typdefinition kann entweder durch Erweiterung oder Restriktion gebildet werden. Bei der Erweiterung werden alle Attribute der Basistypdefinition vererbt. Der Untertyp erhält beide Elementgruppen der Typdefinitionen als eine Sequenz. Dabei wird als erstes die Elementgruppe des Basistyps und danach des Untertyps gefordert.

Bedingt durch die Schemaänderung müssen alle von der Basistypdefinition als zwingend definierten Attribut- und Elementwerte generiert werden.

Komplexe Typdefinition in Hierarchie als Einschränkung einfügen

XML-Schema besitzt einen Mechanismus zur Restriktion von komplexen Typdefinitionen. Der Untertyp funktioniert wie eine Schablone und definiert

alle Attribute und Elemente, die von der Basistypdefinition übernommen werden. Bei einer Restriktion können keine neuen Elemente oder Attribute eingefügt werden.

Die Schemaevolution wird nur durchgeführt, wenn der Untertyp eine Teilmenge des Obertyps ist. Eine Anpassung der Instanzen ist nicht notwendig, da sich der Inhalt des Untertypen nicht ändert.

Entfernen einer Komplexen Typdefinition aus der Hierarchie

Wird die Beziehung zwischen zwei Komplexen Typdefinitionen entfernt, wird zwischen zwei Fällen unterschieden. Ist die Beziehung eine Restriktion, ist kein Evolutionsprozess notwendig. Die durch die beiden Typen definierten Inhalte bleiben gleich.

Ist die Beziehung eine Erweiterung, müssen alle durch den Basistyp definierten Attribute und Elemente aus der Instanz des Untertyps entfernt werden.

3.3.7 Ändern von Beziehungen

Ein Primärschlüssel dient der eindeutigen Identifizierung eines Elementes aus einer Menge von Elementen. Der Schlüssel wird in einer Element-Deklaration definiert. Die für die Identifizierung bestimmten Felder können Elemente und Attribute sein.

Hinzufügen eines Primärschlüssels (key)

Wird ein Primärschlüssel zu einer Element-Deklaration hinzugefügt, müssen die durch die Schlüsseldefinition bestimmten Integritätsbedingungen gelten. Dabei dürfen keine zwei Elemente existieren, die den gleichen Schlüssel besitzen. Duplikate werden aus der Instanz entfernt, wenn der gesamte Inhalt der beiden Elemente übereinstimmt. Besitzen zwei Elemente den gleichen Schlüssel, jedoch unterschiedlichen Inhalts, sollte der Evolutionsschritt abgebrochen werden.

Entfernen eines Primärschlüssels

Wird ein Primärschlüssel aus dem Schema entfernt, dürfen keine Fremdschlüssel auf diesen verweisen. Die Fremdschlüssel müssen separat durch eine Updateanweisung entfernt werden.

Hinzufügen eines optionalen Primärschlüssels (unique)

Ein optionaler Schlüssel setzt nicht die Existenz aller Schlüssel voraus. Die Verarbeitung eines optionalen Schlüssels ist gleich eines zwingenden Schlüssels, mit der Ausnahme, dass nur Elemente beachtet werden, wo alle Felder existieren.

Entfernen eines optionalen Primärschlüssels

Das Entfernen eines optionalen Schlüssels verhält sich wie das Entfernen eines zwingenden Schlüssels.

Hinzufügen eines Fremdschlüssels (keyref)

Vor dem Einfügen eines Fremdschlüssels müssen die Integritätsbedingungen bereits stimmen. Dazu muss auf einen gültigen Primärschlüssel referenziert werden und alle Instanzen müssen überprüft werden. Ist eine Bedingung nicht korrekt, wird die Schemaevolution abgebrochen.

Entfernen eines Fremdschlüssels

Ein Fremdschlüssel kann immer ohne eine Evolution entfernt werden.

3.3.8 Ändern von Identitätsdefinitionen

Der ID/IDREF-Mechanismuss unterscheidet sich zum neuen Schlüsselmechanismus in zwei Punkten. Ein Schlüssel ist auf ein Feld begrenzt und der Gültigkeitsbereich entspricht immer dem gesamten XML-Dokument.

Hinzufügen einer Identität (ID)

Wird ein neues optionales Element oder Attribut als Identität eingefügt, ist keine Anpassung der Instanzen notwendig. Ist das Element oder Attribut zwingend, könnte der Evolutionsprozess mit einem Generator neue eindeutige Identität generieren und die Instanzen erweitern.

Eine weitere Möglichkeit ist das Ändern des Typs eines bestehendes Attributes oder Elementes zu dem Typ Identität. Hierbei müssen alle Instanzen auf Eindeutigkeit überprüft werden. Ist der Inhalt der Instanzen nicht disjunkt kann die Schemaänderung nicht durchgeführt werden.

```
Schema: set type='xs:ID' /books/book/@publisherID

Instanz: -
```

Entfernen der Identität

Wird eine Identität aus dem Schema entfernt, dürfen keinen Identitätsreferenz auf die Identität verweisen. Daher müssen vor dem Löschen der Identität alle Referenzen entfernt werden. Eine Instanzanpassung ist in diesem Fall nicht notwendig.

```
Schema: delete /publisher/@ID

Instanz: -
```

Hinzufügen einer Identitätsreferenz (IDREF)

Wird ein neues optionales Element oder Attribut als Identitätsreferenz eingefügt, ist keine Anpassung der Instanzen notwendig. Ein neues zwingendes Element oder Attribut als Identitätsreferenz kann nicht einfügt werden, da die Referenzen nicht automatisch bestimmt werden können.

Eine weitere Möglichkeit ist das Ändern des Typs eines bestehenden Attributes oder Elementes zu dem Typ Identitätsreferenz. Beim setzen werden alle Instanzen überprüft, ob sie auf existierende Identitäten verweisen. Sind fehlerhafte Verweise vorhanden, wird die Schemaänderung nicht durchgeführt werden.

```
Schema: set type='xs:IDREF' /publisher/@ID

Instanz: -
```

Entfernen einer Identitätsreferenz

Das Entfernen einer Identitätsreferenz ist immer ohne eine weitere Beachtung des Schemas oder der Instanzen möglich.

```
Schema: delete /books/book/@publisherID

Instanz: -
```

Kapitel 4

Sprachvorschlag

Dieses Kapitel enthält einen Sprachvorschlag, der speziell für die Evolution von XML-Schemata entworfen wurde. Im ersten Abschnitt (4.1) werden die Grundkonzepte und Anforderungen der Sprache dargestellt. Die im dritten Kapitel klassifizierten Schemaänderungen sollen dabei alle unterstützt werden. Die Abschnitte 4.2 und 4.3 unterteilen die Sprachdefinition in Navigations- und Operationenteil. XML-Schema besitzt eine reichhaltige Auswahl an Komponenten. Der Abschnitt 4.4 beschreibt die Eigenschaften und Möglichkeiten der Änderung für jede Komponente. Der Abschnitt 4.5 beschreibt die Konstruktoren, die nötig sind, um neue Knoten zu erzeugen. Im letzten Abschnitt (4.6) werden die Funktionen definiert, um Informationen der Schemakomponenten abfragen zu können.

4.1 Einführung

Die XML-Schemaempfehlung des W3C definiert das Post-Schema-Validation-Infoset (kurz. PSVI). Das PSVI verwendet Paradigmen, die aus Objektorientierten Programmiersprachen bekannt sind. Die Schemakomponenten werden als Objekte dargestellt. Je nach Objekttyp besitzen sie gewisse Eigenschaften und Verknüpfungen zu anderen Objekten. Das PSVI beschreibt die Bedeutung der bereitgestellten Informationen, ist jedoch keine konkrete Schnittstelle. Eine Implementierung in Form einer Programmierschnittstelle beschreibt die XML Schema API[Lit04]. Sie definiert eine Schnittstelle für Objektorientierte Programmiersprachen, wie z.B. Java oder C++. Implementiert wurde diese API in den Open-Source Parser Xerces für Java und C++. Der in diesem Kapitel konzipierte Sprachevorschlag verwendet das PSVI als Darstellungsform des Schemas.

Ein XML-Schema hat die Form eines Graphen. Die Navigationssprache

muss es ermöglichen, beliebige Teile des Graphen zu selektieren. Sie muss außerdem deskriptiv und mengenorientiert sein. XPath ist eine Empfehlung vom W3C, die für die Navigation über XML-Instanzen entwickelt wurde. In diesem Kapitel wird ein Vorschlag erstellt, der XPath um das Infoset von XML-Schema zu erweitern.

Der Operationenteil beschreibt die Änderung der selektierten Knoten. Bei der Entwicklung flossen die aktuellen Entwicklungen der Updateerweiterung von XQuery[CFR06] ein.

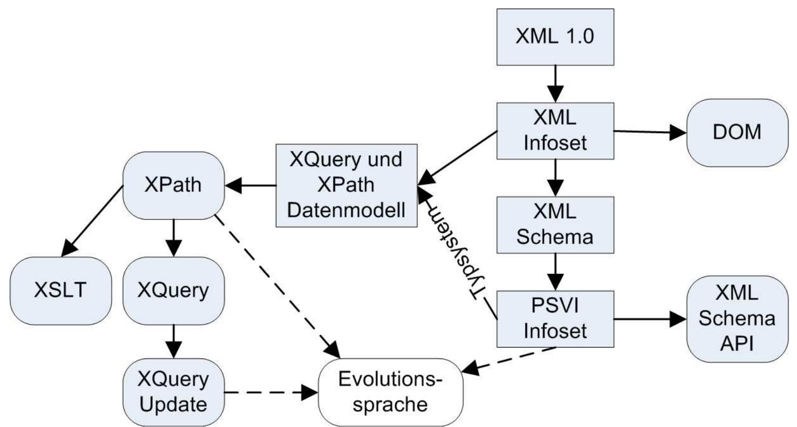


Abbildung 4.1: Überblick der XML-Konzepte

Die Abbildung 4.1 bietet einen Überblick über die verwendeten Konzepte. Dank der nahtlosen Integration in existierende W3C Empfehlungen kann die Schemaevolutionssprache als Erweiterung von XQuery oder als alleinstehende Sprache umgesetzt werden.

4.2 Navigation

Dieser Abschnitt definiert die Syntax und Semantik für die Adressierung von Teilen eines Schemas. Die Navigationssprache ist eine Erweiterung von XPath[BBC+05], welche es ermöglicht, über die Komponenten eines XML-Schemas zu navigieren.

4.2.1 Pfadausdrücke

```
[68] PathExpr         ::= ("/" RelativePathExpr?)
                        | ("//" RelativePathExpr)
                        | RelativePathExpr
[69] RelativePathExpr ::= StepExpr (("/" | "//") StepExpr)*
```

Das Grundprinzip von XPath bleibt bei diesem Vorschlag bestehen. Ein *Pfadausdruck* wird verwendet, um gewünschte Knoten aus dem Schemadokument auszuwählen. Genauer besteht ein *Pfadausdruck* aus einer Serie von *Schritten*, die durch die Symbole '/' oder '//' getrennt werden. Beginnt der *Pfadausdruck* mit dem Symbol '/' handelt es sich um einen *absoluten Pfadausdruck*. Dabei wird immer von dem Wurzelknoten des Schemas ausgegangen. Alle sonstigen *Pfadausdrücke* werden relativ zu einem vorgegebenen Kontextknoten ausgeführt.

Bisher unterscheiden sich XPath und die Spracherweiterung nur in dem geänderten Wurzelknoten.

4.2.2 Schritte

```
[70] StepExpr      ::=  FilterExpr | AxisStep
[71] AxisStep      ::= (ReverseStep | ForwardStep) PredicateList
[72] ForwardStep   ::= (ForwardAxis NodeTest) | AbbrevForwardStep
[75] ReverseStep   ::= (ReverseAxis NodeTest) | AbbrevReverseStep
[82] PredicateList ::=  Predicate*
```

Jeder *Schritt* besteht aus einer *Achse*, einem *Knotentest* und einer Sequenz von *Prädikaten*. Eine *Achse* beschreibt die Beziehung zwischen den Knoten und den zu selektierenden Knoten. Mit Hilfe des *Knotentests* werden die Knoten der *Achse* anhand des Namens oder eines bestimmten Typs ausgewählt. Und als dritten und letzten Schritt wird die Knotenmenge mit Hilfe von *Prädikaten* eingeschränkt.

4.2.3 Achsen

```
[73] ForwardAxis ::= ("child" "::")
                   | ("descendant" "::")
                   | ("attribute" "::")
                   | ("self" "::")
                   | ("descendant-or-self" "::")
                   | ("following-sibling" "::")
                   | ("typedefinition" "::")
                   | ("constraint" "::")
[76] ReverseAxis ::= ("parent" "::")
                   | ("ancestor" "::")
                   | ("preceding-sibling" "::")
                   | ("ancestor-or-self" "::")
                   | ("basetype" "::")
```

```
| ("modelgroup" "::")
| ("reference" "::")
```

Achsen stellen Beziehungen zwischen den Knoten dar. Die Bezeichnung der aus XPath bekannten *Achsen* bleibt bestehen, jedoch ändert sich die Bedeutung dieser. Außerdem werden zusätzlich die *Achsen* 'constraint', 'basetype', 'modelgroup', 'typedefinition' und 'reference' hinzugefügt.

Während eine XML-Instanz einen Baum darstellt, ist die Struktur eines Schemas ein Graph. Die Navigation über eine XML-Instanz geschieht beinahe ausschließlich auf der Ebene der Elemente. Attribut- oder Kommentarknoten können nur Blätter des Baumes sein. Daher haben alle *Achsen*, mit Ausnahme der Vaterachse, ausschließlich von einem Element-Knoten aus eine Funktion.

Ein Schema besteht aus einer Vielzahl unterschiedlicher Komponenten. Die Navigation verläuft dabei über mehrere Ebenen. Z.B. wird von einem Element-Knoten zu der zugewiesenen Typdefinition navigiert und von dieser zum Basistypen der Typdefinition.

Die Bedeutung der *Achsen* ist abhängig von dem Typ des aktuellen Kontextknotens. Im Folgenden wird für jeden Knotentyp die Bedeutung der Achsen definiert.

Element-Knoten Ein Element-Knoten besteht aus einem *Partikel* und einer *Element-Deklaration*. Globale *Element-Deklarationen* besitzen kein *Partikel*.

Folgend die Achsen und ihre Bedeutung:

- **child (Kind)** Bezeichnet alle direkten Nachfahren der Elementgruppe der zugehörigen Typdefinition und alle Anmerkungen und Notations-Deklarationen des Kontextknotens.
- **descendant (Nachfahre)** Sind alle Kinder und die Kindeskinder.
- **attribute (Attribut)** Bezeichnet alle Attribut-Knoten der zum Kontextknoten zugewiesenen Typdefinition.
- **self (Selbst)** Ist der aktuelle Kontextknoten selbst, bestehend aus einem Partikel und der zugehörigen Element-Deklaration.
- **descendant-or-self (Nachfahre und selbst)** Alle Knoten aus *descendant* und der Kontextknoten selbst.
- **following-sibling (Folgende Geschwister)** Alle folgenden Geschwisterknoten des Kontextknotens. Da eine Ordnung nur in einer Sequenz vorgegeben ist, wird zwischen folgenden 3 Situationen unterschieden:

1. Ist der Kontextknoten Teil einer Sequenz, beinhaltet es alle in der Sequenz folgenden Knoten.
2. Ist der Kontextknoten Teil einer Alternative oder Menge, beinhaltet es alle in der Elementgruppe enthaltenen Knoten.
3. Ist der Kontextknoten Teil der Schemawurzel, beinhaltet es alle Konten der Schemawurzel.

- **constraint (Identitätsbeschränkung)** Alle Identitätsbeschränkungs-Definitionen des Kontextknotens.
- **parent (Vater)** Alle Element-Knoten, die den Kontextknoten als direkten Nachfahren haben.
- **ancestor (Vorfahre)** Alle Vorfahren des Kontextknotens.
- **preceding-sibling (Vorangegangene Geschwister)** Alle vorangegangenen Geschwisterknoten des Kontextknotens. Dabei wird zwischen folgenden 2 Situationen unterschieden:
  1. Ist der Kontextknoten Teil einer Sequenz, beinhaltet dies es in der Sequenz vorangegangenen Knoten.
  2. Ist der Kontextknoten Teil der Schemawurzel, einer Menge oder Alternative, ist die Ergebnismenge leer.
- **ancestor-or-self (Vorfahre und selbst)** Alle Knoten aus *ancestor* und dem Kontextknotens selbst.
- **modelgroup (Elementgruppe)** Die Elementgruppe, deren Bestandteil der Kontextknoten ist.
- **reference (Referenz)** Alle Knoten, die auf die Element-Deklaration des Kontextknotens verweisen. Dies ist eine Menge von Paaren aus *Partikeln* und dem Kontextknoten.

Die Abbildung 4.2 veranschaulicht die Navigationsachsen ausgehend von einem Element-Knoten.

Typdefinitions-Knoten Ein Typdefinitions-Knoten besteht aus einer *Typdefinition*. Der Inhalt der Typdefinition entscheidet, ob es sich nach [TBMM04] um eine einfache oder komplexe Typdefinition handelt.

Folgend die Bedeutung der Achsen, ausgehend von einem Typdefinitions-Knoten:

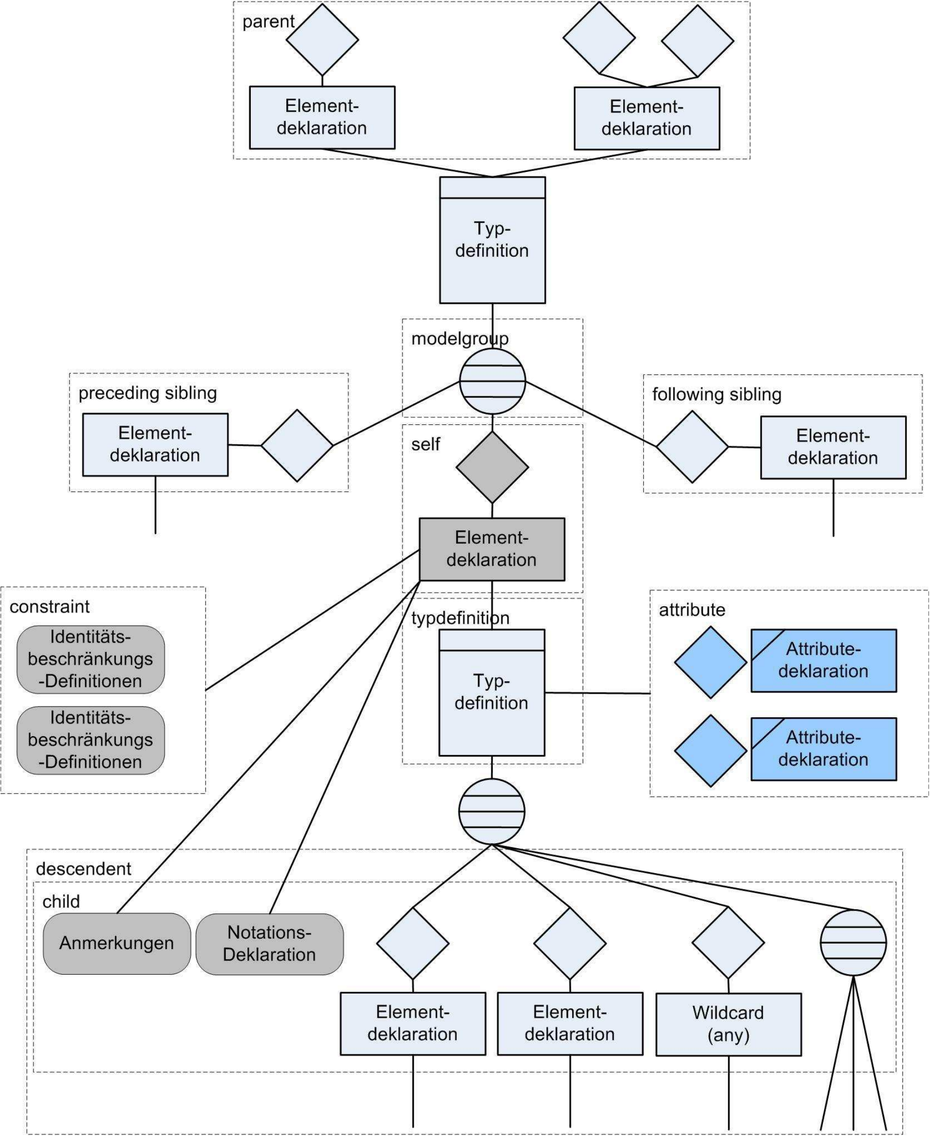


Abbildung 4.2: Bedeutung der Achsen ausgehend von einem Element-Knoten

- **child (Kind)** Alle direkten Nachfolger der Elementgruppe des Kontextknotens und alle Anmerkungen und Notations-Deklarationen des Kontextknotens.
- **descendant (Nachfahre)** Alle Kinder und Kindeskinder.
- **attribute (Attribut)** Alle dem Kontextknoten zugewiesenen Attribut-Knoten.
- **self (Selbst)** Der aktuelle Kontextknoten selbst, bestehend aus einer Typdefinition.
- **parent (Vater)** Alle Element-Knoten die auf den Kontextknoten als Typdefinition verweisen.
- **basetype (Basistyp)** Die Typdefinition, von der die Typdefinition abgeleitet wurde.
- **modelgroup (Elementgruppe)** Die Elementgruppe, die Bestandteil des Kontextknotens ist.

Elementgruppen-Knoten Ein Elementgruppen-Knoten besteht aus einer Elementgruppe, was entweder eine Sequenz, Alternative oder Menge ist.

Die Bedeutung der Achsen ist wie folgt definiert:

- **child (Kind)** Alle direkten Nachfolger des Kontextknotens
- **descendant (Nachfahre)** Alle Kinder und Kindeskinder.
- **self (Selbst)** Der aktuelle Kontextknoten selbst, bestehend aus einer Elementgruppe.
- **following-sibling (Folgende Geschwister)** Alle vorangegangen Geschwisterknoten des Kontextknotens. Dabei wird unterschieden zwischen:
    1. Ist der Vaterknoten (parent) eine Typdefinition, ist die Ergebnismenge leer.
    2. Ist der Vaterknoten eine Sequenz, beinhaltet die Ergebnismenge alle folgenden Knoten.
    3. Ist der Vaterknoten eine Alternative oder Menge, beinhaltet die Ergebnismenge alle der Elementgruppe enthaltenen Knoten.

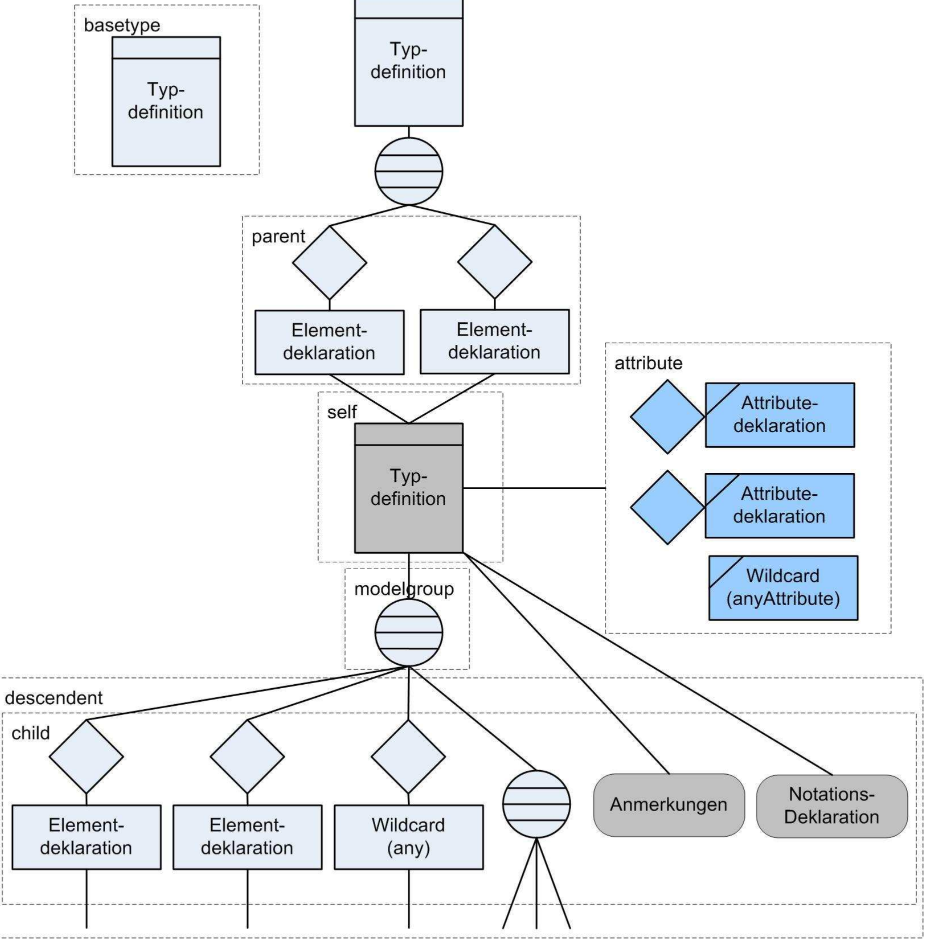


Abbildung 4.3: Bedeutung der Achsen ausgehend von einem Typdefinitions-Knoten

- **parent (Vater)** Entweder eine Elementgruppe oder Typdefinition, deren Bestandteil der Kontextknoten ist.

- **preceding-sibling (Vorangegangene Geschwister)** Alle vorangegangen Geschwisterknoten des Kontextknotens. Dabei wird unterschieden zwischen:

  1. Ist der Vaterknoten (parent) eine Typdefinition, ist die Ergebnismenge leer.
  2. Ist der Vaterknoten eine Sequenz, beinhaltet die Ergebnismenge alle vorangegangenen Knoten.
  3. Ist der Vaterknoten eine Alternative oder Menge, ist die Ergeb-

nismenge leer.

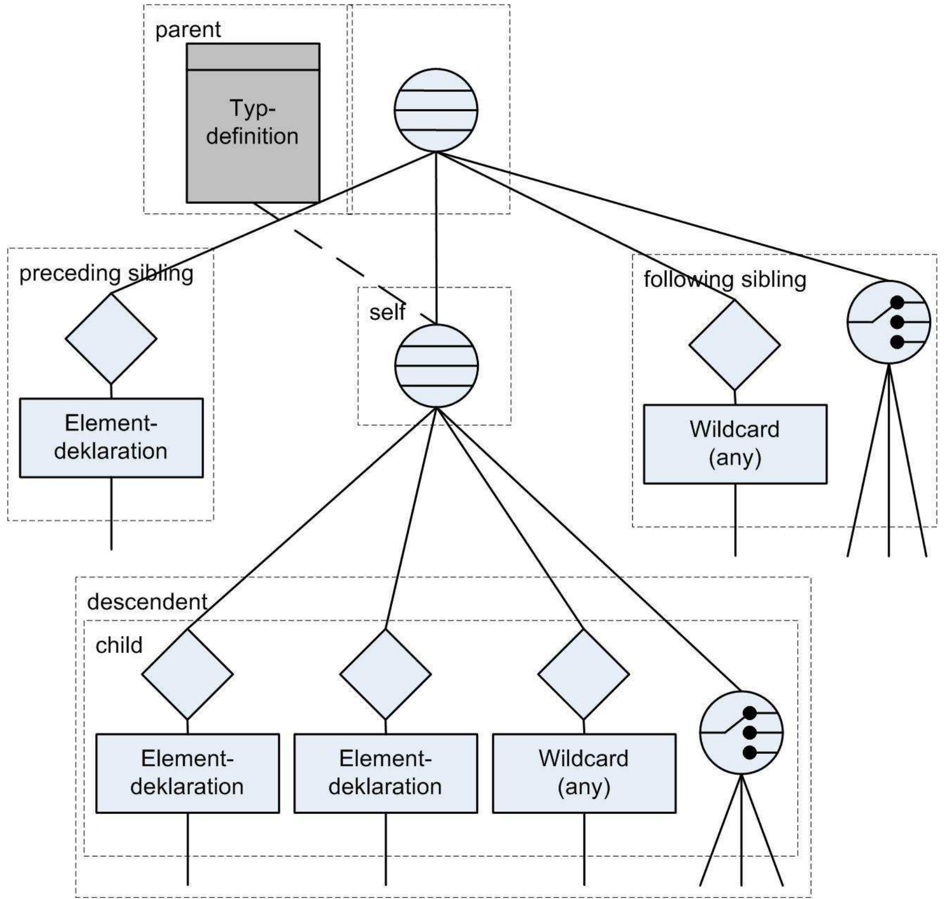


Abbildung 4.4: Bedeutung der Achsen ausgehend von einem Elementgruppen-Knoten

4.2.4 Knotentests

```
[78] NodeTest ::= KindTest | NameTest
[79] NameTest ::= QName | Wildcard
[80] Wildcard ::= "*" | (NCName ":" "*") | ("*" ":" NCName)
```

Im vorherigen Abschnitt wurde die Bedeutung der Achsen abhängig vom Kontextknoten dargestellt. Auf jeden Achsen-Ausdruck folgt ein *Knotentest*. Ein *Knotentest* definiert eine Bedingung, welche auf die Menge der Ergebnismenge der Achse angewendet wird. Stimmt die durch den *Knotentest* bestimmte Bedingung nicht, wird der Knoten aus der Ergebnismenge entfernt.

Ein *Knotentest* kann ein bestimmter Typ eines Knotens sein oder ein Name. Wenn der *Knotentest* eine Namensangabe hat, ist es entweder ein nach [BPSM+04] qualifizierter Name, der anhand des Namens und Namensraumes genau identifiziert oder eine Maske (engl. Wildcard). Mithilfe der Masken können entweder alle Knoten selektiert werden, alle aus einem bestimmten Namensraum oder mit bestimmten Namen unabhängig vom Namensraum.

Die *Knotentests*, welche die Ergebnismenge auf einen bestimmten Typ eingrenzen, sind wie folgt definiert:

```
[123] KindTest ::= SchemaDocumentTest
                 | SchemaElementDeclarationTest
                 | SchemaAttributeDeclarationTest
                 | SchemaParticleTest
                 | SchemaAttributeUseTest
                 | SchemaTypeDefinitionTest
                 | SchemaModelGroupTest
                 | SchemaIdConstraintTest
                 | SchemaWildcardTest
                 | SchemaAttributeWildcardTest
                 | SchemaElementGroupTest
                 | SchemaAttributeGroupTest
                 | SchemaNotationTest
                 | SchemaAnnotationTest
                 | AnyKindTest

[124] AnyKindTest                    ::= "node" "(" ")"
[141] SchemaDocumentTest             ::= "schema-node" "(" ")"
[142] SchemaElementDeclarationTest   ::= "elementdeclaration" "("
                                         (ElementDeclaration)? ")"
[143] SchemaAttributeDeclarationTest ::= "attributedeclaration" "("
                                         (AttributeDeclaration)? ")"
[144] SchemaTypeDefinitionTest       ::= "typedefinition" "("
                                          (TypeDefinition)? ")"
[145] SchemaElementGroupTest         ::= "elementgroup" "("
                                         (ElementGroup)? ")"
[146] SchemaAttributeGroupTest       ::= "attributegroup" "("
                                         (AttributeGroup)? ")"
[147] SchemaIdConstraintTest         ::= "idconstraint" "(" (IDConstraint)? ")"
[148] SchemaModelgroupTest           ::= "modelgroup" "(" ")"
[149] SchemaWildcardTest             ::= "wildcard" "(" ")"
[150] SchemaAttributeWildcardTest    ::= "attributewildcard" "(" ")"
[151] SchemaNotationTest             ::= "notation" "(" ")"
[152] SchemaAnnotationTest           ::= "annotation" "(" ")"
```

Es gibt für alle 13 möglichen Komponenten aus XML Schema einen *Knotentest*. Zusätzlich gibt es *Knotentests* zum Selektieren der Schemawurzel und aller Knoten.

Die *Knotentests* der Element-, Attribut-, Typdefinitions-, Identitäts-, Elementgruppen-Definitionen- und Attributgruppen-Definitions-Knoten besitzen optional einen Parameter, der die Kontenmenge mit der Angabe eines qualifizierten Namens weiter einschränkt.

Kurzform	Langform	Bemerkung
	child::	Standardachse
.	self::node()	aktueller Kontextknoten
..	parent::node()	Vaterknoten
@	attribute::	Attribut des Kontextknotens
/	fn:root(self::node())	Schemawurzel
//	descendant-or-self	Nachfolgeknoten

Tabelle 4.1: Abkürzungen des Pfadausdruckes

4.2.5 Prädikate

```
[83] Predicate ::= "[" Expr "]"
```

Die durch Achse und Knotentest bestimmte Ergebnismenge kann optional mit Hilfe von *Prädikaten* weiter spezifiziert werden. Der Parameter *Expr* ist ein boolescher Ausdruck und kann jeder beliebige Pfadausdruck sein. Ist der Ausdruck nicht leer oder hat den Wert *true* wird das Element der Ergebnismenge eingefügt.

Dazu ein paar Beispiele:

- Es werden alle *book* Elemente selektiert, die ein Unterelement *author* besitzen.

  ```
  //book[author]
  ```

- Es wird die zweite Element-Deklaration mit dem Namen *author* selektiert.

  ```
  /book/author[fn:position() = 2]
  ```

4.2.6 Abkürzungen

```
[74] AbbrevForwardStep ::= "@"? NodeTest
[77] AbbrevReverseStep ::= ".."
```

Abkürzungen erleichtern das Schreiben und Lesen von Pfadausdrücken. In den vorherigen Abschnitten wurden gewisse Abkürzungen definiert. Die Tabelle 4.2 gibt einen Überblick.

Ein paar Beispiele:

- Selektiert die Typdefinition der Element-Deklaration 'books'.

```
Kurzform: /book/books/typedefinition()
Langform: fn:root(self::node())/child::book/child::books/
          typedefinition::typedefinition()
```

- Die Ergebnismenge beinhaltet alle Attribut-Knoten, die Bestandteil der Typdefinition von der Element-Deklaration 'golf' sind.

```
Kurzform: //golf/@price
Langform: fn:root(self::node())/descendant-or-self::node()/
          child::golf/attribute::price
```

- Die Ergebnismenge besteht aus allen Element-Deklarationen, deren Typdefinition die globale Attribut-Deklaration 'isbn' beinhaltet.

```
Kurzform: /@isbn/..
Langform: fn:root(self::node())/attribute::isbn/
          parent::parent()
```

Abkürzung der Navigation

Neben dem Abkürzen von Ausdrücken wird noch eine weitere Art der Abkürzung eingeführt. Besitzt ein Knoten als einziger eine Referenz zu einem Knoten, dessen Knotentyp maximal einmal von dem Ausgangsknoten referenziert werden kann, kann hier die Navigation abgekürzt werden und alle Operationen des Zielknotens sind auf dem Ausgangsknoten anwendbar.

Folgende Abkürzungen werden eingeführt:

- **Element -> Typdefinition**
  Ein Element-Knoten besitzt immer eine Referenz zu einer Typdefinition. Wird die Typdefinition nur von einem Element-Knoten verwendet, können die Operationen der Typdefinition direkt auf den Element-Knoten anwendbar.

- **Typdefinition -> Elementgruppe**
  Eine Typdefinition besitzt genau eine Elementgruppe.

- **Elementgruppen-Definition - > Elementgruppe**
  Eine Elementgruppen-Definition besitzt genau eine Elementgruppe.

Ein paar Beispiele:

- Ein neues Element wird eingefügt.

```
Kurzform: insert element 'isbn' into /book/books
Langform: insert element 'isbn' into
                 /book/books/typedefinition()/modelgroup()
```

- Ein neues Attribut wird eingefügt.

```
Kurzform: insert attribute 'title' into /book/books
Langform: insert attribute 'title' into
                 /book/books/typedefinition()
```

4.3 Operationen

Bei der Entwicklung der Ausdrücke für die Anweisung der Schemaevolution, ist die Syntax der aktuellen Entwicklung von XQuery Update [CR05] als Vorbild mit eingeflossen.

```
[32] ExprSingle ::= FLWORExpr
                  | QuantifiedExpr
                  | TypeswitchExpr
                  | IfExpr
                  | OrExpr
                  | InsertExpr
                  | DeleteExpr
                  | SetExpr
                  | RenameExpr
                  | MoveExpr
```

Der Sprachvorschlag erweitert die Syntax von XQuery um die Ausdrücke 'Insert', 'Delete', 'Set', 'Rename' und 'Move'. Zu beachten ist, dass dies keine Erweiterung der sich noch in Entwicklung befindlichen Sprache, XQuery Update ist. In den nächsten Abschnitten wird die Syntax und Semantik dargestellt.

4.3.1 Insert

```
[153] InsertExpr ::= "insert" SourceExpr
                     (("as first" | "as last")? "into")
                     | "after" | "before") TargetExpr

[158] SourceExpr ::= ExprSingle
[159] TargetExpr ::= ExprSingle
```

Mit Hilfe der Insert-Anweisung wird das Schema an einem bestimmten Knoten um eine Komponente erweitert. Falls der Zielknoten eine Sequenz ist, kann die genaue Position der neuen Komponente noch mit Hilfe von 'as first into' und 'as last into' bestimmt werden. Ist der Zielknoten Bestandteil einer Sequenz wird die Position mit 'after' und 'before' angegeben. Der Ausdruck 'into' wird verwendet, um einen Knoten in eine Menge einzufügen. Wird 'into' auf eine Sequenz angewendet, wird der neue Knoten gleichbedeutend wie 'as last into' am Ende der Sequenz eingefügt.

Anfangsknoten	kann erweitert werden um
Schemawurzel	Element, Attribut, Typdefinition, Attributgruppen-, Elementgruppen-Definition, Notation, Anmerkung
Typdefinition	Elementgruppe, Attribut, Wildcard, Notation, Anmerkung
Elementgruppe	Element, Elementgruppen-Definition,
Element	Identitätsbeschränkungs-Definition, Anmerkung
Attribut	Anmerkung
Identitätsbeschränkungs-Definition	Anmerkung

Tabelle 4.2: Überblick der verschiedenen Knotentypen

Nicht jeder Knoten kann mit beliebigem Inhalt erweitert werden. Die Tabelle 4.2 gibt einen Überblick, welche Anfangsknoten mit welchen Knoten erweitert werden können. Eine detaillierte Beschreibung der Schemaknoten wird im nächsten Abschnitt (4.4) gegeben.

Beispiele

- Die Typdefinition der Element-Deklaration *book* wird um die Attribut-Deklaration *isbn* erweitert.

```
insert attribute 'isbn' into /books/book
```

- Eine neue Element-Deklaration *editor* wird in der Sequenz von *book* nach dem *author* eingefügt.

```
insert element 'editor' of type 'xs:string' after /books/
       book/author
```

4.3.2 Delete

```
[154] DeleteExpr ::=  "delete" TargetExpr
[159] TargetExpr ::=  ExprSingle
```

Mit Hilfe der Delete-Anweisung werden alle durch den Zielausdruck selektierten Knoten aus dem Schema entfernt.

Beispiele

- Das Attribut 'price' wird aus der vom Element 'books' referenzierten Typdefinition entfernt.

  ```
  delete fn:xsd("schema.xsd")/books/book/typedefinition()/
         @price
  oder in Kurzform
  delete /books/book/@price
  ```

- Alle Anmerkungen der Element-Deklarationen werden aus dem Schema entfernt.

  ```
  delete //annotation()
  ```

- Alle nicht verwendeten globalen Typdefinitionen werden entfernt.

  ```
  delete /typdefinition()[references()==0]
  ```

4.3.3 Set

```
[157] SetExpr      ::= "set" PropertyExpr "of" TargetExpr
[161] PropertyExpr ::=
        ("use"              "=" "optional"|"required"|"prohibited")
        | ("default"        "=" Literal)
        | ("fixed"          "=" Literal)
        | ("scope"          "=" "true"|"false")
        | ("type"           "=" QName)
        | ("minoccurs"      "=" NumericLiteral)
        | ("maxoccurs"      "=" NumericLiteral)
        | ("nillable"       "=" "true"|"false")
        | ("abstract"       "=" "true"|"false")
        | ("selector"       "=" PathExpr)
        | ("field"          "=" PathExpr)
        | ("reference"      "=" QName)
        | ("mixed"          "=" "true"|"false")
        // Fassetten einfacher Typdefinitionen
        | ("length"         "=" nonNegativeInteger)
        | ("minLength"      "=" nonNegativeInteger)
        | ("maxLength"      "=" nonNegativeInteger)
        | ("enumeration"    "=" Teil aus dem Wertebereich des Basistyps)
```

```
| ("whitespace"     "=" "preserve" | "replace" | "collapse")
| ("pattern"        "=" RegularExpression)
| ("minExclusive"   "=" Wert aus dem Wertebereich des Basistyps)
| ("maxExclusive"   "=" Wert aus dem Wertebereich des Basistyps)
| ("minInclusive"   "=" Wert aus dem Wertebereich des Basistyps)
| ("maxInclusive"   "=" Wert aus dem Wertebereich des Basistyps)
| ("totalDigits"    "=" positiveInteger)
| ("fractionDigits" "=" nonNegativeInteger)
```

Mit Hilfe der Set-Anweisung werden die Eigenschaften eines Knotens verändert. Die Menge der Eigenschaften hängt vom Typ des Knotens ab. Im nächsten Abschnitt (4.4) Knotentypen und deren Eigenschaften detailliert beschrieben.

Beispiele

- Verändert den Vorgabewert des Einkommens.

  ```
  set default='$3500' of /employee/@income
  ```

- Macht die Angabe des Attributes *religion* optional.

  ```
  set use='optional' of /employee/@religion
  ```

4.3.4 Rename

```
[155] RenameExpr  ::= "rename" TargetExpr "as" NewNameExpr
[160] NewNameExpr ::= QName
[XML] QName       ::= (Prefix ':')? LocalPart
```

Mit der Rename-Anweisung können Element-, Attribut-, Attributgruppen-Definitions-, Elementgruppen-Definitions- und Typdefinitions-Knoten umbenannt werden. Der neue Name besteht aus Namensraum und lokalen Bezeichner. Ist kein Namensraum angegeben, wird der Standardnamensraum des Schemas verwendet.

Beispiel

- Die Attribut-Deklaration *price* wird zu Preis umbenannt.

  ```
  rename /buecher/buch/@price as 'Preis'
  ```

4.3.5 Move

```
[156] MoveExpr ::= "move" SourceExpr
                    (("as first" | "as last")? "into")
                    | "after" | "before") TargetExpr
```

Mit der Move-Anweisung werden Knoten innerhalb des Schemas verschoben. Ziel- und Quellknoten müssen der Tablle 4.2 entsprechen.

Beispiel

- Macht den Publisher als Nachfolger des Editor.

  ```
  move /books/book/publisher after /books/book/editor
  ```

4.4 Eigenschaften der Schemaknoten

In diesem Abschnitt werden die Eigenschaften der einzelnen Knotentypen beschrieben.

4.4.1 Schemawurzel

Die Schemawurzel ist ein Container für die Schemakomponenten. Der Container besteht aus einer Menge von Typdefinitionen, Attribut-Deklarationen, Element-Deklarationen, Attributgruppen-Definitionen, Elementgruppen-Definitionen, Notations-Deklarationen und Anmerkungen.

Jede in der Schemawurzel definierte nicht abstrakte Element-Deklaration kann als Wurzelelement der Instanz verwendet werden.

Eigenschaften

- Menge von **Typdefinitionen**
  Eine Typdefinition wird mit Hilfe der *Insert*-Anweisung eingefügt und mit *Delete* wieder entfernt. Typdefinitionen der Schemawurzel werden als globale Typdefinition bezeichnet. Weiterhin besitzen sie einen Namen und Namensraum, damit andere Komponenten darauf verweisen können. Wird einer lokal definierten Typdefinition ein Name zugewiesen, wird sie in die Schemawurzel verschoben. Die Komponente, deren Bestandteil die Typdefinition vorher war, verweist dann auf die globale Typdefinition.

```
Beispiel: rename /books/book/typedefinition() as
                 'typeOfBook'
```

- Menge von **Attribut-Deklarationen**
  Neue Attribut-Deklarationen werden mit der *Insert*-Anweisung eingefügt und mit *Delete* wieder entfernt werden. Wird die Eigenschaft 'Scope' einer existierenden Attribut-Deklaration auf 'true' gesetzt, wird sie in die Schemawurzeln verschoben. Wird die Eigenschaft 'Scope' einer existierenden Attribut-Deklaration auf 'false' gesetzt, wird sie aus der Schemawurzel entfernt. Alle Komponenten, die diese Attribut-Deklaration referenziert haben, werden mit einer lokalen Kopie erweitert.

  ```
  Beispiel: insert attribute 'id' to /
  ```

- Menge von **Element-Deklarationen**
  Neue Element-Deklarationen werden mit der *Insert*-Anweisung eingefügt und kann mit *Delete* wieder entfernt. Auch die Element-Deklarationen besitzt die Eigenschaft 'Scope' und verhält sich wie eine Attribut-Deklarationen beim Ändern dieser.

  ```
  Beispiel: set scope='true' of /books/book
  ```

- Menge von **Attributgruppen- und Elementgruppen-Definitionen**
  Gruppen-Definitionen dienen der Zusammenfassung von Deklaration und kommen nur in der Schemawurzel vor. Sie werden mit Hilfe der *Insert*-Anweisung eingefügt und mit *Delete* wieder entfernt.

  ```
  Beispiel: insert attributegroup newGroup{/@attr1,/@attr2}
  ```

- Menge von **Notations-Deklarationen und Anmerkungen**
  Notations-Deklarationen und Anmerkungen nehmen nicht an dem Validierungsprozess bei. Sie werden mit Hilfe der *Insert*-Anweisung eingefügt und mit *Delete* wieder entfernt.

  ```
  Beispiel: delete /annotation()
  ```

4.4.2 Attribut-Knoten

Ein Attribut-Knoten besteht aus einer Attribut-Verwendung und Attribut-Deklaration.

Eigenschaften

- Namen und Namensraum - **name = QName**
  Der Name und Namensraum wird mit der *Rename*-Anweisung geändert.

  `Beispiel: rename //book/@isbn as 'newnamespace:newname'`

- Typdefinition - **type = QName**
  Jede Attribut-Deklaration verweist auf eine einfache Typdefinition. Ist keine Typdefinition angegeben, bekommt die Deklaration implizit den Typ 'anySimpleType' zugewiesen. Die Typdefinition wird mit der *Set*-Anweisung geändert.

  `Beispiel: set type='xs:string' //book/@isbn`

- Gültigkeitsbereich - **scope = ('true' | 'false')**
  Der Gültigkeitsbereich (eng. scope) bestimmt, ob die Attribut-Deklaration global oder lokal definiert ist. Mit Hilfe der *Set*-Anweisung kann die Eigenschaft auf 'true' (global) oder 'false' (lokal) gesetzt werden.

  `Beispiel: set scope='true' of //book/@isbn`

- Vorgabewert - **(default | fixed) = String**
  Die Wertebereichsbeschränkung kann ein Standardwert oder ein fixierter Wert sein. Diese Eigenschaft wird mit Hilfe der *Set*-Anweisung verändert.

  `Beispiel: set default='$70.000' of /employee/@income`

- Verwendung - **use = ('optional' | 'required' | 'prohibited')**
  Die Attributverwendung (eng. use) ist entweder optional, zwingend oder verboten. Diese Eigenschaft wird mit Hilfe der *Set*-Anweisung verändert.

  `Beispiel: set use='required' of //book/@isbn`

- Anmerkung - **annotation = any**
  Anmerkungen enthalten Informationen über das Schema für den Benutzer und der Anwendung. Sie nehmen keinen Einfluss auf den Validierungsprozess. Die Anmerkung kann mit der *Insert*-Anweisung erweitert werden und mit *Delete* entfernt werden.

```
Beispiel: insert appinfo content='some information' to
          /@isbn
```

```
Beispiel: delete /@isbn/annotation()
```

4.4.3 Element-Knoten

Ein Element-Knoten besteht aus einem *Partikel* und einer *Element-Deklaration*. Das *Partikel* definiert das minimale und maximale Vorkommen und verweist auf die Element-Deklaration.

Eigenschaften

- Namen und Namensraum - **name = QName**
  Der Name und Namensraum wird mit der *Rename*-Anweisung geändert.

  ```
  Beispiel: rename /book as 'Buch'
  ```

- Typdefinition - **type = (QName | Typdefinition)**
  Jede Element-Deklaration verweist auf eine Typdefinition, die entweder vom einfachen oder komplexen Typ ist. Handelt es sich um eine globale Typdefinition, befindet sie sich in der Schemawurzel und wird durch einen qualifizierten Namen referenziert. Ist es eine lokale Typdefinition, ist sie Bestandteil der Element-Deklaration und kann von keiner anderen Komponente verwendet werden.

  ```
  Beispiel: set type='typeOfBook' of /book
  ```

- Gültigkeitsbereich - **scope = ('true' | 'false')**
  Der Gültigkeitsbereich (eng. scope) bestimmt, ob die Element-Deklaration global oder lokal definiert ist. Mit Hilfe der *Set*-Anweisung kann die Eigenschaft auf 'true' (global) oder 'false' (lokal) gesetzt werden.

  ```
  Beispiel: set scope='true' of //book/author
  ```

- Vorgabewert - **(default | fixed) = string**
  Die Wertebereichsbeschränkung kann ein Standardwert oder ein fixierter Wert sein. Diese Eigenschaft wird mit Hilfe der *Set*-Anweisung verändert.

```
Beispiel: set fixed='EURO' of /book/price/currency
```

- Nullbarkeit - **nillable = ('true' | 'false')**
  Eine Elementdeklaration, die zwingende Elemente beinhaltet, kann trotzdem gültig sein, wenn sie leer ist und die Eigenschaft 'nillable' auf 'true' gesetzt ist. Mit Hilfe der *Set*-Anweisung kann die Eigenschaft auf 'true' oder 'false' gesetzt werden.

  ```
  Beispiel: set nillable='true' of //book/editor
  ```

- Abstrakt - **abstract = ('true' | 'false')**
  Wird ein Element als abstrakt deklariert, darf es nur in Verbindung mit einer Ersetzungsgruppe verwendet werden. Mit Hilfe der *Set*-Anweisung kann die Eigenschaft auf 'true' oder 'false' gesetzt werden.

  ```
  Beispiel: set abstract='true' of /abstract_element
  ```

- Minimales Vorkommen - **minOccurs = NumericLiteral**
  Das Minimale Vorkommen gibt an, wie oft das Element in der Instanz vorkommen muss. Es kann mit der *Set*-Anweisung geändert werden.

  ```
  Beispiel: set minOccurs=0 of /books/book
  ```

- Maximales Vorkommen - **maxOccurs = NumericLiteral**
  Das Maximale Vorkommen gibt an, wie oft das Element in der Instanz vorkommen darf. Es kann mit der *Set*-Anweisung geändert werden.

  ```
  Beispiel: set maxOccurs='unbounded' of /books/book
  ```

- Anmerkung - **annotation = any**
  (gleich der Anmerkung eines Attribut-Knotens)

4.4.4 Elementgruppen-Knoten

Eine Elementgruppe ist entweder eine Sequenz, Auswahl oder eine Menge. Sie kann Element-, Wildcard- und weitere Elementgruppen-Knoten aufnehmen.

Eigenschaften

- Verbundtyp - **compositor = ('sequence' | 'choice' | 'all')**
  Der Verbundtyp der Elementgruppe gibt an, ob der Inhalt eine Liste, Auswahl oder Menge ist. Geändert wird die Eigenschaft mit der *Set*-Anweisung.

  ```
  Beispiel: set compositor='all' of /hobbies/modelgroup()
  ```

- Liste oder Menge von **Element-, Wildcard- und Elementgruppen-Knoten**
  Der Inhalt der Elementgruppe kann mit der *Insert*-Anweisung mit Element-, Wildcard- und Elementgruppen-Knoten erweitert werden. Mit *Delete* werden Bestandteile wieder entfernt.

  ```
  Beispiel: insert element 'swimming' to /hobbies/modelgroup()
            gleichbedeutend mit der Kurzform
            insert element 'swimming' to /hobbies
  ```

- Minimales Vorkommen - **minOccurs = NumericLiteral**
  Das Minimale Vorkommen gibt an, wie oft die Elementgruppe in der Instanz vorkommen muss. Es kann mit der *Set*-Anweisung geändert werden.

- Maximales Vorkommen - **maxOccurs = NumericLiteral**
  Das Maximale Vorkommen gibt an, wie oft die Elementgruppe in der Instanz vorkommen kann. Es kann mit der *Set*-Anweisung geändert werden.

- Anmerkung - **annotation = any**
  (gleich der Anmerkung eines Attribut-Knotens)

4.4.5 Typdefinitions-Knoten

Ein Typdefintions-Knoten beinhaltet eine Typdefinition. Der Inhalt bestimmt, ob sie vom Typ einfach oder komplex ist.

Eigenschaften

- Namen und Namensraum - **name = QName**
  Der Name und Namensraum wird mit der *Rename*-Anweisung geändert. Lokal definierte Typdefinitionen besitzen keinen Namen. Wenn einer

lokal definierten Typdefinition ein Name zugewiesen, wird der Typdefinitions-Knoten in die Schemawurzel verschoben. Der Knoten, deren Bestandteil die Typdefinition vorher war, referenziert dann auf die globale Typdefinition.

```
Beispiel: rename /book/typedefinition() as 'typeOfBook'
```

- **abstract = ('true' | 'false') - Abstrakt**
  Wird eine Typdefinition als abstrakt deklariert, darf sie nicht direkt, sondern nur durch einen abgeleiteten Typen verwendet werden. Mit Hilfe der *Set*-Anweisung kann die Eigenschaft auf 'true' oder 'false' gesetzt werden.

- Basistypdefinition - **basetype**
  Der Basistyp in der Typhierarchie wird mit der Eigenschaft 'basetype' definiert. Ändern lässt sich dies mit der *Set*-Anweisung. Der Basistyp wird durch einen qualifizierten Namen (QName) angegeben.

- Abgeschlossenheit - **finale = ('true' | 'false')**
  Wird eine Typdefinition als abgeschlossen definiert, kann sie nicht als Basistyp einer anderen Typdefinition verwendet werden. Geändert wird die Eigenschaft der Abgeschlossenheit mit der *Set*-Anweisung.

- Menge von **Attribut-Knoten**
  Ein Attribut-Knoten besteht aus einer Attribut-Verwendung und einer Attribut-Deklaration. Mit Hilfe der *Insert*-Anweisung werden Attribut-Knoten eingefügt und mit *Delete* wieder entfernt.

```
Beispiel: insert attribute 'isbn' to /book/typedefinition()
          ist gleichbedeutend mit der Kurzform
          insert attribute 'isbn' to /book

Beispiel: delete /book/typedefinition()/@isbn
          ist gleichbedeutend mit der Kurzform
          delete /book/@isbn
```

- Gemischter Inhalt - **mixed = ('true' | 'false')**
  Ist die Basistypdefinition eine Komplexe Typdefinition, kann mit dieser Option der Inhalt von Zeichenketten erlaubt oder verboten werden.

- Einschränkende Fassetten - **facets**
  Einfache Typdefinitionen werden mit Hilfe der Fassetten *length*, *minLength*, *maxLength*, *enumeration*, *whitespace*, *pattern*, *minExclusive*, *maxExclusive*, *minInclusive*, *maxInclusive*, *totalDigits*, *fractionDigits* eingeschränkt. Für eine genaue Beschreibung der einzelnen Fassetten wird auf [BM04] verwiesen.

- Abgeschlossenheit - **final = (all | list | extension | restriction | union)**
  Mit der Eigenschaft der Abgeschlossenheit werden die Möglichkeiten der Vererbung eingeschränkt.

- Art - **variety = ('atomic' | 'list' | 'union')**
  Die Art der Bildung einfacher Typdefinitionen wird durch die Eigenschaft *varierty* bestimmt.

- Anmerkung - **annotation = any**
  (gleich der Anmerkung eines Attribut-Knotens)

4.4.6 Identitätsbeschränkungs-Definitions-Knoten

Mit Identitätsbeschränkungs-Definitionen können Schlüssel und Schlüssel/Fremdschlüsselbeziehungen auf Knotenmengen definiert werden.

Eigenschaften

- Namen und Namensraum - **name = QName**
  Der Name und Namensraum wird der *Rename*-Anweisung geändert.

- Kategorie - **category = ('key' | 'keyref' | 'unique')**
  Eine Identitätsbeschränkungs-Definition ist entweder ein zwingender Schlüssel, optionaler Schlüssel oder Fremdschlüssel. Das Ändern der Schlüsseleigenschaft von zwingend auf optional und umgekehrt kann mit der *Set*-Anweisung realisiert werden. Ein existierender Fremdschlüssel kann nicht zu einem Schlüssel umgewandelt werden.

  ```
  Beispiel: set category='unique' of
                /book/constraint::schluessel
  ```

- Selektor - **selector = XPath**
  Der Selektor ist ein eingeschränkter, relativer XPath-Ausdruck, der es erlaubt, eine Menge von Knoten der Untermenge des Kontextknotens

zu selektieren. Er stellt den Ausgangspunkt für die Felder dar. Mit Hilfe der *Set*-Anweisung kann der Selektor geändert werden.

```
Beispiel: set selector='./books' of /book
```

- Liste von Feldern - **field = XPath**
  Die Felder sind wie der Selektor eingeschränkte, relative XPath-Ausdrücke. Die Kombination aus Selektor und die Liste der Felder stellen einen Schlüsselausdruck dar. Handelt es sich um einen zwingenden Schlüssel oder eine Schlüsselreferenz, müssen alle Felder in der Instanz existieren. Ist es hingegen ein optionaler Schlüssel, sind nicht alle Felder zwingend. Es werden hier nur Instanzen beachtet, wo alle Felder existieren. Die Liste der Felder kann mit der *Set*-Anweinsung manipuliert werden.

  ```
  Beispiel: set field='@isbn' of /book
  ```

  ```
  Beispiel: set field='@title publisher' of /book
  ```

- Referenz - **reference = QName**
  Eine Fremdschlüsselreferenz verweist mit einem qualifizierten Namen auf eine Schlüsseldefinition. Die Zuweisung kann mit der *Set*-Anweisung geändert werden.

- Anmerkung - **annotation = any**
  (gleich der Anmerkung eines Attribut-Knotens)

4.4.7 Wildcard- und Attribut-Wildcard-Knoten

Ein Wildcard-Knoten besteht aus einem *Partikel* und einer Wildcard. Wildcards ermöglichen das Einfügen von fremdem Inhalt in die XML-Instanz. Die Herkunft des fremden Inhalts kann auf bestimmte Namensräume eingeschränkt werden. Weiterhin gibt es drei verschiedene Validierungsmodis.

Eigenschaften

- Liste von Namensräumen - **namespacelist = ('##any' | '##other' | Liste von '##Namensraum')**
  Der fremde Inhalt kann durch eine Liste aus Namensräumen eingeschränkt werden. Die Eigenschaft wird mit der *Set*-Anweisung gesetzt.

```
Beispiel: set namespacelist='##any' of
              /book/modelgroup()/wildcard()[1]
```

- Validierungsmodus - **processContents = ('lax' | 'skip' | 'strict')**
  Der Validierungsmodus bestimmt, ob der fremde Inhalt gar nicht, streng oder nur wenn eine Typdefinition vorhanden ist, validiert wird. Die Eigenschaft wird mit der *Set*-Anweisung gesetzt.

```
Beispiel: set processContents='skip' of
              /book/modelgroup()/wildcard()[1]
```

- Minimales Vorkommen - **minOccurs = NumericLiteral**
  Das Minimale Vorkommen bestimmt, wie viele der fremden Element-Knoten oder Attribut-Knoten in der Instanz vorkommen müssen. Ändern lässt sich dies mit der *Set*-Anweisung.

```
Beispiel: set minOccurs=0 of /book/modelgroup()/
              wildcard()[1]
```

- Maximales Vorkommen - **maxOccurs = NumericLiteral**
  Das Maximale Vorkommen bestimmt wie viele der fremden Element-Knoten oder Attribut-Knoten in der Instanz vorkommen dürfen. Ändern lässt sich dies mit der *Set*-Anweisung.

```
Beispiel: set maxOccurs='unbounded' of
              /book/modelgroup()/wildcard()[1]
```

- Anmerkung - **annotation = any**
  (gleich der Anmerkung eines Attribut-Knotens)

4.4.8 Elementgruppen-Definitions-Knoten

Elementgruppen-Definitionen ermöglichen es, Element-, Elementgruppen- und Wildcard-Knoten zu einer Einheit zusammenzufassen.

Eigenschaften

- Namen und Namensraum - **name = QName**
  Der Name und Namensraum wird mit der *Rename*-Anweisung geändert.

- Liste oder Menge von **Elementgruppen- und Wildcard-Knoten**
  Elementgruppen- und Wildcard-Knoten können mit der *Insert*-Anweisung eingefügt und mit der *Delete*-Anweisung entfernt werden.

- Anmerkung - **annotation = any**
  (gleich der Anmerkung eines Attribut-Knotens)

4.4.9 Attributgruppen-Definitions-Knoten

Attributgruppen-Definitionen ermöglichen es, Attribut- und Attribut-Wildcard-Knoten zusammenzufassen.

Eigenschaften

- Namen und Namensraum - **name = QName**
  Der Name und Namensraum wird mit der *Rename*-Anweisung geändert.

- Menge von **Attribut-** und **Attribut-Wildcard-Knoten**
  Attribut- und Attribut-Wildcard-Knoten können mit der *Insert*-Anweisung eingefügt und mit der *Delete*-Anweisung entfernt werden.

- Anmerkung - **annotation = any**
  (gleich der Anmerkung eines Attribut-Knotens)

4.4.10 Notation-Definitions-Knoten

Notationen dienen als Verarbeitungshinweise für Software, die XML-Daten verarbeiten.

Eigenschaften

- Namen und Namensraum - **name = QName**
  Der Name und Namensraum wird mit der *Rename*-Anweisung geändert.

  ```
  Beispiel: rename /notation()[1] as 'win32'
  ```

- Öffentlicher Bezeichner - **public = token**
  Der öffentliche Bezeichner enthält eine Typbezeichnung für die Instanz.

```
Beispiel: set public='image/jpeg' of /notation()[1]
```

- Systembezeichner - **system = anyURI**
  Der Systembezeichner verweist auf eine Anwendung, welche die Instanz verarbeiten kann.

  ```
  Beispiel: set system='jpegview.exe' of /notation()[1]
  ```

- Anmerkung - **annotation = any**
  (gleich der Anmerkung eines Attribut-Knotens)

4.4.11 Anmerkungs-Knoten

Eine *Anmerkung* enthält Informationen für die verarbeitende Anwendung und für den Benutzer als Dokumentation. Mit der *Insert*-Anweisung kann eine neue Anmerkung hinzugefügt und eine existierende erweitert werden. Das Manipulieren existierender Bestandteile einer Anmerkung ist nicht vorgesehen. Es ist nur möglich, eine komplette Anmerkung mit der *Delete*-Anweisung zu entfernen.

Eigenschaften

- Menge von Informationen für die Anwendung - **appinfo**
  - Quelle - **source = anyURI**
  - Inhalt - **content = any**
- Menge von Dokumentationen für den Benutzer - **documentation**
  - Quelle - **source = anyURI**
  - Sprache - **language = language**
  - Inhalt - **content = any**

  ```
  Beispiel: insert documentation language="german"
                   content="Dieses Element..." into /@isbn
  (Fügt eine Dokumentation in die Attribut-Deklaration
   "isbn" ein.)

  Beispiel: delete /person/annotation()
  (Entfernt die Anmerkung aus der Element-Deklaration
   "person".)
  ```

4.5 Konstruktoren

```
[81] FilterExpr        ::= Literal | PrimaryExpr
[84] PrimaryExpr       ::= Constructor
[85] Literal           ::= NumericLiteral | StringLiteral
[94] Constructor       ::= DirectConstructor
[95] DirectConstructor ::= DirElemDeclConstructor
                           | DirAttrDeclConstructor
                           | DirTypeConstructor
                           | DirModelGroupConstructor
                           | DirKeyConstructor
                           | DirKeyRefConstructor
                           | DirUniqueConstructor
                           | DirAnyConstructor
                           | DirAnyAttrConstructor
                           | DirElemGroupConstructor
                           | DirAttrGroupConstructor
                           | DirNotationConstructor
                           | DirAnnotationConstructor
```

Dieser Abschnitt definiert für jeden Knotentyp einen Konstruktor. Konstruktoren werden verwendet, um das Schema mit neuen Knoten zu erweitern.

4.5.1 Element-Knoten

```
[162] DirElemDeclConstructor = "element" '"'NCName'"'
                               ("of type" '"'QName'"')?
```

Der Element-Konstruktor erzeugt einen Element-Knoten, bestehend aus einer Element-Deklaration und einem Partikel. Die der Element-Deklaration zugewiesene Typdefinition kann optional angegeben werden.
Der neue Knoten besitzt folgende Standardeigenschaft:

```
type      = 'anyType'
scope     = (abhängig vom Zielknotens)
default   = null
fixed     = null
nillable  = 'false'
abstract  = 'false'
minOccurs = 0
maxOccurs = 1
```

4.5.2 Attribut-Knoten

```
[163] DirAttrDeclConstructor = "attribute" '"'NCName'"'
                              ("of type" '"'QName'"')?
```

Der Attribut-Konstruktor erzeugt einen Attribut-Knoten, welcher sich aus einer Attribut-Deklaration und einer Attribut-Verwendung zusammensetzt. Eine Typdefiniton kann optional definiert werden.
Der neue Knoten besitzt folgende Standardeigenschaft:

```
type    = 'anySimpleType'
scope   = (abhängig vom Zielknoten)
default = null
fixed   = null
use     = 'optional'
```

4.5.3 Typdefinitions-Knoten

```
[164] DirTypeConstructor = "typedefinition" ('"'NCName'"')?
```

Der Konstruktor erzeugt eine leere Typdefinition, die keinen Inhalt erlaubt.
Der neue Knoten besitzt folgende Standardeigenschaft:

```
abstract          = false
basetype          = none
finale            = false
mixed             = false
contenttype       = empty
facets            = -
fundamentalfacets = -
variety           = -
```

4.5.4 Elementgruppen-Knoten

```
[165] DirModelGroupConstructor = ("sequence"|"choice"|"all")
        ("{" (ElementDecalaration|Wildcard|ModelGroup)+"}")?
```

Der Konstruktor erzeugt eine Sequenz, Auswahl oder Menge. Optional kann eine Liste von Element-, Wildcard- und Elementgruppen-Knoten als Inhalt angegeben werden.
Der neue Knoten besitzt folgende Standardeigenschaft:

```
maxOccurs       = 1
minOccurs       = 1
```

4.5.5 Identitätsbeschränkungs-Definitions-Knoten

```
[166] DirKeyConstructor    = "key" NCName "selector=" XPath
                       ("field=" XPath)+
[167] DirKeyRefConstructor = "keyref" NCName "refer=" QName
                       "selector=" $XPath ("field=" $XPath)+
[168] DirUniqueConstructor = "unique" NCName "selector="
                       XPath ("field=" XPath)+
```

Der Konstruktor erzeugt einen Schlüssel- oder Fremdschlüssel-Knoten.

4.5.6 Wildcard-Knoten

```
[169] DirAnyConstructor = "any"
               ("processContents=" ("lax"|"skip"|"strict"))?
               ("namespace=")?
               ("maxOccurs=" nonNegativeInteger)?
               ("minOccurs=" nonNegativeInteger)?
```

Der Konstruktor erzeugt einen Wildcard-Knoten.
Der neue Knoten besitzt folgende Standardeigenschaft:

```
processContents = strict
namespace       = ##any
maxOccurs       = 1
minOccurs       = 1
```

4.5.7 Attribut-Wildcard-Knoten

```
[170] DirAnyAttrConstructor = "anyAttribute"
              ("processContents=" ("lax"|"skip"|"strict"))?
              ("namespace=")?
```

Der Konstruktor erzeugt einen Attribut-Wildcard-Knoten.
Der neue Knoten besitzt folgende Standardeigenschaft:

```
processContents = strict
namespace       = ##any
```

4.5.8 Elementgruppen-Definitions-Knoten

```
[171] DirElemGroupConstructor = "elementgroup" NCName
        ("{" (ElementDecalaration|Wildcard|Elementgroup)+"}")?
```

Der Konstruktor erzeugt einen Elementgruppen-Knoten. Optional kann eine Liste von Element-, Wildcard- und Elementgruppen-Knoten als Inhalt angegeben werden.

4.5.9 Attributgruppen-Definitions-Knoten

```
[172] DirAttrGroupConstructor = "attributegroup" NCName
             ("{" (AttributDeclaration|AttributWildcard)+"}")?
```

Der Konstruktor erzeugt einen Attributgruppen-Knoten. Optional kann eine Liste von Attribut- und Attribut-Wildcard-Knoten als Inhalt angegeben werden.

4.5.10 Notations-Knoten

```
[173] DirNotationConstructor = "notation" NCName
                               "public=" NCName
                               "system=" anyURI}
```

Der Konstruktor erzeugt einen Notations-Knoten.

4.5.11 Anmerkungs-Knoten

```
[174] DirAnnotationConstructor = ("appinfo"|"documentation")
                                 ("source=" anyURI)?
                                 ("language=" language)?
                                 ("content=" any}?
```

Der Konstruktor erzeugt einen Anmerkungs-Knoten.

4.6 Funktionen

In diesem Abschnitt werden die Funktionen definiert. Sie werden benötigt, um bestimmte Informationen des Schemas zu erfragen.

Allgemein

- `"xsd:instance(" SourceExpr ")"`
  `returns List of PathExpr`

  Die Funktion *xsd:instance* dient dazu anhand eines Schemaknotens die Knoten im Instanzdokument aufzufinden. Dazu wird eine Liste von XPath-Ausdrücken generiert, die alle möglichen Instanz-Knoten enthält. Wird die Rekursion erlaubt, ist eine besondere Beachtung der Implementierung nötig. Falls eine Rekursion im Schema auftritt, würde das Resultat eine unendliche Menge von XPath-Ausdrücken sein.

  ```
  Beispiel: insert attribute abc{'new value'} into /a/b/c;

  XQuery-Update: declare function recursion(
                        $ref as PathExpr, $step as PathExpr) {
                   if ($ref) the {
                     do insert attribute abc{'new value'}
                        into $ref;
                     recursion($ref/$step);
                   }
                 }

                 recursion{/a/b/c,/b/c};
  ```

  In dem Beispiel wird ein neues Attribut in die Element-Deklaration *c* eingefügt. In dem Schema gibt es eine Rekursion. Jedes Element *c* besitzt ein optionales Element *b* und jedes Element *b* ein zwingendes Element *c*.

  Die generierte rekursive Funktion erweitert alle Instanz-Knoten mit dem neuen Attribut.

- `"xsd:name(" SourceExpr ")"`
  `returns QName`

  Liefert den qualifizierten Namen eines Knotens.

- `"xsd:scope(" SourceExpr ")"`
  `returns "true" | "false"`

  Gibt den Gültigkeitsbereich des Knotens zurück. Entweder 'true' für global oder 'false' für lokal.

- `"xsd:default(" SourceExpr ")"`
  `returns Literal`

  Gibt den Defaultwert eines Knotens zurück.

- `"xsd:fixed(" SourceExpr ")"`
  `returns Literal`

  Gibt den definierten fixierten Wert eines Knotens zurück.

Attribut-Knoten

- `"xsd:use(" SourceExpr ")"`
  `returns "optional" | "required" | "prohibited"`

  Für Attribut-Knoten gibt es die Attributverwendung zurück.

Element-Knoten

- `"xsd:minOccurs(" SourceExpr ")"`
  `returns NumericLiteral`

  Gibt das Minimale Vorkommen eines Knotens zurück.

- `"xsd:maxOccurs(" SourceExpr ")"`
  `returns NumericLiteral`

  Gibt das Maximale Vorkommen eines Knotens zurück.

- `"xsd:nillable(" SourceExpr ")"`
  `returns "true" | "false"`

  Ist 'true', falls der Zustand Null erlaubt ist und 'false' wenn nicht.

- `"xsd:abstract(" SourceExpr ")"`
  `returns "true" | "false"`

  Ist 'true' falls der Knoten eine abstrakte Komponente darstellt.

- `"xsd:mixed(" SourceExpr ")"`
  `returns "true" | "false"`

  Erlaubt der Element-Knoten gemischten Inhalt, gibt die Funktion 'true' zurück.

Identitätsbeschränkungs-Definitions-Knoten

- `"xsd:selector(" SourceExpr ")"`
  `returns PathExpr`

  Gibt den durch einen XPath-Ausdruck selektieren Teilbaum zurück.

- `"xsd:field(" SourceExpr ")"`
  `returns List of PathExpr`

  Gibt eine Liste von XPath-Ausdrücken zurück.

- `"xsd:reference(" SourceExpr ")"`
  `returns QName`

  Liefert den qualifizierten Namen einer Schlüsseldefinition zurück.

Fassetten einfacher Typdefinitionen

- `"xsd:length(" SourceExpr ")"`
  `"xsd:minLength(" SourceExpr ")"`
  `"xsd:maxLength(" SourceExpr ")"`
  `returns NumericLiteral`

  Diese Funktionen liefern für einfache Typdefinitionen die Fassetten, welche die Länge der Zeichenkette einschränken, zurück.

- `"xsd:enumeration(" SourceExpr ")"`
  `Returns Expr`

  Wurde der Wertebereich auf bestimmte Werte beschränkt, liefert diese Funktion die Menge der erlaubten Werte.

- `"xsd:whitespace(" SourceExpr ")"`
  `returns "preserve" | "replace" | "collapse"`

  Mit dieser Funktion wird der Ablauf der Normalisierung besonderer Zeichen (z.B. Leerzeichen) erfragt.

- `"xsd:pattern(" SourceExpr ")"`
  `returns RegularExpression`

  Diese Funktion liefert den regulären Ausdruck zurück, mit dem der Inhalt der Zeichenkette eingeschränkt werden kann.

- `"xsd:minExclusive(" SourceExpr ")"`
  `"xsd:maxExclusive(" SourceExpr ")"`
  `"xsd:minInclusive(" SourceExpr ")"`

`"xsd:maxInclusive(" SourceExpr ")"`
`returns Literal`

Diese Funktionen liefern die definierten Unter- und Obergrenzen zurück.

- `"xsd:totalDigits(" SourceExpr ")"`
  `returns NumericLiteral`

  Die erlaubten Dezimalstellen können mit dieser Funktion abgefragt werden.

- `"xsd:fractionDigits(" SourceExpr ")"`
  `returns NumericLiteral`

  Diese Funktion liefert die Anzahl der erlaubten Nachkommastellen zurück.

Kapitel 5

Implementierungsdetails

Dieses Kapitel beschreibt die im Rahmen der Diplomarbeit entstandene Implementierung. Es wird ein Überblick über Aufbau, Funktionsweise und dabei aufgetretenen Probleme gegeben. Abschließend wird anhand eines konkreten Beispiels der Ablauf dargestellt.

5.1 Aufgabe

Die Aufgabe ist es, einen Prototyp des Compilers zu entwickeln, der die im vierten Kapitel definierte Sprache umsetzt. Aufgrund der durch die Eingabe gegebenen Evolutionsschritte soll das neue Schema generiert werden. Das Generieren der Updateanweisungen für die Instanzen ist nicht Teil der Aufgabenstellung, wird jedoch ansatzweise umgesetzt.

Beim Prototyping wird zwischen Experimentellem, Explorativem und Evolutionärem Prototyping unterschieden. Der hierbei entstandene Prototyp entspricht dem Experimentellen und Explorativen Prototyping. Mit Hilfe des Experimentellen Prototyping wird die Machbarkeit zur Realisierung geprüft. Gewonnene Erkenntnisse können anschließend in die Spezifikation einfließen. Evolutionäres Prototyping bedeutet, dass die Funktionalität schrittweise erweitert wird.

5.2 Verwendete Technologien

Der Prototyp wurde in der Programmiersprache *Java* implementiert, was die Plattformunabhängigkeit sicherstellt. Zur Entwicklung wurde die IDE[1] *Eclipse* verwendet.

[1] Integrated Development Environment

Zur Generierung des Parser für die Evolutionssprache wurde der frei verfügbare Parsergenerator JavaCC ausgewählt. JavaCC ist ein rein in Java entwickelter Top-Down Generator. Er erlaubt Java-Code direkt in die Grammatik-Definition zu schreiben und bietet Erweiterungen zur automatischen Baumgenerierung.

Für die Verarbeitung der XML-Dokumente und Schemata wurde der XML-Parser Apache Xerces ausgewählt. Xerces besitzt eine Implementierung der XML-Schema API, die ein Vorschlag zur Abbildung des PSVI[2] auf eine Programmierschnittstelle ist. Die Abbildung wird mit Hilfe von IDL[3], einer von der *Object Management Group* entwickelten Sprache zur Definition plattform- und programmiersprachenunabhängiger Schnittstellen, beschrieben. Dabei wird ein XML-Schema auf Basis des Schema-Datenmodells auf einen Graphen abgebildet, worüber dann mit Hilfe der Schnittstelle navigiert und Informationen abgefragt werden können. Ein Vorteil von Xerces ist der verfügbare Quellcode, da es sich um ein Opensource Projekt handelt. Dies erlaubt die Erweiterung der Implementierung, was in den nächsten Abschnitten gezeigt wird.

Verwendung	**Produkt**
Entwicklungsumgebung	Sun JDK v1.5.0_07
IDE	Eclipse v3.2.0
Parser Generator	JavaCC v4.0
XML-Parser inkl. XML Schema API	Apache Xerces v2.8.0 (CVS-Version)
XML-Datenbanksystem	eXist v1.0

5.3 Architektur

Dieser Abschnitt beschreibt den Aufbau des Systems und die Funktionsweise der verschiedenen Module.

Die Abbildung 5.1 gibt einen groben Überblick über die Funktionsweise des Evolutions-Prozessor und der Module.

Evolutions-Prozessor Der Evolutions-Prozessor umfasst alle Module und stellt den Prototypen dar. Als Eingabe benötigt der Prozessor eine Schemaevolutions-Anweisung und die Angabe eines XML-Schemas. Die Ausgabe ist ein neu generiertes Schema oder eine Menge von Fehlermeldungen. Diese

[2] Post Schema Validation Infoset
[3] Interface Definition Language

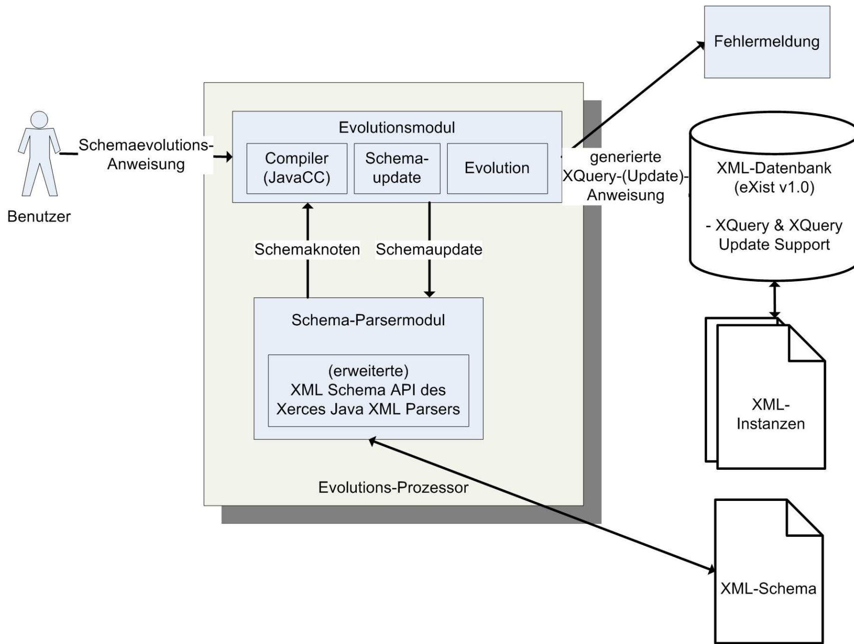


Abbildung 5.1: Überblick der Implementation

Fehler können vom Schema-Parsermodul, dem Compiler oder durch Integritätsverletzungen bei der Schemaänderung hervorgerufen werden. Weiterhin leitet der Prozessor eine generierte XQuery-Update-Anweisung an eine externe Datenbank, in diesem Fall *eXist*, weiter.

Evolutionsmodul Das Evolutionsmodul unterteilt sich in drei Abschnitte. Der Compiler parst die durch den Benutzer eingegebene Evolutionsanweisung. Während des Parsens werden die Pfadausdrücke ausgewertet und die gewünschten Schemaknoten selektiert. Zur Navigation und Selektion wird die Schnittstelle des Schema-Parsermodules verwendet. Sind die Knotenmengen selektiert und die Evolutionsanweisung erkannt, beginnt der Abschnitt Schemaupdate. Anhand der Anweisung und der Knotenmenge werden die erweiterten Funktionen zur Änderung des Schemas des Parsermoduls aufgerufen. Das Parsermodul übernimmt dabei die Prüfung der Integritätsbedingungen. War die Schemaänderung erfolgreich, beginnt der dritte Schritt die Evolution. Dabei werden anhand der bisher gesammelten Informationen und zusätzliche Schemainformationen die XQuery-Updateanweisung generiert.

Schema-Parsermodul Der Kern des Schema-Parsermoduls ist der Xerces XML-Parser. Dank des frei verfügbaren Quellcodes ist es möglich, den Parser um benötigte Funktionen zu erweitern. Nachdem ein Schema geparst wurde,

existiert eine Abbildung im Speicher. Es gibt für jeden Typ der Schemakomponenten eine Klasse, die Funktion zum Navigieren und Abfragen von Informationen bereitstellt. Die XML-Schema API wurde entworfen, um Schemainformationen abzufragen, jedoch nicht zu verändern. Daher wurde der Quellcode soweit verändert, dass das Ändern der Objekte, welche die Schemainformationen enthalten, im Speicher möglich ist. Gleichzeitig wurden die Integrationsbedingungen implementiert.

Wenn die Schemaänderung erfolgreich ausgeführt wurde, wird das Schema wieder in der Datenbank im XML-Format gespeichert. Dazu wurde jedes Schemaobjekt um eine Funktion erweitert, welche die Objektinformation serialisiert und in einem XML-Fragment weitergibt. Wird das Objekt der Schemawurzel serialisiert, impliziert das rekursiv die Erzeugung des gesamten Schemas.

Durch das Erweitern der Klassen ist es nicht nötig, die Schemaänderungen an der XML-Instanz durchzuführen. Gleichzeitig können die Integrationsbedingungen dort implementiert werden, wo sie überprüft werden müssen. Die XML-Instanzen können validiert werden, ohne das Schema vorher neu parsen zu müssen.

5.4 Beispiel

Dieser Abschnitt zeigt den Ablauf des Programms und der Schemaevolution anhand eines Beispiels.

Szenario

Das Beispiel ist eine einfache Literatur-Datenbank. Das Schema besteht aus einem Wurzelelement mit dem Bezeichner *ROOT*. Das Wurzelelement selbst besitzt das optionale Element *Buecher*, welches beliebig viele Unterelemente mit dem Namen *Buch* besitzen kann. Zu jedem *Buch*-Element kann optional ein Verlag angegeben werden. Das komplette Schema ist in Abbildung 5.2 dargestellt. Die Instanz, ersichtlich in Abbildung 5.3, beinhaltet drei Bücher von zwei unterschiedlichen Verlagen.

Schemaevolution

Das Beispiel zeigt, dass zwei Bücher den gleichen Verlag besitzen. Um Redundanz zu vermeiden, sollen alle Verlage ausgelagert werden und die Beziehung zwischen Buch und Verlag mit Hilfe eines Schlüssel/Fremdschlüsselpaares dargestellt werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
           attributeFormDefault="unqualified">
  <xs:element name="ROOT">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Buecher" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Buch" minOccurs="0" maxOccurs="unbound">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Verlag" minOccurs="0">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="Name" type="xs:string"/>
                          <xs:element name="Adresse"/>
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="Titel"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Abbildung 5.2: XML-Schema vor der Schemaevolution

```
<ROOT>
    <Buecher>
        <Buch Titel="Inside XML">
            <Verlag>
                <Name>Addison-Wesley</Name>
                <Adresse>Anschrift2</Adresse>
            </Verlag>
        </Buch>
        <Buch Titel="DB2">
            <Verlag>
                <Name>mitp</Name>
                <Adresse>Anschrift1</Adresse>
            </Verlag>
        </Buch>
        <Buch Titel="SQL In A Nutshell">
            <Verlag>
                <Name>Addison-Wesley</Name>
                <Adresse>Anschrift2</Adresse>
            </Verlag>
        </Buch>
    </Buecher>
</ROOT>
```

Abbildung 5.3: XML-Instanz vor der Schemaevolution

Ablauf

- Als erstes bekommt das *ROOT*-Element ein neues Unterelement mit dem Namen *Verlage*.

```
insert element Verlage into /ROOT;
```

- Als nächstes werden alle Verlage der Büchern in das neue Element *Verlage* kopiert. Da hierbei eine im Schema existierende Element-Deklaration, welche auch Instanzen besitzt, selektiert wird, werden auch die bestehenden Instanzen kopiert.

```
insert //Verlag into //Verlage;
```

- Da jetzt mehr als eine Instanz vom Element *Verlag* erlaubt ist, wird die Anzahl der maximalen Vorkommen erhöht.

```
set maxOccurs='unbound' of /ROOT/Verlage/Verlag;
```

- Da der Name ein eindeutiges Kriterium für einen Verlag ist, wird dieser der neue Primärschüssel. Beim Setzen des Schlüssels werden alle Duplikate der Instanzen entfernt.

```
insert unique VerlagKey selector=Verlag field=Name
       into //Verlage;
```

- Als nächstes wird der Fremdschlüssel definiert.

```
insert keyref VerlagKeyRef refer=VerlagKey
       selector=Verlag field=Name into //Verlage;
```

- Zum Schluss werden noch alle überflüssigen Nicht-Schlüsselelemente aus den Büchern entfernt.

```
delete //Buch/Verlag/Adresse;
```

Das veränderte Schema und die zugehörige Instanz ist in den Abbildungen 5.4 und 5.5 dargestellt.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
           attributeFormDefault="unqualified">
  <xs:element name="ROOT">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Buecher" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Buch" minOccurs="0" maxOccurs="unbound">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Verlag" minOccurs="0">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="Name" type="xs:string"/>
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="Titel"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Verlage" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Verlag" minOccurs="0" maxOccurs="unbound">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Name" type="xs:string"/>
                    <xs:element name="Adresse"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
          <xs:unique name="VerlagKey">
            <xs:selector xpath="./Verlag"/>
            <xs:field xpath="./Name"/>
          </xs:unique>
          <xs:keyref name="VerlagKeyRef" refer="VerlagKey">
            <xs:selector xpath="./Verlag"/>
            <xs:field xpath="./Name"/>
          </xs:keyref>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Abbildung 5.4: XML-Schema nach der Schemaevolution

```
<ROOT>
    <Buecher>
        <Buch Titel="Inside XML">
            <Verlag>
                <Name>Addison-Wesley</Name>
            </Verlag>
        </Buch>
        <Buch Titel="DB2">
            <Verlag>
                <Name>mitp</Name>
            </Verlag>
        </Buch>
        <Buch Titel="SQL In A Nutshell">
            <Verlag>
                <Name>Addison-Wesley</Name>
            </Verlag>
        </Buch>
    </Buecher>
    <Verlage>
      <Verlag>
        <Name>Addison-Wesley</Name>
        <Adresse>Anschrift2</Adresse>
      </Verlag>
      <Verlag>
        <Name>mitp</Name>
        <Adresse>Anschrift1</Adresse>
      </Verlag>
    </Verlage>
</ROOT>
```

Abbildung 5.5: XML-Instanz nach der Schemaevolution

5.5 Ergebnisse

Das Ergebnis ist eine prototypische Implementierung, in der die Kernfunktionen der Sprachdefinition umgesetzt wurden. Ein Teil der XQuery-Update-Generierung wurde auch umgesetzt.

Während der Entwicklung sind verschiedene Prototypen entstanden. Dabei flossen Ergebnisse der Prototypen auch immer wieder zurück in die Sprachdefinition. Ein Beispiel ist z.B. das Problem, dass die Rekursion in einem XML-Schema nicht mit einem XPath-Ausdruck dargestellt werden kann.

Der Prototyp besteht aus einzelnen Modulen, die jeweils austauschbar und erweiterbar sind. Wird z.B. gefordert XSLT- statt XQuery- Anweisungen zu generieren, muss nur das zuständige Modul ausgetauscht werden.

Kapitel 6

Verwandte Arbeiten

Dieses Kapitel gibt einen Überblick der Arbeiten, die mit dem in dieser Diplomarbeit behandelten Themen verwandt sind. Dazu gehören Schemasprachen, die im XML-Umfeld entstanden sind. Andere Arbeiten mit Vorschlägen zum Erweitern existierender Anfragesprachen. Der Prozess der Schemaevolution und die Umsetzung existierender Systeme.

Schemasprachen

Die Struktur eines XML-Dokumentes wird durch das Markup[1] vorgegeben. Wozu braucht man überhaupt Schemasprachen? Sie dienen der Einschränkung der erlaubten Struktur und der Überprüfung der Inhalte gegenüber vorgegebenen Datentypen. Im XML-Umfeld gibt es eine Reihe von Vorschlägen für Schemasprachen.

DTD und XML-Schema [TBMM04] sind Entwicklungen vom W3C. DTD ist ein Erbe der Sprache SGML [Gol96] und Bestandteil der Syntaxdefinition von XML [BPSM+04]. Im Vergleich zu DTD unterscheidet sich XML-Schema vor allem durch die in XML gehaltene Syntax, dem erweiterten Typsystem, der Trennung von Typ- und Elementdeklaration und der Möglichkeit von Vererbungen.

Relax NG [CM01] und Schematron [SCH04] sind Schemasprachen, die nicht vom W3C entwickelt wurden. Relax NG bietet gegenüber XML-Schema Features, wie z.B. die Struktur vom Inhalt der Daten abhängig zu machen. Schematron kann als Erweiterung existierender Schemasprachen verwendet werden. Ein Schema in Schematron besteht aus einer Menge von Behauptungen (engl. Asserts), die mit Hilfe von XPath-Ausdrücken definiert werden.

Die Arbeit [BL01] von Bonifati gibt einen Vergleich existierender XML Schema- and Anfragesprachen.

[1] der Auszeichnung

Spracherweiterungen

XPath [BBC+05], XQuery [BCF+05] und XSLT [Cla99] sind Entwicklungen vom W3C. Davon ist XPath eine Sprache zur Achsennavigation und Basis weiterer W3C Entwicklungen, wie XQuery und XSLT. XQuery ist eine XML-Anfragesprache mit SQL-ähnlichen Konstrukten. In der ersten Version von XQuery wird keine Updatefunktionalität enthalten sein. Arbeiten, wie z.B. die von Tatarinov [TIHW01], Lehti [Leh01] und Hänsel [Hän02], erweitern den existierenden Sprachvorschlag XQuery um Updatefunktionalität.

Schemaevolution - Änderung des Schemas

In den Arbeiten von Zeitz [Zei01] und von Su [SKR02] wird untersucht, welche Auswirkungen Umformungen der DTD auf die XML-Dokumente haben und wie diese angepasst werden können. Dabei beschreibt Zeitz wie mittels XSLT die der DTD zugehörigen XML-Dokumente angepasst werden können. In der Arbeit von Su wird ein Framework vorgestellt, welches eine Menge von Stammfunktionen zur Änderung der DTD besitzt. Die XML-Dokumente werden in eine Objektdatenbank abgelegt. Die DTD wird weiterhin in ein für die Datenbank verständliches OO-Schema umgewandelt.

Schemaevolution - Änderung der Instanzen

In der Arbeit von Klettke, Meyer und Hänsel [KMH05] wird untersucht, welche Auswirkungen ein XML Update auf das durch eine DTD gegebenes XML Schema hat. Updates auf XML-Dokumente können die Struktur verändern und somit das Dokument ungültig gegenüber dem gegebenen Schema machen. Es werden verschiedene Wege zur Lösung dieses Problems geschildert. Eine Möglichkeit, die in der Arbeit detailliert beschrieben wird, ist es die DTD nach einem XML Update anzupassen.

In meiner Studienarbeit [Wil06] wird untersucht, welche XML-Updateoperationen das Schema eines gegebenen XML-Dokumentes ungültig machen. Für diese Fälle werden mögliche Schemaevolutionschritte generiert, welche die Gültigkeit des Schemas erhalten und damit das Update realisierbar machen. Eine prototypische Implementierung zeigt dabei die Möglichkeit einer Umsetzung.

Systeme

Softwarefirmen, wie IBM, Microsoft oder Oracle, erweitern ihre Datenbanksysteme um XML Support. Dabei gibt es die Möglichkeiten, XML auf relationale Tabellen abzubilden oder sie in einem nativen Format abzuspei-

chern. SQL/XML und XQuery sind die Anfragesprachen, die sich als Standard durchgesetzt haben.

Wie sieht es aus mit der Unterstützung von XML Schema und Schema Evolution?

Die Arbeit [BOSdL05] beschreibt, auf welche Weise das von IBM entwickelte Datenbanksystem DB2 die Schemaevolution unterstützt. Um eine Versionisierung von XML Schemata zu ermöglichen, werden XML-Dokumente und Schemata unabhängig voneinander verwaltet. Auf statische Typüberprüfung und die Unterstützung des Imports von Schemata in einem XML-Dokument wird verzichtet. Die XML-Dokumente werden in einer Spalte einer Tabelle mit dem Datentyp XML abgelegt. Die Registrierung der Schemata erfolgt in einem Schemarepository. Jeder XML-Instanz kann nun ein beliebiges XML-Schema aus dem Repository zugewiesen werden, ohne die Instanz dabei zu verändern.

Weiterhin wird ein Protokoll vorgestellt, welches das System auf eine neue Version eines Schemas hinweist. Die neuen Dokumente bekommen automatisch das neue Schema zugewiesen. Für die bereits existierenden Dokumente wird gezeigt, dass bei abwärtskompatiblen Schemata die Zuweisungen automatisch vorgenommen werden können. Wenn keine automatische Anpassung möglich ist, bleibt die Zuordnung der älteren Schemata erhalten. Weiterhin wird gezeigt, wie die Schemaänderungen auch Auswirkungen auf die Anfragen haben können, selbst wenn das Schema abwärtskompatibel ist.

Somit bleibt die Aufgabe der Schemaevolution bei dem Anwender. Die Möglichkeit, gleichzeitig verschiedene Schemata zu verwenden, bringt vor allem bei der Verarbeitung von Anfragen und Änderungen einen deutlichen Mehraufwand. Unter Umständen müssen verschiedene Indizes für die verschiedenen Schemata erstellt werden.

Kapitel 7

Schlussbetrachtung

Dieses Abschließende Kapitel gibt eine Zusammenfassung der Arbeit, wobei Schwerpunkte, Neuansätze aber auch Schwächen des vorgestellten Ansatzes zusammengetragen werden. Darauf folgt ein Ausblick auf noch zu leistende Arbeit.

7.1 Zusammenfassung

Mit dieser Arbeit wurde ein Vorschlag für eine Evolutionssprache für das Datenmodell von XML-Schema[FW04] vorgestellt. Der Sprachvorschlag stellt eine Erweiterung existierender Technologien aus dem XML-Umfeld dar. Die vielseitig verwendete Navigationssprache XPath wurde um das Dantemodell von XML-Schema erweitert. Bei der Entwicklung der Operationen nahm die noch in Entwicklung befindlichen Spracherweiterung XQuery Update Einfluss. Der Sprachvorschlag kann entweder komplett in XQuery integriert werden oder eigenständig sein. Durch die Integration in XQuery steht eine turing-vollständig Anfragesprache bereit.

Evolution bedeutet bei einer Schemaänderung auch die zugehörigen Instanzen anzupassen. Im dritten Kapitel wurde eine Klassifizierung möglicher Schemaänderungen erstellt. Für jede dieser Änderungen wurde gezeigt, welche Auswirkungen sie auf das Schema und der zugehörigen Instanzen haben. Es wurde gezeigt wie XQuery-Updateanweisungen generiert werden können, um die Instanzen dem neuen Schema anzupassen.

Während der Arbeit ist eine prototypische Implementierung der Sprache entstanden. Mit Hilfe der Implementierung wurde die Machbarkeit geprüft und wertvolle Erkenntnisse wurden gewonnen.

7.2 Ausblick und weitere Ideen

In diesem letzten Abschnitt folgt noch ein Ausblick auf absehbare und Richtungsgebende Arbeiten.

Die Ausführungszeit und Speicherbedarf von Anfragen und Updateoperationen ist für größere Datenbanken von entscheidender Bedeutung. Bei der Schemaevolution werden meist Strukturänderungen vorgenommen, die nicht während des normalen Betriebes der Datenbank durchgeführt werden können. Daher sollte Teil einer Arbeit die **Aufwandsabschätzung** der einzelnen Operationen sein. Auf der Aufwandsabschätzung folgt die Untersuchung von **Optimierung**smöglichkeiten der Evolutionsanweisungen selbst und der Umsetzung der Anweisungen.

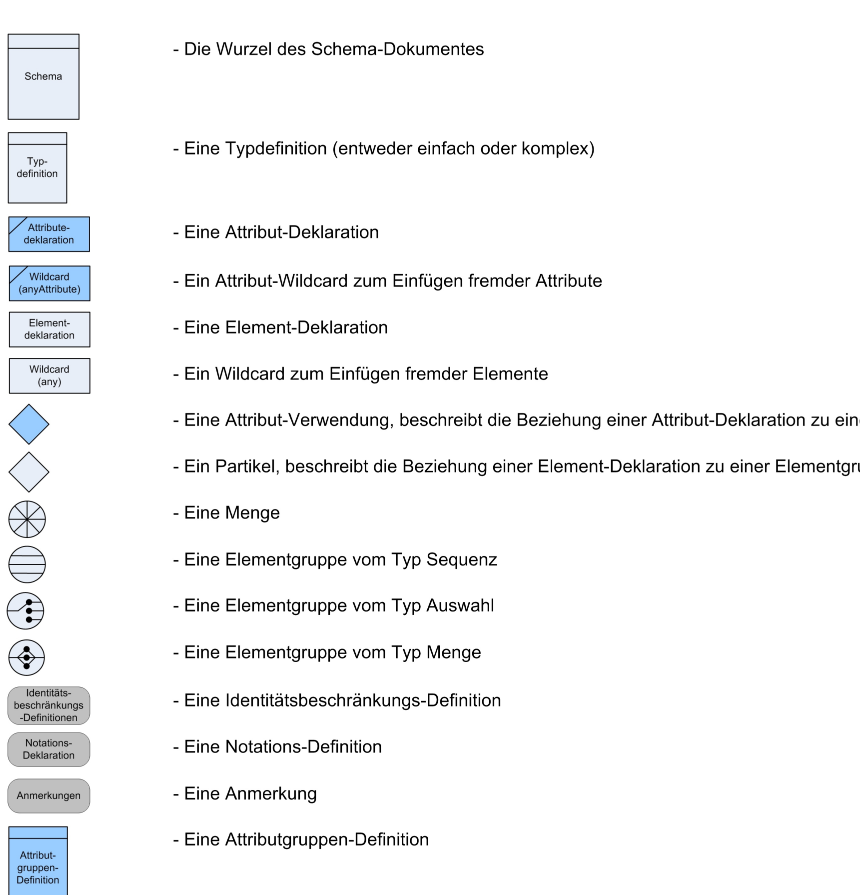
Eine höhere Ausführungszeit wird auch durch die gleichzeitige Ausführung von Anweisungen erreicht. Das Konzept der **Transaktionen** und Serialisierbarkeit könnte Teil einer Arbeit sein. Dazu gehört auch das Entwickeln von geeigneten Speerprotokollen.

Der Sprachumfang enthält keine komplexen Operationen, die aus einer Menge von atomaren Operationen zusammengesetzt wurden. Teil einer Arbeit könnte es sein den Sprachumfang um sinnvolle **komplexe Operationen** zu erweitern. XQuery erlaubt das Definieren von Funktionen und kann somit zum Erstellen der Operationen verwendet werden.

Die Struktur der Daten ist der Mittelpunkt der Softwareentwicklung. Wird die Struktur eines Schemas geändert, hat dies nicht nur Auswirkungen auf die Instanzen. Teil einer Arbeit sollte es sein die Auswirkung der Schemaevolution auf der **Anwendung**sebene zu untersuchen.

Anhang A

Notation der graphischen Symbole



Anhang B

Grammatik

Dieser Abschnitt enthält die ENBF[1] der Spracherweiterung. Die Grundlage der Grammatik stammt aus der XQuery Definition [BCF+05]. Alle mit einem '*' gekennzeichneten Regeln wurden erweitert und mit einem '+' sind neu hinzugekommen.

B.1 XQuery mit Erweiterung

```
[1]     Module                      ::= VersionDecl? (LibraryModule | MainModule)
[2]     VersionDecl                 ::= "xquery" "version" StringLiteral ("encoding" StringLiteral)? Separator
[3]     MainModule                  ::= Prolog QueryBody
[4]     LibraryModule               ::= ModuleDecl Prolog
[5]     ModuleDecl                  ::= "module" "namespace" NCName "=" URILiteral Separator
[6]     Prolog                      ::= ((DefaultNamespaceDecl | Setter | NamespaceDecl | Import) Separator)*
                                        ((VarDecl | FunctionDecl | OptionDecl) Separator)*
[7]     Setter                      ::= BoundarySpaceDecl | DefaultCollationDecl | BaseURIDecl | ConstructionDecl | OrderingModeDecl |
                                        EmptyOrderDecl | CopyNamespacesDecl
[8]     Import                      ::= SchemaImport | ModuleImport
[9]     Separator                   ::= ";"
[10]    NamespaceDecl               ::= "declare" "namespace" NCName "=" URILiteral
[11]    BoundarySpaceDecl           ::= "declare" "boundary-space" ("preserve" | "strip")
[12]    DefaultNamespaceDecl        ::= "declare" "default" ("element" | "function") "namespace" URILiteral
[13]    OptionDecl                  ::= "declare" "option" QName StringLiteral
[14]    OrderingModeDecl            ::= "declare" "ordering" ("ordered" | "unordered")
[15]    EmptyOrderDecl              ::= "declare" "default" "order" "empty" ("greatest" | "least")
[16]    CopyNamespacesDecl          ::= "declare" "copy-namespaces" PreserveMode "," InheritMode
[17]    PreserveMode                ::= "preserve" | "no-preserve"
[18]    InheritMode                 ::= "inherit" | "no-inherit"
[19]    DefaultCollationDecl        ::= "declare" "default" "collation" URILiteral
[20]    BaseURIDecl                 ::= "declare" "base-uri" URILiteral
[21]    SchemaImport                ::= "import" "schema" SchemaPrefix? URILiteral ("at" URILiteral ("," URILiteral)*)?
[22]    SchemaPrefix                ::= ("namespace" NCName "=") | ("default" "element" "namespace")
[23]    ModuleImport                ::= "import" "module" ("namespace" NCName "=")? URILiteral ("at" URILiteral ("," URILiteral)*)?
[24]    VarDecl                     ::= "declare" "variable" "$" QName TypeDeclaration? ((":=" ExprSingle) | "external")
[25]    ConstructionDecl            ::= "declare" "construction" ("strip" | "preserve")
[26]    FunctionDecl                ::= "declare" "function" QName "(" ParamList? ")" ("as" SequenceType)? (EnclosedExpr | "external")
[27]    ParamList                   ::= Param ("," Param)*
[28]    Param                       ::= "$" QName TypeDeclaration?
[29]    EnclosedExpr                ::= "{" Expr "}"
[30]    QueryBody                   ::= Expr
[31]    Expr                        ::= ExprSingle ("," ExprSingle)*
[32]*   ExprSingle                  ::= FLWORExpr | QuantifiedExpr | TypeswitchExpr | IfExpr | OrExpr |
                                        InsertExpr | DeleteExpr | SetExpr | RenameExpr | MoveExpr
[33]    FLWORExpr                   ::= (ForClause | LetClause)+ WhereClause? OrderByClause? "return" ExprSingle
[34]    ForClause                   ::= "for" "$" VarName TypeDeclaration? PositionalVar? "in" ExprSingle
                                        ("," "$" VarName TypeDeclaration? PositionalVar? "in" ExprSingle)*
[35]    PositionalVar               ::= "at" "$" VarName
```

[1] Erweiterte Backus-Naur-Form

```
[36]    LetClause               ::= "let" "$" VarName TypeDeclaration? ":=" ExprSingle
                                    ("," "$" VarName TypeDeclaration? ":=" ExprSingle)*
[37]    WhereClause             ::= "where" ExprSingle
[38]    OrderByClause           ::= (("order" "by") | ("stable" "order" "by")) OrderSpecList
[39]    OrderSpecList           ::= OrderSpec ("," OrderSpec)*
[40]    OrderSpec               ::= ExprSingle OrderModifier
[41]    OrderModifier           ::= ("ascending" | "descending")? ("empty" ("greatest" | "least"))? ("collation" URILiteral)?
[42]    QuantifiedExpr          ::= ("some" | "every") "$" VarName TypeDeclaration? "in" ExprSingle
                                    ("," "$" VarName TypeDeclaration? "in" ExprSingle)* "satisfies" ExprSingle
[43]    TypeswitchExpr          ::= "typeswitch" "(" Expr ")" CaseClause+ "default" ("$" VarName)? "return" ExprSingle
[44]    CaseClause              ::= "case" ("$" VarName "as")? SequenceType "return" ExprSingle
[45]    IfExpr                  ::= "if" "(" Expr ")" "then" ExprSingle "else" ExprSingle
[46]    OrExpr                  ::= AndExpr ( "or" AndExpr )*
[47]    AndExpr                 ::= ComparisonExpr ( "and" ComparisonExpr )*
[48]    ComparisonExpr          ::= RangeExpr ( (ValueComp | GeneralComp | NodeComp) RangeExpr )?
[49]    RangeExpr               ::= AdditiveExpr ( "to" AdditiveExpr )?
[50]    AdditiveExpr            ::= MultiplicativeExpr ( ("+" | "-") MultiplicativeExpr )*
[51]    MultiplicativeExpr      ::= UnionExpr ( ("*" | "div" | "idiv" | "mod") UnionExpr )*
[52]    UnionExpr               ::= IntersectExceptExpr ( ("union" | "|") IntersectExceptExpr )*
[53]    IntersectExceptExpr     ::= InstanceofExpr ( ("intersect" | "except") InstanceofExpr )*
[54]    InstanceofExpr          ::= TreatExpr ( "instance" "of" SequenceType )?
[55]    TreatExpr               ::= CastableExpr ( "treat" "as" SequenceType )?
[56]    CastableExpr            ::= CastExpr ( "castable" "as" SingleType )?
[57]    CastExpr                ::= UnaryExpr ( "cast" "as" SingleType )?
[58]    UnaryExpr               ::= ("-" | "+")* ValueExpr
[59]    ValueExpr               ::= ValidateExpr | PathExpr | ExtensionExpr
[60]    GeneralComp             ::= "=" | "!=" | "<" | "<=" | ">" | ">="
[61]    ValueComp               ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
[62]    NodeComp                ::= "is" | "<<" | ">>"
[63]    ValidateExpr            ::= "validate" ValidationMode? "{" Expr "}"
[64]    ValidationMode          ::= "lax" | "strict"
[65]    ExtensionExpr           ::= Pragma+ "{" Expr? "}"
[66]    Pragma                  ::= "(#" S? QName PragmaContents "#)"
[67]    PragmaContents          ::= (Char* - (Char* '#)' Char*))
[68]    PathExpr                ::= ("/" RelativePathExpr?) | ("//" RelativePathExpr) | RelativePathExpr
[69]    RelativePathExpr        ::= StepExpr (("/" | "//") StepExpr)*
[70]    StepExpr                ::= FilterExpr | AxisStep
[71]    AxisStep                ::= (ReverseStep | ForwardStep) PredicateList
[72]    ForwardStep             ::= (ForwardAxis NodeTest) | AbbrevForwardStep
[73]*   ForwardAxis             ::= ("child" "::") | ("descendant" "::") | ("attribute" "::") | ("self" "::") |
                                    ("descendant-or-self" "::") | ("following-sibling" "::") | ("following" "::") |
                                    ("typedefinition" "::") | ("constraint" "::")
[74]    AbbrevForwardStep       ::= "@"? NodeTest
[75]    ReverseStep             ::= (ReverseAxis NodeTest) | AbbrevReverseStep
[76]*   ReverseAxis             ::= ("parent" "::") | ("ancestor" "::") | ("preceding-sibling" "::") | ("preceding" "::") |
                                    ("ancestor-or-self" "::") | ("basetype" "::") | ("modelgroup" "::") | ("reference" "::")
[77]    AbbrevReverseStep       ::= ".."
[78]    NodeTest                ::= KindTest | NameTest
[79]    NameTest                ::= QName | Wildcard
[80]    Wildcard                ::= "*" | (NCName ":" "*") | ("*" ":" NCName)
[81]    FilterExpr              ::= PrimaryExpr PredicateList
[82]    PredicateList           ::= Predicate*
[83]    Predicate               ::= "[" Expr "]"
[84]    PrimaryExpr             ::= Literal | VarRef | ParenthesizedExpr | ContextItemExpr | FunctionCall | OrderedExpr |
                                    UnorderedExpr | Constructor
[85]    Literal                 ::= NumericLiteral | StringLiteral
[86]    NumericLiteral          ::= IntegerLiteral | DecimalLiteral | DoubleLiteral
[87]    VarRef                  ::= "$" VarName
[88]    VarName                 ::= QName
[89]    ParenthesizedExpr       ::= "(" Expr? ")"
[90]    ContextItemExpr         ::= "."
[91]    OrderedExpr             ::= "ordered" "{" Expr "}"
[92]    UnorderedExpr           ::= "unordered" "{" Expr "}"
[93]    FunctionCall            ::= QName "(" (ExprSingle ("," ExprSingle)*)? ")"
[94]    Constructor             ::= DirectConstructor | ComputedConstructor
[95]*   DirectConstructor       ::= DirElemConstructor | DirCommentConstructor | DirPIConstructor | DirElemDeclConstructor |
                                    DirAttrDeclConstructor | DirTypeConstructor | DirModelGroupConstructor |
                                    DirKeyConstructor | DirKeyRefConstructor | DirUniqueConstructor | DirAnyConstructor |
                                    DirAnyAttrConstructor | DirElemGroupConstructor | DirAttrGroupConstructor |
                                    DirNotationConstructor | DirAnnotationConstructor
[96]    DirElemConstructor      ::= "<" QName DirAttributeList ("/>" | (">" DirElemContent* "</" QName S? ">"))
[97]    DirAttributeList        ::= (S (QName S? "=" S? DirAttributeValue)?)*
[98]    DirAttributeValue       ::= ('"' (EscapeQuot | QuotAttrValueContent)* '"') | ("'" (EscapeApos | AposAttrValueContent)* "'")
[99]    QuotAttrValueContent    ::= QuotAttrContentChar | CommonContent
[100]   AposAttrValueContent    ::= AposAttrContentChar | CommonContent
[101]   DirElemContent          ::= DirectConstructor | CDataSection | CommonContent | ElementContentChar
[102]   CommonContent           ::= PredefinedEntityRef | CharRef | "{{" | "}}" | EnclosedExpr
[103]   DirCommentConstructor   ::= "<!--" DirCommentContents "-->"
[104]   DirCommentContents      ::= ((Char - '-') | ('-' (Char - '-')))*
[105]   DirPIConstructor        ::= "<?" PITarget (S DirPIContents)? "?>"
[106]   DirPIContents           ::= (Char* - (Char* '?>' Char*))
[107]   CDataSection            ::= "<![CDATA[" CDataSectionContents "]]>"
[108]   CDataSectionContents    ::= (Char* - (Char* ']]>' Char*))
```

```
[109]  ComputedConstructor             ::= CompDocConstructor | CompElemConstructor | CompAttrConstructor | CompTextConstructor |
                                            CompCommentConstructor | CompPIConstructor
[110]  CompDocConstructor              ::= "document" "{" Expr "}"
[111]  CompElemConstructor             ::= "element" (QName | ("{" Expr "}")) "{" ContentExpr? "}"
[112]  ContentExpr                     ::= Expr
[113]  CompAttrConstructor             ::= "attribute" (QName | ("{" Expr "}")) "{" Expr? "}"
[114]  CompTextConstructor             ::= "text" "{" Expr "}"
[115]  CompCommentConstructor          ::= "comment" "{" Expr "}"
[116]  CompPIConstructor               ::= "processing-instruction" (NCName | ("{" Expr "}")) "{" Expr? "}"
[117]  SingleType                      ::= AtomicType "?"?
[118]  TypeDeclaration                 ::= "as" SequenceType
[119]  SequenceType                    ::= ("empty-sequence" "(" ")") | (ItemType OccurrenceIndicator?)
[120]  OccurrenceIndicator             ::= "?" | "*" | "+"
[121]  ItemType                        ::= KindTest | ("item" "(" ")") | AtomicType
[122]  AtomicType                      ::= QName
[123] *KindTest                        ::= DocumentTest | ElementTest | AttributeTest | SchemaElementTest | SchemaAttributeTest |
                                            PITest | CommentTest | TextTest | SchemaDocumentTest | SchemaElementDeclarationTest |
                                            SchemaAttributeDeclarationTest | SchemaParticleTest | SchemaAttributeUseTest |
                                            SchemaTypeDefinitionTest | SchemaModelGroupTest | SchemaIdConstraintTest |
                                            SchemaWildcardTest | SchemaAttributeWildcardTest | SchemaElementGroupTest |
                                            SchemaAttributeGroupTest | SchemaNotationTest | SchemaAnnotationTest | AnyKindTest
[124]  AnyKindTest                     ::= "node" "(" ")"
[125]  DocumentTest                    ::= "document-node" "(" (ElementTest | SchemaElementTest)? ")"
[126]  TextTest                        ::= "text" "(" ")"
[127]  CommentTest                     ::= "comment" "(" ")"
[128]  PITest                          ::= "processing-instruction" "(" (NCName | StringLiteral)? ")"
[129]  AttributeTest                   ::= "attribute" "(" (AttribNameOrWildcard ("," TypeName)?)? ")"
[130]  AttribNameOrWildcard            ::= AttributeName | "*"
[131]  SchemaAttributeTest             ::= "schema-attribute" "(" AttributeDeclaration ")"
[132]  AttributeDeclaration            ::= AttributeName
[133]  ElementTest                     ::= "element" "(" (ElementNameOrWildcard ("," TypeName "?"?)?)? ")"
[134]  ElementNameOrWildcard           ::= ElementName | "*"
[135]  SchemaElementTest               ::= "schema-element" "(" ElementDeclaration ")"
[136]  ElementDeclaration              ::= ElementName
[137]  AttributeName                   ::= QName
[138]  ElementName                     ::= QName
[139]  TypeName                        ::= QName
[140]  URILiteral                      ::= StringLiteral
[141]+ SchemaDocumentTest              ::= "schema-node" "(" ")"
[142]+ SchemaElementDeclarationTest    ::= "elementdeclaration" "(" (ElementDeclaration)? ")"
[143]+ SchemaAttributeDeclarationTest  ::= "attributedeclaration" "(" (AttributeDeclaration)? ")"
[144]+ SchemaTypeDefinitionTest        ::= "typedefinition" "(" (TypeDefinition)? ")"
[145]+ SchemaElementGroupTest          ::= "elementgroup" "(" (ElementGroup)? ")"
[146]+ SchemaAttributeGroupTest        ::= "attributegroup" "(" (AttributeGroup)? ")"
[147]+ SchemaIdConstraintTest          ::= "idconstraint" "(" (IDConstraint)? ")"
[148]+ SchemaModelgroupTest            ::= "modelgroup" "(" ")"
[149]+ SchemaWildcardTest              ::= "wildcard" "(" ")"
[150]+ SchemaAttributeWildcardTest     ::= "attributewildcard" "(" ")"
[151]+ SchemaNotationTest              ::= "notation" "(" ")"
[152]+ SchemaAnnotationTest            ::= "annotation" "(" ")"
[153]+ InsertExpr                      ::= "insert" SourceExpr (("as first" | "as last")? "into") | "after" | "before") TargetExpr
[154]+ DeleteExpr                      ::=  "delete" TargetExpr
[155]+ RenameExpr                      ::= "rename" TargetExpr "as" NewNameExpr
[156]+ MoveExpr                        ::= "move" SourceExpr  (("as first" | "as last")? "into") | "after" | "before") TargetExpr
[157]+ SetExpr                         ::= "set" PropertyExpr "of" TargetExpr
[158]+ SourceExpr                      ::= ExprSingle
[159]+ TargetExpr                      ::= ExprSingle
[160]+ NewNameExpr                     ::= QName
[161]+ PropertyExpr                    ::= ("use" "=" "optional"|"required"|"prohibited") | ("default" "=" Literal) | ("fixed" "=" Literal) |
                                            ("declaration" "=" QName) | ("scope" "=" "true"|"false") | ("type" "=" QName) |
                                            ("minoccurs" "=" NumericLiteral) | ("maxoccurs" "=" NumericLiteral) |
                                            ("nillable" "=" "true"|"false") | ("abstract" "=" "true"|"false") | ("selector" "=" PathExpr) |
                                            ("field" "=" PathExpr) | ("reference" "=" QName) | ("mixed" "=" "true"|"false") |
                                            ("length" "=" nonNegativeInteger) | ("minLength" "=" nonNegativeInteger) |
                                            ("maxLength" "=" nonNegativeInteger) | ("enumeration" "=" Expr |
                                            ("whitespace" "=" "preserve" | "replace" | "collapse")| ("pattern" "=" RegularExpression) |
                                            ("minExclusive" "=" nonNegativeInteger) | ("maxExclusive" "=" nonNegativeInteger) |
                                            ("minInclusive" "=" nonNegativeInteger) | ("maxInclusive" "=" nonNegativeInteger) |
                                            ("totalDigits" "=" positiveInteger) | ("fractionDigits" "=" nonNegativeInteger)

[162]+ DirElemDeclConstructor          ::= "element" ’"’NCName’"’ ("of type" ’"’QName’"’)?
[163]+ DirAttrDeclConstructor          :;= "attribute" ’"’NCName’"’ ("of type" ’"’QName’"’)?
[164]+ DirTypeConstructor              ::= "typedefinition" (’"’NCName’"’)?
[165]+ DirModelGroupConstructor        ::= ("sequence"|"choice"|"all") ("{" (ElementDecalaration|Wildcard|ModelGroup)+"}")?
[166]+ DirKeyConstructor               ::= "key" NCName "selector=" XPath ("field=" XPath)+
[167]+ DirKeyRefConstructor            ::= "keyref" NCName "refer=" QName "selector=" $XPath ("field=" $XPath)+
[168]+ DirUniqueConstructor            ::= "unique" NCName "selector=" XPath ("field=" XPath)+
[169]+ DirAnyConstructor               ::= "any" ("processContents=" ("lax"|"skip"|"strict"))? ("namespace=")?
                                            ("maxOccurs=" nonNegativeInteger)? ("minOccurs=" nonNegativeInteger)?
[170]+ DirAnyAttrConstructor           ::= "anyAttribute" ("processContents=" ("lax"|"skip"|"strict"))? ("namespace=")?
[171]+ DirElemGroupConstructor         ::= "elementgroup" NCName ("{" (ElementDecalaration|Wildcard|Elementgroup)+"}")?
[172]+ DirAttrGroupConstructor         ::= "attributegroup" NCName ("{" (AttributDeclaration|AttributWildcard)+"}")?
[173]+ DirNotationConstructor          ::= "notation" NCName "public=" NCName "system=" anyURI}
```

```
[174]+ DirAnnotationConstructor       ::= ("appinfo"|"documentation") ("source=" anyURI)? ("language=" language)? ("content=" any}?
```

Verzeichnis der Abkürzungen

DOM	**D**ocument **O**bject **M**odel
DTD	**D**ocument **T**ype **D**efinition
EBNF	**E**rweiterte **B**ackus-**N**aur-**F**orm
FLWOR	**F**or **L**et **W**here **O**rder by **R**eturn
GUI	**G**raphical **U**ser **I**nterface
HTML	**H**yper**t**ext **M**arkup **L**anguage
IDE	**I**ntegrated **D**evelopment **E**nviroment
IDL	**I**nterface **D**efinition **L**anguage
ISO	**I**nternational **O**rganization for **S**tandardization
JDK	**J**ava **D**evelopment **K**it
ODMG	**O**bject **D**ata **M**anagement **G**roup
OMG	**O**bject **M**anagment **G**roup
OQL	**O**bject **Q**uery **L**anguage
PI	**P**rocessing **I**nstruction
PSVI	**P**ost **S**chema **V**alidation **I**nterface
RELAX	**Re**gular **La**nguage Description for **X**ML
SAX	**S**imple **A**PI for **X**ML
SGML	**S**tandard **G**eneralized **M**arkup **L**anguage
SQL	**S**tructured **Q**uery for **L**anguage
UML	**U**nified **M**odelling **L**anguage
W3C	**W**orld **W**ide **W**eb **C**onsortium
XDM	**X**Query 1.0 and XPath 2.0 **D**ata **M**odel
XEM	**X**ML **E**volution **M**anagement
XML	e**X**tensible **M**arkup **L**anguage
XPath	**X**ML **Path** Language
XQuery	**X**ML **Query** Language
XSD	**X**ML **S**chema **D**efinition
XSL	e**X**tensible **S**tylesheet **L**anguage
XSL-FO	e**X**tensible **S**tylesheet **L**anguage **F**ormatting **O**bjects
XSLT	e**X**tensible **S**tylesheet **L**anguage **T**ransformation

Abbildungsverzeichnis

2.1 Ein Beispiel für ein XML Dokument 6
2.2 Darstellung der Primären Komponenten 8
2.3 Darstellung der Hilfskomponenten 9
2.4 Überblick der Datentypen 11

3.1 Abbildung der Architekturen 18
3.2 Zustandsänderungen, die keinen Einfluss auf die Instanzen haben 25
3.3 Zustandsänderungen, die Einfluss auf die Instanzen haben . . 25
3.4 Übergänge der verschiedenen Elementgruppen 29

4.1 Überblick der XML-Konzepte 44
4.2 Bedeutung der Achsen ausgehend von einem Element-Knoten 48
4.3 Bedeutung der Achsen ausgehend von einem Typdefinitions-Knoten 50
4.4 Bedeutung der Achsen ausgehend von einem Elementgruppen-Knoten 51

5.1 Überblick der Implementation 81
5.2 XML-Schema vor der Schemaevolution 83
5.3 XML-Instanz vor der Schemaevolution 83
5.4 XML-Schema nach der Schemaevolution 85
5.5 XML-Instanz nach der Schemaevolution 86

Literaturverzeichnis

[BBC+05] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. *XML Path Language (XPATH) 2.0.* W3C, http://www.w3.org/TR/2005/CR-xpath20-20051103, 2005.

[BBFV05] Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. *Adding Updates to XQuery: Semantics, Optimization, and Static Analysis.* Second InternationalWorkshop on XQuery Implementation, Experience and Perspectives, 2005.

[BCF+05] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language.* W3C, http://www.w3.org/TR/2005/CR-xquery-20051103, 2005.

[BDH+04] Beatrice Bouchou, Denio Duarte, Mirian Halfeld, Ferrari Alves, Dominique Laurent, and Martin A. Musicante. *Schema Evolution for XML: A Consistency-Preserving Approach.* Mathematical Foundations of Computer Science, 2004.

[Ber03] Philip A. Bernstein. Generic model management: A database infrastructure for schema manipulation. In *BDA*, 2003.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 311–322, New York, NY, USA, 1987. ACM Press.

[BL01] Angela Bonifati and Dongwong Lee. *Technical Survey of XML Schema and Query Languages.* Technical Report, UCLA Computer Science Dept., citeseer.ist.psu.edu/bonifati01technical.html, 2001.

[BM04] Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes Second Edition.* W3C, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028, 2004.

[BOSdL05] Kevin Beyer, Fatma Oezcan, Sundar Saiprasad, and Bert Van der Linden. *DB2/XML: Designing for Evolution.* IBM, 2005.

[BPSM+04] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0 (Third Edition).* W3C, http://www.w3.org/TR/2004/REC-xml-20040204, 2004.

[CD99] James Clark and Steve DeRose. *XML Path Language (XPATH) Version 1.0.* W3C, http://www.w3.org/TR/1999/REC-xpath-19991116, 1999.

[CDF+99] S. Ceri, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. *XML-GL: a Graphical Language for Querying and Restructuring XML Documents.* Università di Milano, http://www8.org/w8-papers/1c-xml/xml-gl/xml-gl.html, 1999.

[CFR06] Don Chamberlin, Daniela Florescu, and Jonathan Robie. *XQuery Update Facility.* W3C, http://www.w3.org/TR/2006/WD-xqupdate-20060508, 2006.

[CJR98] Kajal T. Claypool, Jing Jin, and Elke A. Rundensteiner. SERF: Schema evolution through an extensible re-usable and flexible framework. pages 314–321, 1998.

[Cla99] James Clark. *XSL Transformations (XSLT).* W3C, http://www.w3.org/TR/1999/REC-xslt-19991116, 1999.

[CM01] James Clark and Murata Makoto. *RELAX NG Specification.* OASIS Open, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html, 2001.

[CR05] Don Chamberlin and Jonathan Robie. *XQuery Update Facility Requirements (Working Draft).* W3C, http://www.w3.org/TR/2005/WD-xquery-update-requirements-20050603, 2005.

[CRH00] Kajal T. Claypool, Elke A. Rundensteiner, and George T. Heineman. Evolving the software of a schema evolution system. In *FMLDO*, pages 68–84, 2000.

[CT04] John Cowan and Richard Tobin. *XML Information Set (Second Edition).* W3C, http://www.w3.org/TR/xml-infoset, 2004.

[DFF+05] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics.* W3C, http://www.w3.org/TR/2005/CR-xquery-semantics-20051103, 2005.

[FW04] David C. Fallside and Priscilla Walmsley. *XML Schema Part 0: Primer Second Edition.* W3C, http://www.w3.org/TR/2004/REC-xmlschema-0-20041028, 2004.

[GMR05] Giovanna Guerrini, Marco Mesiti, and Daniele Rossi. *Impact of XML Schema Evolution on Valid Documents.* WIDM'05, 2005.

[Gol96] Charles F. Goldfarb. *The Roots of SGML – A Personal Recollection.* http://www.sgmlsource.com/history/roots.htm, 1996.

[Heu97] Andreas Heuer. *Objektorientierte Datenbanken.* Addison-Wesley, 1997.

[HHW+04] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. *Document Object Model (DOM) Level 3 Core Specification.* W3C, http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407, 2004.

[HS00] Andreas Heuer and Gunter Saake. *Datenbanken: Konzepte und Sprachen.* mitp, 2000.

[Hän02] Birger Hänsel. *Änderungsoperationen in XML-Anfragesprachen.* Universität Rostock, Diplomarbeit, 2002.

[Kep04] Stephan Kepser. *A Simple Proof for the Turing-Completeness of XSLT and XQuery.* Extreme Markup Languages 2004, 2004.

[KM03] Meike Klettke and Holger Meyer. *XML & Datenbanken.* dpunkt, 2003.

[KMH] Meike Klettke, Holger Meyer, and Birger Hänsel. *Schemaevolution und Adaption von XML-Dokumenten und XQuery-Anfragen.* Database Research Group University of Rostock.

[KMH05] Meike Klettke, Holger Meyer, and Birger Hänsel. *Evolution - The Reverse Side of the XML Update Coin*. Database Research Group University of Rostock, 2005.

[KMS02] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. The DSD schema language. *Automated Software Engineering*, 9(3):285–319, 2002. Kluwer. Earlier version in Proc. 3rd ACM SIGPLAN-SIGSOFT Workshop on Formal Methods in Software Practice, FMSP '00.

[Leh01] Patrick Lehti. *Design and Implementation of a Data Manipulation Processor for an XML Query Language*. Technische Universität Darmstadt, Diplomarbeit, 2001.

[Lit04] Elena Litani. *XML Schema API*. W3C, http://www.w3.org/Submission/2004/SUBM-xmlschema-api-20040309, 2004.

[Mar04] Maarten Marx. Conditional xpath, the first order complete xpath dialect, 2004.

[MM05] Jim Melton and Subramanian Muralidhar. *XML Syntax for XQuery 1.0 (XQueryX)*. W3C, http://www.w3.org/TR/2005/CR-xqueryx-20051103, 2005.

[MMW05] Ashok Malhotra, Jim Melton, and Norman Walsh. *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C, http://www.w3.org/TR/2005/CR-xpath-functions-20051103, 2005.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. *XML Query Language (XQL)*. W3C, http://www.w3.org/TandS/QL/QL98/pp/xql.html, 1998.

[SCH04] *Final Committee Draft of ISO Schematron*. www.schematron.com, http://www.schematron.com/iso/dsdl-3-fdis.pdf, 2004.

[SH99] Gunter Saake and Andreas Heuer. *Datenbanken: Implementierungstechniken*. mitp, 1999.

[SKR01] Hong Su, Harumi A. Kuno, and Elke A. Rundensteiner. Automating the transformation of xml documents. In *WIDM*, pages 68–75, 2001.

[SKR02] Hong Su, Diane K. Kramer, and Elke A. Rundensteiner. *XEM: XML Evolution Management*. Worcester Polytechnic Institute, 2002.

[TBMM04] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures Second Edition*. W3C, http://www.w3.org/TR/2004/REC-xmlschema-1-20041028, 2004.

[Tie05] Tobias Tiedt. *Schemaevolution und Adaption von XML-Dokumenten und XQuery-Anfragen*. Universität Rostock, Diplomarbeit, 2005.

[TIHW01] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *SIGMOD Conference*, citeseer.ist.psu.edu/tatarinov01updating.html, 2001.

[VMP03] Yannis Velegrakis, Renée J. Miller, and Lucian Popa. Mapping adaptation under evolving schemas. In *VLDB*, pages 584–595, 2003.

[Wil06] Christian Will. *Ableitung von Schemaevolutionsschritten aus XML-Updateoperatioen*. Universität Rostock, Studienarbeit, 2006.

[Woo98] Lauren Wood. *Document Object Model (DOM) Level 1 Specification*. W3C, http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001, 1998.

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

Thesen

1. XML ist eine der Kerntechnologien des Informationszeitalters.
2. Anwendungen werden zunehmend semistrukturierte Daten verarbeiten.
3. Ein Schema unterliegt einem ständigem Evolutionsprozess.
4. XML-Schema wird DTD als Schemasprache vollkommen ersetzen. Es wird jedoch auch weiterhin mehr als eine XML Schema Sprache geben.
5. Strukturänderungen sollten auf Basis des Datenmodells der Schemasprache beschrieben werden.
6. Die Anfrage- und Manipulationssprache XQuery ist turing-vollständig und damit zu komplex für Datenbanken und Datenverarbeitende Systeme. Eine eingeschränkte Version muss definiert werden, um die Sicherheit der Anfragen zu garantieren.
7. Die Schemasprache XML-Schema ist sehr komplex und bietet viele Möglichkeiten dieselbe Struktur zu definieren. Eine vereinfachte Variante, die auch auf redundante Konzepte wie den ID/IDREF-Mechanismus verzichtet, würde die Akzeptanz der Sprache vergrößern.