Normalform für XML-Schema

Studienarbeit

Tobias Tiedt

Lehrstuhl Datenbank
und Informationssysteme
Fachbereich Informatik
Universität Rostock

Albert-Einstein-Str. 21
D-18059 Rostock

Zusammenfassung

XML wird häufig als Dokumentenaustauschformat benutzt oder es findet Einsatz im Datenbankbereich oder in föderierten Informationssystemen. Dabei ist es aber wichtig, die Korrektheit von XML-Dokumenten garantieren zu können oder Vergleichbarkeit auf den Dokumenten zu realisieren. Ein Mechanismus zur Definition, Beschreibung und Validierung von XML-Dokumentklassen ist XML-Schema.
Diese Arbeit beschreibt eine Normalform für XML-Schema, mit Hilfe dessen zum Beispiel leichtere Vergleichbarkeit realisiert werden kann. Genauer geht es um die Frage, wie die XML-Schema-Konstrukte aus beliebigen Schemata in die Normalform überführt werden können und welche Auswirkungen diese Transformationen zum Beispiel auf die Komplexität haben.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In föderierten Informationssystemen oder allgemein dort, wo mehrere XML-Dokumentkollektionen auftreten, ist es sinnvoll, diese vergleichen zu können. Dabei ist eine XML-Dokumentkollektion oder eine XML-Klasse eine Menge von einzelnen XML-Dokumenten, welche alle durch eine DTD oder ein Schema definiert und beschrieben werden. Hierbei ist XML-Schema als Definition aber um ein Wesentliches komplexer als die DTD und mehrere verschiedene Schemata können ein und dieselbe Klasse von XML-Dokumenten beschreiben. Um aber nun eine leichtere Vergleichbarkeit oder mögliche inhaltserhaltende Umformungen und Vereinfachungen von solchen XML-Klassen oder von Schemata zu haben, ist es sinnvoll, diese in eine eindeutige Normalform zu überführen.
Es ist auch denkbar, dass durch eine solche Normalform ein besserer automatischer Datenbankentwurf erfolgen kann. Darüber hinaus kann XML-Schema auch zur Klassendefinition in objektorientierten Systemen eingesetzt werden, da mittels eines Schemas leicht Klassendefinitionen oder Typhierarchien aufgebaut werden können.

1.2 Normalform

Der Begriff einer Normalform ist schwer einheitlich zu definieren, da in den verscheidenen Bereichen der Informatik unterschiedliche Anforderungen existieren.
Eine Normalform kann man als ein gewisses Regelwerk auffassen. Eine Menge von Bedingungen und Regeln, welche gelten müssen, damit die Normalform gegeben ist. Im Datenbankbereich, speziell bei relationalen Datenbanken, sind die dort vorkommenden Normalformen eben eine solche Zusammenfassung von Bedingungen, die gelten müssen. So ist die *erste Normalform 1NF* nur dann gegeben, wenn die Attributwerte der Relationen atomar sind. Die *zweite Normalform 2NF* nur dann, wenn keine Abhängigkeiten unter Schlüsselteilen existieren. Die *dritte Normalform 3NF* ist nur dann gegeben, wenn zusätzlich zur Bedingung der 2NF keine transitiven Abhängigkeiten auftreten [STS97].
Es existieren darüber hinaus noch weitere Normalformen wie die *Boyce-Codd-Normalform* und die *4NF*, welche mehrwertige Abhängigkeiten behandelt, mehr darüber in [HS00] und [Heu97]. Alle diese Normalformen erfüllen die Vorstellung, dass eine Normalform eine Menge von Bedingungen ist.
Eine Normalform kann auch nur eine Definition sein, eine Festlegung, welche eindeutig ist und darüber hinaus den selben Sachverhalt ausdrückt und eine einfachere Darstellung bietet. Als Beispiel kann man in der Mathematik Geraden in der Ebene auf verschiedene Weisen darstellen, eine Möglichkeit ist die *Hessische Normalform*, welche lediglich eine Festlegung ist und den selben Sachverhalt ausdrückt.
Die in dieser Arbeit vorgestellte Normalform ist nun eine Menge an Bedingungen und Regeln, welche die selben Instanzdokumente definieren und ein eindeutiges Schema für eine XML-Klasse definieren.

1.3 Aufbau der Arbeit

Ziel dieser Arbeit ist die Definition einer Normalform für XML-Schema und die Entwicklung einer grafischen Notation.

In **Kapitel 2** erfolgt eine Einführung in XML-Schema und eine detaillierte Vorstellung der Konstrukte zur Definition von XML-Dokumenten. Diese werden an einem Beispiel erläutert und gezeigt, wie eine mögliche XML-Instanz aussieht.
Weiterhin werden in diesem Kapitel Designprinzipien erklärt und warum sich diese gebildet haben.
In **Kapitel 3** wird dann die Normalform eingeführt und definiert. Ebenso werden Metriken für DTDs für XML-Schema adaptiert, mit Hilfe dessen man Betrachtungen über Güte und Komplexität durchführen kann. In Kapitel 3 wird ebenfalls die graphische Notation eingeführt und es werden Transformationschritte erläutert, mit Hilfe dessen man ein beliebiges Schema in die Normalform überführen kann.
Anwendungen dieser Normalform werden **Kapitel 4** gezeigt und erläutert. Hauptaugenmerk der Normalform dieser Arbeit ist dabei die Vergleichbarkeit von Schemata.
Anschließend wird in **Kapitel 5** eine Zusammenfassung gegeben und dieses Kapitel zeigt einen Ausblick auf Themen aufbauend auf dieser Arbeit oder Themen, welche diese Arbeit weiterführen könnten.

2 XML-Schema

2.1 Einführung

XML 1.0 wurde mit dem Konzept der Wohlgeformtheit und der Validität[1] eingeführt. Die Wohlgeformtheit ist leicht zu prüfen, jedoch bedarf es zur Validierung eines weiteren Mechanismus. Mit Hilfe der DTD (*Document Type Definition*) ist es möglich, XML-Dokumente auf ihre Korrektheit zu validieren. Jedoch sind viele Anforderungen aufgrund der geringen Komplexität und des geringen Umfangs von der DTD nicht erfüllbar.
Aus diesen Gründen wurde XML-Schema am 2. Mai 2001 vom W3C (*WorldWideWeb-Konsortium*) als Recommodation verabschiedet ([Hol00] und [BSL01]). Im XML-Schema-Vorschlag des W3C wurden viele Anforderungen umgesetzt. Es sollten dieselben Möglichkeiten wie bei der DTD realisierbar sein, z.B. die Definition der Reihenfolge und die Festlegung der Anzahl des Auftretens von Elementen und Attributen. Darüber hinaus ist die Integration von XML-Namensräumen, die die DTD nicht beherrscht, ein Typkonzept, welches bei der DTD nur rudimentär vorhanden ist, ein Vererbungsmechanismus und Ableitungsmechanismus gefordert[W3C01a]. Ebenfalls gefordert ist eine gewisse Optimierung der Interoperabilität, dazu gehören Mechanismen um einzelne Schema in andere einzufügen. Dies geschieht durch die include- und import-Anweisungen, welche in 2.2.1 näher beschrieben werden. XML-Schema sollte desweiteren leicht in vorhandene Applikationen integrierbar sein, welche schon XML verarbeiten können, es sollte selbstbeschreibend sein, leicht für das Internet nutzbar oder einsetzbar sein und die XML-Syntax benutzen [Gul01]. So definiert die XML-Schema-Spezifikation ein gewisses XML-Vokabular, welches dazu dient, XML-Dokumente zu definieren oder zu beschreiben. Durch den Gebrauch der XML-Syntax ist ein Schema nun leicht in Applikationen integrierbar, welche schon mit XML umgehen können, im Gegensatz dazu die DTD, welche eine eigene Syntax besitzt. Jedoch hat XML-Schema dadurch auch Nachteile, da es wegen der Syntax schnell lang und unübersichtlich werden kann und bei weitem nicht so kompakt ist wie eine DTD.
Insgesamt gehen die XML-Schema-Spezifikationen davon aus, dass immer zwei XML-Dokumente benutzt werden. Zum Ersten das Instanz-Dokument und zum Zweiten ein Schema-Dokument, gegen jenes das Instanz-Dokument validiert werden kann. Eine XML-Instanz ist dabei ein Element, ein konkretes Dokument, aus der Menge von XML-Dokumenten (*XML-Klasse* oder *XML-Dokumentkollektion*), welche alle durch ein Schema beschrieben werden. Dabei ist der Unterschied zwischen Schema und Instanz ähnlich dem Zusammenhang zwischen Klasse und Objekt aus dem objektorientierten Bereich [BSL01]. Der Grundgedanke bei XML-Schema beruht auch hauptsächlich auf Typen und nicht auf Tag-Namen. Durch diesen Fakt und durch das erweiterte Typkonzept (siehe 2.2.2) lässt sich XML-Schema auch besser in Programmiersprachen und Datenbankumgebungen integrieren.

2.2 Vorschlag des W3C

Der XML-Schema Vorschlag (*XML-Schema Recommodation*) umfaßt drei Teile, wobei der erste Teil (*Primer*) eine Kurzeinführung darstellt. Die anderen beiden Teile (*Structure* und *Datatype*) beinhalten eine umfassende Beschreibung von XML-Schema.
Dabei enthält der zweite Teil eine genaue Beschreibung der Strukturen, welche in XML-Schema vorkommen. Diese Strukturen garantieren dieselbe Funktionalität wie mit der DTD umsetzbar ist. Zusätzlich sind die Beschreibungen von den Vererbungsmechanismen und Ableitungsmechanismen

[1] genauer unter [W3C00]

enthalten und die Möglichkeiten der Aufteilung eines Schemas auf mehrere Dokumente und dessen Zusammenführung werden beschrieben. Ebenso sind Namensräume für Schemata beschrieben, welche als eine Anforderung in den Schema-Vorschlag hineingenommen worden sind.
Der dritte Teil beschreibt die Datentypen, welche in XML-Schema integriert sind. Ebenso werden Methoden zur Definition eigener Datentypen beschrieben. Das Typkonzept hinter XML-Schema ist um ein Vielfaches komplexer als jenes in der Document Type Definition und findet starke Anlehnung an schon vorhandene Typsysteme wie in Datenbanksystemen und in Programmiersprachen.
Diese beiden Teile werden nun im einzelnen beschrieben und an einem Beispiel erläutert. Genauer kann man es auf den Seiten des W3C ([W3C01a]) und in [Gul01] nachlesen.

2.2.1 Strukturen

Der Strukturteil verfolgt mehrere Ziele, zum Einen einen Mechanismus zur "Einschränkung" der Dokumentstruktur und des Dokumentinhaltes. Jenes bedeutet, dass man im Schema Richtlinien definiert, welche von den einzelnen XML-Instanzen einzuhalten sind, damit diese valide sind.
Die genauen Ziele sind

- Mechanismus zur Einschränkung der Dokumentstruktur
  - Namensräume (*Namespaces*)
  - Elemente
  - Attribute

  und des Dokumentinhaltes
  - Datentypen
  - Entities
  - Vermerke
- Mechanismen zur Vererbung und Wiederverwendung von Typdefinitionen und Elementdeklarationen
- Möglichkeit zur eingebetteten Dokumentation
- Mechanismus zur Nutzung primitiver Datentypen
- Mechanismus zur Nutzung von Verarbeitungsanweisungen (*Processing-Instructions*)

2.2.1.1 Namensräume

Namensräume sind der Definition des W3C nach Mengen von Namen, die durch eine URI-Referenz[2] identifiziert werden. Diese Bezeichner werden dazu verwendet, Elemente, Typen und Attribute in Informationsmengen einzuteilen.
Jenes bedeutet, dass Namensräume ein Mechanismus sind, um zu bestimmen, welche XML-Tags zu welcher Informationsmenge gehören. Sie bilden darüber hinaus eine Möglichkeit, eindeutige Identifizierer zu definieren, da z.B. beim Zusammenfügen von mehreren Schemata zu einem großen Schema, die Eindeutigkeit der Bezeichner garantiert bleiben muss. Um eine Eindeutigkeit zu erreichen, greift man auf URI's (*Uniform Resource Identifier*) zurück. Die meisten angegebenen Namensraumbezeichner suggerieren jedoch, dass sich hinter der Bezeichnung eine URL befindet. Jedoch ist zu beachten, dass es hier nur um die Eindeutigkeit geht, so sind auch andere Namensraumbezeichner denkbar.
Durch diese Bezeichnung der Namensräume werden auch die Informationsmengen eindeutig getrennt. Bei XML-Schemata werden im Schema-Wurzelelement `schema` die Namensräume angegeben, aus denen Definitionen und Deklarationen verwendet werden. Zur weiteren Unterscheidung

[2] URI's sind in der RFC-Seite 2396 definiert, siehe http://www.ietf.org/rfc/rfc2396.txt

werden noch Kurzformen festgelegt, welche dann vor die verwendeten Tags mit Doppelpunkt abgetrennt gesetzt werden. Die Deklaration der verwendeten Namensräume realisiert man über das `xmlns`-Attribut, dabei steht xmlns fuer XML-NameSpace. Als Attribut-Werte werden die URI's angegeben, das Kürzel legt man fest, indem man es durch einen Doppelpunkt abgetrennt, hinter `xmlns` notiert.
Desweiteren beschreibt ein Schema einen Zielnamensraum, in dem die Informationen des Schemas, im Einzelnen die Element- und Attributdeklarationen, zusammengefasst sind. In Instanzdokumenten muss dann dieser Zielnamensraum erneut angegeben werden, damit Parser wissen, aus welchem Namensraum die Informationen stammen. Bei selbstdefinierten Namensräumen muss zusätzlich noch der Ort angegeben werden, wo das Schema zu finden ist, damit dagegen validiert werden kann. Dieses geschieht mit dem Attribut `schemaLocation`, welches sich im Namensraum `http://www.w3.org/2001/XMLSchema-instance` befindet, wobei dieser Namensraum der zum Namensraum `http://www.w3.org/2001/XMLSchema` entsprechende Namensraum für Instanzdokumente ist. Wenn dann ein Parser die Definitionen des Namensraumes im Instanzdokument findet, diese übereinstimmen (Angabe des Namensraumes und Zuordnung des Ortes zu diesem Namensraum) und hinter dem angegebenen Ort ein Schema gefunden werden kann, dann validiert der Parser das Instanzdokument gegen das gefundene Schema.
Abbildung 2.1 zeigt die Verwendung der Attribute um Namensräume anzugeben und Abbildung 2.2 zeigt die Verwendung in einem Instanzdokument.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="urn:myNameSpace">
    ...
</xs:schema>
```

Abbildung 2.1: Namensraumfestlegung

```
<person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="urn:myNameSpace"
        xsi:schemaLocation="urn:myNameSpace
            http://localhost/NameSpaces/myNameSpace.xsd"
/>
```

Abbildung 2.2: Namensraumverwendung in Instanzen

Genaueres kann man auf den Internetseiten des W3C ([W3C99b]) nachlesen. Um nun weitere Namensräume nutzen zu können, muss dieses über die `import`-Anweisung geschehen (siehe 2.2.1.10).

2.2.1.2 Elementdeklaration

Elemente sind ein fundamentaler Bestandteil von XML. Jedes XML-Dokument beginnt mit einem *Wurzel*-Element, welches das *Eltern*-Element oder *Parent*-Element für das gesamte Dokument ist. Dieses Wurzel-Element oder auch root-Element kann weiterere Subelemente besitzen, diese wiederum ebenfalls weitere Subelemente usw. Aus diesem Grund kann man XML-Dokumente als Baum darstellen, wobei die Elemente des Dokumentes die Knoten des Baumes sind.
Die Deklaration eines Elementes im Schema geschieht mit dem Schlüsselwort `element` und das Inhaltsmodell und den Typ des Elementes bestimmt man durch eine Reihe von Attributen.

```
<xs:element name="Vorname" maxOccurs="3"
    type="xs:string"/>
```

Abbildung 2.3: Elementdeklaration

Das Beispiel 2.3 deklariert ein Element mit dem Namen `Vornamen` und hat den Typ `string`, es ebenfalls wird festgelegt, dass das Element maximal dreimal auftreten darf.

Die möglichen Inhaltsmodelle von Elementen sind

- leerer Inhalt
- strukturierter Inhalt (nur Kindknoten)
- unstrukturierter Inhalt (textueller Inhalt)
- gemischter Inhalt (`mixedContent`) (Kindknoten und textueller Inhalt)

Nimmt man das Beispiel als XML-Konstrukt, also ein Tag mit Namen `xs:element`, so besitzt es als Inhaltsmodell den leeren Inhalt, da das Beispielelement keine weiteren Subelemente oder Kindknoten besitzt und auch keinen textuellen Inhalt hat. Im Gegensatz dazu besitzt das Tag in einer möglichen XML-Instanz den unstrukturierten Inhalt als Inhaltsmodell, da es vom Typ `string` ist. Eine mögliche Instanz könnte so aussehen

```
<Vorname>Kai</Vorname>
```

XML-Schema unterstützt dabei dieselben Inhaltsmodelle wie bei der DTD möglich sind, zusätzlich hat man bei XML-Schema noch `ANY` und `ALL`.
Wie genau welche Typen eingesetzt werden können, um das Inhaltsmodell des Elementes mitzubestimmen, wird im Abschnitt über nutzerdefinierte Datentypen in 2.2.2.2 genauer erläutert.
Die durch XML-Schema definierte Charakteristika erreicht man durch eine Anzahl möglicher Attribute des element-Tags:

- **abstract**: dieses Attribut kann die Werte *true* oder *false* annehmen, wobei true bedeutet, dass das Element ausschließlich der Schemastrukturierung dient und in Instanzen nicht vorkommen darf.
- **block**: mit diesem Attribut wird festgelegt, inwiefern das Element weiterverwendet werden kann. Dabei sind mögliche Attribute
  - `restriction:` es dürfen keine einschränkend abgeleiteten Typen anstelle des Originaltyps verwendet werden
  - `extension:` es dürfen keine erweiterten Typen verwendet werden[3]
  - `substitution:` der Typ darf nicht in einer Substitutionsgruppe enthalten sein (siehe Abschnitt Substitutionsgruppen (2.2.1.6))
  - `#all:` generelles Verbot einer Weiterverwendung
- **default**: simpleType's können mit einem Defaultwert belegt werden (siehe Abschnitt über simpleType (2.2.2.3))
- **final**: die Nutzung des Elementes in einer Ableitung wird verboten. Dabei verbietet final im Gegensatz zu block die Ableitung schon auf Schemaebene
- **fixed**: ermöglicht eine konstante Belegung des Elementes mit einem Wert
- **form**: form gibt an, ob im Instanzdokument Namensraumpräfixe erscheinen müssen oder nicht. Es existieren die beiden Belegungen *qualified* und *unqualified*, wobei nach der Angabe von qualified das Präfix angegeben werden muss.
- **id**: eindeutige Kennzeichnung des Elementes

[3] Die Erklärung, was einschränkende und erweiternde Ableitung ist, findet man im Abschnitt über Vererbung (2.2.1.8)

- **minOccurs/maxOccurs**:wie oft kann ein Element auftreten. Es wird die minimale und maximale Anzahl des Elementes festgelegt.
- **name**: Namensgabe, wobei der Name vom Typ NCName[4] ist.
- **nilable**: es erlaubt, dass das Element in der Instanz mit einem Null-Wert belegt ist. In diesem Falle ist der Wert *true* ansonsten *false*
- **ref**: es wird auf eine andere Elementdeklaration verwiesen, referenziert, welche global deklariert ist, mehr dazu im Abschnitt über Wiederverwendung
- **substitutionsGroup**:Auflistung von Elemente, welche gleichberechtigt anstelle der anderen im Instanzdokument auftreten können, mehr dazu im Abschnitt über Substitutionsgruppen
- **type**: Festlegung des Typen des Elementes, wobei vordefinierte, als auch anwenderdefinierte Typen benutzt werden können[5]

2.2.1.3 Attributdeklaration

Die Attributdeklararation geschieht analog zur Elementdeklaration durch das Schlüsselwort `attribute`. Jedoch werden hier mehr Möglichkeiten der Deklaration als bei der DTD unterstützt. So können Attribute deklariert werden, welche optional, zwingend, konstant und verboten sind. Dieses geschieht durch die möglichen Werte des `use`-Attributes `optional`, `required`, `fixed` und `prohibited`.

```
<xs:attribute name="PNr" use="required" type="xs:integer"/>
```

Abbildung 2.4: Attributdeklaration

Das Beispiel 2.4 zeigt ein Attribut, welches den Namen PNr hat und vom Typ `integer` ist und zwingend vorkommen muss.
Ein mögliches Instanzfragment könnte

```
<Person PNr="4326">...</Person>
```

sein.
Analog zur Elementdeklaration besteht die Möglichkeit, selbst definierte Typen zuzuweisen, oder das `type`-Attribute nicht zu nutzen und den Typen lokal und anonym zu definieren. Dieses wird in Abschnitt 2.2.2.2 detailliert beschrieben.
Aber durch diese Angabe von speziell gebildeten Typen, sind die Möglichkeiten zur Deklaration vielfältiger als bei der DTD, da nun auch mittels Listenbildung und Aufzählungen ein Attribut deklariert werden kann. Die Charakteristika des deklarierten Attributes bestimmt man durch folgende Attribute:

- **default**: Angabe eine Vorgabewertes, welcher benutzt wird, wenn der Nutzer in der Instanz keinen Wert spezifiziert
- **final**: die Nutzung des Elementes in einer Ableitung wird verboten. Dabei verbietet final im Gegensatz zu block die Ableitung schon auf Schemaebene
- **fixed**: konstantes Attribut, welches vom Nutzer nicht geändert werden kann
- **form**: wie bei der Elementdeklaration
- **id**: eindeutige Kennzeichnung des Attributes durch eine schemaweite eindeutige Zeichenkette

[4] NCName ist ein String ohne Doppelpunkt.
[5] siehe Abschnitt 2.2.2

- **name**: Namensgabe, wobei der Name vom Typ NCName[6] ist.
- **ref**: wie bei der Elementdeklaration
- **type**: Zuweisung eines Typen aus der Menge der simpleType's[7], wenn kein Typ angegeben wird, ist der Typ `anyType`
- **use**: legt fest, wie das Attribut verwendet werden soll, ob es optional (`optional`) ist, notwendig (`required`) oder verboten ist (`prohibited`). Die Angabe, dass ein Attribut verboten ist, bedeutet, dass es in einer Instanz nicht auftreten darf, sondern es wurde lediglich zur Strukturierung im Schema verwendet und wurde nicht zur allgemeinen Nutzung freigegeben.

2.2.1.4 Attributgruppen

Wiederbenutzung erreicht man durch den Referenzierungsmechanismus von XML-Schema, welcher im Abschnitt Wiederverwendung/Referenzen genauer erklärt wird. Eine weitere Möglichkeit sind Attributgruppen, bei denen man mehrere Attributdeklarationen zusammenfassen kann und ebenfalls über den Referenzierungsmechanismus einem Element zuweisen kann.

```
<xs:attributeGroup name="itemInfoGroup">
    <xs:attribute name="PNr"
        type="xs:integer"/>
    <xs:attribute name="Personenkennziffer" type="xs:string"/>
    <xs:attribute name="Groesse" type="xs:float"/>
    ...
</xs:attributeGroup>

<xs:element name="Person">
    <xs:complexType>
        <xs:attributeGroup ref="itemInfoGroup"/>
    </xs:complexType>
</xs:element>
```

Abbildung 2.5: Attributgruppe

Das Beispiel 2.5 zeigt eine Attributgruppe, welche drei verschiedene Attribute beinhaltet. In der Elementdeklaration wird nun auf die Gruppe referenziert, so dass in einer Instanz das Element bis zu drei Attribute besitzt.
Die Vorteile von Attributgruppen sind, dass man die Attribute global definiert und sie somit jederzeit wiederbenutzen kann. Ausserdem strukturieren sie das Schema und steigern die Lesbarkeit, da man mehrere Attribute mit nur einer Referenz in einem Element benutzt werden können. Desweiteren kann man über Referenzen auch Gruppen von Gruppen darstellen.

2.2.1.5 Elementgruppen und Modellgruppen

Die Deklaration von Elementgruppen erfolgt analog zu der Deklaration von Attributgruppen. Das hierzu verwendete Schlüsselwort ist `group`.
Genauer sind jenes Elementgruppendefinitionen, da weitere Elementgruppen oder Modellgruppen durch die Schlüsselwörter `all`, `choice`, `sequence`, `union` und `list` deklariert werden. Wie am Beispiel ersichtlich ist, kapselt eine Elementgruppendefinition ein Elementgruppe. Die Funktionalität und die Anwendung von `all`, `choice` und `sequence` wird im Abschnitt über selbstdefinierte Typen (2.2.2.4) näher erläutert.

[6] NCName ist ein String ohne Doppelpunkt.
[7] siehe Abschnitt 2.2.2.3

```
<xs:group name="elementGroup" >
    <xs:sequence>
        <xs:element name="Beruf" type="xs:string"/>
        <xs:element name="Familienstand" type="xs:string"/>
        ...
    </xs:sequence>
</xs:group>

<xs:element name="Person">
    <xs:complexType>
        <xs:group ref="elementGroup"/>
    </xs:complexType>
</xs:element>
```

Abbildung 2.6: Elementgruppe

2.2.1.6 Substitutionsgruppen

Hinter einer Substitutionsgruppe steht eine Gruppierung von Elementen, welche gegeneinander ausgetauscht oder vielmehr substituiert werden können. Über das Attribut `substitutionGroup` legt man fest, zu welcher Gruppe ein deklariertes Element gehört. Hierbei ist der Wert des Attributes ein anderes Element, wobei die Typen aller in einer Gruppe vereinten Elemente gleich sein müssen.

```
<xs:element name="Telefonnr" type="xs:string"/>
<xs:element name="US-Telefonnr" type="xs:string"
    substitutionsGroup="Telefonnr"/>
<xs:element name="Int-Telefonnr" type="xs:string"
    substitutionsGroup="Telefonnr"/>

<xs:element ref="Telefonnr"/>
```

Abbildung 2.7: Substitutionsgruppe

Im Beispiel 2.7 wurden drei Elemente deklariert, wobei durch die Angabe der Substitutionsgruppe alle drei zusammengefasst wurden. Die vierte Elementdeklaration zeigt, wie man eine Substitutionsgruppe weiterverwendet.
In einem Instanzdokument kann nun an der Stelle, wo eine `Telefonnr` stehen muss, auch eine `US-Telefonnr` oder eine internationale `Int-Telefonnr` stehen.
Durch den Einsatz von Substitutionsgruppen erreicht man so eine hohe Flexibilität bei der Erstellung von Instanzen, welche gegenüber dem Schema gültig sein müssen.

2.2.1.7 Annotationen

Annotationen dienen zum Kommentieren des Schemas oder oft auch zur Angabe spezieller Instruktionen für spezielle Parser und Validierer. Diese Annotationen bedienen sich dabei keiner speziellen Syntax wie Kommentare und bieten darüeber hinaus auch eine größere Flexibilität, da noch zwischen Applikationsinformation und Dokumentation unterschieden wird.

2.2.1.8 Vererbung

Der XML-Schema-Vorschlag sieht einen Mechanismus vor, der Vererbung ermöglicht. Dabei handelt es sich ähnlich wie bei objektorientierten Programmiersprachen um die Bildung von neuen Datentypen mittels schon definierter Datentypen.

```
<xs:annotation>
    <xs:appinfo>
        Schema zur Beschreibung eines Personenkontaktes
    </xs:appinfo>
    <xs:documentation>
        Dieses Element dient...
    </xs:documentation>
</xs:annotation>
```

Abbildung 2.8: Annotation

Hierbei wird unterschieden unterschieden, ob die Vererbung eine Ableitung durch Einschränkung ist oder eine Ableitung durch Erweiterung.

Ableitung durch Einschränkung Bei diesem Vererbungstyp beschränkt der Subtyp den Supertyp (*derivation by restriction*). Dass bedeutet, dass der Subtyp durch eine engere Definition im Wertebereich engeschränkt wird. Ein Beispiel wäre der Typ `int`, dessen Wertebereich $-2^{31} <= int <= 2^{31} - 1$ beträgt, wobei hingegen der Typ `short` eine Ableitung von int durch Einschränkung ist, da der Wertebereich ($-2^{16} <= short <= 2^{16} - 1$) eine Teilmenge des Wertebereiches von int ist.
Die Terminus zur Umsetzung ist die *Fassette (facet)*, wobei es grundlegende Fassetten gibt, welche dazu dienen, den Wertebereich eines Typen semantisch zu beschreiben. Im Schema existieren fünf grundlegende Fassetten.

- Gleichheit
- Ordnung
- Wertebereichsgrenzen
- Kardinalität
- numerischer/nicht-numerischer Wertebereich

Diese Fassetten sind nicht manipulierbar und ihnen unterliegen alle Datentypen. Dem gegenüber stehen die einschränkenden Fassetten, welche dazu dienen, den Wertebereich eines Typs einzuschränken, wobei sie durch den Nutzer manipulierbar sind. Zu ihnen zählen

- **length**: Längenangabe des Typs, wobei diese je nach Typ, von dem abgeleitet wird, variieren kann, bei String z.B. Zeichenanzahl, bei `hexBinary` Byteanzahl
- **minLength**: minimale Länge
- **maxLength**: maximale Länge
- **pattern**: der lexikalische Bereich wird auf Literale beschränkt, welche auf ein bestimmtes Muster zutreffen, das Muster muss ein regulärer Ausdrück sein (siehe Unix-RegEx's)
- **enumeration**: Einschränkung des Wertebereiches auf die benannten Werte
- **whiteSpace**: Beschränkung des Wertebereiches von Typen, welche von String abgeleitet sind. Dies geschieht durch Bearbeitung des Whitespaces (Tabulatoren, Wagenrücklauf, Zeilenvorschub und Freizeichen)
  - *preserve:* keine Behandlung des Whitespace
  - *replace:* Tabulatoren (#x9), Zeilenvorschub (#xA) und Wagenrücklauf (#xD) werden durch Freizeichen (#x20) ersetzt

  - *collapse:* aufeinander vorkommende Freizeichen werden zu einem gekürzt und Freizeichen als Präfix oder Suffix werden entfernt.

- **maxInclusive**: gibt die obere Schranke inklusive an
- **maxExclusive**: gibt die obere Schranke exklusive an
- **minExclusive**: gibt die untere Schranke exklusive an
- **minInclusive**: gibt die untere Schranke inklusive an
- **totalDigits**: gibt die Gesamtanzahl von Dezimalstellen für vom Typ `decimal` abgeleitete Typen an
- **fractionDigits**: gibt die Gesamtanzahl der Nachkommastellen an

```
<xs:element name="PLZ">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:pattern value="\d{5}"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
```

Abbildung 2.9: Vererbung: Ableitung durch Einschränkung (1)

In Beispiel 2.9 wird dem Element `PLZ` anonym und lokal ein Typ zugewiesen, welcher aus der Klasse simpleType ist (siehe 2.2.2.3), dabei wird dieser lokale Typ durch Ableitung mittels Einschränkung erzeugt. Geerbt wird vom Typ `string` und soweit eingeschränkt, dass `PLZ` den Wertebereich von 5 Zahlen (siehe Unix-RegEx's) hintereinander hat.

```
<simpleType name="Summe">
    <restriction base="decimal">
        <totalDigits value="8"/>
        <fractionDigits value="2" fixed="true"/>
    </restriction>
</simpleType>
```

Abbildung 2.10: Vererbung: Ableitung durch Einschränkung (2)

In Beispiel 2.10 wird ein einfacher Typ (siehe 2.2.2.3) mit dem Namen `Summe` definiert und der Wertebereich wird auf Zahlen beschränkt, welche insgesamt 8 Stellen und davon 2 Nachkommastellen besitzen.
Die Anwendung der anderen Fassetten erfolgt analog, wobei die möglichen anwendbaren Fassetten von Datentyp zu Datentyp variieren, da ein Wert vom Typ String z.B. keine Nachkommastellen aufweist.

Ableitung durch Erweiterung Bei diesem Vererbungstypen erweitert der erbende Subtyp den Supertyp *(derivation by extension)*. Das bedeutet, dass diese Ableitung nicht verändernd auf die Elemente des Supertyps eingreift, sondern neue hinzufügt. Analog zu der Ableitung durch Einschränkung weist hier das Schlüsselwort `extension` die Vererbungsart aus.
In Beispiel 2.11 wurde ein Typ `Student` definiert, welcher ein Element `Matrikelnr` umfasst. Danach wurde ein zweiter komplexer Typ definiert, welcher von `Student` erbt und so erweitert, dass ein zweites Element hinzugenommen wird. Dadurch umfasst der Typ `HiWi` sowohl ein Element `Matrikelnr` als auch ein Element `Gehalt`.
Durch das Hinzufügen von Elementen ist der Subtyp mächtiger als der Supertyp.

```
<xs:complexType name="Student">
    <xs:sequence>
        <xs:element name="Matrikelnr" type="xs:int"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="HiWi">
    <xs:complexContent>
        <xs:extension base="Student">
            <xs:sequence>
                <xs:element name="Gehalt"
                    type="xs:float"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Abbildung 2.11: Vererbung: Ableitung durch Erweiterung

2.2.1.9 Wiederverwendung/Referenzen

Neben der Vererbung sieht der XML-Schema-Vorschlag auch einen Mechanismus zur Wiederverwendung bereits deklarierter Attribute und Elemente vor. Dieses geschieht durch das Attribut `ref`, wobei auf ein Element oder Attribut referenziert wird. Bei dieser Art der Referenzierung wird zwingend die vollständige Definition übernommen, dh. Name, Typ und bei Elementen Übernahme des Inhaltsmodelles (siehe dazu Abschnitt 2.2.1.2).

```
<xs:attribute name="phone">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:pattern value="\+\d{2}\-\d{2,6}\-\d{2,12}"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>

<xs:element name="business">
    <xs:complexType mixed="true">
        <xs:attribute ref="phone"/>
    </xs:complexType>
</xs:element>
```

Abbildung 2.12: Referenzierung von Attributen/Elementen

Es wird in 2.12 ein Attribut `Telefonnr` mittels Ableitung durch Einschränkung deklariert und dann im deklarierten Element `Geschaeft` durch Referenzierung verwendet.
Eine zweite Möglichkeit der Wiederverwendung von Schema-Konstrukten ist die Definition von eigenen Datentypen, von denen entweder geerbt werden kann oder in der Definition weiterer Datentypen verwendet werden können. Dabei erfolgt die Weiterverwendung einfach durch Zuweisung über das `type`-Attribut, genaueres über die Definition von eigenen Datentypen im Abschnitt über simpleType und complexType (2.2.2.2).
Bei der Wiederverwendung durch Referenzen oder durch Weiterverwendung von definierten Typen gibt es lediglich die Einschränkung, dass es nur mit Elementen, Attributen und Typen, welche global definiert und deklariert sind, realisierbar ist. Dabei ist ein Element global deklariert, wenn

die Deklaration direktes Kind vom `schema`-Knoten ist. Wobei der schema-Knoten der Wurzel-Knoten des Schema-Baumes ist (siehe 2.2.1.1). Im Kapitel über die grafische Notation (3.2) wird zudem gezeigt, wie sich ein Schema als Baum visualisieren lässt.

2.2.1.10 import/include/redefine

Ein weiterer Mechanismus um Wiederverwendbarkeit zu erlangen, ist die Möglichkeit, Schemas in mehrere aufzuteilen und diese über `import` und `include` wieder zu einem Schema zusammenzufügen. Dadurch kann man Teile auslagern und sie beliebig wiederverwenden.

include Es ist der einfachste Weg, verschiedene Schemata einzubinden, jedoch unterliegt es vielen Einschränkungen im Gebrauch. Die `include`-Anweisung muss ein direktes Kind vom `schema`-Knoten sein und das erste oder die ersten, da mehrere `include`-Anweisungen gültig sind. Der Effekt der include-Anweisung ist derselbe, als wenn man die im eingebundenen Schema enthaltenen Deklarationen und Definitionen im Dokument, welches das Schema einbindet, vorgenommen hätte. Das heißt, der Effekt ist derselbe wie bei der #include-Direktive in der Programmiersprache C++.

```
<xs:include id="ID" schemaLocation="http://localhost/mySchema.xsd"/>
```

Abbildung 2.13: include-Anweisung

Das Beispiel 2.13 zeigt den Gebrauch der `include`-Anweisung.
Durch diesen Effekt ist auch klar, dass nur die im eingebundenen Schema global deklarierten und definierten Element, Attribute und Typen verwendet können. Diese Restriktion ist dieselbe wie bei der Wiederverwendung durch das `ref`-Attribut.
Die nächte Einschränkung beim Gebrauch ist, dass alle über include eingebundenen Schemata denselben `targetNamespace`[8] haben müssen. Der Grund dafür ist der Vorgang beim Parsen, bei denen die geparsten eingebundenen Schemata in denselben Namensraum des einbindenen Dokumentes gelegt werden. Deswegen kann man Schemata, die keinen Zielnamensraum angegeben haben einbinden, aber keine, die einen unterschiedlichen Zielnamensraum deklarieren. Damit man Schemata mit unterschiedlichen Zielnamensräumen einbinden kann, wurde die import-Anweisung definiert.

import Die Arbeitsweise der `import`-Anweisung ist ähnlich der `include`-Anweisung, jedoch gibt es die Möglichkeit, den Namensraum mit anzugeben, in welchem das importierte Schema liegt. Man sollte ebenfalls den Namensraum des importierten Dokumentes deklarieren und ebenfalls einen Präfix spezifizieren.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="urn:myNameSpace"
    xmlns:mySchema="http://localhost/mySchema"/>

    <xs:import
        namespace="http://localhost/mySchema"
        schemaLocation="http://localhost/mySchema.xsd"/>
    ...
</xs:schema>
```

Abbildung 2.14: import-Anweisung

Das Beispiel 2.14 zeigt den Gebrauch der `import`-Anweisung.

[8] siehe 2.2.1.1

redefine Ein Problem beim Gebrauch von `import` und `include` ist die Annahme, dass man mit den importierten Komponenten in exakt der Form, wie sie importiert wurden, weiterarbeitet. XML-Schema bietet aber einen Mechanismus, mit dem man Schemata importieren kann und die dort deklarierten Komponenten umdefinieren kann. Dieses geschieht mit dem Schlüsselwort `redefine`. Auch hier gibt es gewisse Einschränkungen im Gebrauch.

- `redefine` muss das erste Kind vom schema-Knoten sein
- nur Schemata, die zum selben Zielnamensraum[9]gehören oder keinen Zielnamensraum besitzen, können redefiniert werden
- nur global deklarierte Komponenten können redefiniert werden

Die Verwendung sieht ähnlich der Verwendung von Attribut- und Element-Deklarationen oder Typdefinitionen aus. So umschließt das `redefine`-Tag eine Definition, welche einen Typen aus dem importierten Schema neudefiniert.

```
<xs:redefine schemaLocation="http://localhost/schema.xsd">
    <xs:element name="Telefonnr">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:pattern value="\+\d{2}\-\d{2,6}\-\d{2,12}"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:element>
</redefine>
```

Abbildung 2.15: Umdefinierung einer Elementdeklaration

Das Beispiel in der Abbildung 2.15 zeigt, wie Teile eines importierten Schemas neudefiniert werden können. Die Komponenten, welche nicht neudefiniert wurden, werden unverändert übernommen.

2.2.2 Datentypen

Datentypen definieren den Typ, den Daten innerhalb eines XML-Dokumentes haben können. Das Typkonzept innerhalb XML-Schema ist wesentlich komplexer und umfangreicher als innerhalb der DTD. Da die Technologie XML auch im Umfeld der Programmiersprachen und Datenbanken Einsatz findet, ist es sinnvoll, das dortige Typkonzept nutzen zu können.
Aus diesem Grund wurden viele Datentypen nach XML-Schema adaptiert und integriert. So sind die Ziele des Datentyp-Teils von XML-Schema

- Darstellung von primitiven Datentypen
  - byte, integer, float,...
  - string, date,...
  - Datentypen von SQL
  - primitive Datentypen von Java oder C++
- Definition eines Typsystems, welches eine adäquate Importierung in und Exportierung aus Datenbanksystemen erlaubt
- Möglichkeit zur Definition nutzerspezifischer Datentypen
  - complexType und simpleType
  - Ableitung von Datentypen
  - Vererbung und Einschränkung von Datentypen

[9]siehe 2.2.1.1 auf Seite 8

2.2.2.1 Built-In-Typen

Im Vorschlag des W3C sind 44 Datentypen vordefiniert, die sogenannten *Built-In-Types*. Jenes sind elementare oder primitive Datentypen, wie sie in Programmiersprachenumgebungen und in Datenbanksystemen vorkommen können. Abbildung 2.16 zeigt eine Übersicht über die verfügbaren Built-In-Typen von XML-Schema. Es können jedoch keine neuen primitiven Datentypen erzeugt

Abbildung 2.16: Typ-Hierarchie in XML-Schema (©W3C)

werden, der einzige Weg ist, diese im W3C-Vorschlag zu integrieren, aber sie sind Grundlage dafür, neue Typen zu erzeugen, entweder durch die Ableitung oder Vererbung oder durch die Möglichkeit neue strukturierte nutzerdefinierte Typen zu definieren. Dabei hängt es stark von dem vererbenden Typen ab, welche einschränkenden Fassetten benutzt werden können (genauer in [Gul01] S.192ff und im W3C-Vorschlag Teil3 [W3C01a]).

2.2.2.2 Nutzerdefinierte Datentypen

Eine weitere Anforderung an XML-Schema war es, neue Typen erzeugen zu können. Möglichkeiten zur Realisierung wurden schon im Strukturteil angedeutet, siehe dazu 2.2.1.8 und 2.2.1.9.
In XML-Schema werden zwei grundsätzliche Arten von Typen unterschieden, einfache Typen (`simpleType`) und komplexe Typen (`complexType`). Die Typ-Hierarchie ist in Abbildung 2.16 gezeigt. Die Basis für alle Typen in XML-Schema ist der Urtyp `anyType`. Desweiteren ist `anySimpleType`

die Basis für alle einfachen Typen und somit auch für alle Built-In-Typen. Die Basis für die komplexen Typen bildet `anyComplexType`. Dabei bedeutet Basis in diesem Zusammenhang, dass diese Typen die Vorfahren aller anderen Typen in der Hierarchie sind.

2.2.2.3 simpleType

SimpleType's nennt man einfache Typen, da sie weder Elementinhalt noch Attributinhalt besitzen können, das bedeutet, dass ein Element, dessen Typ ein Kindelement deklariert, immer aus der Menge der komplexen Typen ist. Daraus ist ersichtlich, dass einfache Typen nur von einfachen Typen erben können. Insgesamt gibt es drei Möglichkeiten neue einfache Typen durch Ableitung zu bilden.

- Ableitung durch Einschränkung
- Ableitung durch Mengenbildung
- Ableitung durch Listenbildung

Die Ableitung durch Erweiterung ist nicht möglich, da ein einfacher Typ durch Erweiterung zu einem komplexen Typen werden kann, indem man z.B. ein Element hinzufügt.

Ableitung durch Einschränkung Diese Art der Typbildung ist die am häufigsten vorkommende. Sie beruht auf der Manipulation von einschränkenden Fassetten (*constraining facets*), welche jeder Datentyp besitzt.
Die Definition eines einfachen Typen geschieht durch das Schlüsselwort `simpleType`. Ein einfacher Typ besitzt drei Attribute, den Namen, den der Typ erhalten soll, eine schemaweite eindeutige ID und das Attribut `final`, welches angibt, ob der Typ weitervererbt werden kann oder nicht.

```
<xs:simpleType name="Vorname" id="ID001" final="extension">
    ...
</xs:simpleType >
```

Abbildung 2.17: simpleType-Definition

Die Abbildungen 2.9 (S.15) und 2.10 (S.15) zeigen ein detaillierteres Beispiel für die Definition eines einfachen Typen durch die Ableitung mit Einschränkung.

Ableitung durch Mengenbildung Eine weitere Form der Ableitung bei einfachen Typen und eine weitere Form der Wiederverwendung ist die Ableitung durch Vereinigung, bei der mehrere einfache Typen zu einem zusammengefassten Typen werden können.
Die Vereinigung geschieht über das Element `union`, welches mehrere einfache Typdefinitionen in einer neuen Definition vereinigt.

Ableitung durch Listenbildung Die dritte und letzte Möglichkeit, neue Typen zu definieren, ist die Ableitung mittels Listen. Im Gegensatz zur Fassette `enumeration`[10] kann eine Liste mehr als einen Wert aufnehmen, wobei die Werte durch Whitespace-Zeichen[11] getrennt werden. Die Definition einer solchen Liste erfolgt durch Angabe des Grundtypen wie `xs:string`, `xs:integer` oder beliebige definierte einfache Typen und unter Angabe der minimalen Länge und maximalen Länge der Liste.
Beispiel 2.19 zeigt eine Definition einer Liste vom Grundtyp `string`, deren minimale Länge ein Eintrag und deren maximale Länge fünf Einträge sind.
`<color>rot gruen blau orange schwarz weiß braun</color>` wäre deswegen ein nicht valides Beispiel, da die Liste mit sieben Einträgen die maximale definierte Länge überschreitet.

[10] siehe Ableitung durch Einschränkung Abschnitt 2.2.1.8
[11] Whitespacezeichen sind Zeilenumbruch, Wagenrücklauf, Leerzeichen, Tabulatoren, etc.

```
<xs:attribute name="size">
    <xs:simpleType>
        <xs:union>
            <xs:simpleType>
                ...
            </xs:simpleType>
            <xs:simpleType>
                ...
            </xs:simpleType>
        </xs:union>
    </xs:simpleType>
</xs:attribute>
```

Abbildung 2.18: Ableitung durch Vereinigung

```
<xs:element name="color">
    <xs:simpleType>
        <xs:list itemtype="xs:string">
            <xs:restriction>
                <xs:minLength value="1">
                <xs:maxLength value="5">
            </xs:restriction>
        </xs:list>
    </xs:simpleType>
</xs:element>
```

Abbildung 2.19: Ableitung durch Listenbildung

2.2.2.4 complexType

Den einfachen Typen stehen die komplexen Typen gegenüber, welche Elementinhalt und oder Attributinhalt besitzen können. Aus diesem Grund können komplexe Typen auch durch Ableitung mittels Erweiterung definiert oder wiederverwendet werden. Analog zu den einfachen Typen können auch komplexe Typen unabhängig von Elementdeklarationen definiert werden. So können sie weiter vererbt werden oder über Referenzen verwendet werden. Insgesamt existieren zwei Ableitungsmöglichkeiten um komplexe Typen zu bilden.

- Ableitung durch Einschränkung
- Ableitung durch Erweiterung

Beispiele zu den Ableitungsformen sind schon im Abschnitt über Vererbung (2.2.1.8) gegeben. Die Abbildungen 2.9 (S.15) und 2.10 (S.15) zeigen einschränkende Ableitung, Abbildung 2.11 (S.16) erweiternde Ableitung.

Modellgruppen Bei der Definition von Typen, welche als Inhaltsmodell nur Elementinhalt besitzen, fasst man die Elementdeklarationen in einer Modellgruppe zusammen. Diese Modellgruppen geben an, wie der Elementinhalt organisiert wird.

- `<choice>` definiert eine Liste an Elementen, von denen eines im Instanzdokument gesetzt werden muss.
- `<sequence>` definiert eine Liste an Elementen, von denen alle Elemente in der definierten Reihenfolge[12] in der Instanz vorkommen müssen.

[12] die Reihenfolge wird durch die Folge der Deklarationen vorgegeben

- `<all>` definiert eine Liste an Elementen, von denen alle Elemente in der Instanz vorkommen müssen, dabei ist die Reihenfolge nicht einzuhalten.

Es gibt bei den Modellgruppen je nach Art gewisse Einschränkungen im Gebrauch. Eine `choice`-Gruppe und eine `sequence`-Gruppe können Attribute wie `minOccurs`, `maxOccurs`, ID und ein Namensattribut mit allen möglichen Werten aus dem Wertebereich besitzen, im Gegensatz dazu darf das `all`-Element `minOccurs` und `maxOccurs` nur mit den Werten 0 oder 1 beinhalten. Diese Einschränkung bezieht sich auch auf die `minOccurs` und `maxOccurs`-Attribute der Elementdeklarationen innerhalb einer all-Gruppe.
Ebenfalls können Sequenzen und Auswahlen ineinander verschachtelt werden, `all`-Gruppen dürfen hingegen keine weiteren Modellgruppendefinitionen beinhalten und dürfen auch nicht innerhalb einer `choice`-Gruppe oder `sequence`-Gruppe vorkommen. Das bedeutet, dass das `all`-Element direktes Kind von `complexContent` oder `complexType` sein muss.

simpleContent/complexContent Anstelle einer anonymen lokalen Definition von komplexen Typen kann man gleichwertig das Inhaltsmodell des Elementes angeben. Dabei unterscheidet man zwischen `simpleContent` und `complexContent`. Ein komplexer Typ stellt gleichzeitig einen komplexen Inhalt dar und analog dazu die einfachen Typen, welche `simpleContent` darstellen.

Ableitung durch Einschränkung Eine Ableitung durch Einschränkung erhält man durch das Element `restriction` genauso wie bei den einfachen Typen. Der Unterschied ist jedoch, dass der Basistyp dupliziert wird. Das bedeutet, dass zwischen dem umschließenden `restriction`-Tag die Definition des Basistypen übernommen wird. Der Part, welcher eingeschränkt werden soll, wird dabei einfach neudefiniert. Wenn man diese Duplizierung nicht durchführt, dann schränkt man den Basistypen in der Hinsicht ein, dass der neue Typ "leer" ist, da XML-Schema nicht weiß, welche Komponente der Definition eingeschränkt werden soll. Da einfache Typen atomar[13] sind, erfolgt die Einschränkung direkt auf dem Definitionsbereich und man muss die Basistyp-Definition nicht duplizieren, wärend dessen komplexe Typen aus mehreren Komponenten zusammengesetzt sind.

Ableitung durch Erweiterung Im Gegensatz zu einfachen Typen unterstützen komplexe Typen die Ableitung durch Erweiterung. Durch das `extension`-Tag gibt man die Art der Ableitung als Ableitung durch Erweiterung an. Dabei muss man den Basistypen nicht duplizieren, in dem man seine Definition erneut aufführt. Jenes ist nicht nötig, da die beim Basistyp definierten Komponenten mit in den ableitenden Typen übernommen werden, zusätzlich zu den in dem umschließenden `extension`-Tag definierten Komponenten. Dieses entspricht der Vererbung in objekt-orientierten Programmiersprachen, wo die erbende Klasse automatisch die Methoden und Attribute der vererbenden Klasse übernimmt (mit Ausnahmen von speziell ausgezeichneten Komponenten mittels private oder protected[14]).

2.2.3 Gültigkeitsbereich

Unabhängig ob Elementdeklaration oder Typdefinition, alle Komponenten haben einen gewissen Gültigkeitsbereich (*scope*). Dieser Gültigkeitsbereich teilt sich in *lokal* und *global* auf. Der globale Gültigkeitsbereich beschränkt sich auf die Komponenten, die als direkte Kinder des `schema`-Knotens definiert oder deklariert wurden. Die Vorteile bei global deklarierten Komponenten sind einerseits die einfachere Struktur des Schemas, andererseits die Wiederverwendbarkeit, welches der wichtigste Grund für globale Deklarationen und Definitionen ist. Definierte Typen und deklarierte Attribute, Elemente und Gruppen können von jeder anderen Position im Schema aus genutzt werden, entweder durch Ableitungen, Referenzen oder Typzuweisungen.

[13] bei atomaren Typen kann man nicht auf einzelne Teile zugreifen, Beispiel ist der Built-In-Typ date, welcher die Notation yyyy-mm-dd hat und man nicht auf die Tag-Komponente zugreifen kann, dafür muss man den Typ gday benutzen.
[14] siehe Programmiersprache C++

Der lokale Gültigkeitsbereich ist verglichen zur DTD neu, wo alle Elementdeklarationen global waren. Dabei erreicht man lokale Gültigkeit, indem Deklarationen oder Typdefinitionen als Kinder von anderen Deklarationen oder Definitionen vorgenommen werden. In der Abbildung 2.9 auf Seite 15 wird ein Element `PLZ` deklariert und der Typ des Elementes lokal angegeben.
Der Vorteil, wenn man alle Definitionen und Deklarationen lokal vornimmt, ist, dass das Schema in der Struktur den Instanzdokumenten entspricht. Jedoch können lokal definierte und deklarierte Komponenten nicht über Referenz oder Vererbung etc. weiterverwendet werden.
Zu den Gültigkeitsbereichen kommt noch die Namensgebung. Im Beispiel 2.9 sieht man, dass der lokal definierte einfache Typ über kein Namensattribut verfügt. Solche Definitionen ohne Namensgebung nennt man *anonym*, da sie nicht über den Namen ansprechbar sind und somit ebenfalls nicht wiederverwendet werden können.
Es gibt bei der Verwendung von anonymen oder benannten Typen, bei lokalen oder globalen Definitionen und Deklarartion verschiedene Vor- und Nachteile, welche auch vom konkreten Einsatzgebiet abhängen (siehe 2.3).

2.2.4 IDs und Schlüssel

In der Document Type Definition existiert ein ID/IDREF-Mechanismus, der es erlaubt, global[15] eindeutige ID's zu definieren und diese ID's über IDREF zu referenzieren. Allerdings dürfen nur Attribute vom Typ ID sein. Wenn man diesen Mechanismus mit den Eindeutigkeits-Bedingungen (*Uniqueness*) und den referentiellen Integritätsbedingungen von Datenbanksystemen vergleicht, ist der ID/IDREF-Mechanismus sehr limitiert.
Im Vergleich dazu bietet XML-Schema die Möglichkeit, Elemente und Attribute als ID's zu definieren. Ebenso ist in dem W3C-Vorschlag ein Mechanismus zur Beschreibung von Uniqueness und referentieller Integrität mittels Schlüssels vorhanden. Darüber hinaus kann man den Gültigkeitsbereich über XPath-Ausdrücken[W3C99a] festlegen, indem die Eindeutigkeit gelten soll.

2.2.4.1 selector

Das `selector`-Element wählt den Gültigkeitsbereich der Attribute oder Elemente aus, welche als eindeutig oder als Schlüssel definiert wurden. Das `selector`-Element an sich hat keinen Inhalt, einzige Ausnahme sind Annotationen. Mit dem `xpath`-Attribut spezifiziert man einen XPath-Ausdruck, welcher dann den Gültigkeitsbereich bestimmt.

2.2.4.2 field

Das `field`-Element hat ebenfalls keinen Inhalt mit Ausnahme der Annotationen und als einzige Attribute wie das `selector`-Element ein `id`-Attribut und ein `xpath`-Attribut. Mit einem XPath-Ausdruck selektiert man die Komponente, Attribut oder Element, welche eindeutig (*uniqueness*) oder Schlüssel sein soll.

2.2.4.3 unique

Die `selector`- und `field`-Elemente sind der fundamentale Bestandteil der `unique`- und `key/keyref`-Elemente.
Das `unique`-Element wird dazu benutzt, ein Attribut oder ein Element oder eine Kombination aus mehreren Komponenten als in dem über den Selektor bestimmten Bereich eindeutig zu markieren.

2.2.4.4 key/keyref

Über das `key`-Element einen Schlüssel anzulegen, verläuft analog zur Verwendung des `unique`-Elementes. Es besitzt als einzigen erlaubten Inhalt das `annotation`-Element, einen Selektor und ein Feld. Mit Hilfe des Selektor-Elementes wählt man einen Gültigkeitsbereich aus und mit dem `field`-Element bestimmt man die Komponente, welche Schlüssel sein soll.

[15] global bedeutet hier, Instanzdokumentenweit

Dadurch kann man diese Komponente als Identifikator benutzen und über das `keyref`-Element darauf referenzieren.
Das `keyref`-Element sieht dabei genauso aus wie das `unique`-Element, nur dass es ein weiteres Attribut besitzt, das `refer`-Attribut, welches angibt, auf welchen Schlüssel referenziert werden soll.

```
<xs:unique name="Beispiel">
    <xs:selector xpath=".//Name"/>
    <xs:field xpath="@Telefon"/>
</xs:key>

<xs:key name="Schluesselname">
    <xs:selector xpath=".//Kontakt"/>
    <xs:field xpath="Kontakt/Telefon"/>
</xs:key>

<xs:keyref name="TelefonRef" refer="Schluesselname">
    <xs:selector xpath=".//Kontakt"/>
    <xs:field xpath="Kontakt/Telefon"/>
</xs:keyref>
```

Abbildung 2.20: Verwendung des unique, key und keyref-Elementes

Das Beispiel aus Abbildung 2.20 zeigt die Verwendung der `unique`, `key` und `keyref`-Elemente. Im `unique`-Beispiel wird vorgegeben, dass ein zuvor deklariertes Element `Name` der Gültigkeitsbereich einer eindeutigen Telefonnummer ist, wobei `Telefon` ein Attribut von `Name` ist. Im Beispiel zu key/keyref wurde ein Schlüssel mit Namen `Schluesselname` definiert und die Referenz `TelefonRef` verweist darauf. In diesem Falle ist der Gültigkeitsbereich das Element `Kontakt` und der direkte Schlüssel ist ein Kindelement `Telefon`. Dabei arbeiten die XPath-Ausdrücke auf der Struktur der Instanzdokumente.

2.3 Designprinzipien

Der W3C-Vorschlag beschreibt ausführlich die Möglichkeiten, welche mit XML-Schema realisierbar sind und welche Komponenten dafür verwendet werden müssen. Jedoch gibt es viele Fragen, die der W3C-Vorschlag nicht beantwortet oder nicht beantworten kann, da sie sehr abhängig von dem jeweiligen Einsatzgebiet von XML-Schema sind.
Hierbei geht es speziell um das gesamte Design und weniger um die spezielle Syntax von XML-Schema. Wie eine Elementdeklaration und eine Typdefinition auszusehen hat, wird fest vorgegeben und jeder, der ein Schema entwirft, wird dieses auf die selbe Weise tun. Die Frage aber, welches das beste Design für eine spezielle Anwendung ist, kann der W3C-Vorschlag nicht beantworten, da es zu viele Möglichkeiten des Einsatzes von XML-Schema gibt.
Die wichtigsten Fragen zum Einsatz der XML-Schema-Komponenten sind:

- Wann soll ein einzelnes großes Schema eingesetzt werden und wann viele kleine Schemata, welche dann zusammengefügt werden?
- Wann ist eine Elementdeklaration und wann eine Typdefinition sinnvoll?
- Sind globale Deklarationen und Definitionen sinnvoller oder lokale?
- Wann setzt man Attribute ein und wann Elemente?
- ...

Aufgrund der eben genannten Fragen und noch vieler mehr, sind Designprinzipien entstanden, welche ganz unterschiedliche Antworten auf die Fragen bezüglich des Gültigkeitsbereiches, Multipartschema und Definitionen und Deklarationen haben. Alle diese Prinzipien haben Vor- und Nachteile und haben ihre ganz unterschiedlichen Einsatzgebiete. Die drei großen Designprinzipien, welche sich herauskristallisiert haben, werden nun ausführlich beschrieben und ihre Vorteile und Nachteile genannt.

2.3.1 Russian Doll Design

Das *Russian Doll Design* (kurz RDD) sieht eine Nestung aller Definitionen und Deklarationen vor. Das bedeutet, dass alle Elemente und Typen lokal deklariert und definiert werden, mit der Ausnahme des Wurzel-Knotens der Instanz-Dokumente. Das Ergebnis ist ein Schema, welches die Instanz direkt wieder spiegelt, da die Struktur im Schema genauso festgelegt wurde, wie die Elemente und ihr Inhalt im Instanzdokument stehen[16].
Ein Beispiel eines Schemas im Russian Doll Design ist im Anhang A.2 aufgeführt, wobei ein mögliches Instanzdokument in Anhang A.1 aufgeführt ist.
Am Beispiel erkennt man sofort einen der gravierendsten Nachteile des RDD und von XML-Schema gegenüber einer DTD. Durch die strikte Nestung aller Deklarationen wird das Schema schnell unübersichtlich und durch die Verwendung der XML-Syntax durch XML-Schema, bläht sich die Schemadefinition auf.
Ein weiterer Nachteil sind Rekursionen, welche durch ein Schema im RDD nicht darstellbar sind, da die einzelnen Deklarationen und Definitionen unendlich oft in sich selber verschachtelt würden.
Der große Vorteil eines Schemas ist die Wiederverwendung von Teilen der Definition. Das RDD ist jedoch zu unflexibel, da alle Deklarationen und Definitionen lokal durchgeführt werden, können sie weder im Schema selbst, noch bei `import/include` weiterverwendet werden. Dies ist ein weiterer Grund, warum das Schema unübersichtlich lang werden kann, da viele Definitionen mehrfach gemacht werden, siehe dazu das Beispiel in Anhang A.2, bei dem die Definition des Telefontyps dreimal lokal und anonym vorgenommen werden muss.
Diesen Nachteilen stehen die Vorteile des einfachen Entwurfes und der schnellen Entwicklung gegenüber. Darüber hinaus können aus einem Schema im RDD schnell Instanzen gebildet werden, da eben das Schema die selbe Struktur besitzt. Die Unflexibilität gegenüber der Wiederverwendung von Teilen des Schemas bildet aber auch Vorteile, da durch die lokalen Definitionen und Deklarationen der Namensraum örtlich eingegrenzt wird. Deswegen kann bei Verwendung des Schemas in anderen Schemata entweder über die Namensraumdeklaration (siehe 2.2.1.1) oder über den `include` und `import`-Mechanismus (siehe 2.2.1.10) nur auf das Wurzel-Element zugegriffen werden. Damit minimiert man die Folgeänderungen in diesen Schemata, wenn man am Schema im RDD etwas ändert. Es findet in gewissem Sinne auch ein *Information Hiding* statt, da die interne Struktur des Schemas "verborgen" bleibt, da man nur das Wurzel-Element "sieht" und darauf zugreifen kann.
Durch diese Art der Definition des Schemas leitet sich auch der Name des Designprinzips ab. Wenn man um jede deklarierte Komponente ein Rechteck ziehen würde, könnte man das Design auch als *Box-in-Boxes*-Design bezeichnen, ähnlich dem Matruschka-Prinzip, daher der Name Russian Doll Design.

2.3.2 Salami Slice Design

Das *Salami Slice Design* (kurz SSD) ist zum RDD gegensätzlich angelegt. Es sieht nur Deklarationen von Elementen und Attributen vor, welche ausschließlich global erfolgen. Die bei den Deklarationen verwendeten Typen sind entweder Built-In-Typen oder werden lokal definiert. Ebenso werden diese Typdefinitionen nicht verschachtelt.
Die einzigen Ausnahmen bei den Deklarationen sind Elemente oder Attribute, welche in komplexen Typen deklariert werden und als Typ einen Built-In-Typ besitzen, diese werden lokal deklariert. Ansonsten werden die Deklarationen über Referenzen (siehe 2.2.1.9) durchgeführt.

[16] what you see is what you get (*WYSIWYG*)

Ein komplettes Schema welches dem Salami Slice Design folgt, ist im Anhang A.3 aufgeführt. Es definiert dabei die selbe Menge an Instanzdokumenten wie das Schema im RDD aus A.2.
Durch die Vorgabe der globalen benannten Deklarationen ergeben sich viele Vorteile, welche das RDD nicht hat. So ist ein Schema im SSD leicht in mehrere Schemata aufteilbar und mittels `include` und `import` wieder zusammensetzbar. Da die Komponenten global deklariert sind, kann man auf alle diese Komponenten zugreifen und sie verwenden. Dadurch ist das SSD bezüglich der Wiederverwendbarkeit flexibler als das Russian Doll Design. Ebenso ist es klarer strukturiert und dadurch teilweise kürzer und besser lesbar als das RDD, wobei die Lesbarkeit eher subjektiv als objektiv ist.
Die Nachteile bei diesem Designprinzip sind die Namensräume, da man bei einem Schema den Zielnamensraum mit angeben kann, welcher dann in den Instanzen ebenfalls angegeben werden muss und als Präfix vor die Elementnamen geschrieben wird, sofern die Elemente oder Attribute global deklariert sind. Jenes kann man über ein bestimmtes Attribut in der Elementdeklaration und Attributdeklaration ausschalten. Das `form`-Attribut bestimmt den Umgang mit den Präfixen (siehe Abschnitte über Elemente 2.2.1.2 und über Attribute 2.2.1.3) bei Angabe eines Zielnamensraumes. Durch diese Angaben kann es leicht zu Namensraumkonflikten in der Instanz kommen. Da die ganzen definierten Namensraumpräfixe angegeben werden müssen, wird so die Unleserlichkeit der Instanzen erhöht. Ebenso ist im Schema die Angabe, ob die Präfixe in der Instanz genutzt werden müssen, sehr aufwendig, da sie für jedes einzelne deklarierte Element und Attribut durchgeführt werden muss.
Wenn man sich das Schema genauer anschaut, dann sieht man, dass die verschachtelten Deklarationen und Definitionen aus dem Russian Doll Design aufgelöst wurden und alle global deklariert wurden und hintereinander notiert wurden. Durch diesen Umstand leitet sich der Name des Prinzips ab, weil es so ähnlich aussieht, wie eine aufgeschnittene Salami.

2.3.3 Venetian Blind Design

Das *Venetian Blind Design* (kurz VBD) ist dem SSD sehr ähnlich, da hier ebenfalls Definitionen und Deklarationen nicht verschachtelt werden und diese global erfolgen. Jedoch werden beim VBD keine Elemente und Attribute global deklariert, sondern die verwendeten Typen im Schema werden global definiert und den Elementen und Attributen über das `type`-Attribut zugewiesen. Die einzige Ausnahme bildet das Wurzel-Element, welches als einziges Element global deklariert wird. Im Anhang A.4 ist das Beispiel im Venetian Blind Design angegeben, welches gleich die Vorteile des VBD vermittelt.
Im Vergleich zum SSD ist das VBD noch flexibler, was die Wiederbenutzung angeht, wobei die Gründe in den Referenzen liegen. Wenn man zwei Elemente haben möchte, mit exakt dem selben Typ aber mit unterschiedlichen Namen, wie es im Beispiel bei den Telefonnummern der Fall ist, dann muss man die Elemente zweimal deklarieren, da bei Referenzierungen neben dem Inhaltsmodell und Typ auch der Name unveränderbar übernommen wird. Beim VBD hat man dagegen nur Typen definiert, welche man ohne weiteres mehrfach verwenden kann. Dadurch kann man ein Schema im Venetian Blind Design wesentlich verkürzen, wenn man mehrere Komponenten mit den gleichen Typen hat.
Ein weiterer Vorteil ist die Namensraumkontrolle. Durch die globalen Typdefinitionen hat man keine globalen Elemente und Attribute, weswegen eine bessere Namensraumkontrolle erreicht wird, als dies beim SSD der Fall ist. Der Grund dafür ist die Möglichkeit, bei lokal deklarierten Komponenten global festzulegen, wie Namensraumpräfixe in Instanzen verwendet werden sollen. Dieses geschieht über spezielle Attribute des `schema`-Elementes:

- `elementFormDefault`
- `attributeFormDefault`

Die möglichen Werte sind die selben wie beim `form`-Attribut bei Komponentendeklarationen und auch ihre Bedeutung ist die gleiche.

- *qualified*: ein angegebenes Präfix muss verwendet werden
- *unqualified*: ein angegebenes Präfix wird nicht verwendet

Daraus leitet sich auch der Name für das Design ab, da man global festlegen kann, ob Verwendung der Präfixe stattfinden muss oder nicht, ähnlich einer Jalousie, welche man auf und zuklappt[17].

[17] Jalousie, engl. venetian blind

3 Normalform XSDNF

Es existieren mehrere Ausarbeitungen über Normalformen von XML-Dokumenten, welche sich in einem Schema, unabhängig ob DTD oder XML-Schema, niederschlagen [AL02]. Das Ziel bei diesen Normalformen wie *NF-SS* (näheres in [WLL+02]) oder *XNF* (siehe [Sun02]) ist das Erreichen der in Datenbanken vorkommenden Bedingungen für Normalformen, so dass damit die Redundanz in XML-Dokumenten vermindert, wenn nicht sogar entfernt werden kann. Dabei wird wie bei Datenbanken ein Dekompositionsalgorithmus[1] entwickelt, womit die Redundanzen unter Angabe von funktionalen Abhängigkeiten entfernt werden sollen. Dabei wird eine DTD oder ein XML-Schema in mehrere Teile aufgesplittet und somit im Endeffekt auch die Instanzen, welche dann, ähnlich wie mit Fremdschlüsseln und Verbunden in Datenbanken, mittels XLink und XPointer [W3C01b] zusammengesetzt werden können, so dass man die ursprüngliche Information wieder erhält.
Unabhängig von solchen Versuchen verfolgt die hier vorgestellte Normalform, kurz *XSDNF* von XML-Schema-Definition Normalform, andere Ziele wie unter anderem eine bessere Vergleichbarkeit von Schemata oder eine bessere Möglichkeit der Abbildung solcher Schemata in Klassendesigns oder Datenbankentwürfe.
Dabei spielt weniger die Redundanzfreiheit in Instanzdokumenten eine Rolle, als vielmehr eine geringe Komplexität wie

- geringe Komplexität des Schema
- geringe Komplexität bei der Verarbeitung des Schemas

In diesem Kapitel soll eine Normalform definiert werden, welche solche Eigenschaften aufweist und es sollen Betrachtungen bezüglich der Komplexität und der Güte des Schemas durchgeführt werden.

3.1 Definition

Die Normalform XSDNF ist stark an das Venetian Blind Design angelehnt, da dieses Designprinzip die meisten Vorteile in sich vereint, welche XML-Schema bietet. So steht die Wiederverwendung von Komponenten im Vordergrund, die klare lesbare Struktur des Schemas und bei einer Baumdarstellung des Schemas die geringe Baumtiefe. Daraus ergeben sich Vorteile in diversen Anwendungsgebieten. So ist die geringe Baumtiefe ein Vorteil beim Vergleich zweier Schemata, beim Venetian Blind Design reduziert sich der Vergleich auf den Vergleich zweier Listen, welche als Listenelemente die globalen Typen haben (siehe 4.1). Die Definition von Typen ermöglicht eine leichte Abbildung in einen Entwurf für ein relationales, objektrelationales oder objektorientiertes Datenbanksystem, da hier die Grundprinzipien ebenfalls auf Datentypen beruhen. Eine Abbildung in eine Klassenhierarchie einer objektorientierten Sprache wie Java oder C++ ist ebenfalls leicht realisierbar.
Diese Vorteile werden nun anhand einer formalen Definition und gewissen Metriken auf Schemata und den Aussagen, die diese Metriken zulassen, erläutert.

3.1.1 formale Definition

Da ein XML-Dokument als Baum repräsentierbar ist, kann ein XML-Schema aufgrund der Verwendung der XML-Syntax ebenfalls als Baum repräsentiert werden. Aus diesem Grund werden die Eigenschaften der Normalform anhand Baumeigenschaften definiert.

[1] siehe [HS00, S.251,264] und [Heu97]

Schema:
Sei ein Schema $\mathcal{S}$ in Baumnotation, dann gilt $\mathcal{S} = (N, E)$, wobei N eine Menge an zulässigen Knoten ist und E eine Menge an Kanten und es gilt $E \subset N \times N$. Das hier die Beziehung der echten Teilmenge gilt, wird schon durch die Eigenschaft der Irreflexivität der Relation E begründet. Kein Knoten steht mit sich selber in Beziehung.
Der Wurzelknoten eines Schemas $\mathcal{S}$ ist immer der `schema`-Knoten, wobei definierte Präfixe für verwendete Namensräume vernachlässigt werden.

zulässige Knoten und Kanten:
Die Menge der zulässigen Knoten N besteht aus den Bezeichnern der XML-Elemente in einem Schema, das bedeutet, dass z.B. `complexType` und `element` zulässige Knoten sind. Somit wird diese Menge und die Beziehung untereinander, welche durch die Relation $E \subset N \times N$ ausgedrückt wird, durch den Namensraum `http://www.w3.org/2001/XMLSchema` bestimmt, ähnlich wie das erstellte Schema die Knoten und Beziehungen in Instanzdokumenten bestimmt.

Baumlevel:
Sei ein Schema $\mathcal{S} = (N, E)$ in grafischer Notation und ohne Rekursionen. Dann bestimmt die Länge des Weges von der Wurzel zu einem Knoten $n \in N$ auch die Baumstufe oder Baumlevel, in der sich der Knoten befindet, was auch durch den Pfad ausgedrückt wird. Die Baumlevelfunktion $level : N \rightarrow \mathbb{N}$ bildet nun einen beliebigen Knoten auf den Wert des Baumlevels, in dem sich der Knoten befindet.

Vater-Kind-Beziehung:
Sei $\mathcal{S} = (N, E)$ ein Schema und $u \in N$, dann ist die Menge der Kinder von u definiert durch:

$$children(u) = \{v \in N \mid (u, v) \in E\}$$

Dabei heisst u Vater von v und v heisst Kind von u. Da folgendes gilt: $level(u) < level(v)$, befindet sich u auf einer kleineren Baumstufe als v, aus diesem Grund wird die Vater-Kind-Beziehung wie folgt notiert:

$$v \in children(u) : u < v$$

Pfad:
Sei $\mathcal{S} = (N, E)$ ein Schema, dann ist ein Pfad $\rho_{u_n}^{u_1}$ eine Sequenz von Knoten $u_1, u_2, \ldots, u_n \in N$, wobei $u_i < u_{i+1}$ oder $(u_i, u_{i+1}) \in E$ mit $(1 < i < n - 1)$ und $n \geq 2$ gelten muss. Jenes bedeutet, dass ein Pfad ρ mindestens zwei Knoten umfasst.
Dabei ist die Länge eines Pfades definiert als

$$\mid \rho_{u_n}^{u_1} \mid = n$$

Daraus ergibt sich auch eine bessere Definition der Baumlevel-Funktion, so kann man die Baumlevel-Funktion auch wie folgt definieren:

$$u \in N \wedge \exists \rho_u^{\texttt{schema}} : level(u) = \mid \rho_u^{\texttt{schema}} \mid$$

wobei `schema` der Wurzelknoten des XML-Schemas ist.

Vorfahr-Nachfahr-Beziehung:
Die Vorfahr-Nachfahr-Beziehung zweier Knoten aus N lässt sich wie folgt definieren:

$$(u, v \in N \wedge \exists \rho_v^u) \Rightarrow u \prec v$$

Dann ist u Vorfahre von v und v ist Nachfahre von u. Dabei wird das $\prec$ Zeichen verwendet, weil $level(u) < level(v)$ gilt.

Bezeichner:
Ein Schema ist ein bezeichneter Baum, dies bedeutet, dass die einzelnen Knoten im XML-Schema-Baum über einen Bezeichner verfügen, welcher aus der Menge der zulässigen Bezeichner sein muss. Dabei ist diese Menge durch den W3C-Vorschlag von XML-Schema definiert und im Namensraum `http://www.w3.org/2001/XMLSchema` zusammengefasst.
Die Abbildung $label(u)$ bildet dabei einen Knoten u auf seinen Bezeichner ab.

$$u \in N \Rightarrow label(u) \in \texttt{http://www.w3.org/2001/XMLSchema}$$

Ein Bezeichner kann als Element eines Namensraumes aufgefasst werden, da ein Namensraum als Menge von Informationen[2] bezüglich erlaubter Namen und Beziehungen definiert ist.

Mächtigkeitsabbildung:
Sei $\mathcal{S}$ ein Schema und N die Menge der Knoten des Schemas.
Die Mächtigkeitsabbildung $\mathfrak{M}_K : N \rightarrow \mathbb{N}_0$ bildet eine beliebige Knotenmenge N und einen Knotenbezeichner K in die Menge der natürlichen Zahlen inklusive der Null ab. Dabei entspricht das Ergebnis der Abbildung gerade der Anzahl der Knoten aus N mit dem Bezeichner K.
Ein Beispiel wäre,

$$\mathcal{S} = (N, E) : \mathfrak{M}_{\texttt{schema}}(N) = 1$$

da in einem XML-Schema-Dokument der Knoten `schema` nur einmal auftreten darf.
Aus dieser Definition ergibt sich auch folgende logische Schlussfolgerung für ein Schema $\mathcal{S} = (N, E)$:

$$u \in N \Rightarrow \mathfrak{M}_{label(u)}(N) \geq 1$$

XSDNF:
Ein Schema $\mathcal{S} = (N, E)$ ist nun in der Normalform XSDNF, wenn folgende Bedingungen gelten:

$$\begin{aligned}
&\forall u \in children(\texttt{schema}) : \\
&\quad label(u) \in \{\texttt{complexType}, \texttt{simpleType}, \texttt{include}, \texttt{import}, \texttt{redefine}, \texttt{annotation}, \texttt{element}\} \\
&\quad \wedge \quad \mathfrak{M}_{\texttt{element}}(children(\texttt{schema})) = 1 \\
&\quad \wedge \quad \mathfrak{M}_{\texttt{complexType}}(children(\texttt{schema})) = \mathfrak{M}_{\texttt{complexType}}(N) \\
&\quad \wedge \quad \mathfrak{M}_{\texttt{simpleType}}(children(\texttt{schema})) = \mathfrak{M}_{\texttt{simpleType}}(N)
\end{aligned}$$

Tabelle 3.1: Bedingungen der Normalform XSDNF

3.1.2 Erläuterung

Aus der formalen Definition der XSDNF kann man den Aufbau der Normalform erkennen. Wie jedes Schema beginnt es mit dem Wurzelknoten `schema`, aber die einzigen erlaubten Kindknoten des `schema`-Knotens sind Typ-Knoten wie `simpleType` und `complexType`, `include`, `import`, `redefine` und zur Dokumentation der `annotation`-Knoten. Ebenso erlaubt ist der `element`-Knoten, es darf aber nur ein einziges Element global deklariert werden, eben das Wurzel-Element der Instanzdokumente, welches mit der Bedingung

$$\mathfrak{M}_{\texttt{element}}(children(\texttt{schema})) = 1$$

gefordert wird. Im Abschnitt über die grafische Notation (siehe 3.2) wird dieser Knoten dabei speziell gekennzeichnet. Alle anderen Konstrukte wie Attributgruppen und Substitutionsgruppen sind nicht erlaubt und müssen bei einer Transformation in die Normalform in adäquate Typ-Konstrukte umgeformt werden (siehe 3.4).

[2] Siehe 2.2.1.1 auf Seite 8

Durch die Bedingungen

$$\mathfrak{M}_{\texttt{complexType}}(children(\texttt{schema})) = \mathfrak{M}_{\texttt{complexType}}(N) \wedge$$
$$\mathfrak{M}_{\texttt{simpleType}}(children(\texttt{schema})) = \mathfrak{M}_{\texttt{simpleType}}(N)$$

wird gefordert, dass alle Typdefinitionen global erfolgen, also die Anzahl der Typknoten, welche direkte Kinder des schema-Knotens sind, muss gleich der Anzahl aller Typknoten des gesamten Schemas sein.
Diese Bedingung wird nicht für die `include`, `import` und `redefine`-Knoten gefordert, da sie per Definition von XML-Schema nur global deklariert sein dürfen.
Durch die Bedingung, dass einzig nur das Wurzelelement der Instanz global deklariert sein darf, werden auch Referenzen aus der Normalform entfernt, da Referenzen nur auf global deklarierte Elemente erfolgen darf (siehe 2.2.1.9). Einzige Ausnahme sind hierbei Rekursionen z.B. Referenzen auf das global deklarierte Element aus Typdefinitionen heraus, da aber Rekursionen in den meisten Fällen Schwierigkeiten bereiten, sollte man diese entfernen. Bei der Transformation in die Normalform kann eine Referenz auf das Wurzel-Element auch dadurch aufgelöst werden, indem man die Referenz durch eine Typzuweisung ersetzt (siehe 3.4).
Wenn weitere Schemata über den `include` und `import` Mechanismus eingefügt werden, wird zusätzlich gefordert, dass diese ebenfalls in der Normalform vorliegen müssen.

3.2 graphische Notation

Bei der formalen Definition war es von Vorteil, ein Schema als einen Baum zu betrachten, bei dem die einzelnen Knoten die Schema-Konstrukte waren. Abbildung 3.1 zeigt einen solchen Schema-Baum, welcher dem Schema in VBD aus dem Anhang A.4 entspricht. Man kann leicht erkennen, dass die Forderungen aus der Tabelle 3.1 erfüllt sind, somit ist dieses Schema in XSDNF.
Jedoch ist diese Form der grafischen Repräsentation nicht sehr aussagekräftig, wenn es um die Semantik der Definitionen geht, da man nur erkennen kann, wie oft zum Beispiel ein komplexer Typ definiert wurde, aber nicht, wie die Namen der Typen sind und welche Elemente diese Typen zugewiesen bekommen.
Aus diesem Grund wird für die grafische Notation der Schemata eine andere Variante eingeführt, welche die Semantik der Definitionen sichtbar macht. Dabei werden die einzelnen Konstrukte nicht als Knoten dargestellt, sondern als spezielle geometrische Form der Knoten und die Knotenbezeichner sind die Namen der definierten Typen und deklarierten Elemente und Attribute. Die Modellgruppen und Vererbungsarten werden ebenfalls erfasst und als Knoten dargestellt und die Typzuweisungen werden durch zusätzliche Kanten kenntlich gemacht.
Abbildung 3.2 zeigt eine kurze Übersicht über die verwendeten Symbole. So wird der `schema`-Knoten durch eine ausgefüllte Ellipse dargestellt, Elemente als Achteck und Attribute als Sechseck. Wie schon erwähnt, wird das Wurzel-Element oder root-Element der Instanzdokumente speziell ausgezeichnet und durch ein doppelt umrahmtes Achteck dargestellt. Die Modellgruppen wie `sequence, all` und `choice` werden als Ellipse dargestellt, genauso wie Elementgruppen und Attributgruppen. Typen werden, sofern sie als `simpleType` oder `complexType` definiert wurden, als Rechteck dargestellt, Built-In-Typen werden nicht als Knoten dargestellt, da sie die Komplexität des Schemabaumes nicht vergrößern. Wie Abbildung 3.3 darstellt, werden `include`, `import` und `redefine`-Knoten, sofern sie im Schema vorkommen, speziell ausgezeichnet und zusätzlich mit einem Rechteck umrahmt.
Referenzen werden durch eine Kantenbeschriftungen ausgewiesen, die Kanten welche auf die referenzierten Attribute und Elementen verweisen, bekommen den Schriftzug *ref.* Genauso ist es beim `key/keyref`-Mechanismus, bei dem die Kanten vom Schlüsselverweis zum Schlüssel hin mit *key* beschriftet werden. Typzuweisungen werden dagegen prinzipiell durch eine gepunktete Kante dargestellt.
Die Tabelle 3.2 zeigt nochmal eine Übersicht über die verwendeten Symbole der grafischen Notation. Dabei geben die in der letzten Tabellenspalte kursiv dargestellten Beschriftungen Schlüsselwörter aus XML-Schema an.

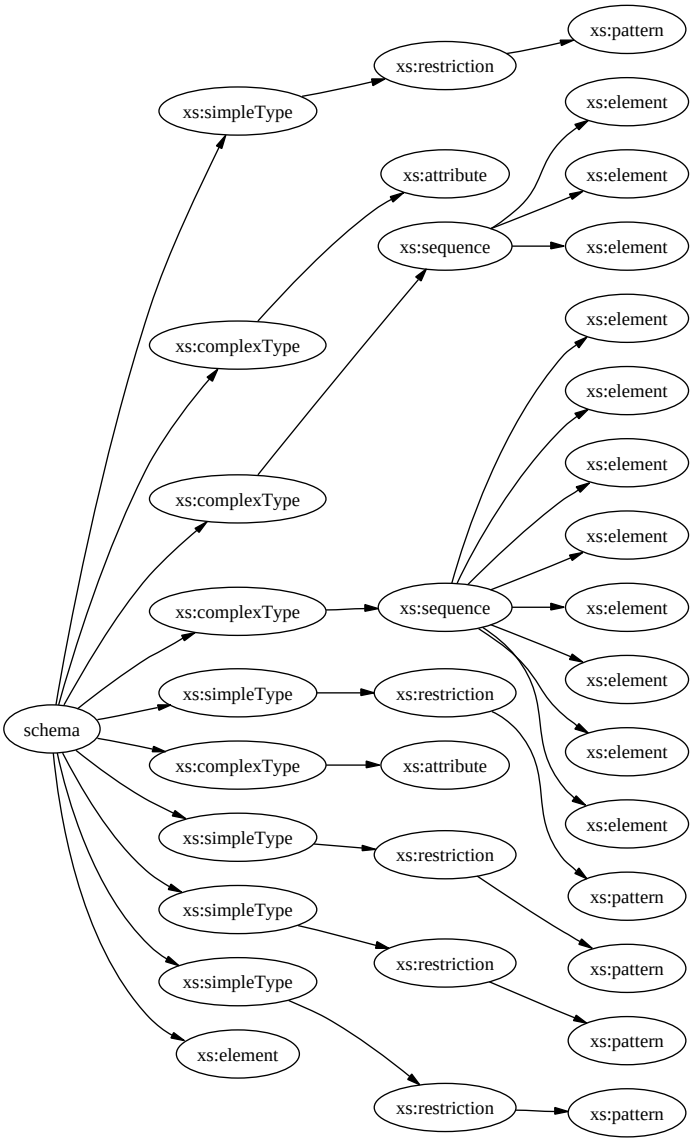

Abbildung 3.1: XML-Schema-Baum

Im Anhang B sind die grafischen Notationen der Schemata aus Anhang A aufgelistet.
Die Auswahl der erfassten Knoten in der grafischen Notation ist schon in Hinblick auf die Auswertung mittels Metriken geschehen, da z.B. alle aufgeführten Knoten die Komplexität des Schemas erhöhen.

3.3 Metriken auf XML-Schema

Es existieren zahllose Metriken in der Softwaretechnik, welche gewisse Maßzahlen sind, mit Hilfe deren man Aussagen über Komplexität und Güte von Softwareentwürfen treffen kann. Zusätzlich werden noch verschiedene Arten von Metriken unterschieden, welche ganz unterschiedliche Bereiche abdecken. So existieren Produktmetriken, welche ein Schema direkt beschreiben, Ressourcenmetriken, welche z.B. Speicherbedarf, Prozessorzeit zur Verarbeitung und weiteres beschreiben und es existieren Prozessmetriken, welche den Entwurfsprozess direkt beschreiben. Im folgenden werden nur Produktmetriken vorgestellt, welche Aussagen über Komplexität und Güte zulassen.

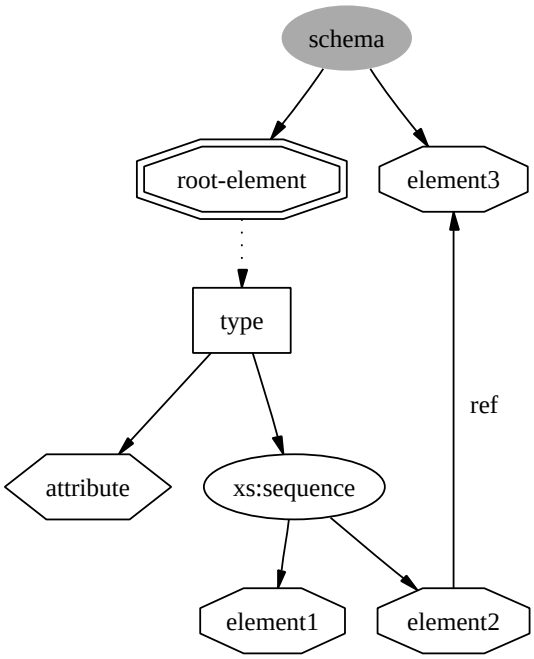

Abbildung 3.2: Legende der Symbole für die grafische Notation

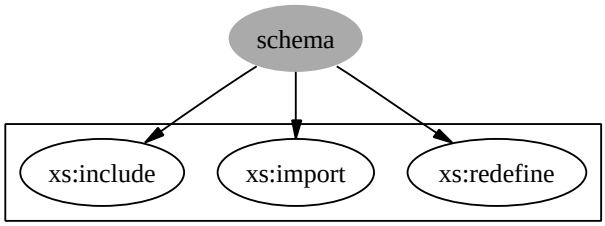

Abbildung 3.3: Kapselung von import und include in der grafische Notation

Diese Metriken kann man auch auf die Document Type Definition anwenden und somit auch für XML-Schema.

3.3.1 Metriken auf DTD-Dokumenten

Die folgend vorgestellten Metriken wurden aus [Sch02] entnommen und werden auf die XML-Schema-Definition angewendet.

3.3.1.1 Größe

Die erste Metrik ist auch gleich die einfachste und leitet sich aus der Software-Metrik *Lines of Codes* (LOC) ab. Wenn man diese Metrik auf DTD's anwendet, repräsentiert sie die Anzahl der Attribute, Elemente, Entities und Notationen und entspricht so in gewissem Sinne der Größe der DTD.

$$Size(DTD) = n_{EL} + n_A$$

Dabei ist n_{EL} die Anzahl der Elementdeklarationen in der DTD und n_A die Anzahl der Attributdeklarationen.

Komponente	Darstellung	Beschriftung
schema	ausgefüllte Ellipse	*schema*
Element	Achteck	Elementname
Instanzwurzel	doppelt umrahmtes Achteck	Elementname
Attribut	Sechseck	Attributname
definierte Typen	Rechteck	Typname
sequence	Ellipse	*sequence*
choice	Ellipse	*choice*
all	Ellipse	*all*
restriction	Ellipse	*restriction*
extension	Ellipse	*extension*
include	Ellipse	*include*
import	Ellipse	*import*
redefine	Ellipse	*redefine*
Attributgruppe	Ellipse	Gruppenname
Elementgruppe	Ellipse	Gruppenname
Substitutionsgruppe	Ellipse	Gruppenname
Referenz	Kantenbeschriftung	ref
Schlüsselbeziehung	Kantenbeschriftung	key
Typzuweisung	gepunktete Kante	

Tabelle 3.2: Übersicht über die verwendeten Symbole der grafischen Notation

Durch diese Berechnung entspricht diese Metrik gleichermaßen der Anzahl der Knoten im DTD-Baum.

3.3.1.2 Strukturkomplexität

Diese Metrik repräsentiert die Komplexität einer DTD indem hier der DTD-Baum ausgewertet wird und sich die Metrik wie folgt berechnet:

$$Compl(DTD) = e - n + 1$$

Dabei ist e die Anzahl der Kanten und n die Anzahl der Knoten des Baumes, wobei dem Baum zusätzliche Kanten hinzugefügt wurden, wann immer Quantifizierer in der Definition auftreten. Hinzu kommt, dass ID/IDREF-Beziehungen die Komplexität weiter erhöhen würden, so dass diese Metrik noch um die Anzahl der IDREF und IDREFS Attribute erweitert werden muss.

$$Compl(DTD) = e - n + 1 + n_{IDREF}$$

3.3.1.3 Strukturtiefe

Eine weitere Möglichkeit, um die Größe und Komplexität einer DTD herauszufinden, ist die Betrachtung der Strukturtiefe des DTD-Baumes.
Da bei dieser Metrik für jeden Knoten die Tiefe berechnet wird, müssen zuerst Rekursionen entfernt werden, da ansonsten die Metrik unendlich groß wird.
Dabei hat ein Blatt eine Tiefe von 0 und die Strukturtiefe jedes weiteren Knotens im DTD-Baum ist die maximale Tiefe seiner Kinder + 1. So berechnet sich die Tiefe für einen Knoten n wie folgt.

$$Depth(n) = \begin{cases} 0 & : n \text{ ist Blatt} \\ max(Depth(n_i)) + 1 & : n_i \text{ ist Kind von } n \end{cases}$$

3.3.1.4 Fan-In

Eine weitere Metrik, welche sich direkt aus dem DTD-Baum ableitet, ist die sogenannte Fan-In-Metrik, welche die Anzahl der Kindknoten der einzelnen Knoten beschreibt und somit nicht die Tiefe sondern die Breite des DTD-Baumes.

$$\mathit{Fan\text{-}In}(n) = \begin{cases} 0 & : n \text{ ist Blatt} \\ \mid \mathit{children}(n) \mid = \mid \{n_i \mid n_i \text{ ist Kindknoten von } n\} \mid & : \text{sonst} \end{cases}$$

daraus ergibt sich, dass Elemente mit einer größeren Fan-In-Metrik komplexer sind als andere.

3.3.1.5 Fan-Out

In gewissem Sinne gegensätzlich zur Fan-In-Metrik ist die Fan-Out-Metrik, da sie zur Berechnung nicht die Anzahl der Kindknoten heranzieht, sondern die Anzahl der Elternknoten. Dadurch beschreibt diese Metrik die Wiederverwendbarkeit von Elementdeklarationen.

$$\mathit{Fan\text{-}Out}(n) = \begin{cases} 0 & : n \text{ ist Wurzelknoten} \\ \mid \{n_i \mid n_i \text{ ist Elternknoten von } n\} \mid & : \text{sonst} \end{cases}$$

3.3.1.6 Zusammenfassung

Die Metriken der Größe *Size* und Komplexität *Compl* arbeiten global auf der gesamten DTD, während man für die Fan-In-Metrik und Fan-Out-Metrik zuerst die einzelnen Knoten berechnen muss, bevor man Aussagen über die DTD treffen kann.
Dabei lassen *Size* und *Compl* Aussagen über die Verständlichkeit, Verwendbarkeit und somit für die Benutzbarkeit und Instandhaltung zu. Die Metrik *Depth* hingegen trifft Aussagen über die Effizienz, da eine flache Tiefe des DTD-Baumes auch ein flaches Instanzdokument wiederspiegelt. Somit ist speziell bei Anfragesprachen, welche XPath zur Ansteuerung der Knoten nutzen, diese Metrik interessant, da bei flachen Hierarchien die XPath-Ausdrücke einfacher sind und somit effizient bearbeitbar. Dieses ist auch ein Grund, warum eine flache Hierarchie bei der XSDNF von Vorteil sein kann, wenn man XPath-Anfragen direkt an das Schema und nicht an die Instanz stellt.
Die Fan-Out-Metrik beschreibt dagegen die Wiederverwendbarkeit von Elementen und Elementdeklarationen und somit beschreibt sie auch die Veränderbarkeit, da Elemente mit einer hohen Fan-Out-Metrik schwerer anzupassen sind, wenn Änderungen an der DTD vorgenommen werden.

3.3.2 Metriken auf XML-Schema-Dokumenten

Bei der Adaption der Metriken von der DTD auf XML-Schema-Definitionen ist darauf zu achten, dass ein XML-Schema-Dokument wesentlich mehr Elemente beinhaltet und somit in der grafischen Notation auch einen wesentlich komplexeren Baum ergibt. Während bei der Definition eines Elementes in einer DTD ein Knoten im Baum hinzukam, ist bei einem XML-Schema noch der Typ des Elementes zu berücksichtigen, ob eine Wiederverwendung mittels Referenzen oder Vererbung stattgefunden hat, wieviele Attribute das Element besitzt etc. Weiterhin ist fraglich, ob die eingeführten Kanten zur Beschreibung der Typzugehörigkeit oder die Kanten für die Referenzen in den Metriken mit berücksichtigt werden sollen.

3.3.2.1 Größe

Damit die Größe den Schema-Baum ähnlich beschreibt, wie es bei der DTD der Fall ist, werden nicht die Elementdeklarationen gezählt, sondern die Knoten der grafischen Notation. Dabei werden Elementdeklarationen, Attributdeklarationen, Typdefinitionen unabhängig ob lokal oder global, Vererbungsarten und die Gruppierungen berücksichtigt.
Sei $\mathcal{S} = (N, E)$ ein Schema und liegt in der in Abschnitt 3.2 eingeführten graphischen Notation vor, wobei N die Knoten in der grafischen Repräsentation sind und E die dort vorkommenden

Kanten, dann berechnet sich die Größe wie folgt.

$$Size(\mathcal{S}) = \mid N \mid$$

3.3.2.2 Strukturkomplexität

Die Strukturkomplexität ergibt sich aus der Anzahl der Kanten und der Knoten. Jedoch erhöhen der key/keyref-Mechanismus und der ID/IDREF-Mechanismus die Komplexität der Struktur. Der ID/IDREF-Mechanismus wurde aus Kompatibilitätsgründen mit in den XML-Schema-Vorschlag aufgenommen und wird über Typen realisiert. Bei der DTD wirkt sich auch die Angabe von Quantifizierern aus, bei XML-Schema verläuft es analog mit den Häufigkeitsangaben durch die `minOccurs` und `maxOccurs`-Attributen bei den Elementdeklarationen und Elementgruppendeklarationen. Deswegen werden analog zur DTD-Metrik *Compl* ebenfalls zusätzliche Kanten für ID/IDREF-Beziehungen eingeführt und wenn `minOccurs` und `maxOccurs` Angaben erfolgen, werden ebenfalls zusätzliche Kanten eingezeichnet. Ebenso ist es bei den Referenzen und bei Schlüsseln, da diese aber schon in der grafische Notation berücksichtigt wurden, müssen keine neuen Kanten hinzugefügt werden.
In den Beispielen aus Anhang A wurde für das Element `Adresse` das Attribut `maxOccurs` auf 2 gesetzt, da Personen über einen Zweitwohnsitz verfügen können, bei dem deklarierten Element `Mobiltelefon` wurde `minOccurs` auf 0 und `maxOccurs` auf unbounded also unendlich gesetzt, da eine Person beliebig viele Mobiltelefone besitzen kann. Abbildung 3.4 zeigt den Baum des Beispiels im Russian Doll Design mit den zusätzlichen Kanten, welche dick eingezeichnet wurden.

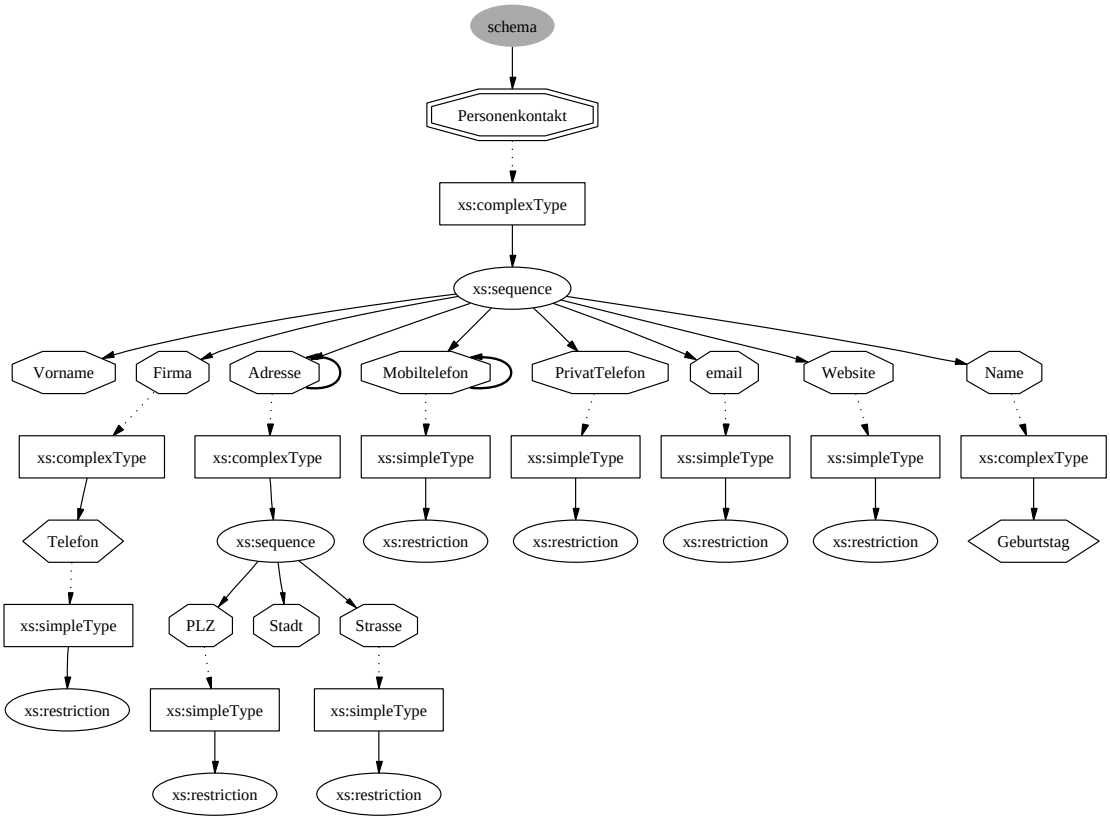

Abbildung 3.4: Beispiel vom RDD aus Anhang B mit zusätzlichen Kanten für minOccurs/maxOccurs

Die Strukturkomplexität berechnet sich für ein Schema $\mathcal{S} = (N, E)$ wie folgt.

$$Compl(\mathcal{S}) = \mid E \mid - \mid N \mid + (e_{Occurs} + e_{IDREF}) + 1$$

Dabei ist e_{Occurs} die Anzahl der zusätzlichen Kanten wegen den `minOccurs` und `maxOccurs` Attributen und e_{IDREF} ist die Anzahl der Kanten für die ID/IDREF-Beziehungen.
Für das Beispiel wäre die Komplexität $Compl(\mathcal{S}) = 34 - 35 + (2 + 0) + 1 = 2$.
Tabelle 3.3 zeigt eine Übersicht über die Beispiele aus dem Anhang B und alle dazu berechneten Metriken.

3.3.2.3 Strukturtiefe

Um die Strukturtiefe eines Schemas berechnen können, müssen zuerst die Rekursionen entfernt werden, da ansonsten die Metrik wie bei der DTD unendlich groß wird. Die Berechnung erfolgt dann analog wie bei der DTD. Referenzenkanten und Typkanten werden vernachlässigt, da die Referenzen und Typzuweisungen nur über Attribute realisiert werden und somit den Baum nicht vertiefen. Ausnahme sind lokal definierte Typen, wobei diese dann direkte Kinder der Elemente und Attribute sind und in der grafischen Notation direkt darunter angeordnet sind. In den Abbildungen aus Anhang B wurden die Ebenen der Baumstruktur dargestellt, beim Venetian Blind Design wurde der Übersichtlichkeit wegen eine Darstellung von links nach rechts verwendet.

$$Depth(n) = \begin{cases} 0 & : n \text{ ist Blatt} \\ max(Depth(n_i)) + 1 & : n_i \text{ ist Kind von } n \end{cases}$$

Eine weitere Möglichkeit zur Berechnung der Strukturtiefe wäre die in 3.1.1 eingeführte Baumlevelfunktion:

$$\forall u \in N \, \forall \, Pfade \, \rho : Depth(n) = max(level(u)) - 1 = max(|\rho|) - 1$$

3.3.2.4 Fan-In-Metrik

Die Fan-In-Metrik ist analog zur Metrik auf DTDs definiert und berechnet sich für ein Schema $\mathcal{S}$ wie folgt:

$$Fan\text{-}In(n) = \begin{cases} 0 & : n \text{ ist Blatt} \\ \mid children(n) \mid & : \text{sonst} \end{cases}$$

3.3.2.5 Fan-Out-Metrik

Die Fan-Out-Metrik berechnet sich wie bei der DTD aus der maximalen Anzahl der Elternknoten eines Knotens.

$$Fan\text{-}Out(n) = \begin{cases} 0 & : n \text{ ist Wurzelknoten} \\ \mid \{n_i \mid n_i \text{ ist Elternknoten von } n\} \mid & : \text{sonst} \end{cases}$$

Dabei werden die Typzuweisungen wie auch in der Fan-In-Metrik mit berücksichtigt, da die Fan-Out-Metrik die Wiederverwendbarkeit von definierten Komponenten beschreibt.

3.3.2.6 Zusammenfassung

Bei der DTD repräsentierte der DTD-Baum auch die Instanzdokumente und somit können die auf DTDs definierten Metriken auch auf die Instanzdokumente angewendet werden. Im Gegensatz dazu steht XML-Schema, da durch die vielfältigen Möglichkeiten bei der Deklaration von Elementen und Attributen ganz unterschiedliche Schemata die selben Instanzdokumente beschreiben. So definieren alle im Anhang aufgeführten Beispiele an Schemata die selbe Menge an Instanzdokumenten, eine mögliche Instanz ist in Anhang A.1 aufgeführt. Da die einzelnen Metriken auf die grafische Notation der Schemata angewendet werden, kann man sie nicht ohne weiteres für die Instanzdokumente benutzen. Die Ausnahme ist die Größe, welche man am leichtesten so anpassen kann, dass sie auch Aussagen für die Instanzen aus der XML-Klasse zulässt. Somit dienen die adaptierten Metriken nur für Vergleiche zwischen Schemata, welche die selben Instanzen definieren. Da aber die Metriken

auf die grafische Notation angewendet wurde und da die grafische Notation sehr unterschiedlich definiert werden kann, ist bei Vergleichen der Metriken von verschiedenen Schemata zu beachten, dass dieselbe Notation zugrunde liegt.
Tabelle 3.3 zeigt eine Übersicht über die berechneten Metriken für die Schemata aus Anhang A und ihren graphischen Darstellungen aus Anhang B.

Bsp	Size	Compl	Depth	Fan-In	Fan-Out
RDD (B.1)	35	2	9	8	1
SSD (B.2)	34	12	4	11	2
VBD (B.4)	31	12	3	10	3

Tabelle 3.3: Metriken der Beispiel-Schemata

3.3.3 Vergleich der Designprinzipien und der Normalform

Durch die errechneten Metriken der Designprinzipien von den Beispielen aus Anhang A lassen sich folgende Aussagen ableiten.

Durch die Metrik *Depth* erkennt man die Baumtiefe, welche bei dem Russian Doll Design am größten ist, was durch die lokalen Verschachtelungen der Deklarationen entsteht. Im Gegensatz dazu stehen das Salami Slice und das Venetian Blind Design. Das SSD hat im ungünstigsten Falle eine um eins größere Tiefe als das Pendant im Venetian Blind Design, wenn nämlich die deklarierten Elemente über einen selbst definierten lokalen Typen verfügen[3]. Beim VBD dagegen fällt die globale Deklaration der Elemente weg[4], statt dessen werden nur globale Typdefinitionen vorgenommen und aus diesem Grund ist die maximale Tiefe um eins kleiner als beim SSD. Einzige Ausnahme ist bei Verwendung der `redefine`-Klausel[5].

Wenn mehrere Elemente denselben Typen besitzen sollen, aber unterschiedliche Namen, dann kommt die Wiederverwendbarkeit zum Tragen, welche durch die *Fan-Out*-Metrik beschrieben wird. Dabei hat das VBD immer die größte Fan-Out-Metrik, da dieses Design die beste Wiederverwendbarkeit von Komponenten darstellt. Das SSD hat eine kleinere Fan-Out-Metrik, da bei Referenzen nicht nur der Typ sondern auch der Name übernommen wird und so für verschiedene Elemente des gleichen Typs aber unterschiedlichen Namens mehrere Elemente deklariert werden müssen und darauf verwiesen werden muss. Das Russian Doll Design verwendet deklarierte oder definierte Komponenten nur einmal, daher die Fan-Out-Metrik von 1.

Die *Fan-In*-Metrik beschreibt die maximale Anzahl an Kindelementen und da alle verwendeten Typen oder Elemente beim SSD und VBD global definiert und deklariert werden, haben diese beiden Designprinzipien im Vergleich zum RDD eine höhere Fan-In-Metrik.

Die Größe *Size* ist ebenfalls ein Indiz für die Wiederverwendung von Komponenten, deswegen wird in den meisten Fällen die Größe des VBD am geringsten sein, da hier viele Teile wiederverwendet werden und so die gesamte Größe des Schemas kleiner wird als z.B. beim RDD.

Bei der Metrik *Compl* spiegelt sich auch der Aufwand wieder, welcher investiert werden muss, damit eine hohe Wiederverwendung von Komponenten stattfindet oder damit eine geringe Baumtiefe erzielt wird. Deswegen hat das RDD die kleinste Komplexität.

[3] z.B. ρ=(`schema`,`element`,`complexType`,`sequence`,`element`)
[4] z.B. ρ=(`schema`,`complexType`,`sequence`,`element`)
[5] ein möglicher Pfad wäre dann ρ = (`schema`, `redefine`, `complexType`, `choice`, `element`)

Da die Normalform sehr an das Venetian Blind Design angelehnt ist, ergeben sich für die Normalform dieselben Metriken wie für das VBD. Die Komplexität ist weniger entscheidend als die geringe Baumtiefe und die hohe Wiederverwendbarkeit. Dadurch ergeben sich Vorteile bei Vergleichen zweier Schemata[6].

3.4 Transformation in die Normalform

Im folgenden Abschnitt wird gezeigt, wie XML-Schema-Dokumente in die Normalform umgeformt werden können. Die einzelnen XML-Schema-Konstrukte werden so umgeformt um die XSDNF zu erreichen, dabei wird zwischen zwei Arten von Konstrukten unterschieden. Die einen werden durch adäquate Konstrukte ersetzt, andere werden unter Umständen nur in der Baumhierarchie versetzt.

3.4.1 Umformungen

3.4.1.1 Typdefinitionen

Typdefinitionen werden in der Baumhierarchie so verschoben, dass sie nach der Transformation global definiert sind, also direkte Kinder des `schema`-Knotens sind. Sofern sie anonym definiert wurden, also über kein Namensattribut verfügen, wird neben der Verschiebung noch ein Name zugewiesen. Der Name ist dabei beliebig wählbar, der Einfachheit halber wird dem Typen der Name des Elementes oder Attributes zugewiesen. Dies widerspricht nicht der Forderung der Eindeutigkeit von Bezeichnern, da die Eindeutigkeit nur innerhalb eines Komponententyps gefordert wird. Deswegen gibt es keine Konflikte, wenn Typen und Elemente die selben Namen haben.
Das folgende Beispiel zeigt, wie ein lokal definierter anonymer Typ transformiert werden muss. Dabei sind die Umformungen kursiv hervorgehoben.

```
<xs:schema>
  <xs:element name="Name">
    <xs:complexType>
      <xs:attribute name="Geburtstag"
        use="required" type="xs:date"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<xs:schema>
  <xs:element name="Name" type="name"/>
    <xs:complexType name="name">
      <xs:attribute name="Geburtstag"
        use="required" type="xs:date"/>
  </xs:complexType>
</xs:schema>
```

Tabelle 3.4: Transformation von simplen oder komplexen Typen

Die Transformation in der grafischen Notation zeigt Abbildung 3.5, dabei ist die Verschiebung der Typdefinition durch einen roten gepunkteten Pfeil markiert.

3.4.1.2 Referenzen

Referenzen werden aufgelöst, indem aus der Elementdeklaration, auf die referenziert wird, ein simpler oder komplexer Typ gemacht wird. Dabei wird das `element`-Tag entfernt und dem Typ bei Bedarf ein Name gegeben. Die Elemente, die darauf referenzieren, bekommen den Namen des aufgelösten Elementes und der neu entstandene Typ wird zugewiesen.
Wenn das referenzierte Element oder Attribut als Typ nur eine Zuweisung besitzt, also keinen lokal definierten Typen, wird allen referenzierenden Elementen oder Attributen dieser Typ zugewiesen und der dazugehörige Name des referenzierten Elementes gegeben. Das referenzierte Element wird entfernt.

[6] Siehe 4.1

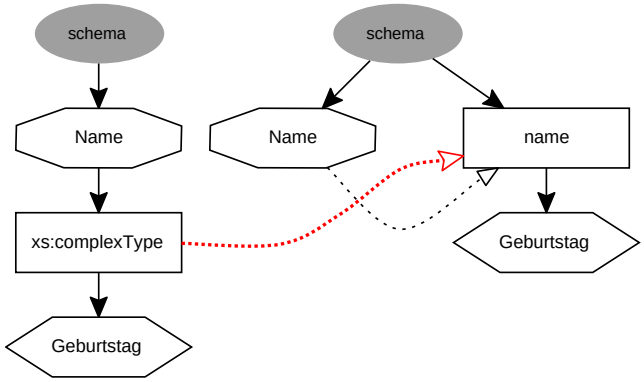

Abbildung 3.5: Typtransformation in grafischer Notation

Tabelle 3.5 zeigt die Auflösung einer Referenz auf ein Element mit komplexem Typ und Abbildung 3.6 zeigt die dazugehörige Transformation in graphischer Notation.
Da nur auf global deklarierte Elemente referenziert werden kann, muss nach der Umformung in eine Typdefinition keine Verschiebung in der Baumhierarchie erfolgen.

```
<xs:schema>
  <xs:complexType name="adresse">
    <xs:sequence>
      <xs:element ref="PLZ"/>
      ...
    </xs:sequence>
  </xs:complexType>

  <xs:element name="PLZ">
    <xs:simpleType>
      <xs:restriction>
          ...
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

```
<xs:schema>
  <xs:complexType name="adresse">
    <xs:sequence>
      <xs:element name="PLZ" type="plz"/>
        ...
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="plz">
    <xs:restriction>
      ...
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Tabelle 3.5: Auflösung von Referenzen

3.4.2 Ersetzungen

3.4.2.1 Attributgruppen

Attributgruppen werden bei der Transformation in die Normalform eliminiert und durch einen komplexen Typen ersetzt. Dabei werden die Attributdeklarationen unverändert übernommen, der Name der Attributgruppe wird zum Namen des Typen.
In dem Beispiel aus Tabelle 3.6 wird die Attributgruppe `itemInfoGroup` in einen komplexen Typen mit demselben Namen umgeformt. Zusätzlich wird der lokal definierte Typ vom Element `Produkt` umgeformt, in diesem Falle entspricht der neue globale Typ dem komplexen Typen, welcher aus der Eliminierung der Attributgruppe entstanden ist. Abbildung 3.7 verdeutlicht die Umformung noch einmal.

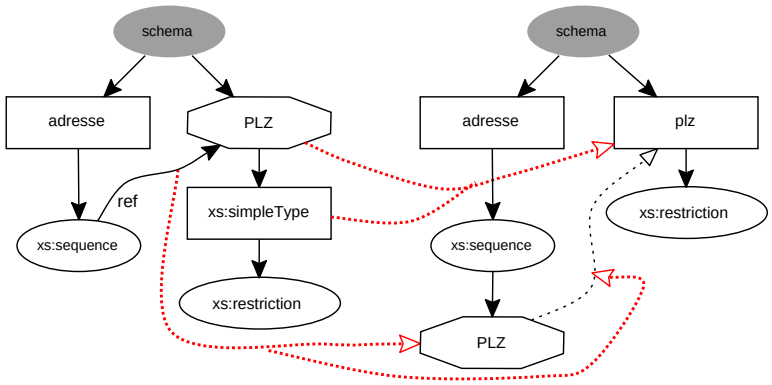

Abbildung 3.6: Referenzumformung in grafischer Notation

```
<xs:schema>
  <xs:attributeGroup
    name="itemInfoGroup">
    <xs:attribute name="Preis"
      type="xs:string"/>
    <xs:attribute name="Groesse"
      type="xs:string"/>
    <xs:attribute name="Farbe"
      type="xs:string"/>
  </xs:attributeGroup>
  <xs:element name="Produkt">
    <xs:complexType>
      <xs:attributeGroup
        ref="itemInfoGroup"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

```
<xs:schema>
  <xs:complexType
    name="itemInfoGroup">
    <xs:attribute name="Preis"
      type="xs:string"/>
    <xs:attribute name="Groesse"
      type="xs:string"/>
    <xs:attribute name="Farbe"
      type="xs:string"/>
  </xs:complexType>
  <xs:element name="Produkt"
    type="itemInfoGroup" />
</xs:schema>
```

Tabelle 3.6: Ersetzung von Attributgruppen

3.4.2.2 Elementgruppen

Elementgruppen umfassen mehrere Elementdeklarationen ähnlich wie es komplexe Typen können. Die möglichen Inhaltsmodelle sind dabei `choice`, `sequence` und `all`, welches dieselben sind, wie bei Typdefinitionen komplexer Typen. Nachdem eine Elementgruppe deklariert wurde, wird bei der Verwendung zum Beispiel aus einer `all`-Gruppe darauf referenziert.
Tabelle 3.7 zeigt links die Verwendung einer Elementgruppe und recht wie die Elementgruppendeklaration eliminiert werden kann, indem sie als komplexer Typ definiert wird. Es wird dann nicht mehr darauf verwiesen, sondern über Typzuweisung verwendet. Abbildung 3.8 zeigt dabei das Beispiel in grafischer Notation, die rot markierten Pfeile zeigen die Umformung in die Normalform hinein.

3.4.2.3 Substitutionsgruppen

Substitutionsgruppen bildet man, indem man mehrere Elemente deklariert und den Elementen zusätzlich den Namen eines weiteren Elementes als Wert des `substitutionGroup`-Attributs zuweist, gegen jenes die Elemente substituiert werden können. Dabei wird auf das Element, welches die Substitutionsgruppe bestimmt, referenziert. Dieses ist die gängigste Anwendung von Substitutionsgruppen.

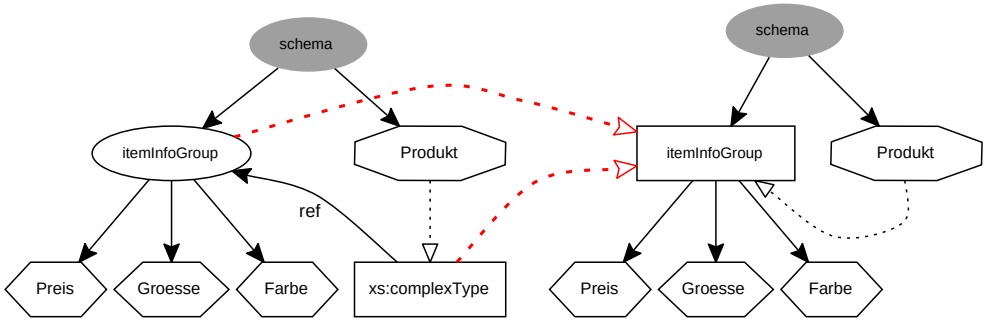

Abbildung 3.7: Eliminierung von Attributgruppen in grafischer Notation

```
<xs:schema>
<xs:group name="person">
    <xs:all>
        <xs:element name="Nachname"
            type="xs:string"/>
        <xs:element name="Vorname"
            type="xs:string" />
        <xs:element name="Geburtsdatum"
            type="xs:date" />
    </xs:all>
</xs:group>
<xs:element name="Person">
    <xs:complexType>
        <xs:all ref="person"/>
    </xs:complexType>
</xs:element>
</xs:schema>
```

```
<xs:schema>
<xs:complexType name="person">
    <xs:all>
        <xs:element name="Nachname"
            type="xs:string"/>
        <xs:element name="Vorname"
            type="xs:string" />
        <xs:element name="Geburtsdatum"
            type="xs:date" />
    </xs:all>
</xs:complexType>
<xs:element name="Person"
    type="person"/>
</xs:schema>
```

Tabelle 3.7: Ersetzung von Elementgruppen

Substitutionsgruppen werden nun aufgelöst, indem man die Referenz durch eine `choice`-Elementgruppe ersetzt. Dies ist möglich, da auf globaler Schemaebene keine Referenzierungen stattfinden können, da sonst die Eindeutigkeit des Wurzel-Elementes nicht erfüllt ist. Dadurch wird nur innerhalb komplexer Typen oder Elementgruppen auf Elemente referenziert. Einzige Ausnahme ist, wenn auf die Substitutionsgruppe nicht referenziert wird. Da Substitutionsgruppen nur global erfolgen, ist der Effekt auf das Instanzdokument derselbe, als wenn man bei den Elementen das Attribut `substitutionGroup` weglässt. Dann wird das Wurzelelement der Instanz variabel gehalten, alle global deklarierten Elemente können also Wurzelelement sein.
Im Beispiel in Tabelle 3.8 wird eine Substitutionsgruppe `phone` deklariert, zu der die drei Elemente `phone`, `usphone` und `intphone` gehören. Diese Gruppe wird nun aufgelöst, indem das referenzierende Element durch die `choice`-Elementgruppe, bestehend aus den drei Elementen, ersetzt wird. In der Abbildung 3.9 zeigen die rot markierten Pfeile, wie die Referenz und die Substitutionsgruppe `phone` zu einem `choice`-Element verschmolzen werden um die Substitutionsgruppe aufzulösen.

3.4.2.4 include/import/redefine

Da angenommen wird, dass bei einem Importieren anderer Schemata über `include`, `import` oder `redefine` diese importierten Schemata ebenfalls in Normalform vorliegen, ist es nicht notwendig, diese Komponenten zu entfernen. Deswegen sind in Tabelle 3.1 auf Seite 31 mit den Bedingungen für die XSDNF bei den möglichen `schema`-Kindelementen `include`, `import` und `redefine` auch enthalten. Das Problem bei der Auflösung der `include`-, `import`- oder `redefine`-Elemente ist

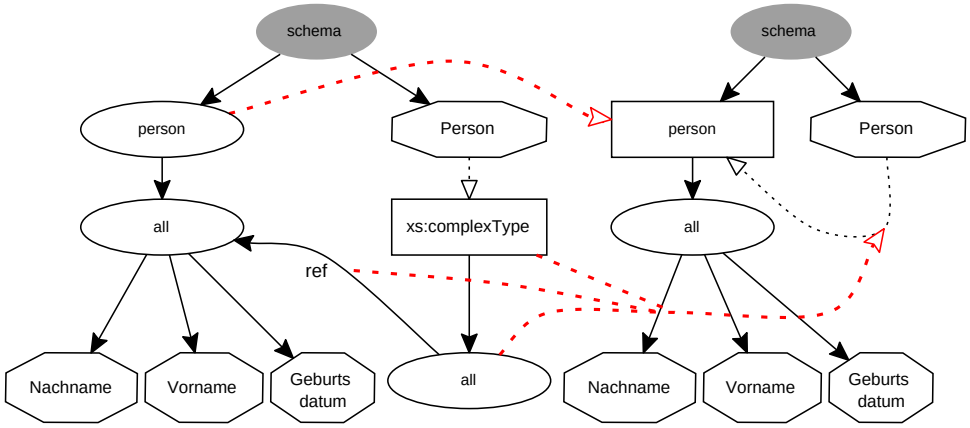

Abbildung 3.8: Eliminierung von Elementgruppen in grafischer Notation

```
<xs:schema>
  <xs:element name="phone"
    type="xs:string"/>
  <xs:element name="usphone"
    type="xs:string"
    substitutionGroup="phone/>
  <xs:element name="intphone
    type="xs:string"
    substitutionGroup="phone"/>
  ...
  <xs:complexType name="kontakt">
    <xs:sequence>              ...
      <xs:element ref="phone"/>
      ...
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

```
<xs:schema>
  ...
  <xs:complexType name="kontakt">
    <xs:sequence>
      ...
      <xs:choice>
        <xs:element
          name="phone"
          type="xs:string"/>
        <xs:element
          name="usphone"
          type="xs:string"/>
        <xs:element
          name="intphone
          type="xs:string"/>
      </xs:choice>
      ...
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Tabelle 3.8: Ersetzung von Substitutionsgruppen

das transitive Fortsetzen, da die eingebundenen Schemata wiederum weitere Schemata einbinden können, was möglicherweise auch ein endloser Prozess sein kann, wenn Schemata sich gegenseitig einbinden. Ebenso werden die so entstehenden Schemata sehr groß und aufgebläht, was unter anderem auch die Effizienz bei der Vergleichbarkeit schmälert.
Der Vollständigkeit halber wird eine Möglichkeit zur Auflösung von `include`- und `redefine`-Elementen aufgeführt. Da `import` auf Schemata mit verschiedenen Zielnamensräumen arbeitet, kann es nur sehr schwer eliminiert werden, da dort leicht Namensraumkonflikte auftreten können, wie zum Beispiel ein deklariertes Element mit dem selben Namen im gleichen Namensraum.
Im Gegensatz dazu können `include` und `redefine` nur auf Schemata angewendet werden, welche den selben Zielnamensraum festlegen.

include Der Effekt der `include`-Anweisung ist derselbe, als wenn man den Inhalt des eingebundenen Schemas in das einbindende hinein kopiert. Deswegen werden diese Schemata expandiert um die `include`-Anweisung aufzulösen, was bedeutet, dass alle Deklarationen und Definitionen des eingebundenen in das einbindende Schema übernommen werden. Durch die Festlegung, dass

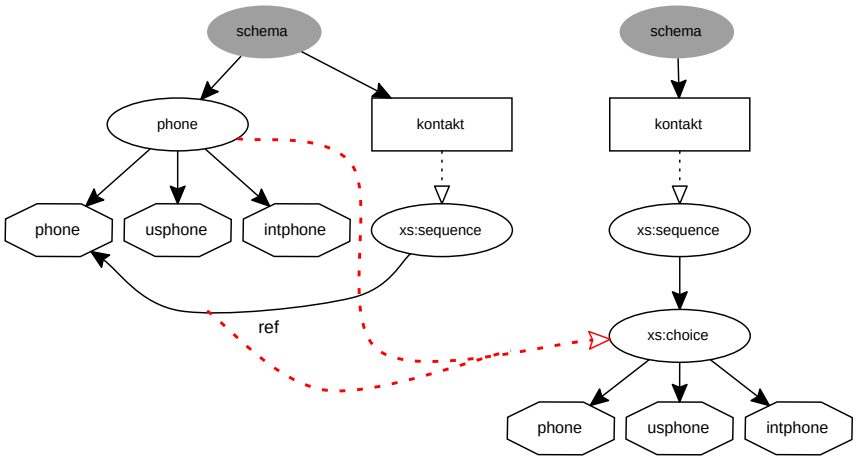

Abbildung 3.9: Eliminierung von Substitutionsgruppen in grafischer Notation

`include` nur auf Schemata mit gleichen Zielnamensräumen arbeitet, kommt es hier nicht zu Namensraumkonflikten.

redefine Der Effekt der `include`-Anweisung ist derselbe wie bei `redefine`, wenn man keine Deklarationen oder Definitionen innerhalb der `redefine`-Tags neu definiert. So wird das einbindende Schema genauso expandiert, wie bei einem `include`, jedoch werden die deklarierten oder definierten Komponenten, welche im einbindenden Schema redefiniert werden, durch diese neuen Definitionen ersetzt. Bei der Expansion von `include` und `redefine` gibt es nun zwei Abarbeitungsreihenfolgen

- zuerst expandieren und dann in die Normalform transformieren oder
- das eingebundene Schema zuerst in die Normalform überführen und dann expandieren oder einbinden.

3.5 Zusammenfassung

Durch die festgelegte Normalform erhalten XML-Schemata, welche in diese Normalform transformiert wurden, ein einheitliches Aussehen wie zum Beispiel, dass alle definierten Typen global definiert werden müssen, wodurch eine bessere Vergleichbarkeit ermöglicht wird (siehe 4.1). Ebenso ist die Normalform idempotent, was bedeutet, dass eine mehrfache Überführung in die Normalform dasselbe Ergebnis liefert. Würde man also die Transformation in die Normalform als mathematische Abbildung $Trans_{XSDNF}(schema.xsd)$ betrachten, so würde die Idempotenz wie folgt ausgedrückt werden,

$$Trans_{XSDNF}(Trans_{XSDNF}(schema.xsd)) = Trans_{XSDNF}(schema.xsd)$$

wobei schema.xsd ein beliebiges XML-Schema-Dokument darstellt.
Ein Problem bilden noch Rekursionen, welche man vor der Transformation eliminieren sollte. Wenn man die graphische Notation als Graphen ansieht, sind Rekursionen Zyklen in diesem Graphen. Eliminiert man diese Zyklen nicht, kann man bei dem Prozess der Referenzauflösung in eine Endlosschleife geraten, ebenso ist bei vorhandenen Rekursionen die Metrik *Depth*, also die Metrik bezüglich der Baumtiefe, nicht aussagekräftig, da sie unendlich wird. Rekursionen könnte man über XLink und XPointer (siehe [W3C01b]) auflösen, indem man dem Element, welches auf das Wurzelelement referenziert, den Typen `anyURI` zuweist, wobei der Wert dann eine URL auf ein anderes Dokument wäre, eben jenes mit dem Inhalt, auf das man referenzieren möchte. Denkbar

wäre auch, wenn man nicht auf das Wurzelelement verweist, dass man das ganze Schema in mehrere aufspaltet, welche dann unterschiedliche Instanzen beschreiben, wobei dann wieder der XLink-Mechanismus benutzt werden muss, um auf die Dokumente aus der anderen Instanz zu referenzieren.

4 Anwendungen

Hauptaufgabe der Normalform war es, eine bessere Vergleichbarkeit auf Schema-Ebene realisieren zu können. Diese Vergleichbarkeit ist wichtig, wenn man in föderierten Informationssystemen ganze XML-Klassen speichern möchte, da man so vorher überprüfen kann, ob diese XML-Klasse schon durch ein gespeichertes Schema beschrieben wird.

4.1 Vergleichbarkeit von Schemata

Wenn man zwei Schemata, welche man miteinander vergleichen möchte, in die Normalform überführt hat, hat man im Endeffekt zwei Bäume, sofern man die Kanten der Typzuweisungen in der graphischen Notation vernachlässigt. Durch die Metrik *Depth* sieht man, dass Schemata in der Normalform die geringste Baumtiefe haben, im Gegensatz zum RDD oder SSD, da nur Pfade ρ existieren mit ρ=*(schema, Typdefinition, Gruppe/Vererbung/Attribute, ..., Element/Attribute)*, wobei $Typdefinition$ für den Namen eines komplexen oder einfachen Typs steht und *Gruppe/Vererbung/Attribute* steht für eine Elementgruppe `all`, `sequence`, `choice` oder für die Vererbungsarten `extension`, `restriction` oder für deklarierte Attribute. Bei der Berechnung der Strukturtiefe *Depth* wird dabei der `schema`-Knoten vernachlässigt. Die einzige Ausnahme bildet die `redefine`-Klausel, wenn man andere Schemata einbindet und dort Teile neu definieren möchte. Wenn diese eingebundenen Schemata ebenfalls in Normalform vorliegen, dann kann man nur global definierte Typen umdefinieren, weswegen dann die Metrik *Depth* des einbindenden Schemas den selben Wert hat, wie ein Schema im Salami Slice Design.
In der Abbildung B.4 aus Anhang B erkennt man, dass alle definierten Typen ein Baumlevel von zwei haben. Deswegen reduziert sich der Vergleich zweier Schemata auf den Vergleich der definierten Typen, welche leicht in einer Liste organisiert werden können. Beim Vergleich dieser Listen muss jedoch jedes Element mit jedem der anderen verglichen werden, um herauszufinden, ob die Typen in den beiden Schemata gleich sind. Dabei wirkt es sich von Nachteil aus, dass die Typen variable Bezeichnungen besitzen können, so dass man die Typen nicht anhand ihres Namens miteinander vergleichen kann, sondern es müssen die einzelnen Teile der Typdefinition miteinander verglichen werden, wie Wertebereich, Anzahl der deklarierten Elemente oder Attribute und deren Typen und Inhaltsmodell wie `all`, `sequence` und `choice`.
Ein weiterer Nachteil ist, dass die Reihenfolge, wann die Typen im Schema definiert werden, nicht fest ist, sondern variabel ist. Dadurch muss man jeden Typen mit jedem vergleichen. Eine Verbesserung des Vergleichalgorithmus wäre die Definition einer Sortierung, wobei hier ebenfalls wegen der variablen Namen nicht die Namen der Typen als Sortierkriterium verwendet werden dürfen, sondern zum Beispiel Anzahl der deklarierten Elemente und Attribute oder ebenfalls das Inhaltsmodell. Dadurch könnte man den Algorithmus dahin verbessern, dass beide Listen jeweils nur einmal sequentiell durchlaufen werden müssen.
Zusätzlich müssen noch die Typzuweisungen überprüft werden, ob die selben Elemente die selben Typen zugewiesen bekommen. Dabei ist die Schwierigkeit, die in beiden Schemata gefundenen Elemente, welchen die selben Typen zugewiesen werden, zu vergleichen, da vielleicht der Name der Elemente oder Attribute unterschiedlich sein kann, aber der Typ gleich ist. Im Gegensatz zu den Typnamen sind die Elementnamen nicht syntaktisch sondern semantisch von Bedeutung. Ein Vergleich, ob die Semantik übereinstimmt, könnte man über ein Wörterbuch oder mittels Thesauren realisieren, in dem man nachschaut, ob die beiden unterschiedlichen Elementnamen die gleiche Bedeutung haben. Man kann auch andere Verfahren aus dem Information-Retrieval heranziehen, wie Stammwortreduktion, so dass man die Elementnamen auf ihre Stammform zurückführt und so die Semantik überprüfen kann.

Im Gegensatz dazu ist es aber auch denkbar, dass man Schemata, welche die selben Typen definieren, aber mit Elementen oder Attributen mit unterschiedlichen Bezeichnungen als allgemein unterschiedlich annimmt, da im Endeffekt die unterschiedlichen Namen der Elemente auch andere Elementnamen in den Instanzen sind, weswegen die Instanzen auch syntaktisch nicht übereinstimmen.

4.2 Abbildung von XML-Schemata auf Datenbankentwürfe

4.2.1 Abbildung auf relationale Datenbankentwürfe

Bei der Abbildung eines XML-Schemas auf einen Datenbankentwurf für relationale Datenbanken hat man die Schwierigkeit, ein hierarchisches Schema auf eine flache Relation, also einer Relation in erster Normalform[1], adäquat abzubilden. Ebenso ist die Schwierigkeit, dass Beziehungen von Datensätzen untereinander nur über eine Schlüssel/Fremdschlüssel-Beziehung darstellbar sind. Dieses ist zwar in XML-Schema integriert, siehe dazu 2.2.4.4, aber es muss bei der Abbildung eines Schemas in eine Relation die Hierarchie aufgebrochen werden, weswegen zusätzliche Schlüssel/Fremdschlüssel eingeführt werden müssen. Dieses ist notwendig, da jede weitere Hierarchiestufe in der Instanz eine neue Tabelle erfordert.
Allgemein bildet man XML-Elemente und XML-Attribute auf die Attribute einer Relation ab. Angaben bei den `minOccurs` und `maxOccurs`-Attributen der Elementdeklarationen oder bei dem `use`-Attribut bei der Attributdeklaration bestimmen bei den Tabellendefinitionen, ob Nullwerte erlaubt sind oder nicht.
Tabelle 4.1 zeigt die Abbildung des Beispiels im VBD aus Anhang A.4 auf eine Tabellendefinition.

```
create table Personenkontakt{
    Name            varchar(40) NOT NULL,
    Geburtstag      date  NOT NULL,
    Vorname         varchar(40) NOT NULL,
    Firma           varchar(40) NOT NULL,
    Telefon         varchar(40) NOT NULL,
    AdresseID       ID          NOT NULL,
    Mobiltelefon    varchar(40),
    PrivatTelefon   varchar(40),
    email           varchar(40) NOT NULL,
    Website         varchar(40) NOT NULL,
    FOREIGN KEY(AdresseID)
        REFERENCES Adresse (AdresseID)
        }
```

```
create table Adresse{
        AdresseID     ID,
        Strasse     varchar(40) NOT NULL,
        PLZ         varchar(40) NOT NULL,
        Stadt       varchar(40) NOT NULL,
        PRIMARY KEY (AdresseID)
        }
```

Tabelle 4.1: Abbildung auf ein relationales DB-Schema

Es existieren jedoch viele Probleme bei der Abbildung eines Schemas auf eine Relationendefinition. Wenn man keine erweiterbare Datenbank hat, kann man keine eigenen Typen definieren. Deswegen muss man bei den simplen Typdefinitionen auf den Basisdatentyp zurückgreifen, von dem diese Typen erben. Aus diesem Grund hat die Spalte `PrivatTelefon` bei der Definition in Tabelle 4.1 den Typ `varchar`, obwohl im Schema dafür ein eigener Typ definiert wurde. Ein weiteres Problem ist, wie man mit Attributen umgeht, ob man diese in eine eigene Tabelle ablegt oder wie hier im Beispiel, die Attribute neben ihren Elementen in ein und derselben Tabelle speichert.
Bei komplexen Typen, welche weitere Elemente deklarieren, legt man eine neue Tabelle an und speichert darin die deklarierten Elemente und in der Haupttabelle speichert man eine ID oder einen anderen Schlüssel, damit man die Informationen über Verbunde wieder herleiten kann. Dabei geben die Gruppen an, wie die Definition auszusehen hat, bei `sequence` und `all` müssen alle

[1] siehe [HS00]

Elemente darin vorkommen, bei `choice` dagegen nur eines der deklarierten Elemente. Deswegen müssen bei der Definition der Tabellen bei einer `choice`-Gruppe oder ansonsten bei `minOccurs` bzw. `maxOccurs` Angaben oder bei Verwendung des `nillable`-Attributes Nullwerte zugelassen werden. Bei der `sequence`-Gruppe ist die Reihenfolge der deklarierten Elemente von entscheidender Bedeutung, diese Reihenfolge kann man aber in den Tabellen einer relationalen Datenbank nicht aufrechterhalten.

4.2.2 Abbildung auf objekt-relationale Datenbankentwürfe

Im Gegensatz zu relationalen Datenbanken kann man bei objektrelationalen Datenbanken (kurz ORDB) die Hierarchie adäquat abbilden, da bei ORDB geschachtelte Relationen erlaubt sind. Ebenso hat man bei ORDB die Möglichkeit eigene Typen zu definieren, weswegen das Abbilden des Schemas einfacher ist.

Durch die Möglichkeit der Typdefinitionen ist die Abarbeitungsreihenfolge anders als bei relationalen Datenbanken, wo man mit dem Instanzwurzelelement begonnen hat und eine Tabelle angelegt hat und bei jedem Element sich dessen Typ angeschaut hat. Bei einem komplexen Typen wurde dann bei Bedarf eine neue Tabelle angelegt und Schlüssel und Fremdschlüsselwerte eingesetzt.

Bei ORDB kann man das Schema parsen und die Typdefinitionen mit gewissen Einschränkungen auf Typdefinitionen der ORDB abbilden. Analog zu den relationalen Datenbanken existieren so Einschränkungen zum Beispiel bei einfachen Typdefinitionen, welche reguläre Ausdrücke oder andere Pattern benutzen, um den Wertebereich einzuschränken. In diesem Fall muss man wieder auf den Basisdatentyp zurückgreifen, von dem geerbt wird. Ist dieser Basisdatentyp wieder ein definierter einfacher Typ, muss auf dessen Basistyp zurückgegriffen werden. Bei Bedarf wird so rekursiv durchgegangen, bis man einen Built-In-Typen hat, welchen man adäquat auf einen Datenbanktypen abbilden kann.

Nach den Typdefinitionen werden den Elementen bei Bedarf genau wie im Schema diese Typen zugewiesen, so dass sich ein ähnliche Struktur zum Schema ergibt. Dieses wird in Tabelle 4.2 verdeutlicht, wo das Beispiel aus Anhang A.4 in eine Tabellendefinition[2] einer ORDB abgebildet wurde.

Im Beispiel erkennt man das Vorgehen bei definierten einfachen Typen, dessen Restriktionen man nicht adäquat abbilden kann. In diesem Falle wurde der Basistyp genommen und überall auch dort verwendet, wo man den definierten Typen zuweist. Deswegen sind in dem Beispiel aus Tabelle 4.2 weniger definierte Typen als noch im XML-Schema.

4.2.3 Abbildung auf objekt orientierte Datenbankentwürfe

Da bei objektorientierten Datenbanken (kurz OODB) ebenfalls Hierarchien unterstützt werden, läuft die Abbildung eines Schemas ähnlich der Abbildung bei ORDB.

Es spielt jedoch keine Rolle, ob man für jeden Typ eine Klasse der OODB definiert und dann mit Komponentenobjekten arbeitet oder ob man die Definitionen lokal vornimmt, so dass man komplexe Objekte bekommt. Aus diesem Grund ist es nicht wichtig, ob ein Schema in Normalform vorliegt oder nicht. Wenn man nun das Beispiel in VBD aus Anhang A.4 so abbildet, wie es aufgebaut ist, arbeitet man mit Komponentenobjekten, welche durch die Klassen-Komponentenklassen-Beziehung zusammengeführt werden. Im Beispiel[3] aus Tabelle 4.3 wäre das Objekt `Adresse` der Klasse `Personenkontakt` ein Komponentenobjekt.

Die Schwierigkeiten bei der Abbildung von den einfachen Typen sind genauso wie bei den objektrelationalen Datenbanken, da diese Restriktionen nicht adäquat durch Klassendefinitionen dargestellt werden können.

[2] die verwendete Syntax wurde aus [SM99] übernommen
[3] die Syntax ist aus [Heu97] entnommen

```
create type name_t (
    Name        varchar(40),
    Geburtstag  date)

create type firma_t (
    Firma       varchar(40),
    Telefon     varchar(40))

create type adresse_t (
    Strasse     varchar(40),
    PLZ         varchar(40),
    Stadt       varchar(40))

create type personenKontakt_t (
    Name            name_t,
    Vorname         varchar(40),
    Firma           firma_t,
    Adresse         adresse_t,
    Mobiltelefon    varchar(40),
    PrivatTelefon   varchar(40),
    email           varchar(40),
    Website         varchar(40))

create table Personenkontakt of type personenKontakt_t;
```

Tabelle 4.2: Abbildung auf ein objekt-relationales DB-Schema

4.3 Abbildung von XML-Schemata auf Java-Klassen

Bei der Abbildung auf Java-Klassen ist es ebenso wie bei OODB nicht von Wichtigkeit, ob das Schema in Normalform vorliegt oder nicht. Bei Java-Klassen speziell ist es teilweise von Vorteil, innere Klassen zu definieren, ähnlich zu lokalen anonymen Typdefinitionen in XML-Schema. Es gäbe die Möglichkeit über ein annotiertes Schema festzulegen, welche definierten Typen dann auf innere Java-Klassen abgebildet werden sollen ähnlich den annotierten Schemata von Oracle, welche dazu dienen, die Typen von XML-Schema adäquat auf die speziellen Datentypen von Oracle abzubilden. Ein annotiertes Schema erreicht man, indem man speziell entwickelte Schemata über die Namensraumdefinitionen einbindet und so nutzen kann, jedoch müssen Parser diese Erweiterungen verarbeiten können.
Wenn man nun lediglich normale Klassen benötigt, bietet sich ein Schema in Normalform an, bei Bedarf von inneren Klassen muss man die Normalform aufbrechen und Typen auch lokal definieren.

```
CLASS name
    ATTRIBUTES
        Name:STRING,
        Geburtstag:DATE

CLASS firma
    ATTRIBUTES
        Firma:STRING,
        Telefon:STRING

CLASS adresse
    ATTRIBUTES
        Strasse:STRING,
        PLZ:STRING,
        Stadt:STRING

CLASS personenKontakt
    ATTRIBUTES
        Name:name,
        Voranme:STRING,
        Firma:firma,
        Adresse:adresse,
        Mobiltelefon:STRING,
        PrivatTelefon:STRING,
        email:STRING,
        Website:STRING
```

Tabelle 4.3: Abbildung auf ein objekt-orientiertes DB-Schema

```
import java.util.*;
import java.lang.*;

class name {
        public String   Name;
        public Date     Geburtstag;
}
class firma {
        Firma:STRING,
        Telefon:STRING
}
 class adresse {
        public String Strasse;
        public String PLZ;
        public String Stadt;
}
public class personenKontakt {
        public name     Name;
        public String   Voranme;
        public firma    Firma;
        public adresse  Adresse;
        public String   Mobiltelefon;
        public String   PrivatTelefon;
        public String   email;
        public String   Website;
        }
```

Tabelle 4.4: Abbildung auf Java-Klassen

5 Schlußbetrachtung

Eine Möglichkeit zur Strukturdarstellung von XML ist XML-Schema, welche es zusätzlich erlaubt, XML-Dokumente gegen ein zugehöriges Schema zu validieren und so die Korrektheit zu prüfen. Jedoch ist die Darstellung mittels Schema sehr vielseitig, so kann man Typinformationen in Typdefinitionen oder über Element- und Attributdeklarationen darstellen. Ebenso kann man Typen global mit Namen definieren oder lokal und anonym. Ziel dieser Arbeit war es, eine Normalform zu definieren, in die man beliebige Schemata überführen kann, um so eine einheitliche Struktur zu erhalten. Einsatzmöglichkeiten für dieses Verfahren oder diese Normalform ist die Vergleichbarkeit von Schemata, um herauszufinden, welche identisch sind und somit die selben XML-Klassen beschreiben. Dies ist besonders in föderierten Informationssystemen von Bedeutung.

5.1 Zusammenfassung

Diese Arbeit gibt eine grobe Einführung in XML-Schema und gewissen Designprinzipien, wodurch die Unterschiede und Vielfältigkeiten in der Erstellung von Schemata aufgedeckt werden.
Es wird nun in dieser Arbeit ein Vorschlag für eine Normalform gegeben und zusätzlich Transformationschritte, welche anhand einer graphischen Notation verdeutlicht werden, um von beliebigen Schemata in die Normalform zu gelangen. Dabei erfüllt die Normalform die Eigenschaft, dass sie inhaltserhaltend ist, das bedeutet, dass nach einer Transformation dieselben XML-Klassen beschrieben werden. Ebenso ist die Normalform idempotent, so dass man bei mehrfacher Anwendung der Transformationschritte immer zu derselben Normalform gelangt.
Abschließend wurde ein Überblick gegeben, wo man diese Normalform einsetzen kann und inwiefern sich die Vergleichbarkeit dadurch leichter realisieren lässt.
Bei Anwendungsgebieten wie der automatischen Abbildung eines Schemas, welches in Normalform vorliegt, in ein Datenbankdesign oder eine Klassenhierarchie einer objektorientierten Programmiersprache ergeben sich dieselben Probleme wie bei beliebigen Schemata, da man bei dieser Abbildung wiederum mehrere Varianten zur Auswahl hat. Ebenso bestehen Schwierigkeiten darin, dass man Schema-Konstrukte nicht adäquat abgebildet werden können.

5.2 Ausblick

Um eine allumfassende Bewertung der “Normalform für XML-Schema” durchführen zu können, wäre es denkbar:

- Nicht nur Produktmetriken, wie vorgestellt, zu betrachten, sondern Ressourcenmetriken auf XML-Schema zu definieren und zu betrachten, wie z.B. die Verarbeitungszeit durch einen DOM- oder SAX-Parser oder die Effizienz beim Zugriff mittels XQuery.
- Implementierung eines Prototypen zur automatischen Transformation, wobei unter anderem XSLT möglich wäre, aber auch DOM-Parser, über die man die Baumstruktur verändern kann.
- Effizienzüberprüfung der Normalform mit beliebigen Schemata, was die Vergleichbarkeit angeht, so wäre die Implementation eines Prototypen zum Vergleichen zweier Schemata denkbar.

Anhang A (Beispiel Kontakt)

Nachfolgend ist ein komplettes Beispiel eines XML-Dokumentes und der möglichen dazugehörigen Schemata abgebildet. Die einzelnen Schemata liegen im Russian Doll, Salami Slice und Venetian Blind Design vor. Im Anhang B sind die zugehörigen Darstellungen in Baum-Notation zu finden.

A.1 XML-Dokument

```
<PersonenKontakt
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="sa_beispiel_salami.xsd">
    <Name Geburtstag="1970-01-01">Name</Name>
    <Vorname>Vorname</Vorname>
    <Firma Telefon="012/34567">Firma</Firma>
    <Adresse>
        <Strasse>Strasse 12a</Strasse>
        <PLZ>98765</PLZ>
        <Stadt>Rostock</Stadt>
    </Adresse>
    <Mobiltelefon>0170/1122990</Mobiltelefon>
    <PrivatTelefon>012/897334</PrivatTelefon>
    <email>email@www.com</email>
    <Website>http://www</Website>
</PersonenKontakt>
```

A.2 Schema im Russian Doll Design

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
 <xs:element name="PersonenKontakt">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="Name">
     <xs:complexType mixed="true">
      <xs:attribute name="Geburtstag" type="xs:date"/>
     </xs:complexType>
    </xs:element>
    <xs:element name="Vorname" type="xs:string"/>
    <xs:element name="Firma">
     <xs:complexType mixed="true">
      <xs:attribute name="Telefon">
       <xs:simpleType>
        <xs:restriction base="xs:string">
         <xs:pattern value="\d{3,6}/\d+"/>
        </xs:restriction>
       </xs:simpleType>
      </xs:attribute>
     </xs:complexType>
    </xs:element>
    <xs:element name="Adresse" maxOccurs="2">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Strasse">
        <xs:simpleType>
         <xs:restriction base="xs:string">
          <xs:pattern value="[a-zA-Z\-.]+ \d{1,3}[a-z]?"/>
```

```
        </xs:restriction>
       </xs:simpleType>
      </xs:element>
      <xs:element name="PLZ">
       <xs:simpleType>
        <xs:restriction base="xs:string">
         <xs:pattern value="\d{5}"/>
        </xs:restriction>
       </xs:simpleType>
      </xs:element>
      <xs:element name="Stadt" type="xs:string"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name="Mobiltelefon" nillable="true" minOccurs="0" maxOccurs="unbounded">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:pattern value="\d{3,6}/\d+"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:element>
   <xs:element name="PrivatTelefon" nillable="true">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:pattern value="\d{3,6}/\d+"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:element>
   <xs:element name="email">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:pattern value="[\w.-_]*@[\w.-_]*"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:element>
   <xs:element name="Website">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:pattern value="http://[\w._-/]*"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

A.3 Schema im Salami Slice Design

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
 <xs:element name="Name">
  <xs:complexType mixed="true">
   <xs:attribute name="Geburtstag" type="xs:date"/>
  </xs:complexType>
 </xs:element>
 <xs:attribute name="Telefon">
  <xs:simpleType>
   <xs:restriction base="xs:string">
    <xs:pattern value="\d{3,6}/\d+"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
 <xs:element name="Firma">
  <xs:complexType mixed="true">
```

```
  <xs:attribute ref="Telefon"/>
 </xs:complexType>
</xs:element>
<xs:element name="Strasse">
 <xs:simpleType>
  <xs:restriction base="xs:string">
   <xs:pattern value="[a-zA-Z\-.]+ \d{1,3}[a-z]?"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
<xs:element name="PLZ">
 <xs:simpleType>
  <xs:restriction base="xs:string">
   <xs:pattern value="\d{5}"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
<xs:element name="Adresse">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="Strasse"/>
   <xs:element ref="PLZ"/>
   <xs:element name="Stadt" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="Mobiltelefon">
 <xs:simpleType>
  <xs:restriction base="xs:string">
   <xs:pattern value="\d{3,6}/\d+"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
<xs:element name="PrivatTelefon">
 <xs:simpleType>
  <xs:restriction base="xs:string">
   <xs:pattern value="\d{3,6}/\d+"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
<xs:element name="email">
 <xs:simpleType>
  <xs:restriction base="xs:string">
   <xs:pattern value="[\w.-_]*@[\w.-_]*"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
<xs:element name="Website">
 <xs:simpleType>
  <xs:restriction base="xs:string">
   <xs:pattern value="http://[\w._-/]*"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
<xs:element name="PersonenKontakt">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="Name"/>
   <xs:element name="Vorname" type="xs:string"/>
   <xs:element ref="Firma"/>
   <xs:element ref="Adresse" maxOccurs="2"/>
   <xs:element ref="Mobiltelefon" nillable="true" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element ref="PrivatTelefon" nillable="true"/>
   <xs:element ref="email"/>
   <xs:element ref="Website"/>
  </xs:sequence>
 </xs:complexType>
```

```
 </xs:element>
</xs:schema>
```

A.4 Schema im Venetian Blind Design

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">
    <xs:complexType name="name" mixed="true">
        <xs:attribute name="Geburtstag" type="xs:date"/>
    </xs:complexType>
    <xs:complexType name="firma" mixed="true">
        <xs:attribute name="Telefon" type="telefon"/>
    </xs:complexType>
    <xs:simpleType name="strasse">
        <xs:restriction base="xs:string">
            <xs:pattern value="[a-zA-Z\-.]+ \d{1,3}[a-z]?"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="plz">
        <xs:restriction base="xs:string">
            <xs:pattern value="\d{5}"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:complexType name="adresse">
        <xs:sequence>
            <xs:element name="Strasse" type="strasse"/>
            <xs:element name="PLZ" type="plz"/>
            <xs:element name="Stadt" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="telefon">
        <xs:restriction base="xs:string">
            <xs:pattern value="\d{3,6}/\d+"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="email">
        <xs:restriction base="xs:string">
            <xs:pattern value="[\w.-_]*@[\w.-_]*"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="website">
        <xs:restriction base="xs:string">
            <xs:pattern value="http://[\w._-/]*"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:complexType name="personenKontakt">
        <xs:sequence>
            <xs:element name="Name" type="name"/>
            <xs:element name="Vorname" type="xs:string"/>
            <xs:element name="Firma" type="firma"/>
            <xs:element name="Adresse" type="adresse" maxOccurs="2"/>
            <xs:element name="Mobiltelefon" type="telefon" nillable="true"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="PrivatTelefon" type="telefon" nillable="true"/>
            <xs:element name="Email" type="email"/>
            <xs:element name="Website" type="website"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="PersonenKontakt" type="personenKontakt"/>
</xs:schema>
```

Anhang B (grafische Notation)

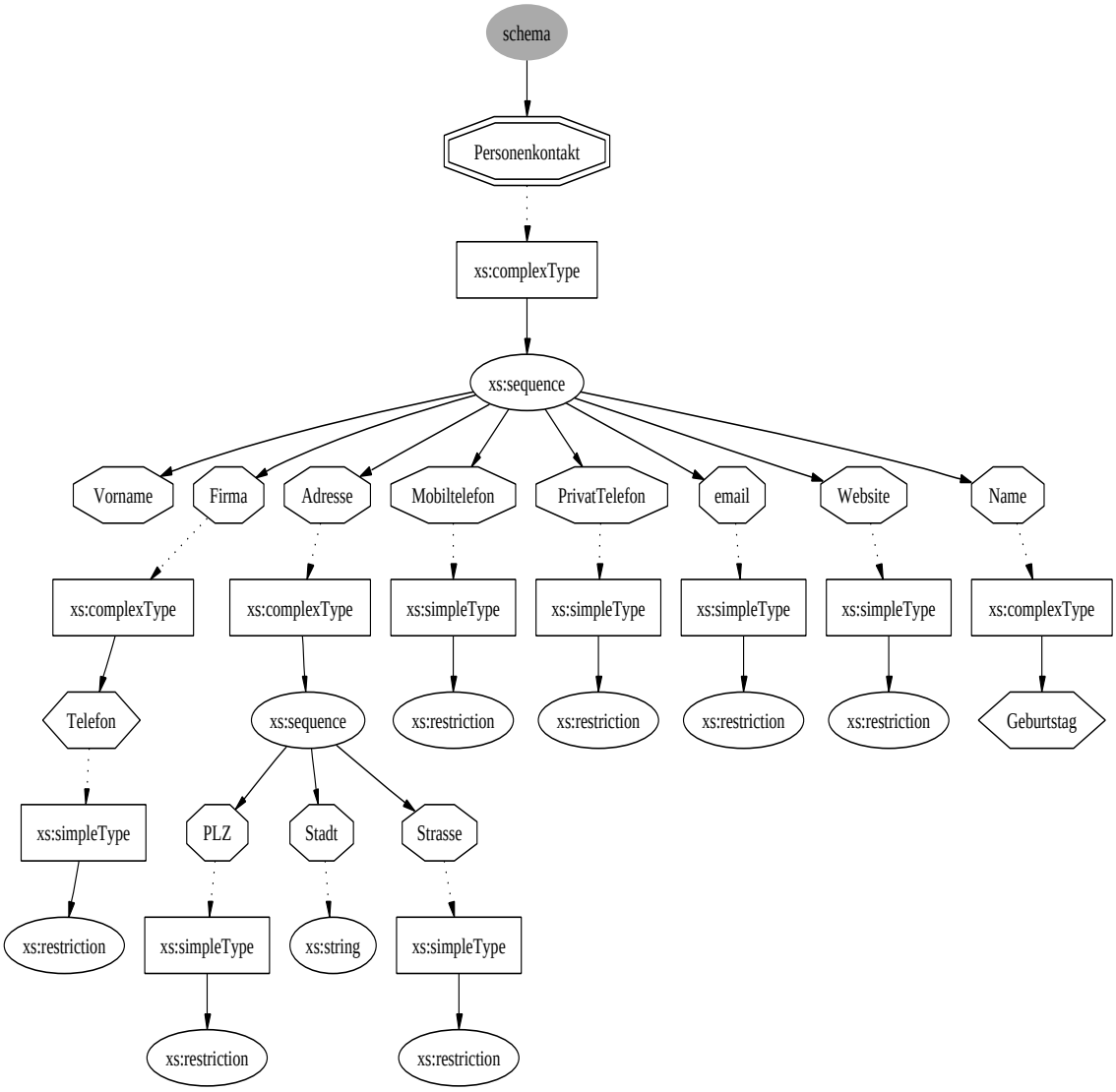

Abbildung B.1: Russian Doll Design in grafischer Notation

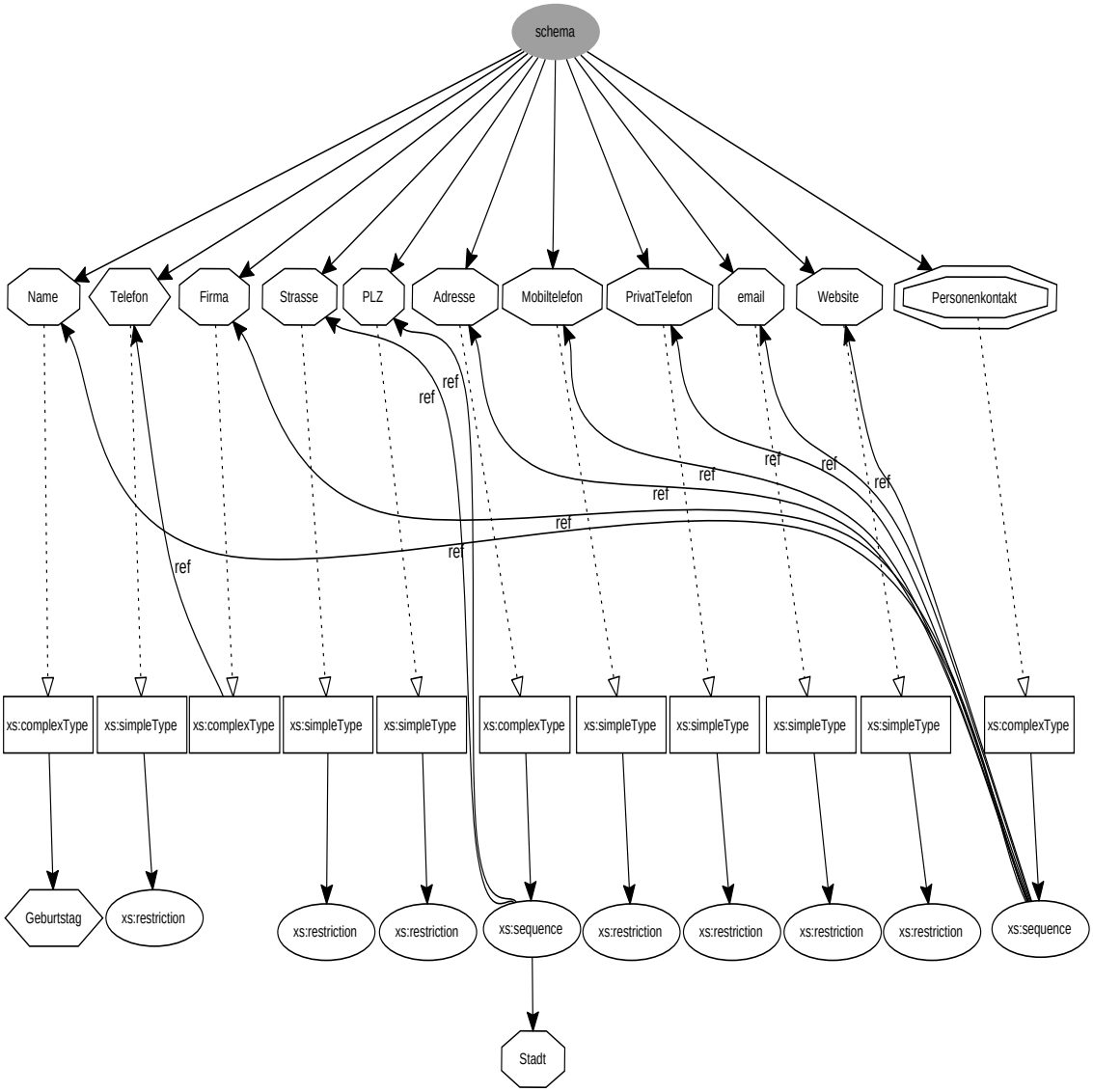

Abbildung B.2: Salami Slice Design in grafischer Notation

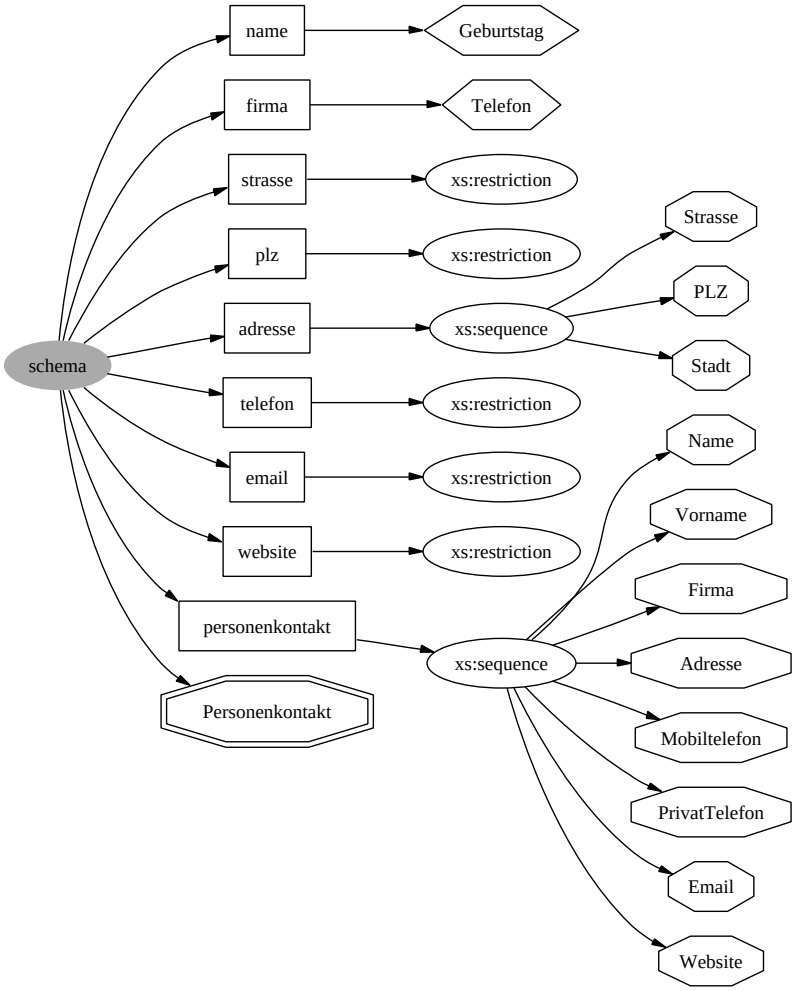

Abbildung B.3: Venetian Blind Design in grafischer Notation ohne Angabe der Typzugehörigkeiten

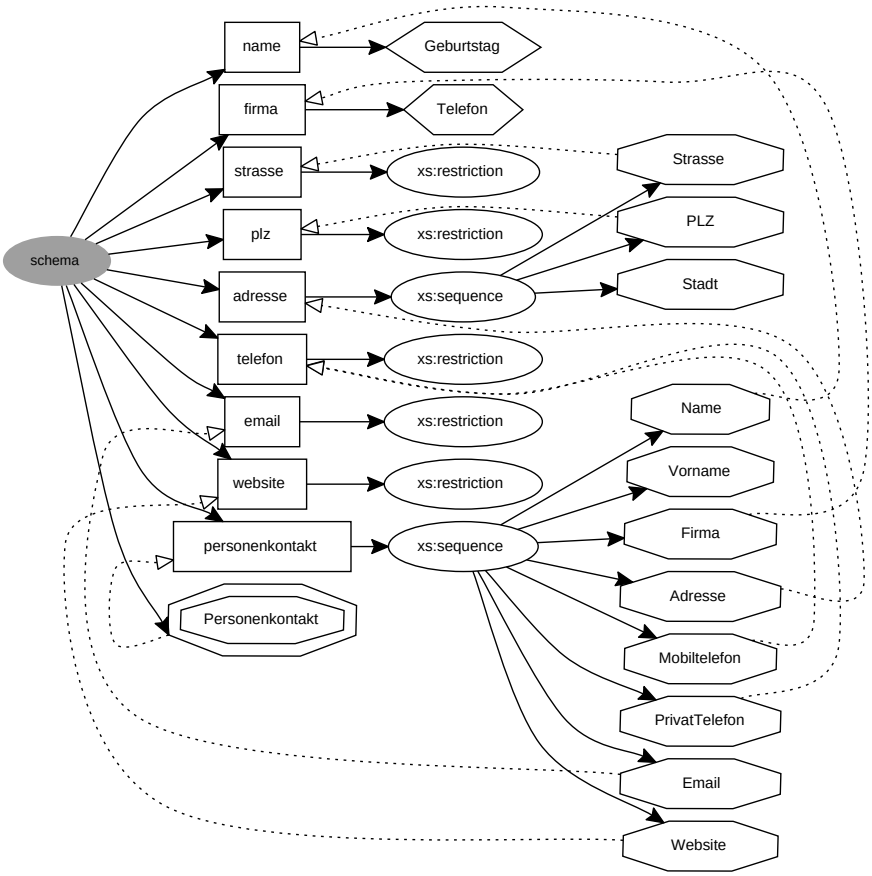

Abbildung B.4: Venetian Blind Design in grafischer Notation mit Angabe der Typzugehörigkeiten

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AL02] Marcelo Arenas and Leonid Libkin. *A Normal Form for XML Documents*. University of Toronto, citeseer.nj.nec.com/582104.html, 2002.

[AL03] Marcelo Arenas and Leonid Libkin. *An Information-Theoretic Approach to Normal Forms for Relational and XML Data*. University of Toronto, citeseer.nj.nec.com/arenas03informationtheoretic.html, 2003.

[Boo94] Grady Booch. *Objektorientierte Analyse und Design*. Addison-Wesley, 1994.

[BSL01] Don Box, Aaron Skonnard, and John Lam. *Essential XML*. Addison-Wesley, 2001.

[FP97] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics*. International Thomson Computer Press, 1997.

[Gul01] David Gulbransen. *Using XML Schema*. QUE, November 2001.

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

[Hol00] Steven Holzner. *XML Inside*. New Riders New York, 2000.

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

[HV02] Christina Hille and Dietgar Völzke. *XML-Schema und XQuery*. Institut Informatik Universität Münster, Mai 2002.

[KH81] Dennis G. Kafura and Sallie M. Henry. Software structure metric based on information flow. *IEEE Transactions on Software Engeneering*, September 1981.

[McL01] Brett McLaughlin. *Java and XML*. O'Reilly, 2001.

[Min01] Steffen Mintert. *XML-Schema*. editionW3C, http://www.edition-w3c.de/, März 2001.

[Raa93] Jörg Raasch. *Systementwicklung mit strukturierten Methoden*. Carl Hanser Verlag, 1993.

[Rom01] Christian Romberg. *Untersuchung zur automatischen XML-Schema-Ableitung*. Universität Rostock, Diplomarbeit, Oktober 2001.

[Sch02] Lars Schneider. *Entwicklung von Metriken für XML-Dokumentkollektionen*. Universität Rostock, Diplomarbeit, 2002.

[SM99] Michael Stonebraker and Dorothy Moore. *Ojektrelationale Datenbanken*. Carl Hanser Verlag, 1999.

[Som92] Ian Sommerville. *Software Engineering*. Addison-Wesley, 1992.

[STS97] Gunter Saake, Can Türker, and Ingo Schmitt. *Objektdatenbanken*. Thomson Publishing, 1997.

[Sun02] Yan Sun. *XNF - Eine Normalform für XML Dokumente*. Institut Theoretische Informatik Phillips-Universität Marburg, September 2002.

[Vli02] Eric van der Vlist. *XML Schema*. O'Reilly, Juni 2002.

[W3C99a] James Clark, Steve DeRose / W3C. *XML Path Language (XPath) Version 1.0*. W3C, http://www.w3.org/TR/xpath, November 1999.

[W3C99b] Tim Bray, Dave Hollander, Andrew Layman / W3C. *Namespaces in XML*. W3C, http://www.w3.org/TR/REC-xml-names/, Januar 1999.

[W3C00] Tim Bray, Jean Paoli, Eve Maler, C.M. Sperberg-McQueen / W3C. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C, http://www.w3.org/TR/REC-xml, Oktober 2000.

[W3C01a] David C. Fallside, Henry S. Thompson, Paul V. Biron et al. / W3C. *XML-Schema*. W3C, http://www.w3.org/XML/Schema, März 2001.

[W3C01b] Eve Maler, Steve DeRose, David Orchard/ W3C. *XML Linking Language (XLink) Version 1.0*. W3C, http://www.w3.org/TR/xlink/, Juni 2001.

[Wal90] Ernest Wallmüller. *Software-Qualitätssicherung in der Praxis*. Carl Hanser Verlag, 1990.

[WLL+02] Xiaoying Wu, Tok Wang Ling, Sin Yeung Lee, Mong Li Lee, and Gillian Dobbie. *NF-SS: Normal Form for Semistructured Schema*. University of Singapore / University Auckland, 2002.