Schemaevolution und Adaption von XML-Dokumenten und XQuery-Anfragen

Diplomarbeit

Tobias Tiedt

Lehrstuhl Datenbank- und Informationssysteme
Institut Informatik
Universität Rostock

Albert-Einstein-Str. 21
D-18059 Rostock

1. Gutachter:	Prof. Dr. Andreas Heuer
2. Gutachter:	Prof. Dr. Clemens H. Cap
Betreuer:	Dr.-Ing. Meike Klettke

eingereicht am: 06.03.2005

Zusammenfassung

XML wird häufig als Dokumentenaustauschformat benutzt oder es findet Einsatz im Datenbankbereich oder in föderierten Informationssystemen. Durch den Einsatz von XML-Schema zur Validierung ergeben sich hohe Anforderungen an die Datenbestände bei der Evolution der Schemata.
Diese Arbeit beschreibt eine Änderungssprache an XML-Schema, um die Evolutionsschritte durchzuführen und die anschließende Generierung neuer Anfragen an die XML-Dokumente und XQuery-Anfragen zur Adaption der Evolutionsschritte in den Datenbeständen.

Abstract

XML is often used as a data exchange format or it is used for data storage in database systems and distributed information systems. While the usage of XML-Schema for validating the documents, several demands exist what will happen with documents next to the evolution of schemas.
This paper introduces an update-language especially for XML-Schema to carry out the steps of evolution. It also explains how to generate new queries for adapting the documents or XQuery-queries to the evolutionsteps.

CR-Klassifikation
E.1, H.2.1, H.3.1, H.3.3, I.7.1, I.7.2

Schlüsselwörter
XML, XML-Schema, XQuery, Anfragesprache, Änderungssprache, Evolution, inkrementelle Validierung, Dokumentenanpassung

Key Words
XML, XML-Schema, XQuery, query language, update language, evolution, incremental validation, document adaptation

Inhaltsverzeichnis

Kapitel 1

Einleitung

Daten werden zunehmend mittels XML (e**X**tensible **M**arkup **L**anguage) gespeichert und die einzelnen XML-Dateien in föderierten Informationssystemen gehalten oder in Datenbanksystemen gespeichert. Die Vorteile von XML sind unter anderem Maschinenlesbarkeit und die Lesbarkeit durch den Anwender. Dieses ist durch relativ einfache Daten gewährleistet, denen eine strenge Syntax und eine logische und semantische Struktur anhaftet. Dabei entspricht die Strenge der Struktur den Anforderungen an Wohlgeformtheit der XML-Dokumente und der strengen Hierarchie, so dass XML-Dokumente einer Baumstruktur oder hierarchischen Struktur entsprechen. Die Liste der Vorteile kann noch erweitert werden, da XML in hohem Maße flexibel ist, weil die zugrunde liegende logische Struktur frei definiert werden kann. Dieses ist durch die Definition nutzerspezifischer Elemente, so genannten *Tags*, realisierbar, sofern gewisse Regeln, unter anderem die Wohlgeformtheit, nicht verletzt werden. Dadurch hat man mit XML neben dem Einsatz als Metasprache die Möglichkeit, es als Datenaustauschformat für verschiedenste Anwendungen zu nutzen.

Mit dieser zunehmenden Nutzung werden hohe Anforderungen an XML gestellt und es werden immer mehr Erweiterungen festgelegt. An erster Stelle steht dabei sicherlich die Möglichkeit zur Definition der Struktur der einzelnen XML-Dokumente. Realisiert wird dies durch die Mechanismen DTD (**D**ocument **T**ype **D**efinition) [W3C04e] und XML-Schema [W3C04a]. Durch sie erhält man die Möglichkeiten, die Struktur und die Typen der einzelnen Elemente zu definieren und zu deklarieren. Gleichermaßen dienen sie dazu, dass man XML-Dokumente gegen sie validiert, so dass neben der Wohlgeformtheit auch die Gültigkeit bezüglich eines Schemas gegeben sein muss. Vor allem im Bereich des Datenaustausches ist dies von hoher Wichtigkeit.

Andere Erweiterungen im XML-Bereich sind die Möglichkeiten für Anfragen an XML-Datenbestände. Ein Beispiel dafür ist mit XQuery [W3C05] vorgesehen und realisiert worden. Bei fortlaufender Nutzung der Datenbestände kommen auch immer mehr Fragen auf, welche schon bei den etablierten Datenbanksystemen, gleich ob relationalem oder objektorientiertem System, diskutiert wurden. Diese können unter dem Begriff der Evolution zusammengefasst werden. Dabei wird das den Daten zugrunde liegende Schema an neue Gegebenheiten angepasst und die Dokumente mit den neuen Schemata abgestimmt. Im Informationssystem abgelegte Anfragen müssen ebenfalls auf die neuen Dokumente abgestimmt werden, damit sie weiterhin ausführbar sind und korrekte Ergebnisse liefern.

Die vorliegende Diplomarbeit bearbeitet das Thema der Anfragen an XML-Schemata in XML-Schema-Notation, so dass mit diesen Anfragen eine Evolution auf der Schema-Ebene durchgeführt werden kann und anschließend die notwendigen Änderungen der Dokumente und XQuery-Anfragen bestimmt und vollzogen werden können. Dabei ist die im Zuge dieser Diplomarbeit definierte Anfragesprache keine speziell an XML-Schema angepasste, sondern eine allgemeine Änderungssprache mit Operationen allgemein für XML-Dokumente. Dies ist sinnvoll und möglich, da XML-Schema eine konkrete Ausprägung von XML ist. Deswegen sind XML-Schemata gültige und wohlgeformte XML-Dokumente. Einzig bei der Adaption der Änderungen bei XQuery-Anfragen sind spezielle Operationen notwendig, da XQuery nicht in XML-Syntax vorliegt.

Der Aufbau dieser Diplomarbeit staffelt sich wie folgt: in Kapitel 2 wird ein Überblick über das Thema Evolution gegeben, in dem die grundsätzlichen Ideen dahinter und die damit verbundenen Schwierigkeiten bei deren Umsetzung erläutert werden. In Kapitel 3 wird ein Überblick über Anfragesprachen mit möglichen Änderungsoperationen gegeben und deren Vorteile und Nachteile beschrieben und im Kapitel 4 werden bereits existierende Ansätze zur Evolution auf Schemata im XML-Bereich vorgestellt. Darüber hinaus wird in Kapitel 5 der Entwurf einer Änderungssprache XSEL für XML-Schema vorgestellt. Kapitel 6 umfasst die Beschreibung der konkreten Umsetzung bis hin zur prototypischen Implementierung eines Parsers und den Modulen zur Umsetzung der Sprache. Die Besonderheiten und Schwierigkeiten, die bei der Umsetzung der Evolutionssprache bezüglich der Adaption der Dokumente und XQuery-Anfragen bestehen, werden in Kapitel 7 dargelegt. Abschließend findet in Kapitel 8 eine Zusammenfassung und ein Ausblick auf mögliche Erweiterungsideen und Verbesserungen statt.

Kapitel 2

Evolution

In diesem Kapitel wird ein kurzer Überblick über die Thematik der Evolution gegeben. Es wird darauf eingegangen, welche Gründe für die Evolution an definierenden Schemata existieren und welche Konsequenzen die Evolution auf die Datenbestände hat.

Der Begriff Evolution ist immer von Bedeutung, sobald Daten und deren Schemata langfristig zum Einsatz kommen und ist weitläufig anwendbar auf jeden Aspekt der Datenhaltung und Datenorganisation. Im Folgenden wird nur die Evolution auf Datenbeständen, die im XML-Format vorliegen, und genauer die Evolution von XML-Schemata in XML-Schema-Notation betrachtet. Diese Einschränkung ist möglich, weil die Grundprinzipien unabhängig vom konkreten Anwendungsfall und die Konsequenzen unabhängig von den darunter liegenden Daten sind. Zur Umsetzung der Evolution eines Datenbankschemas sind also die gleichen Schritte zu machen und die gleichen Bedingungen zu beachten, wie dies für die Evolution auf einem Schema für XML-Daten und der Adaption der XML-Dokumente notwendig ist.

Im Rahmen dieser Arbeit wird davon ausgegangen, dass jenes den XML-Daten zugrunde liegende Schema in XML-Schema-Notation, wie es im Entwurf des W3C [W3C04a] definiert ist, vorliegt und nicht als DTD.

2.1 Einführung

Die Evolution bezeichnet einen Entwicklungsprozess, der im Bereich der Datenbanken und auch im Bereich der semistrukturierten Daten auf der Ebene des Schemas stattfindet. Damit beschreibt die Evolution die Änderung des Datenmodells. So wird während dieser Entwicklung das Schema geändert und damit neuen Aufgaben angepasst.

Die Gründe für diese neuen Aufgaben können sehr vielfältig sein. Ein Beispiel wären langlebige Datenbestände, welche über die Zeit immer wieder ergänzt und erweitert werden müssen wie z.B. Daten über Personenkontakte. Besaßen Personen in den siebziger Jahren des 20. Jahrhunderts überhaupt kein Mobiltelefon, so können nun Personen davon sogar mehrere besitzen. Betrachtet man diesen Aspekt der zeitlichen Veränderung der möglichen Daten, so ist die Evolution die Anpassung des Schemas an solche Gegebenheiten. Wenn also Daten über Personen, welche mindestens eine Mobilfunknummer aufweisen, in den Datenbestand aufgenommen werden sollen, muss das Schema aus den siebziger Jahren an diese neuen Umstände angepasst werden. Dass in den siebziger Jahren des 20. Jahrhunderts XML noch nicht definiert worden war und noch nicht existierte, sollte dem Sinn des Beispiels keinen Abbruch

tun.

Weitere Gründe für die Evolution sind bezüglich des Datenbestandes geänderte Anforderungen des Nutzers. Es bedarf der Evolution auch, wenn der anfängliche Entwurf eines Schemas ungenügend ist oder sogar Fehler enthält. Dann ist es dringend notwendig, dass das Schema angepasst wird. In diesen Anpassungsschritten werden die Fehler beseitigt und die ungenügenden Aspekte dahingehend erweitert bzw. reduziert, dass sie den Anforderungen entsprechen.

Im Weiteren wird von Evolution immer als Evolution auf der Schema-Ebene gesprochen. Evolution kann man nicht als einen abgeschlossenen endlichen Prozess betrachten, müssen doch im ungünstigsten Fall immer wieder Änderungen am Schema vorgenommen werden. Evolution ist also ein fortwährender Prozess, der aus einzelnen Änderungen besteht oder aus einer Abfolge von Evolutionsschritten. In Kapitel 5 findet eine Konkretisierung des Begriffes Evolutionsschritt statt.

2.2 Bedingungen

Unabhängig vom konkreten System und der konkreten Realisierung der Evolution unterliegt diese immer gewissen Bedingungen, die bei der Umsetzung der einzelnen Evolutionsschritte beachtet und eingehalten werden müssen.

An erster Stelle muss ein System, das die einzelnen Änderungsschritte auf dem Schema durchführt, darauf achten, dass die Evolution nicht die Regeln des Schemas verletzt. Dies bedeutet, dass nach jedem einzelnen Schritt wieder ein gültiges Schema vorliegen muss.

Das System muss ebenfalls dafür sorgen, dass die Evolutionsschritte auch auf die XML-Dokumente angewendet werden. Dies ist zwingend notwendig, da nach der Änderung des Schemas die gespeicherten Dokumente bezüglich dieses Schemas nicht mehr valide sein müssen. Aus diesem Grund müssen die Dokumente adaptiert werden, damit sie wieder der Validität gegenüber dem Schema genügen. Dabei bedeutet Validität, dass die Inhalte der Dokumente den Definitionen und Deklarationen aus dem Schema deren Eigenschaften genügen.

Außerdem können im Informationssystem Anfragen abgelegt worden sein, welche dazu dienen, Daten zu extrahieren und zu verarbeiten. Ein Beispiel für solche Anfragen sind XQuery-Anfragen (siehe 3.4). Diese sind speziell auf die Dokumente zugeschnitten, was bedeutet, dass die XQuery-Anfragen aufgrund der Struktur der Dokumente und der Namen einzelner Knoten Daten extrahieren. Deswegen müssen diese Anfragen nach der Dokumentanpassung ebenfalls abgeändert werden. Im weiteren Verlauf der Arbeit wird einschränkend davon ausgegangen, dass im Informationssystem nur XQuery-Anfragen abgelegt sind.

Diese Evolutionsbedingungen und die damit verbundenen Anpassungen dienen dazu, dass nach erfolgten Evolutionsschritten weiterhin valide Daten vorliegen und dass die abgelegten Anfragen weiterhin ausführbar sind. Diese Anpassungen sind mitunter schwierig, da jeder Evolutionsschritt anders ausfallen kann und sie dementsprechend immer differenziert betrachtet werden müssen. Eine Übersicht der Schwierigkeiten bei der Adaption ist in Kapitel 7 gegeben.

Im Idealfall wird die Evolution ohne Datenverlust ausgeführt. Dies wäre unter anderem dann möglich, wenn nicht das Schema geändert wird und dann die Daten angepasst werden, sondern

indem Versionierung betrieben wird, also mehrere Versionen eines Schemas und verschiedene Datenbestände existieren. Dieser Ansatz wirft aber weitere Probleme wie z.B. Kostenkontrolle auf. Dabei sind mit den Kosten die Zeit der Verarbeitung und der Speicherplatz gemeint. In [Bou03] wird daher der Versuch gemacht, Evolution von Datenbankschemata so umzusetzen, dass eine Kombination von Versionierung und Schemaänderung durchgeführt wird.

Wird aber eine Kombination mit Versionierung des Schemas nicht in Betracht gezogen, so wäre eine weitere Bedingung an die Evolution, dass bei der Datenadaption der Daten- oder der Informationsverlust minimal sein soll, was ebenfalls bei der Anpassung der Daten berücksichtigt werden muss. Damit sind die notwendigen Änderungen abhängig von den Evolutionsschritten, aber auch von dem zugrunde liegenden Informationsmaß, welches den Informationsgehalt eines Dokumentes charakterisiert.

Die folgende Auflistung zeigt noch mal einen kurzen Überblick über mögliche Bedingungen:

- Ein Schema muss vor und nach erfolgter Evolution valide bezüglich den Konventionen eines Schemas sein.
- Die Daten müssen entsprechend angepasst werden, so dass sie weiterhin valide sind.
- Bei der Anpassung muss ein möglicher Informationsverlust so gering wie möglich sein.

2.3 Anpassung der Daten

Im vorangegangenen Abschnitt wurde erläutert, warum Anpassungen der Dokumente nicht ausbleiben können, wobei diese Anpassungen direkt abhängig von den zuvor durchgeführten Evolutionsschritten sind. Es existieren verschiedene Ansätze, um die Evolution zu spezifizieren und daraus die notwendigen Anpassungen abzuleiten.

In [Zei01] wird die Evolution auf einer DTD als Schema für XML beschrieben. Dabei wird zunächst eine Einteilung vorgenommen, ob es sich um Änderungen auf Attribut- oder auf Elementebene handelt. Weiterhin wird danach unterschieden, wie sich die Informationskapazität bei den DTD's und den Dokumenten verhält. Dadurch ergeben sich je nach Änderung des Schemas andere Informationskapazitäten, die eine Einteilung vornehmen lassen, inwieweit die Dokumente angepasst werden müssen. Es wird grob unterschieden, ob sich die Informationskapazität verringert, erweitert oder gleich bleibt. Der Vorschlag in [Zei01] zur Adaption der Schemaänderungen in den Dokumenten sieht XSLT (e**X**tensible **S**tylesheet **L**anguage **T**ransformation) [1] als Transformationssprache vor.

In [Rei92] wird als ein anderer Ansatz die Formalisierung der Evolution durchgeführt. Aufgrund dieser Formalisierung lassen sich logische Axiome aufstellen, die dann in einer konkreten logischen Programmiersprache implementiert werden können. Es wird ein initiales Schema vorausgesetzt und dann bei einzelnen Evolutionsschritten versucht, aus dem neuen resultierenden Schema die Daten der Dokumente logisch abzuleiten. Ist dies möglich, wird der Evolutionsschritt durchgeführt und eine Datenanpassung ist nicht notwendig. Ansonsten sind gerade die Teile, welche nicht abgeleitet werden konnten, abzuändern.

[1] siehe dazu [W3C99a]

In [Fau01] werden zur Evolution und Adaption Operationen spezifiziert, welche in so genannten Trafo-Ausdrücken beschrieben werden. Diese Trafo-Ausdrücke werden dann auf das Schema angewendet. Anschließend wird beim Parsen der Dokumente für die einzelnen Elemente über Typzuweisungen die definierende Stelle im Schema herausgefunden. Dadurch kann der entsprechende Trafo-Ausdruck bereitgestellt werden. Dieser wird nun auf den unter dem Instanzelement befindlichen Teilbaum im Dokument angewendet. Nach der Anwendung wird der Trafo-Ausdruck an die über dem Teilbaum befindlichen Elemente weitergereicht, damit Angaben zur Häufigkeit des Auftretens, behandelt werden können.

Die Schwierigkeit speziell bei XML-Schema besteht darin, dass eine eindeutige Abbildung der Namen von deklarierten Elementen zu deren Typen nicht gegeben ist. Im Gegensatz dazu ist dies bei der DTD der Fall. In XML-Schema ist dieses nicht gegeben, da je nach Kontext, die Elemente verschiedene Typen aufweisen können. Mit Kontext ist dabei die Position gemeint, in der das Element in der Baumhierarchie des Dokumentes anzufinden ist. Die Definition einer solchen Abbildung ist aber sinnvoll, da nach der Änderung des Schemas die Teile im Dokument, welche unter Umständen transformiert werden müssen, durch eine solche Abbildung genau lokalisiert werden können.

Beispiele für notwenige Änderungen der Dokumente sind unter anderem das Löschen von Teilen des Schemas. Da Teile der Dokumente nun keine Entsprechung im Schema finden, müssen sie gelöscht werden. Ein anderes Beispiel sind Änderungen von Quantoren, in XML-Schema sind diese durch die Attribute `minOccurs` und `maxOccurs` der Elementdeklarationen repräsentiert. Wenn sich durch die Änderung der Quantoren im Schema ein größerer Wertebereich bei den Dokumenten ergibt, zum Beispiel wenn ein Quantor von “muss” auf “kann” geändert wird, so hat dies keinen Einfluss auf die Dokumente. Ist dies umgekehrt der Fall, bei Änderung von “kann” auf “muss”, müssen alle Dokumente dahingehend angepasst werden, dass an den entsprechenden Stellen Standardwerte oder Nullwerte eingefügt werden, sofern noch keine Werte vorhanden sind. Dabei stellt sich aber die Frage, wie Standardwerte entsprechend ihrer Datentypen aussehen sollen. In Kapitel 7 wird dieser Frage nachgegangen.

2.4 weitere Anpassungen

Nach der Evolution des Schemas und der Adaption der Daten sind weitere Anpassungen notwendig, da die meisten Datenbanksysteme oder Informationssysteme die Möglichkeit für Anfragen vorsehen. Diese Anfragen können im System abgelegt sein und werden von außen, z.B. durch Anwendungsprogramme, zur Ausführung gebracht. Es können entweder SQL-Anweisungen oder *Stored Procedures*, speziell im Datenbankbereich, oder Anfragen diverser XML-Anfragesprachen wie XQuery sein, sofern die Daten im XML-Format vorliegen. Mit Hilfe dieser Anfragen werden Daten aus den Dokumenten extrahiert. Deswegen müssen sie eventuell angepasst werden, da ansonsten nicht sichergestellt werden kann, dass sie ausgeführt werden können oder dass ihre Ergebnisse von den Informationen her korrekt sind.

Bei Anfragen ergeben sich ähnliche Anforderungen wie bei den Daten. Es muss geklärt werden, welche Anfragen abgeändert werden müssen, welche nicht und wie man dies erkennen

kann oder woraus man dies ableiten kann. Speziell für die zu ändernden Anfragen muss geklärt werden, welche Teile transformiert werden müssen, da viele Mehrdeutigkeiten auftreten können.

Aufgrund dessen, dass von XML-Daten ausgegangen wird, soll als Beispiel XQuery dienen, da es die am weitesten verbreitete XML-Anfragesprache ist. Ein genauer Überblick zu XQuery ist Abschnitt 3.4 zu entnehmen.

Nach entsprechender Änderung der Dokumente gilt es die Anfragen herauszufinden, die Teile von den geänderten Dokumenten auswerten. Wegen der Definition von XQuery ist es relativ leicht herauszufinden, ob eine Anfrage geändert werden muss, da der W3C-Vorschlag [W3C05] zu XQuery eine feste Struktur der Anfragen vorsieht. Der wichtigste Ausdruck in XQuery ist die so genannte FLWR (**F**or **L**et **W**here **R**eturn) -Klausel oder die FLWOR (**F**or **L**et **W**here **O**rder by **R**eturn) -Klausel[2]. Dabei wird in der `FOR`- und `LET`-Anweisung festgelegt, auf welches XML-Dokument die Anfrage abzielt. Über einen XPath[3]-Ausdruck (siehe 3.2) wird ein Referenzknoten innerhalb dieses Dokumentes spezifiziert.

Die Schwierigkeiten bei XQuery sind neben der speziellen Syntax das Variablenkonzept. So kann durch die Nutzung von Variablen die direkte Interpretation der XPath-Ausdrücke nicht durchgeführt werden, sondern die Variablen müssen zuerst expandiert werden. Es werden in der `FOR`- und `LET`-Anweisung die durch XPath-Ausdrücke ermittelten Knotenmengen in Variablen abgelegt, welche dann in den `WHERE`- oder `RETURN`-Anweisungen durch XPath-Ausdrücke weiter eingeschränkt werden können. Eine weitere Schwierigkeit ergibt sich durch den Einsatz von XPath als Selektions- und Navigationsmechanismus. Die vielfachen Änderungsmöglichkeiten und die damit verbundenen Schwierigkeiten sind in Abschnitt 7.2 genauer beschrieben. Siehe dazu auch das Beispiel 7.1 im Abschnitt 7.2.

[2] als *flower* ausgesprochen
[3] XPath (**X**ML **Path** Language)

Kapitel 3

Anfragesprachen

Im nun folgenden Kapitel wird ein Überblick über existierende Ansätze zur XML-Verarbeitung gegeben. Speziell Anfragesprachen werden dabei betrachtet und abschließend ein Vergleich zur Möglichkeit der Manipulation aufgestellt.

3.1 Anforderungen an eine Anfragesprache

Die Anforderungen, welche an eine Anfragesprache gestellt werden, können in großen Teilen von den Anforderungen an SQL im Datenbankbereich abgeleitet werden. Zusätzlich können noch XML-spezifische Anforderungen aufgestellt werden.

Die folgende Auflistung zeigt in der Übersicht, welche Anforderungen an Anfragesprachen existieren:

- **Deskriptivität**: Umschreiben des gewünschten Ergebnisses und nicht Angabe, wie dies zustande kommen soll.
- **Sicherheit**: Korrekte Anfragen müssen ein endliches Ergebnis in endlicher Zeit liefern.
- **Kompaktheit**: Eine kompakte Notation soll als Grundlage der Anfragesprache dienen.
- **Ad-hoc-Formulierbarkeit**: Anfragen können direkt auch ohne komplette Programme erstellt werden.
- **Mengenorientiertheit**: Operationen sollen auf Mengen von Objekten arbeiten.
- **Adäquatheit**: Alle Konstruktoren des Datenmodells werden unterstützt.
- **Abgeschlossenheit**: Ergebnis einer Anfrage kann Eingabe einer neuen Anfrage sein.
- **Orthogonalität**: Anfragekonstrukte sollen uneingeschränkt miteinander kombinierbar sein.
- **Vollständigkeit**: Alle Daten lassen sich verlustfrei extrahieren.
- **Optimierbarkeit**: Verschachtelte und kombinierte Operationen sind bezüglich der Ausführungseffizienz optimierbar.

Eine genauere Erläuterung zu den einzelnen Punkten kann [KM02] entnommen werden. Darin wird auch ein Überblick gegeben, welche XML-spezifischen Anforderungen gestellt werden können. Zum Beispiel die Einbettung von Anfragen in Dokumente oder die Unterstützung von Hyperlinks.

Allgemein kann gesagt werden, dass Anfragesprachen dazu dienen, die Daten aus dem Daten- oder Dokumentmodell zu extrahieren und weiter zu verarbeiten. Es ist auch angedacht, dass mittels Anfragen dieses Datenmodell manipuliert werden kann. Um diese Möglichkeiten umzusetzen, kann im Bereich der XML-Anfragesprachen noch die Notwendigkeit der Navigation betrachtet werden. Da XML einer Baumstruktur entspricht, ist es sinnvoll, dass über Ausdrücke im Baum navigiert werden kann, damit die Daten z.B. iterativ extrahiert werden können. Es ist ebenso sinnvoll, dass über spezielle Ausdrücke Knoten in diesem Baum selektiert werden, welche dann über die Anfragen weiterverarbeitet oder transformiert werden. Es existieren Ansätze, welche die Möglichkeiten zur Navigation oder zur Selektion spezieller Knoten umsetzen. Beispiele sind dabei die Mechanismen wie XPath oder DOM. Diese Mechanismen sind durch das W3C beschrieben und dienen häufig als Grundlage für weitere vom W3C vorgeschlagene Mechanismen wie XSL (e**X**tensible **S**tylesheet **L**anguage) oder XLink.

3.2 XPath

Da XML-Dokumente wohlgeformt sein müssen und damit die strikte Einhaltung der Hierarchie gefordert wird, kann man XML-Dokumente als Baum betrachten. Es existiert mit XLink und XPointer (für einen genauen Überblick siehe [W3C01]) die Möglichkeit, mehrere Dokumente miteinander zu verketten, indem man so genannte Links angibt, die auf andere Dokumente verweisen und dabei genauer auf speziell ausgewählte Knoten innerhalb dieser Dokumente.

Um diese Verkettung und vor allem eine sichere Navigation in den Bäumen, die XML-Dokumente repräsentieren, zu unterstützen, wurde vom W3C der Vorschlag zu XPath gemacht. Der folgende Abschnitt gibt einen groben Überblick über die Technik, für tiefergehende Informationen siehe [W3C99b] oder [KM02].

3.2.1 Einführung

XPath ist im engeren Sinn keine Anfragesprache, sondern dient dazu, über spezielle Ausdrücke gewisse Knotenmengen zu selektieren. Die Verarbeitung dieser selektierten Knotenmengen wird von anderen Mechanismen und Systemen durchgeführt. Damit dient XPath zur Navigation in den Bäumen, weswegen XPath die Grundlage für viele Mechanismen zur Baumverarbeitung ist. Aus diesem Grund wird XPath zur Knotenselektion bei XSLT eingesetzt, das dazu dient, das Dokument einzulesen und über Stylesheets zu transformieren. Damit bekannt ist, welche Knoten transformiert werden sollen, werden so genannte Templates definiert, die über XPath-Ausdrücke Knoten selektieren.

Ein weiterer Einsatz für XPath sind die schon angesprochenen Mechanismen XLink und XPointer. XPointer erweitert das XPath-Konzept um die Begriffe Position und Positionsmen-

ge, damit XPointer als Referenzierungsmechanismus dienen kann. Mit Hilfe von XPath kann kein Dokumentteil selektiert werden, der in der Mitte eines Textknotens beginnt und in einem anderen Textknoten innerhalb eines benachbarten Teilbaumes endet, da XPath nur mit "ganzen" Knoten umgehen kann.

XPath wird auch in XQuery-Anfragen benutzt, um die Knotenmengen anzugeben, welche verarbeitet werden sollen.

Aufgrund dieses häufigen Einsatzes wird XPath hier aufgeführt. Ein zweiter Grund ist, dass XPath auch zur Selektion in XSEL, siehe dazu Kapitel 5, verwendet wird.

Das primäre syntaktische Gebilde in XPath ist ein Ausdruck, dessen Auswertung folgende Objekte liefern kann:

- Knotenmenge (jeder Knoten tritt in der Menge nur einmal auf)
- Wahrheitswert
- reelle Zahl
- Zeichenkette

Darüber hinaus findet die Verarbeitung in einem gewissen Kontext statt, zu dem Folgendes zählt:

- der aktuelle Knoten (Kontextknoten)
- die Position des Knotens innerhalb des Kontextes (Kontextposition)
- die Größe des Kontextes, also die Anzahl der Knoten einer initialen Knotenmenge (Kontextgröße)
- eine Menge an Variablenbindungen (Kombination aus Variablenname und Ausdruck, der den Wert der Variable liefert)
- eine Menge an Funktionen
- ein Satz von Namensraumdeklarationen

Der Kontextknoten ist der aktuell von einem System, wie einem XSLT-Prozessor oder einer XQuery-Anfrage, verarbeitete Knoten. Die Knotenmenge ist nun gerade die Menge an Knoten, die über den XPath-Ausdruck selektiert wird. Die Variablenbindungen und die Funktionen werden von außen in die Verarbeitung hineingereicht.

Dabei sieht der Aufbau eines XPath-Ausdrucks einer URL (**U**niform **R**esource **L**ocator) ähnlich, da er wie eine URL die Aufgabe hat, etwas zu lokalisieren. Bei XPath dienen die Ausdrücke der konkreten Lokalisierung von Knoten.

Neben der Angabe eines Kontextknotens ist der so genannte *Location-Path* das Entscheidende, da er alle Anweisungen enthält, die notwendig sind, damit eine Knotenmenge (oder ein anderes Objekt von den oben aufgeführten) selektiert werden kann, die als Ergebnis der

Auswertung zurückgegeben wird. Somit ist der Location-Path der wichtigste Teil eines XPath-Ausdrucks. Darin werden so genannte Achsen angegeben, welche die Navigationsrichtung im Baum vorgeben. Zusätzlich können Prädikate und Funktionen dazu eingesetzt werden, um die zu selektierende Knotenmenge genauer zu spezifizieren.

3.2.2 Location-Path

Der *Location-Path* kann als eine Art Wegbeschreibung aufgefasst werden, die es ermöglicht, einzelne Knoten zu selektieren und zu erreichen. Dabei unterscheidet man zwei Typen von Location-Paths:

- relative Location-Paths
- absolute Location-Paths

Vom Aufbau sind sie ähnlich, da beide jeweils aus so genannten *Location-Steps* bestehen, welche in der Verarbeitung wie ein Schritt im Baum behandelt werden. Der relative Location-Path wird vom Kontextknoten ausgehend verarbeitet, der absolute beginnt mit einem Schrägstrich (/) und legt damit fest, dass die Verarbeitung nicht vom Kontextknoten ausgeht, sondern von der Wurzel des Baumes. Verschiedene Location-Steps können über die Schrägstriche miteinander verkettet werden.

Anfänglich wird mit einem Location-Step die Richtung festgelegt, in der im Baum navigiert werden soll. Die Richtung ist dabei durch einen Achsenbezeichner gegeben. Folgende Achsen, welche jeweils eine genau festgelegte Anzahl an Knoten beinhalten, sind dabei möglich:

- **ancestor**: die Vorfahren des Kontextknotens ohne den Kontextknoten selbst
- **ancestor-or-self**: wie ancestor, inklusive dem Kontextknoten
- **parent**: der Elternknoten des Kontextknotens
- **self**: der Kontextknoten
- **child**: die Kindknoten oder direkten Nachfahren des Kontextknotens
- **descendant**: die Kinder und Kindeskinder des Kontextknotens ohne diesen
- **descendant-or-self**: wie descendant, der Kontextknoten gehört dabei mit zur Achse
- **following-sibling**: alle folgenden Geschwisterknoten des Kontextknotens
- **preceding-sibling**: alle vorangegangenen Geschwisterknoten
- **following**: alle Knoten nach dem Kontextknoten, außer seine Nachfahren
- **preceding**: alle Knoten vor dem Kontextknoten, außer seine Vorfahren
- **namespace**: die gültige Namensraumdeklaration für den Kontextknoten
- **attribute**: alle Attribute des Kontextknotens

Die Achsen **ancestor**, **descendant**, **following**, **preceding** und **self** zerlegen dabei den Dokumentbaum in disjunkte Teilbereiche, die in der Summe wiederum alle Knoten des Dokumentes beinhalten. Abbildung 3.1 zeigt die Navigationsachsen als Baumübersicht.

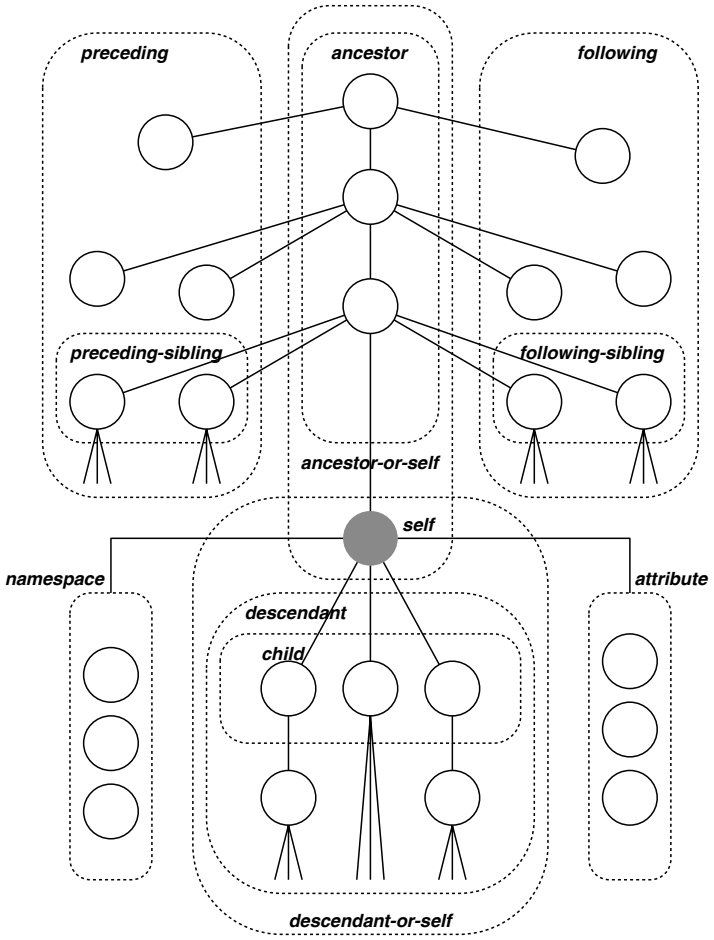

Abbildung 3.1: Navigationsachsen von XPath

3.2.3 Knotentest, Prädikate, Funktionen

Neben der Möglichkeit, einzelne Location-Steps miteinander zu einem Ausdruck zu verketten, gibt es noch die Möglichkeiten, so genannte Knotentests und Prädikate einzusetzen. Bei der Angabe der Achse wird eine Vorauswahl getroffen. Alle Knoten, die auf dieser Achse liegen, werden erfasst. Mittels Knotentests und Prädikaten besitzt man die Möglichkeit, die Ergebnismenge weiter einzuschränken und genauer zu spezifizieren.

Knotentest

Der Knotentest besteht entweder aus dem Namen eines Knotens an sich oder der Abfrage, von welchem Typ der Knoten ist. Damit werden in die Knotenmenge nur die Knoten auf-

genommen, die entweder dem angegebenen Namen entsprechen oder vom gewünschten Typ sind. Für die Typüberprüfung gibt es folgende Testfunktionen:

- **text()**: selektiert alle Textknoten
- **comment()**: selektiert alle Kommentare
- **processing-instruction()**: selektiert alle Prozessorinstruktionen (engl: *processing instructions*)
- **node()**: deckt alle Arten an Knoten ab, somit selektiert **node()** alle Knoten der Achse

Zur Überprüfung des Namens gibt man diesen anstelle der Funktionsaufrufe hinter dem Achsenbezeichner an. Alle Knotentests werden dabei durch zwei Doppelpunkte (::) vom Achsennamen getrennt.

Prädikate

Durch die Angabe von Prädikaten kann die Knotenmenge noch weiter spezifiziert werden. Sie erlauben die Angabe komplexerer Bedingungen, welche ausgewertet werden. Dabei können nicht nur die Knotentypen berücksichtigt werden, sondern es kann zum Beispiel konkret abgefragt werden, ob ein Attribut einen speziellen Inhalt hat oder ob unter der aktuell getroffenen Auswahl bestimmte Elemente existieren. Damit ist das Prädikat ein gewisser Filter für die Knotenmenge, welche durch die Achse und die Knotentests bestimmt wurde. Die Anwendung dieses Filters erzeugt wiederum eine Knotenmenge mit den Knoten, die dem Prädikat genügen.

Funktionen

Neben Prädikaten und Knotentests hat man noch die Möglichkeit vordefinierter Funktionen in XPath, welche entweder eine zusätzliche Bestimmung der gewünschten Knotenmenge erlauben oder eine gewisse Verarbeitung durchführen.

Es existieren dabei numerische und Boole'sche Funktionen, Funktionen zur Zeichenkettenmanipulation und Funktionen, welche auf Knotenmengen arbeiten.

Die folgende Auflistung zeigt im Überblick die in der Funktionsbibliothek definierten Funktionen für Knotenmengen. Dabei ist der Rückgabewert der Funktion kursiv und der Funktionsname fett notiert.

- *number* **last()**: liefert die Kontextgröße zurück
- *number* **position()**: liefert die Position des Kontextknotens im Kontext zurück
- *number* **count(node-set)**: liefert die Anzahl der Elemente der übergebenen Knotenmenge *node-set* zurück
- *node-set* **id(object)**: liefert eine Knotenmenge, deren Elemente die ID *object* aufweisen

- *string* **local-name(node-set?[1])**: liefert den lokalen Namen des ersten Elementes der übergebenen Knotenmenge oder wenn nichts übergeben wird, den lokalen Namen des Kontextknotens
- *string* **namespace-uri(node-set?)**: liefert den Namensraum-URI[2] des ersten Elementes oder wenn nichts übergeben wird, den des Kontextknotens
- *string* **name(node-set?)**: der Name, der lokale Name mit Namensraumpräfix, des ersten Knotens der übergebenen Knotenmenge, ansonsten den des Kontextknotens

Die folgende Auflistung gibt einen Überblick über die integrierten numerischen Funktionen:

- *number* **number(object?)**: versucht *object* als Nummer zu interpretieren und gibt die Zahl zurück, wenn die Interpretation fehlschlägt wird NaN (**N**ot **a** **N**umber) zurückgegeben
- *number* **sum(node-set)**: die Summe der Zahlenwerte der Elemente aus der übergebenen Knotenmenge
- *number* **ceiling(number)**: die kleinste ganze Zahl, die größer als *number* ist
- *number* **floor(number)**: die größte ganze Zahl, die kleiner als *number* ist
- *number* **round(number)**: rundet den Wert von *number* zur nächsten ganzen Zahl

Nachfolgend sind die möglichen Zeichenkettenfunktionen aufgeführt:

- *string* **string(object?)**: versucht *object* oder bei Fehlen des Argumentes den Kontextknoten als Zeichenkette zu interpretieren
- *string* **concat(string, string ... string)**: fügt alle übergebenen Zeichenketten zu einer zusammen
- *boolean* **starts-with(string, string)**: überprüft ob die erste Zeichenkette mit der zweiten Zeichenkette beginnt, wenn ja, liefert die Funktion *true* zurück, ansonsten *false*
- *boolean* **contains(string, string)**: überprüft, ob die zweite Zeichenkette an irgendeiner Stelle in der ersten Zeichenkette vorkommt
- *string* **substring-before(string, string)**: liefert den Teil der ersten Zeichenkette *vor* der Position des ersten Auftretens von der zweiten in der ersten Zeichenkette
- *string* **substring-after(string, string)**: liefert den Teil der ersten Zeichenkette *nach* der Position des ersten Auftretens von der zweiten in der ersten Zeichenkette
- *string* **substring(string, number, number)**: liefert den Teil ab der Position der ersten Zahl bis zur Position, angegeben durch die zweite Zahl

[1] das Fragezeichen entspricht dem Quantor "?" eines regulären Ausdrucks und gibt die Häufigkeit des Auftretens 0 oder 1 an

[2] URI (**U**niform **R**esource **I**dentifier)

- *number* **substring-length(string)**: liefert die Länge der Zeichenkette

Zum Schluss erfolgt noch die Auflistung der integrierten Boole'schen Funktionen:

- *boolean* **boolean(object?)**: versucht *object* als Boole'schen Wert zu behandeln
- *boolean* **not(boolean)**: negiert den übergebenen Boole'schen Wert
- *boolean* **true()**: liefert immer *wahr* oder *true*
- *boolean* **false()**: liefert immer den Wert *falsch* oder *false*
- *boolean* **lang(string)**: überprüft, ob der Kontextknoten sich innerhalb der Sprache *string* befindet. Wenn kein `xml:lang`-Attribut gefunden werden kann, werden der Reihe nach alle Knoten der ancestor-or-self-Achse überprüft, ob diese ein `xml:lang`-Attribut besitzen.

Einen genauen Überblick über XPath und den vorhandenen Funktionsbibliotheken kann man [W3C99b, Bac00] entnehmen.

3.2.4 Ausdrücke

Es wurde schon gesagt, dass der Ausdruck das primäre syntaktische Gebilde in XPath ist. Es wurde aber nicht näher darauf eingegangen, was genau ein Ausdruck sein kann.

Grundlegend ist ein Ausdruck in XPath entweder

- ein Literal
- eine Zahl
- eine Variablenreferenz
- ein Location-Path bzw. eine Verknüpfung mehrerer
- oder ein Funktionsaufruf.

Mit Hilfe von XPath-Ausdrücken können nun ganz bestimmte Knoten selektiert werden oder man kann in geringem Maße gewisse Informationen aus den Bäumen extrahieren, wie zum Beispiel mit der Node-Set-Funktion *name()*, welche den Namen des ersten Elementes aus einer Knotenmenge als Resultat liefert.

Am häufigsten wird XPath jedoch zur Navigation und damit zur Knotenselektion eingesetzt.

3.3 DOM

In diesem Abschnitt wird ein Überblick über das DOM (**D**ocument **O**bject **M**odel) gegeben, einem Vorschlag des W3C [W3C00]. DOM ist keine wirkliche Anfragesprache, sondern eine Programmierschnittstelle, die es ermöglicht, ein auf eine Baumstruktur abgebildetes Dokument in seiner Baumstruktur zu manipulieren.

3.3.1 Einführung

Im Gegensatz zu SAX[3], das durch Diskussionen in der XML-*dev*-Mailing-Liste entworfen und entwickelt worden ist, ist das DOM eine Empfehlung des W3C. Das DOM ist nicht speziell für eine Programmiersprache entworfen wurde, sondern stellt den Inhalt und das Modell von Dokumenten allgemein dar. Der Vorschlag zu DOM untergliedert sich in bisher 3 Entwicklungsstufen, *Levels* genannt, wobei in diesen eine gewisse Funktionalität festgelegt ist. DOM Level 3 ist noch in der Entwicklung begriffen, weswegen ein kurzer Überblick bezüglich Level 2 gegeben wird. Fortan ist mit DOM das DOM Level 2 gemeint.

Durch die Definition des DOM als die Festlegung der Struktur und des Inhalts eines Dokumentes ist für den Einsatz in konkreten Programmiersprachen eine Definition von Schnittstellen erforderlich. Im Vorschlag wurden deshalb auch Sprachanbindungen angeboten, unter anderem für Java, welche unter *http://www.w3.org/TR/DOM-Level-2/java-binding.html* eingesehen werden kann.

Hauptsächlich wird DOM neben SAX dazu verwendet, XML-Parser zu implementieren. DOM und SAX verfolgen dabei zwei unterschiedliche Strategien. Während SAX ereignisorientiert arbeitet und XML-Dokumente als Sequenz von Methoden-Aufrufen betrachtet, also eine Streaming-Schnittstelle ist, arbeitet DOM auf einer Baum-Struktur. Neben der Möglichkeit XML-Daten zu parsen, ist im DOM die Möglichkeit zur Manipulation der Struktur vorgesehen.

3.3.2 Objektmodell

Das DOM kann als Projektion des XML-Infoset [W3C04b] aufgefasst werden. Dabei beschreibt das XML-Infoset, welche Knoten in einem Dokument auftreten können und unterscheidet dabei folgende Typen:

- Dokument (*Document Information Item*)
- Element (*Element Information Items*)
- Attribut (*Attribute Information Items*)
- Prozessorinstruktion (*Processing Instruction Information Items*)
- Entitätenreferenz (*Unexpanded Entity Reference Information Items*)
- Text (*Character Data Information Items*)
- Kommentar (*Comment Information Items*)
- Dokumenttyp (*Document Type Declaration Information Item*)
- ungeparste Entität (*Unparsed Entity Information Items*)
- Notation (*Notation Information Items*)

[3] SAX (**S**imple **A**PI for **X**ML)

- Namensraum (*Namespace Information Items*)

Zusätzlich werden im XML-Infoset die Eigenschaften der einzelnen Knoten erläutert. Zum Beispiel besitzt ein Element nach dem XML-Infoset zwei Namen, einen lokalen und einen zweiten, der vor dem lokalen Namen das Namensraumpräfix beinhaltet. Elemente können Kindknoten und Attribute haben, besitzen einen Vater und weitere Namensraumangaben. Ein Attribut hingegen besitzt keinen Vater, sondern ein so genanntes *owner element*, das Element, zu dem das Attribut gehört.

Das Objektmodell von DOM stellt dieses Infoset nun als Graph mit Knoten und gerichteten Kanten in einer Baumstruktur dar. Weiterhin spezifiziert das DOM für jeden Knoten die zu unterstützende Schnittstelle, deren Syntax und Semantik in einer konkreten Implementation berücksichtigt und erfüllt werden muss. Zusätzlich werden im DOM die Beziehungen der Knoten untereinander definiert. Aufgrund von vorhandenen Beziehungen kann man die Kanten im Dokumentgraphen als gerichtet betrachten. Die Schnittstellen, welche die Knoten unterstützen müssen, sind im Abschnitt 3.3.3 kurz aufgeführt und erläutert.

Es werden nun die Knoten differenziert betrachtet, so dass verschiedene Arten von Knoten unterstützt werden, wobei die Knotentypen aus dem Infoset entnommen sind. In Tabelle 3.1 werden die möglichen Typen aufgeführt und deren Bedeutung mit angegeben.

DOM	**Erläuterung**
Document	stellt den Dokument-Knoten dar oder allgemeiner das Dokument-Objekt
DocumentFragment	Verweis auf einen Teil eines Dokumentes
DocumentType	Verweis auf das Element `<DOCTYPE>`
EntityReference	Verweis auf eine Entität
Element	stellt einen Element-Knoten dar
Attr	stellt einen Attribut-Knoten dar
ProcessingInstruction	Verarbeitungsanweisung
Comment	Kommentar
Text	repräsentiert den Textinhalt eines Elementes oder Attributes
CDATASection	`CDATA`-Abschnitt
Entity	Angabe einer Entität
Notation	stellt eine Notation dar

Tabelle 3.1: Knotentypen im DOM

3.3.3 Schnittstellen

Die einzelnen Schnittstellen, die sich hinter jedem Knotentyp verbergen, sind speziell auf die Eigenschaften, die im DOM für die Knoten definiert wurden, abgestimmt. All diese Schnittstellen leiten sich von der Schnittstelle *Node* ab, weswegen in einer konkreten Implementation das Navigieren in der Struktur einheitlich ist, da alles ein Knoten vom Typ *Node* ist. Abbil-

dung 3.2 aus [BSL01] zeigt eine UML[4]-Darstellung zu den Schnittstellen des DOM.

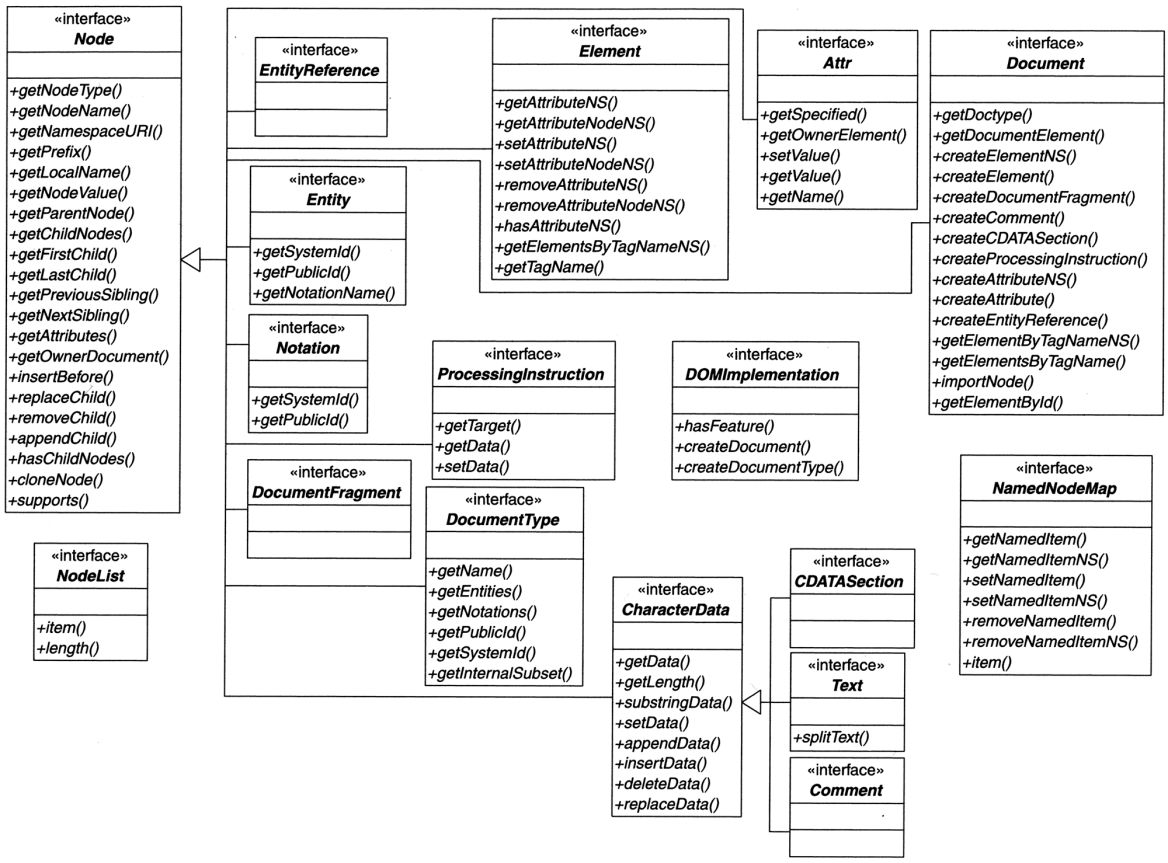

Abbildung 3.2: UML-Darstellung der DOM-Schnittstellen

Die grundsätzliche Funktionalität ist durch die Schnittstelle *Node* realisiert, welche einerseits Methoden bereithält, um durch den Baum navigieren zu können und andererseits die meisten Methoden zur Manipulation des Baumes bereitstellt. Diese Schnittstelle ist auch wegen der Vererbungsbeziehungen, da sie die Basisschnittstelle für alle anderen ist, der Dreh- und Angelpunkt. Beim Schnittstellenentwurf wurde vom W3C versucht, einen Kompromiss zwischen Typsicherheit und Bequemlichkeit, die man mit einer solchen Basisschnittstelle hat, zu finden. So sind manche Operationen der *Node*-Schnittstelle nicht auf alle Knotentypen anwendbar, existiert unter anderem die Abfrage der *Node*-Schnittstelle *hasChildren()* für Attribute nicht, da diese keine Kindknoten besitzen können. Um eine größere Allgemeingültigkeit zu erlangen, wurden die DOM-Schnittstellen so angelegt, dass sie vom Vorhandensein von RTTI (**R**un**t**ime **T**ype **I**dentification) einer Zielprogrammiersprache unabhängig sind. Zur Überprüfung der Kompatibilität gegenüber einer abgeleiteten Schnittstelle wird das Attribut *nodeType* der *Node*-Schnittstelle herangezogen. Das bedeutet, dass die Informationen über die Knoten des Dokumentes im Typsystem einer Programmiersprache eingebettet sind. Als Beispiel dient der nachfolgende Quelltextausschnitt, vorliegend in der Programmiersprache

[4]UML (**U**nified **M**odelling **L**anguage)

Java. Dieser zeigt, wie die Überprüfung, ob ein Knoten vom Typ *Element* ist, erfolgt. Diese Überprüfung wird durch die in der Schnittstelle definierte Funktion *isElement(...)* realisiert, welche nicht auf die in Java vorhandene RTTI zurückgreift

```
public boolean isElement(org.w3c.dom.Node n){
        return n instanceof org.w3c.dom.Element;
}
```

sondern das Attribut *nodeType* der *Node*-Schnittstelle nutzt

```
public boolean isElement(org.w3c.dom.Node n){
        return n.nodeType == org.w3c.dom.Node.ELEMENT_NODE;
}
```

3.3.4 Beziehung zwischen Knoten

Das DOM definiert auch die Beziehung der Knoten untereinander, wobei sich diese Beziehung hauptsächlich aus der Eltern-Kind-Beziehung des Infoset ableitet. Diese Informationen werden über gewisse Attribute oder Variablen einer konkreten Programmiersprache realisiert, welche angeben, ob ein Knoten über Kindknoten verfügt und welche Knoten diese Kindknoten, Geschwisterknoten oder Elternknoten repräsentieren. Abbildung 3.3 zeigt, welche Attribute in der *Node*-Schnittstelle vorhanden sind und welche Position im Dokumentbaum sie vertreten.

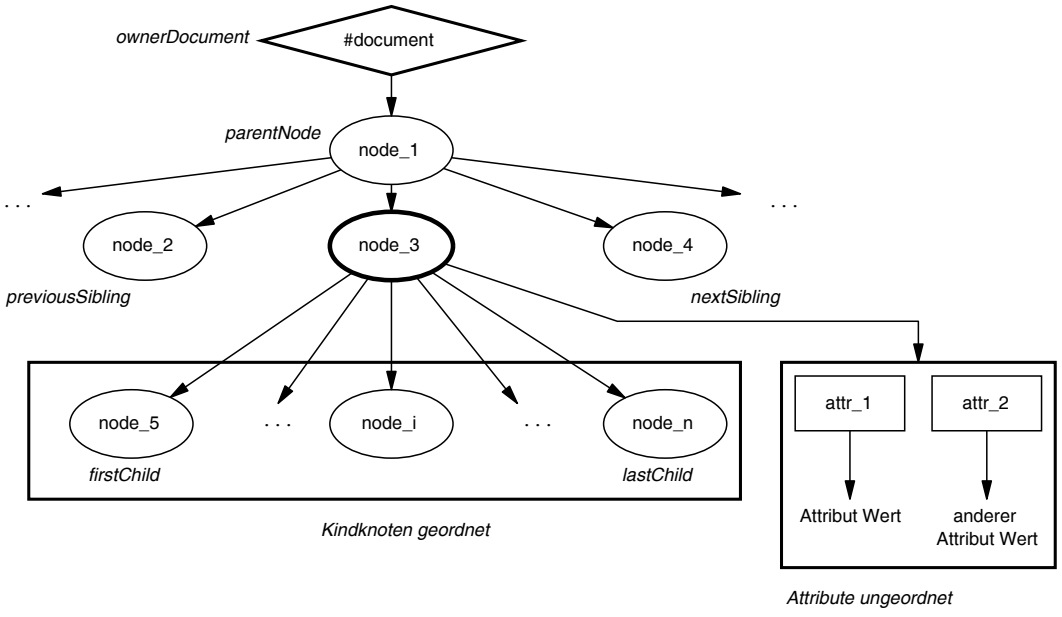

Abbildung 3.3: Beziehungen der Node-Schnittstelle

In der Abbildung sind die Namen der Schnittstellenattribute kursiv notiert. Die Darstellung der einzelnen Knoten eines Dokumentes wird so festgelegt, dass Elemente als Ellipsen,

Attribute als Rechtecke, der Dokumentknoten als Raute und Textknoten oder Werte von Attributen ohne Rahmen dargestellt werden. Wegen einer besseren Zuordnung wurden die Kindknoten des Beispielknotens *node_3* und dessen Attribute mit einem zusätzlichen Rahmen versehen. Die Abbildung wurde aus [BSL01] entnommen. Dort findet man auch mehrere Beispiele zur Anwendung der Schnittstellen im DOM.

3.4 XQuery

Wegen der rasanten XML-Entwicklung wurde es notwendig, eine Möglichkeit zu besitzen, Daten aus XML-Dokumenten zu extrahieren. Deswegen wurden viele Entwicklungen in Bezug auf eine Anfragesprache für XML gemacht. Dabei nahmen all diese Entwicklungen wie Yatl, Lorel oder XQL (**X**ML **Q**uery **L**anguage) Einfluss auf die Entwicklung von Quilt, das als Vorläufer von XQuery gilt. Daneben wurde XQuery aus dem Bereich von SQL (**S**tructured **Q**uery **L**anguage) und der objektorientierten Datenbanksysteme und dort im Speziellen durch OQL (**O**bject **Q**uery **L**anguage) mit beeinflusst. Die Orthogonalität von XQuery ist aus OQL entlehnt und die syntaktischen Klauseln von XQuery sind ähnlich dem Block SFW (**S**elect-**F**rom-**W**here-Klausel) von SQL aufgebaut.

Neben Anfragesprachen aus verschiedenen Bereichen spielte bei der Entwicklung zu XQuery auch XPath eine Rolle. Im Standard zu XPath [W3C99b] ist festgelegt, dass XML-Dokumente einer Baumstruktur folgen. XQuery verwendet nun ein Datenmodell, welches auf dem XPath-Standard 1.0 aufbaut und dementsprechend auch einer Baumstruktur folgt.

Wegen der Notwendigkeit einer Anfragesprache im XML-Bereich mit den in Abschnitt 3.1 genannten Bedingungen wurde 2001 vom W3C eine Arbeitsgruppe mit der Aufgabe der Definition einer Anfragesprache, die XML-Query-Arbeitsgruppe, zusammengestellt. Mit der Arbeit am aktuellen Arbeitsentwurf [W3C05] ging ebenfalls die Arbeit an einem Entwurf zu Version 2.0 von XPath [W3C04d] einher. Ein genauer Überblick zu XQuery kann aus den Webseiten des W3C [W3C05, W3C04c, W3C99b, W3C04d] und aus [KM02] entnommen werden.

3.4.1 Einführung

Wie schon erwähnt, ist XQuery eine Entwicklung, basierend auf vielen entworfenen Prototypen einer XML-Anfragesprache. Dabei können alle Konstrukte der Sprache XQuery orthogonal verwendet werden, das bedeutet, dass sie frei miteinander kombiniert werden können. Dies ist auf den Einfluss der objektorientierten Anfragesprache OQL der ODMG (**O**bject **D**ata **M**anagement **G**roup) zurückzuführen. Diese Sprachkonstrukte sind so genannte Ausdrücke, von denen folgende unterstützt werden:

- FLWR/FLWOR-Ausdrücke, Anfragen ähnlich dem SFW-Block von SQL
- XPath-Ausdrücke zur Selektion von Dokumentteilen
- Elementkonstruktoren zur Erstellung neuer XML-Konstrukte
- XML-Literale allgemein

- datenspezifische Operationen
- Funktionsaufrufe, unter anderem Standardfunktionen oder nutzerdefinierte Funktionen
- bedingte Anweisungen zur Auswertungssteuerung
- quantifizierte Ausdrücke mit `ALL` und `ANY` als mögliche Quantoren
- Test der Datentypen oder Typkonvertierungen

Jeder Ausdruck kann aufgrund der Orthogonalität einen oder mehrere Teilausdrücke enthalten, welche wiederum Konstrukte von XQuery sein können. Diese XQuery-Konstrukte haben einen gewissen Typ, wobei keine Restriktionen vorliegen, was den Ergebnistyp einer XQuery-Anweisung betrifft. Ausnahmen sind spezielle Anweisungen wie die bedingte If-Anweisung oder die Verwendung von numerischen Operationen, bei denen ein Boole'scher oder ein numerischer Wert vorausgesetzt wird.

3.4.2 Datenmodell

Das Datenmodell von XQuery beschreibt, wie die Daten strukturiert sind und wie darauf zugegriffen wird. XML-Dokumente besitzen eine baumartige Struktur, wie in XPath festgelegt, und das Datenmodell folgt dabei dieser Struktur. Somit können XML-Dokumente, Sequenzen von Dokumenten oder Dokumentfragmente beschrieben werden, welche als geordnete Menge von Knoten betrachtet werden. Diese geordnete Knotenmenge wird in XQuery als Sequenz bezeichnet.

Durch die Sequenzen kann eine geordnete Menge an Bäumen dargestellt werden, da Knoten geschachtelt sein können und wiederum Knoten enthalten können. Sequenzen dagegen sind flach, da sie selber keine anderen Sequenzen beinhalten können. Im Unterschied zu XPath, wo in der Knotenmenge die Knoten eindeutig waren, können in den Sequenzen Duplikate von Knoten auftreten. Darüber hinaus ist auf Sequenzen immer eine Ordnung definiert und es wird keine Unterscheidung zwischen einem Knoten und einer Sequenz der Länge 1 gemacht.

Die Knoten einer Sequenz haben einen bestimmten Typ, der entweder ein Grunddatentyp und aus XML-Schema entlehnt ist, z.B. `string` oder `float`, oder der Typ eines Knotens ist ein nutzerdefinierter Typ.

3.4.3 Typsystem

XQuery ist streng typisiert und basiert auf statischer Typprüfung, welche zu Beginn der Auswertung einer XQuery-Anfrage vollzogen wird und durch die der Typ einer Anfrage bestimmt wird. Im späteren Verlauf wird darauf geachtet, dass der Wert des Ausdrucks eine Instanz des herausgefundenen Typs ist.

Der Wert eines XQuery-Ausdrucks ist eine Sequenz von mehreren Knoten. Diese Knoten sind entweder einfach und haben einen Wert aus dem Wertebereich eines Basisdatentyps wie er in XML-Schema definiert ist oder die Knoten der Sequenz sind komplex. Das bedeutet,

dass sie vom Wert her einem der sieben möglichen Knotentypen entsprechen wie einem Element, Attribut oder Namensraumknoten. Dann besitzen diese Knoten einen Namen, einen Zeichenkettenwert oder einen typisierten Wert, welcher wiederum eine Sequenz ist.

Im Vorschlag zu XQuery sind spezielle Ausdrücke definiert, mit denen man Knoten auf ihre Typen explizit überprüfen oder eine Typumwandlung durchführen kann.

Typüberprüfung

Zur Typüberprüfung stehen dem Nutzer folgende Ausdrücke zur Verfügung:

- `INSTANCE OF [ONLY]`
- `TYPESWITCH`

Mit dem Operator `INSTANCE OF` kann der Wert eines Ausdrucks überprüft werden, ob er von einem speziellem Typ oder einem dessen Untertypen ist. Wenn das der Fall ist, dann liefert der Operator *true* zurück, ansonsten *false*. Mit der Angabe `ONLY` kann die Überprüfung zusätzlich eingeschränkt werden, so dass der Wert eines Ausdruck exakt vom angegebenen Typ sein muss, Untertypen sind dann nicht erlaubt.

Mit dem Operator `TYPESWITCH` kann ein Ausdruck auf seinen Typ überprüft werden und passend zum vorliegenden Typ können verschiedene Werte zurückgegeben werden.

Typumwandlung

Neben der Möglichkeit die Typen von Ausdrücken zu überprüfen, ist in XQuery die Möglichkeit gegeben, Ausdrücke in ihrem Typ umzuwandeln. Die Operatoren für diese expliziten Typumwandlungen sind folgende:

- `CAST AS`
- `TREAT AS`

Durch den Operator `CAST AS` wird eine Typumwandlung der Basisdatentypen oder elementaren Datentypen vorgenommen. Für nutzerdefinierte Typen ist es notwendig, dass der Anwender Funktionen implementiert, die diese Umwandlung realisieren.

Im Gegensatz dazu wandelt der Operator `TREAT AS` die Typen nicht um, sondern überprüft einen Ausdruck auf den angegebenen Typ oder einen der Untertypen und bricht ab, wenn der Ausdruck dem nicht entspricht.

Zusätzlich wird in XQuery noch eine implizite Typumwandlung vorgenommen, wenn Ausdrücke oder Funktionsparameter einen speziellen Typen verlangen. Es werden dabei gewisse Regeln zur Konvertierung benutzt, wenn der angegebene Ausdruck nicht dem geforderten Zieltyp entspricht. Schlägt diese Umwandlung fehl, dann wird eine Typfehler-Ausnahme ausgelöst und die Bearbeitung abgebrochen.

3.4.4 FLWOR-Ausdrücke

Neben Vergleichsausdrücken und Vergleichsoperatoren, welche zwei Operanden oder Ausdrücke z.B. auf Gleichheit oder Ungleichheit überprüfen, oder Boole'schen Ausdrücken und Pfadausdrücken, welche im Allgemeinen auf XPath 1.0 basieren, ist der wohl wichtigste Ausdruck in XQuery die so genannte FLWOR-Klausel, der For-Let-Where-Order-Return-Ausdruck.

Der Name basiert auf der Syntax dieses Ausdrucks, da die Grundklauseln `FOR`, `LET`, `WHERE`, `ORDER BY` und `RETURN` sind. Ein FLWOR-Ausdruck besteht somit im Einzelnen aus einer Reihe von `LET` und `FOR`-Klauseln, denen eine `WHERE`-Klausel und `ORDER BY`-Klausel folgen können und abschließend folgt die `RETURN`-Klausel.

Über die Klauseln `LET` und `FOR` kann man die Ergebnisse von Ausdrücken an eine oder mehrere Variablen binden. Bei einem Pfadausdruck oder XPath-Ausdruck entsprechen nach der Bindung die Variablen den durch einen XPath-Ausdruck selektierten Knoten. Variablen werden durch das Dollarzeichen `$` speziell ausgezeichnet, damit eine Unterscheidung zu Namen von Elementen oder Attributen aus dem Dokument erfolgen kann. Nach der Bindung von Werten an die Variablen sind diese unveränderbar, da im Gegensatz zu imperativen Programmiersprachen keine Werte zugewiesen werden können. Allerdings können die Variablenbindungen jederzeit überschrieben werden. Zusätzlich werden an die Variablen weitere Bedingungen gestellt. So ist die Variablenbindung nur innerhalb des aktuellen Ausdrucks sichtbar, was auch die im aktuellen Ausdruck eingeschlossenen Ausdrücke beinhaltet. Nach der Abarbeitung des aktuellen Ausdrucks sind die Variablen ungebunden und ein Zugriff auf eine ungebundene Variable hat eine Ausnahme zur Folge. Wurde eine Variablenbindung überschrieben, so ist im aktuellen Ausdruck die unmittelbar zuletzt geschehene Bindung gültig. Der Typ einer Variablen entspricht dabei dem Typ des Ergebnisses, das an die Variable gebunden wurde.

Bei der `LET`-Klausel wird nun die gesamte Ergebnismenge an eine Variable gebunden, während hingegen bei der `FOR`-Klausel jedes Element der Ergebnismenge einzeln an eine Variable gebunden wird. Dadurch wird die `FOR`-Klausel zur Iteration über die Ergebnismenge verwendet.

Über die optionale `WHERE`-Klausel kann die gebundene Ergebnismenge weiter eingeschränkt werden und über die `ORDER BY`-Klausel kann die Menge nach einem gewissen Kriterium geordnet werden.

Abschließend wird in der `RETURN`-Klausel das Ergebnis des XQuery-Ausdrucks konstruiert. Bei Verwendung der `FOR`-Klausel wird die `RETURN`-Klausel bei jedem Iterationsschritt evaluiert und das Ergebnis zu einem Zwischenergebnis hinzugefügt, das nach einem kompletten Durchlauf der Schleife als Gesamtergebnis zurückgegeben wird.

In Beispiel 3.1 ist ein FLWR-Ausdruck aufgeführt, entnommen aus [KM02]. In diesem Beispiel werden aus einem Dokument alle Elemente mit Namen `hotel` selektiert und mittels der `FOR`-Klausel darüber iteriert. In einer zweiten `FOR`-Klausel werden alle Kindelemente `zimmertyp` der Elemente `hotel` selektiert und an eine Variable gebunden, über die anschließend iteriert wird. In der `WHERE`-Klausel wird die Ergebnismenge dahingehend eingeschränkt, dass nur Zimmer mit einem Preis unter 100 verarbeitet werden. Dabei ist der Preis als Attri-

but `preis` der Elemente `zimmertyp` angegeben. In der `RETURN`-Klausel wird dann das Ergebnis konstruiert, welches ein XML-Konstrukt ist, in dem die Namen und Preise aller Hotels aufgeführt werden, die ein Zimmer anbieten, welches weniger als 100 kostet.

Beispiel 3.1 (Beispiel eines FLWR-Ausdrucks in XQuery)

```
<billighotels>{
FOR $h IN //hotel
    FOR $z in $h/zimmertyp
    WHERE $z/@preis <= 100
    RETURN <hotel>
        <name>{$h/@name}</name>
        <preis>{$z/@preis}</preis>
    </hotel>
}</billighotels>
```

3.5 XUpdate

Die Sprache XUpdate ist ein Entwurf der *XML:DB Initiative for XML Databases*. Dabei ist die XML:DB eine Unternehmung, die von der Industrie mit dem Ziel initiiert wurde, die Entwicklung und Spezifizierung des Datenmanagements in XML-Datenbanken voranzutreiben. Für das XUpdate-Projekt wurden von der XML:DB zwei Vorschläge [XML00b, XML00a] mit dem Status eines Arbeitsentwurfes (engl: *Working Drafts*) gemacht.

3.5.1 Einführung

Die Anforderungen, die in [XML00a] gestellt wurden, sind sehr allgemein gehalten und beziehen sich auf Möglichkeiten zur Anfrage und Änderungen auf XML-Dokumenten. Ferner sieht der Vorschlag zu XUpdate vor, dass Anfragen in XML-Syntax vorliegen, was die Vorteile hat, dass keine zusätzlichen Parser notwendig sind oder das die Anfragen in XML-Dokumenten eingebettet werden können. Allerdings kann die verwendete XML-Syntax die Komplexität und die Lesbarkeit der Anfragen zum Negativen hin beeinflussen.

Aufgrund der Beeinflussung durch die Standards zu XSLT und XPath ergibt sich das zugrunde liegende Datenmodell aus der Integration von XPath-Ausdrücken, welche wiederum auf das XML-Infoset abgebildet werden. Dadurch kann man für die Dokumente eine Baumstruktur annehmen, auf der die Anfragen arbeiten. Für dieses Datenmodell wurden folgende Knotentypen festgelegt: *Element*, *Attribut*, *Text*, *Processing-Instruction* und *Kommentare*. Hingegen fehlen die anderen Knotentypen aus dem XML-Infoset wie Namensraumknoten und der Dokumentknoten, welcher unter anderem im DOM definiert ist oder in XPath den Wurzelknoten darstellt.

3.5.2 Operationen

In [XML00b] sind die definierten Operationen aufgeführt und es wird eine kurze Beschreibung gegeben, wie diese Operationen in ihrer Syntax aussehen und welche Effekte sie auf die XML-Dokumente haben.

Im Vorschlag zu XUpdate wird durch die Angabe von XPath-Ausdrücken die Selektion realisiert. Diese Selektion dient aber nur zur Bestimmung der Knoten, welche weiterverarbeitet werden sollen. Somit verfügen alle Operationen über ein `select`-Attribut, welches einen XPath-Ausdruck als Wert besitzt.

Die nachfolgende Auflistung zeigt alle Operationen, wie sie in [XML00b] gefordert werden.

- xupdate:insert-before
- xupdate:insert-after
- xupdate:append
- xupdate:update
- xupdate:remove
- xupdate:rename
- xupdate:variable
- xupdate:value-of
- xupdate:if

Die Operationen sind dabei mit der ID `xupdate`, welche normalerweise zur Auszeichnung von XUpdate-Ausdrücken verwendet wird, aufgeführt. Der Namensraum-URI für XUpdate ist *http://www.xmldb.org/xupdate*.

Aus der Auflistung sieht man, dass auch ein Mechanismus zur Variablenbindung vorgesehen ist, welcher über die Operation `variable` realisiert wird. Mit der Operation `value-of` kann der Wert einer Variablen ausgewertet werden.

Neben diesen Operationen zum Einfügen, Anhängen, Ändern, Löschen und Umbenennen hat man in XUpdate noch die Möglichkeit, mehrere Operationen zusammenzufassen. Die Gruppierung erreicht man durch das Einschließen der Operationen in einem Element *xupdate:modifications*. Im Gegensatz zu XQuery existiert zu jedem Typ von Knoten ein expliziter Konstruktor, z.B. `xupdate:element` zur Erzeugung eines neuen Elementes oder *xupdate:text* zur Erzeugung eines neuen Textknotens.

3.6 Vergleich

In Tabelle 3.2 sind noch einmal die wichtigsten Kriterien einer Anfragesprache aufgeführt und für alle vorgestellten Ansätze angegeben, inwieweit diese den Forderungen gerecht werden.

Ein wirklicher Vergleich kann nicht vorgenommen werden, da das DOM und XPath keine "richtigen" Anfragesprachen in dem Sinne darstellen. Deshalb weisen sie in vielen Bereichen Lücken auf, die von Anfragesprachen erfüllt werden sollten. Allen voran die Orthogonalität, welche angibt, dass die Sprachkonstrukte uneingeschränkt miteinander kombierbar sein sollen, was aber bei DOM nicht möglich ist, da es lediglich eine Programmierschnittstelle darstellt. XQuery und XUpdate sind hingegen Vertreter von Anfragesprachen im Bereich von

Kriterium	**XPath**	**DOM**	**XQuery**	**XUpdate**
Deskriptivität	+	-	+	+
Adäquatheit	+	+	+	-
Optimierbarkeit	+	-	+	+
Orthogonalität	-	-	+	+
Abgeschlossenheit	+	+	+	+
Vollständigkeit	+	+	+	+
Datenextraktion	+	+	+	+
Datenmanipulation	-	+	-	+
Operationen auf dem Schema	-	-	-	-
Transaktionen	-	-	-	-

Tabelle 3.2: Vergleich der Ansätze zur XML-Verarbeitung

XML, welche als solche geplant waren, weswegen die Eigenschaften von Anfragesprachen von vornherein beim Entwurf berücksichtigt wurden. Änderungsoperationen sind im Vorschlag zu XQuery noch nicht vorhanden, aber für spätere Versionen vorgesehen. Aus der Tabelle erkennt man, dass die Unterstützung von Operationen auf dem Schema bei keinem der Mechanismen vorhanden ist. Die Entwicklung zu XQuery sieht diese Möglichkeit vor, ist in der aktuellen Version [W3C05] aber noch nicht integriert. Das Konzept der Transaktionen ist ebenfalls bei keinem der Ansätze vorhanden. Der Vergleich wird im Kapitel 5 um XSEL erweitert.

Kapitel 4

Existierende Ansätze

Aus dem Vergleich in der Tabelle 3.2 wird ersichtlich, dass die im XML-Umfeld entwickelten und existierenden Anfragesprachen entweder keine oder nur sehr wenige Änderungsoperationen definieren. Hinzu kommt, dass diese Änderungsoperationen für den Einsatz auf den Dokumenten konzipiert wurden und die Möglichkeit von Operationen auf Schemaebene nicht gegeben ist. Aus diesem Umstand heraus begründet sich die Notwendigkeit von Sprachentwicklungen speziell zur Umsetzung der Evolution auf der Schemaebene.

4.1 XEM

In den Arbeiten [RKS02] und [RCC+02] wird erläutert, wie der Entwurf zu XEM (**X**ML **E**volution **M**anagement) aussieht. Die Gründe für den Entwurf zu XEM liegen darin, dass die meisten Systeme, welche XML-Daten speichern und Zugriff auf diese gewähren, auf Datenbanksystemen aufbauen. Diese Datenbanksysteme wurden entweder um einen Typ XML erweitert oder die XML-Daten werden als einfache Zeichenkette oder als CLOB (**C**haracter **L**arge **Ob**ject) aufgefasst. Andere Ansätze sehen vor, die DTD auf ein Datenbankschema abzubilden, um die Daten so in eine Datenbank importieren zu können. Unabhängig vom Ansatz bedeutet Evolution in diesen Fällen, dass die Evolutionsschritte nicht auf dem zugrunde liegenden Schema der XML-Dokumente arbeiten, sondern auf das darunter befindliche Datenbankschema abgebildet werden und dieses dann ändern. XEM ist nun ein erster Entwurf für ein echtes XML-Management-System, bei dem Evolution auf dem Schema für XML stattfindet.

In den Arbeiten liegt den XML-Dokumenten die DTD als Schema zugrunde. Die Dokumente und die DTD werden als Graph definiert, die Dokumente speziell als Baum und es wird eine bidirektionale Abbildung zwischen DTD und Dokument festgelegt. Diese Abbildung dient nach vorgenommenen Änderungen dazu, die anzupassenden Teile im Dokument zu lokalisieren.

Neben diesen Definitionen wird eine Menge an Operationen definiert, die speziell für die Anwendung auf der DTD zugeschnitten sind. Zusätzlich wird eine Menge an korrespondierenden Operationen zur Adaption der Dokumente entworfen. Die DTD-Operationen sind mit einer kurzen Erläuterung ihrer Auswirkungen in Tabelle 4.1 aufgeführt und in Tabelle 4.2 die Dokumentoperationen.

Damit die Operationen die Wohlgeformtheit und Korrektheit nicht verletzten, werden Vor-

DTD-Operation	Erläuterung
createDTDElement(s,t)	erzeugt einen neuen Elementtyp mit Namen s und Inhaltsmodell t
destroyDTDElement()	löscht Elementtyp
insertDTDElement(e,i,q,v)	fügt Elementtyp e mit Quantor q, Standardwert v an der Position i in der DTD ein
removeContentParticel(i)	löscht den Inhalt eines DTD-Teils an der Position i
changeQuant(i, q)	ändert den Quantor des DTD-Teils an der Position i zum neuen Wert q
convertToGroup(start,end,t)	gruppiert alle DTD-Teile von der Position *start* bis zur Position *end* zu einem Typ t
flattenGroup(i)	löst die Gruppierung eines Typs an der Position i
addDTDAttr(s,t,d,v)	fügt Attributtyp mit Namen s und Typ t (Standardtyp d) und dem Standardwert v zu einem Elementtyp hinzu
destroyDTDAttr(s)	löscht Attributtyp mit Namen s aus der DTD

Tabelle 4.1: Übersicht der DTD-Operationen in XEM

Dokument-Operation	Erläuterung
createDataElement(e,v)	erzeugt neues Element mit Typ e und Wert v
addDataElement(e,i)	fügt Element e an der Position i im Dokument ein
destroyDataElement()	löscht Element aus dem Dokument
addDataAttr(s,v)	fügt ein Attribut s und Wert v zu einem Element hinzu
destroyDataAttr(s)	löscht ein Attribut mit Namen s aus Element

Tabelle 4.2: Übersicht der Dokumentoperationen in XEM

und Nachbedingungen festgelegt, welche zu beachten sind. Es wird keine Operation ausgeführt, wenn die Vorbedingungen nicht erfüllt sind und die Änderungen durch die Ausführung der Operation werden rückgängig gemacht, sofern die Nachbedingungen nicht gültig sind. Zu jeder definierten DTD-Operation wird in [RKS02] ein genauer Überblick gegeben, welche Vor- und Nachbedingungen gelten müssen und wie die Operationen für die Dokumente aussehen, die die Daten nach erfolgter DTD-Änderung adaptieren.

4.2 Inkrementelle Validierung

Ein wichtiger Teil in der konkreten Umsetzung der Evolution ist die inkrementelle Validierung. Nachdem ein Schema geändert wurde, sind die Dokumente unter Umständen nicht mehr valide und müssen angepasst werden. Über die inkrementelle Validierung kann man in einzelnen Schritten überprüfen, welche Eigenschaften welcher Teile in den Dokumenten nicht mehr erfüllt sind.

4.2.1 Validierung durch Regeln

Für die inkrementelle Validierung wird das Schema als Menge von gewissen Bedingungen aufgefasst, die erfüllt sein müssen, damit z.B. ein XML-Dokument valide bezüglich des Schemas ist. Diese Bedingungen werden nacheinander geprüft.

In [KSR02] werden vier Grundregeln definiert, die bei der inkrementellen Validierung überprüft werden müssen:

1) Quantorenbedingung

   (1.1) r.`minOccurs` $\leq \mid r \mid$

   (1.2) r.`maxOccurs` $\geq \mid r \mid$

2) Attributbedingung

   (2.1) a.`use=''required''` $\Rightarrow \exists\ a \in attrs$

   (2.2) a.`use=''optional''` $\Rightarrow (\exists\ a \in attrs)\ \vee\ (\forall\ attr \in attrs\ :\ attr \neq a)$

3) Validität der Elementknoten

4) Validität der Attributknoten

Die Quantorenbedingung sagt aus, dass die Anzahl des Elementes r in einer Instanz sich innerhalb der Grenzen bewegen muss, die durch den Wert des `minOccurs`- und `maxOccurs`-Attributes der Deklaration von r festgelegt wird.

Bei der Attributbedingung wird gesichert, dass das Auftreten eines Attributes a (in der Attributliste $attrs$ eines Elementes) dem Wert des `use`-Attributes seiner Deklaration folgt. Das bedeutet, dass ein Attribut auftreten muss, wenn der Wert des `use`-Attributes `required` ist. Ist der Wert des `use`-Attributes dagegen `optional`, dann kann das Attribut auftreten, muss es aber nicht. Bei `fixed` muss das Attribut mit einem bestimmten Wert auftreten und bei `prohibited` darf das Attribut nicht auftreten.

In der dritten Regel sind alle Bedingungen und Eigenschaften von Elementen gruppiert, welche überprüft werden und gültig sein müssen. Diese Eigenschaften sind der Name des Elementes und die Korrektheit des Typs. Dazu gehören alle im Typ deklarierten Attribute, deren Auftreten durch die zweite Regel sicher gestellt wird. Zum Typ gehören ebenfalls alle deklarierten Kindelemente, die valide sein müssen und in der richtigen Anzahl in der Instanz auftreten müssen, was durch die erste und dritte Regel gesichert wird.

Die Regel über die Validität von Attributknoten umfasst die Eigenschaften von Attributen, die eingehalten werden müssen. Dazu zählen der Name und der Typ der deklarierten Attribute und dass bei einem Element nicht zwei Attribute mit dem gleichen Namen auftreten dürfen. Ebenso muss die zweite Regel gelten.

In [KSR02] wird zusätzlich noch eine Menge von möglichen Änderungsoperationen, in Tabelle 4.3 aufgeführt, angegeben und für die einzelnen Operationen genau spezifiziert, welche Regeln bei der inkrementellen Validierung überprüft werden müssen.

DTD-Operation	**Regel**	Erläuterung
delElePassed *child*	1 (Bedingung 1.1)	löscht Kindknoten
delAttrPassed()	2	löscht Attributknoten
insElePassed *child* [before|after](*child*)	3	fügt Element ein
insAttrPassed(*attr*)	4	fügt Attribut hinzu
renameElt *child* to	3 & 1	benennt ELement um
renameAttr *child* to	2 (Bedingung 2.1) & 4	benennt Attribut um
replaceE *child* with c	1 & 3	ersetzt Elementknoten
replaceEV *child* with	1 & 3	ersetzt Elementwert eines Blattknotens
replaceA *child* with new attr	2 & 4	ersetzt Attributknoten
replaceAV *child* with	2 & 4	ersetzt Attributwert

Tabelle 4.3: Operationen und Regelbeachtung bei der inkrementellen Validierung

4.2.2 Validierung über Erfüllbarkeit / Baumautomaten

Der Vorschlag zur inkrementellen Validierung in [PV03] basiert auf der inkrementellen Validierung von Zeichenketten. Die Überprüfung, ob ein Wort einem regulären Ausdruck oder einer Grammatik genügt, ist der Ausgangspunkt zur Überprüfung, ob ein XML-Dokument einer DTD genügt. Die XML-Dokumente werden hierfür als Bäume betrachtet, auf denen eine Ordnung definiert ist. Darüber hinaus besitzen die Knoten des Baumes gewisse Bezeichner (engl: *label*). Durch die definierte Ordnung und die Konkatenation der einzelnen Knotenbezeichner werden die Zeichenketten gebildet. Es genügt, nur diese Zeichenketten auf Valididtität zu kontrollieren, da die Überprüfung der einzelnen Datenwerte im Dokument bezüglich der Gültigkeit ihres Definitionsbereiches nicht die Hauptschwierigkeit darstellt.

In einer DTD kann man die Elementnamen nicht von den für die Elemente angegebenen Typen trennen. Diese Einschränkung wird durch die Verwendung einer spezialisierten DTD [PV00] umgangen. Zwischen einer spezialisierten DTD und einem Baumautomaten besteht zusätzlich die Verbindung, dass sie beide eine reguläre Baumsprache definieren. Baumautomaten sind den Automaten, die Zeichenketten einlesen und akzeptieren oder zurückweisen, ähnlich. Aus diesem Grund wird die Validierung auf das Problem der Zeichenkettenvalidierung zurückgeführt.

Um auf der Sequenz der Knotennamen einen Baumautomaten anwenden zu können, wird eine Hilfsstruktur eingeführt und aufgebaut. Diese Hilfsstruktur ist ein B-Baum (siehe [HS99]), dessen Knoten so genannte Übergangsrelationen (engl: *transition relations*) sind. Diese Übergangsrelationen bilden die einzelnen Sequenzen auf eine Menge ab, deren Elemente Tupel aus zwei Automatenzuständen sind. Ein Tupel umfasst dabei den Ausgangszustand des Automaten und den Automatenzustand, in den die Sequenz ihn vom Ausgangszustand überführt. Dabei bilden die Blätter des B-Baumes einen Knotennamen auf einzelne Übergänge im Automaten ab. Die Elternknoten entsprechen der Zusammensetzung oder "Konkatenation" aus

ihren Kindknoten, so dass der Wurzelknoten des Baumes die Sequenz repräsentiert, die den Baumautomaten von einem Startzustand in einen seiner Endzustände überführt.

Bei Änderung der Knotennamen, Löschen oder Einfügen von Knoten wird der B-Baum der Hilfsstruktur von unten nach oben neu berechnet. Anschließend genügt zur Validierung der Sequenz die Kontrolle, ob sich in der neuberechneten Hilfsstruktur eine Übergangsrelation befindet, die den Startzustand und einen Endzustand des Automaten beinhaltet.

Eine weitere in [PV03] vorgeschlagene Möglichkeit zur Validierung ist der Weg über codierte Binärbäume. Dabei werden die XML-Bäume über spezielle Regeln in Binärbäume umgewandelt, welche anschließend durch einen Bottom-Up-Baumautomaten überprüft werden.

4.3 Evolution und Adaption mittels XSLT

Zur Ableitung der durchzuführenden Änderungen auf den Dokumenten wird untersucht, wie man die Änderungen auf der DTD beschreiben kann. Existierende Möglichkeiten sind die genaue Beschreibung der Änderungen, die auf einer DTD durchgeführt werden können, z.B. durch die Angabe eines Regelwerkes. Ein anderer Ansatz ist der Vergleich zweier DTDs und die Ableitung der zwischen ihnen existierenden Unterschiede. Es liegen aber nur unzureichende semantische Informationen vor, so dass dieser Ansatz nur in gewissen Fällen erfolgreich ist. Neben dem Vergleich zweier DTDs können zwei XML-Dokumente miteinander verglichen werden. Die beiden XML-Dokumente sind valide bezüglich einer DTD und über die Unterschiede zwischen den Dokumenten kann man die Unterschiede der beiden DTDs ableiten. Bei diesem Ansatz liegen ebenfalls nicht genügend semantische Informationen vor, so dass unter anderem nicht erkannt werden kann, dass zwei Dokumente mit gleicher Struktur sich nur in den Namen ihrer Elemente unterscheiden.

Aufgund dieser Mängel wird in [Zei01] die erste Möglichkeit gewählt und beschrieben, wie das Regelwerk in Form der möglichen Operationen aussieht. Die Art des Schemas wird dabei als DTD festgelegt.

In diesem Regelwerk wird grundlegend unterschieden, ob die durchzuführenden Änderungen auf Element-, Attribut- oder Entitätenebene ablaufen. Zusätzlich wird der Begriff der Informationskapazität eingeführt. Damit wird eine weitere Unterscheidung vorgenommen, ob sich bei einer Transformation die Kapazität erhöht oder reduziert oder ob die Informationskapazität gleichbleibt und wie sich die Informationen der Dokumente verhalten. Anschließend wird für die Operationen, die im Zuge der Evolution möglich sein sollen, eine informelle Beschreibung ihrer Wirkung angegeben und so das Regelwerk aufgebaut. Einen genauen Überblick über die Operationen und welche Transformationen auf der DTD erlaubt sind, kann [Zei01] entnommen werden.

Nach den Änderungen auf dem Schema werden die Daten mittels XSLT dahingehend transformiert, dass sie bezüglich der evolvierten DTD wieder valide sind.

4.4 Evolution mittels logischer Ausdrücke

Ein anderer Ansatz zur Formatevolution sieht die Verwendung von logischen Ausdrücken vor. In [Fau01] werden die möglichen Änderungen, die durch eine Menge an definierten Operationen beschrieben werden, in logische Ausdrücken zusammengefasst, so genannten Trafo-Ausdrücken. Diese Trafo-Ausdrücke dienen als Basis für die Schema- und Dokumenttransformation.

Das Schema, in XML-Schema-Notation, wird als Menge von regulären Ausdrücken über einem bestimmten Alphabet betrachtet. Die Trafo-Ausdrücke werden unter Berücksichtigung der durchzuführenden Operation und dem Ausgangsformat, einem regulären Ausdruck, gebildet und sind so angelegt, dass sie den Dokumenten in den Blättern ähneln, weswegen sie die gleichen Dokumentfragmente wie die regulären Ausdrücke akzeptieren. Für einen genauen Überblick zur Bildung der Trafo-Ausdrücke wird auf [Fau01] verwiesen.

Ein Evolutionsschritt besteht aus zwei Teilen, der Formattransformation und der Dokumenttransformation. Die Formattransformation beginnt zunächst mit der Interpretation des Trafo-Ausdrucks und der Anwendung auf dem Schema, wobei die Änderungen, die das Format modifizieren, erst am Schluss übernommen werden, da ansonsten bei Umbenennungen die Korrespondenz zwischen Schema und dem Trafo-Ausdruck verloren gegen kann. Aus dem Trafo-Ausdruck zur Schemaänderung wird dann ein Trafo-Ausdruck zur Adaption der Dokumente generiert.

Bei der Änderung der Dokumente, der Dokumenttransformation, wird zuerst über den Identifikator, dem Namen des deklarierten Elementes, die Position dieses Elementes im Dokument festgestellt. Beim Parsen wird zusätzlich durch die Überprüfung, ob die Unterelemente mit denen im Typ deklarierten übereinstimmen, eine Typzuweisung realisiert. Dadurch kann man auf den Trafo-Ausdruck, der zuvor das Schema modifiziert hat, zugreifen. Dieser Ausdruck wird zunächst auf die Kindelemente des Elementes angewendet und diese auf diesem Weg transformiert. Danach werden der Ausdruck und die Liste der akzeptierten Elemente an das Elternelement weitergereicht, so dass dadurch die Angaben von Quantoren behandelt werden können.

4.5 SERF

Da Datenbanksysteme wesentlich früher entwickelt wurden als die Technologie um XML und dadurch länger existieren als die Systeme, welche die Daten mittels XML strukturieren und speichern, sind bereits viele Ansätze zur Umsetzung der Evolution im Bereich der Datenbanksysteme vorhanden. Wegen der zahlreichen Unterschiede zwischen der Struktur von XML-Dokumenten und der Struktur in relationalen Datenbanken kann man die Ansätze nur bedingt übernehmen und auf XML-Schemata anwenden.

Die logische Struktur der Daten in objektorientierten Datenbanksystemen lässt sich besser mit der Struktur von XML vergleichen. Darunter ist zu verstehen, dass sich die hierarchische und baumähnliche Struktur von XML-Dokumenten leicht auf objektorientierte Strukturen abbilden lassen.

In OODBS existieren vordefinierte Operationen zur Umsetzung von Schemaänderungen, allerdings sind diese sehr eingeschränkt. Ein Versuch, dem Nutzer mehr Freiheiten zu geben, was die Evolution und allgemein die Transformation betrifft, ist SERF.

Die Grundidee ist, dass über die vorgegebenen Operationen im OODBS komplexe Transformationen zusammengesetzt werden können. Diese werden dann vereinfacht und verallgemeinert, bekommen einen Namen und werden als Vorlagen, so genannten Templates, gespeichert, damit sie wieder verwendet werden können. Die Verallgemeinerung zum Template hin umfasst unter anderem die Substitution von konkreten Namen durch Variablenbezeichner. An die Templates können dann konkrete Werte als Parameter übergeben werden.

Der Ablauf der Evolution gliedert sich in vier große Teilschritte:

- Abfragen der Metadaten
- Ändern des Schemas
- Abfragen der zu ändernden Objekte
- Ändern der Objekte

Am Anfang müssen die Metadaten über die Klassen und Objekte des Systems abgefragt werden, damit die komplexen nutzerdefinierten Transformationen in Templates zusammengefasst werden können, welche dann auf alle Klassen anwendbar sind. Diese Metadaten werden dazu benutzt, die Verallgemeinerung der zusammengesetzten Transformationen durchzuführen.

Anschließend werden die Transformationen durchgeführt und die darin gekapselten im System vordefinierten Operationen auf das Schema angewendet.

Der dritte Schritt beinhaltet die Identifikation aller Objekte, die durch die Schematransformation beeinflusst wurden. Diese werden dann im vierten Schritt dahingehend angepasst, dass sie bezüglich des geänderten Schemas gültig sind. Die Anpassung geschieht wieder durch eine Anfragesprache wie OQL im Bereich der OODBS.

Eine genaue Beschreibung zu SERF kann [CJR98b, CR99, CJR98a] entnommen werden.

Sofern man ein XML-Informationssystem hat, in dem Grundprimitive für Änderungen bereits definiert sind, die Schemata in XML-Schema-Notation vorliegen und eine Anfragesprache wie XQuery einsetzbar ist, kann der Grundgedanke von SERF auf das System übertragen werden. XML-Schema ist hierbei eine Bedingung, da es in XML-Syntax vorliegt und deswegen mittels XQuery verarbeitet werden kann, was für den Schritt 1 der Metadatenextraktion notwendig ist.

4.6 Sangam

Ein ähnlicher Ansatz wie bei SERF wird mit dem Framework Sangam [CR03] vorgestellt. Die Grundideen und Grundschritte sind die gleichen. Der Unterschied besteht darin, dass die benötigten Grundoperationen als Operationen auf einer Algebra definiert sind. Diese Algebra wird Cross-Algebra genannt und lässt sich auf die Algebren, die einer Anfragesprache wie SQL oder XQuery zugrunde liegen, abbilden.

Die Grundoperationen dienen dann zur Modellierung komplexer Transformationsaufgaben auf dem Schema, welche ebenfalls als Templates gespeichert werden. Dadurch erhält man wie bei SERF die Möglichkeit, diese Transformationen zu optimieren, ähnlich den algebraischen Optimierungen bei SQL.

Da Sangam für Transformationen auf verschiedenen Schemata eingesetzt werden soll, muss das Framework mit verschiedenen Datenmodellen umgehen können. Als allgemeines Datenmodell dient deswegen ein Graph, der so genannte Sangam-Graph, der spezielle Eigenschaften besitzt. Er ist ein gerichteter Graph, wobei jedem Knoten des Graphen zugeordnet wird, ob er einen komplexen oder atomaren Typ besitzt. Weiterhin werden zwei verschiedene Kantentypen unterschieden, eine Besitzt-Kante (*property*) und eine Enthält-Kante (*containment*). Zusätzlich wird eine Kantenbeschriftung vorgenommen, die angibt, mit welcher Anzahl die Knoten in Beziehung stehen, ähnlich wie bei einem ER-Diagramm[1]. Auf den von einem Knoten wegführenden Kanten wird außerdem eine lokale Ordnung definiert.

Die Bildung komplexer Transformationen geschieht in zwei Schritten. Zuerst erfolgt die Definition der grundlegenden Operationen, als *Bricks* bezeichnet, vom englischen Wort für Ziegelstein (engl: *brick*) abgeleitet und danach die Möglichkeiten des Zusammensetzens und Kombinierens der Operationen, als *Mortar* bezeichnet, vom englischen Wort für Mörtel (engl: *mortar*).

4.6.1 Bricks: die Grundoperationen

Die Grundoperationen in Sangam sind folgende:

- Cross-Operator
- Connect-Operator
- Smooth-Operator
- Subdevide-Operator

Mit Hilfe des Cross-Operators werden Knoten des Graphen verarbeitet. Als Eingabe dient ein Knoten, aus dem eine Ausgabe erzeugt wird, wobei die Kindknoten des Eingabeknotens vollständig und ohne Veränderungen als Kindknoten des erzeugten übernommen werden.

Mit dem Connect-Operator lassen sich Kanten zwischen Knoten erzeugen, sofern diese Knoten zuvor mit Hilfe des Cross-Operators erzeugt wurden. Die neue Kante übernimmt alle Eigenschaften der Kante, die zwischen den Knoten existiert, die als Eingabe für den Cross-Operator gedient haben. Also jene Knoten, aus denen die Knoten erzeugt wurden, zwischen denen die neue Kante angelegt werden soll.

Der Smooth-Operator kombiniert zwei Beziehungen zwischen Knoten zu einer neuen. Die Knoten in der neuen Beziehung müssen dabei zuvor mit dem Cross-Operator aus den Knoten der alten Beziehung erzeugt worden sein. Der dazu inverse Operator ist der Subdevide-Operator, der eine gegebene Beziehung aufspaltet, sofern die neuen Knoten mittels des Cross-Operators aus den alten erzeugt wurden.

[1] ER (**E**ntity **R**elationship)

4.6.2 Mortar: Techniken der Kombination

Die zuvor kurz beschriebenen Operationen können zu neuen und komplexeren Transformationsanweisungen kombiniert werden. Es werden dabei zwei Grundtechniken unterschieden, die Verbindung von Operationen über Kontextabhängigkeit (engl: *context dependency*) und über Ableitung (engl: *derivation*). Eine Kombination der zwei Techniken ist ebenfalls möglich.

- Kontextabhängigkeit
- Ableitung
- Kombination Kontextabhängigkeit und Ableitung

Bei der Kombination durch Kontextabhängigkeit können Operationen so miteinander verbunden und kombiniert werden, dass die einzelnen Operationen auf Teilgraphen arbeiten und dadurch ein neuer Gesamtgraph erzeugt wird. Die Auswertung dieser komplexen Ausdrücke erfolgt dabei von rechts nach links, so dass sichergestellt wird, dass die Kindknoten vor ihren Elternknoten verarbeitet werden. Die Auswertungsreihenfolge der Geschwisterknoten ist dabei nicht von Bedeutung. Die Kontextabhängigkeit impliziert für einen aus op_1 und op_2 zusammengesetzten Ausdruck $op_1 \rightarrow op_2$, welche Informationen durch op_1 verarbeitet werden. Diese Informationen beschreiben die Eingabe für und die Ausgabe von op_2 und welche Operation (inklusive der Bedingungen) sich hinter op_1 verbirgt. Die Operation op_1 ist sozusagen Bedingung für op_2.

Bei der Ableitung können Operationen so miteinander kombiniert oder verschachtelt werden, dass die Ausgaben der einen als Eingabe einer anderen Operation dienen. Bei einem Ausdruck der Form $op_3(op_1(\cdots), op_2(\cdots))$ wird gesagt, dass die Ausgabe der Operation op_3 aus den Ausgaben von op_1 und op_2 abgeleitet wurde.

Die zusammengesetzten Operationen kann man als Graph darstellen, wobei die Abarbeitung der einzelnen Operationen in Post-Order stattfindet. Dies bedeutet, dass zuerst die Operationen abgearbeitet werden, deren Ausgabe als erneute Eingabe dienen, oder die Bedingung für andere Operationen sind.

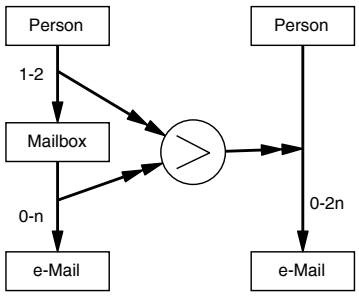

Abbildung 4.1: Beispiel für den Smooth-Operator

Abbildung 4.1 zeigt den Smooth-Operator wie die Beziehung zwischen Person und Mailbox und die Beziehung Mailbox und e-Mail zu einer neuen zusammenfasst. Damit der Operator auf diese beiden Knoten angewendet werden kann, müssen zunächst zwei neue Knoten mit

dem Cross-Operator erzeugt werden. Wie der Cross-Algebra-Graph zu den drei kombinierten Operationen aussieht, ist in Abbildung 4.2 gezeigt. Im Beispiel liegt Kontextabhängigkeit vor. Abbildung 4.3 zeigt einen kompletten Sangam-Graphen mit Kantenbeschriftung und der definierten Ordnung und zeigt, wie sich dieser Graph ändert, wenn der Smooth-Operator angewendet wird. Die dabei betroffenen und zusammengezogenen Beziehungen, die sich als Kanten im Sangam-Graph darstellen, sind dicker eingezeichnet.

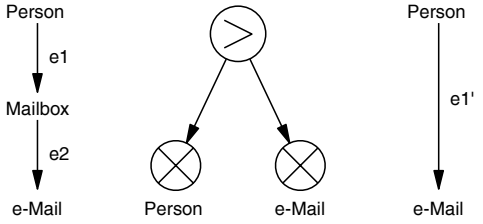

Abbildung 4.2: Cross-Algebra-Graph der Operationen im Beispiel

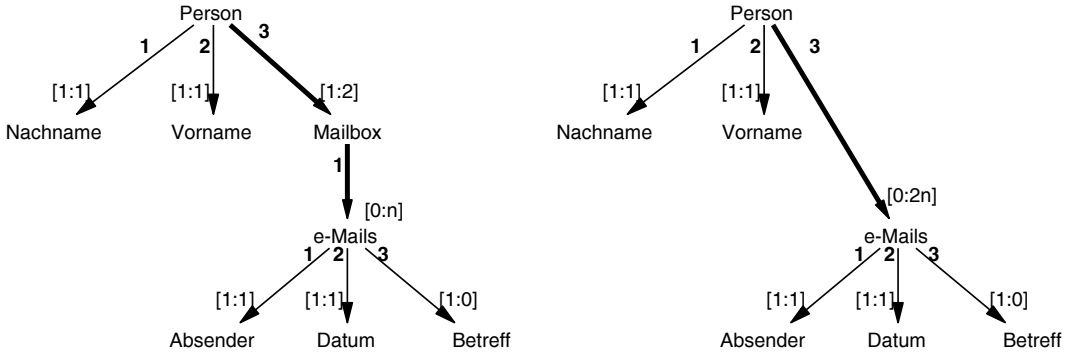

Abbildung 4.3: Änderung im Sangam-Graph

In einer konkreten Umsetzung gibt es verschiedene Arten der Abarbeitung. Entweder bildet man die Algebraausdrücke auf konkrete Anfragesprachen wie SQL oder XQuery ab, oder man benutzt eigene Algorithmen wie in [CR03], welche XML-Schemata oder Datenbankschemata zuerst in einen Sangam-Graphen übersetzen, auf dem die Cross-Algebra-Ausdrücke direkt angewendet werden können.

4.7 Evolution mittels Graphenoperationen

Ein weiterer Ansatz der Beschreibung von Evolution auf einem Schema ist die Betrachtung des Schemas als Graph und der Definition von Operationen, die diese Graphenstruktur manipulieren. In [Sim04, Coo03, CS03] wird die als Schema zugrunde liegende DTD als Graph interpretiert. Die DTD wird als eine Abbildung von einem Alphabet V in die Klasse C von Sprachen über V angenommen:

$$DTD : V \longrightarrow C$$

Um aus der DTD einen Graphen aufzubauen, wird diese zuerst vereinfacht. Eine vereinfachte DTD unterscheidet sich dadurch, dass als einzige Quantorenangabe "*" zugelassen ist

und nur die Gruppierung “(...)” bei Elementdefinitionen gültig ist. Eine Umformung zu einer vereinfachten DTD erreicht man durch

- Substitution der Entitäten durch ihre Definitionen,
- Interpretation von `REQUIRED` und `IMPLIED` als Terme von “*” “ ”,
- Definition von V als Vereinigung der Attribut- und Elementnamen,
- Aufnahme jeder Element- (`<!ELEMENT>`) und Attributlistendefinition (`<!ATTLIST>`) in eine Tabelle von gewissen Abbildungen,
- und Ersetzen aller Alternativen “|” durch die Gruppierung.

Elemente und Attribute werden somit als Knoten der Graphen betrachtet und die Kanten zwischen den Knoten beschreiben die Verschachtelung zwischen ihnen oder deren Eltern-Kind-Beziehung. Anschließend wird der DTD-Graph noch faktorisiert. Das schließt unter anderem das Zusammenfassen der Knoten von Elementen und ihren Attributen zu einem Knoten ein, wobei Attribute, die von mehreren Elementen verwendet werden, weiterhin als eigenständige Knoten im faktorisierten DTD-Graphen auftreten. Wenn Elemente Kinder besitzen, die wiederum nur ein einziges Kindelement besitzen, dann werden diese ebenfalls zu einem Knoten zusammengefasst. Ein genauer Überblick und eine formale Beschreibung der Faktorisierung kann [Sim04] entnommen werden.

- AddAttribute
- RemoveAttribute
- AddEdge
- RemoveEdge
- AddVertex
- RemoveVertex
- MergeVertex
- SplitVertex

Die definierten Operationen arbeiten auf dem faktorisierten DTD-Graphen. Die genauen Bedingungen für das Ausführen der Operationen und welche Veränderungen sie genau verursachen sind in [Sim04] aufgeführt. In den Arbeiten wurden jedoch keine Vorschläge zur Adaption der Daten gemacht.

Kapitel 5

Anfragesprache XSEL

Es existieren verschiedene Entwürfe zu Anfragesprachen im XML-Bereich, wie in Kapitel 3 gezeigt wurde. Diese Anfragesprachen oder die vorgestellten Mechanismen wie XPath dienen zur Selektion von Dokumentteilen, zur Datenextraktion und zur Weiterverarbeitung. In manchen Ansätzen sind Änderungsoperationen schon vorhanden oder sie befinden sich noch in der Entwicklung wie bei XQuery. Im Speziellen sind sie zur Anwendung auf XML-Dokumenten zugeschnitten, so dass mit deren Hilfe Daten aus den XML-Dokumenten extrahiert werden können. Aus diesem Grund sind sie auch auf XML-Schemata anwendbar, da XML-Schema ein XML-Dialekt ist. Mit XML-Dialekt ist dabei eine konkrete Ausprägung von XML gemeint, d.h. die Dokumente des Dialekts sind gültige wohlgeformte XML-Dokumente. Die vorgestellten Anfragesprachen sind jedoch nicht speziell zur Umsetzung von Schemaevolutionsschritten entworfen.

Der Entwurf zur Sprache XSEL (**X**ML-**S**chema **E**volution **L**anguage) ist speziell zur Evolutionsumsetzung auf XML-Schemata geschehen. Die definierten Operationen sind hingegen allgemein gehalten, das bedeutet, dass beim Entwurf auf die Definition von schemaspezifischen Operationen wie Typkonstruktoren verzichtet wurde. Dadurch können die Anfragen der Sprache XSEL auf jedes XML-Dokumentangewendet werden.

Für das den Dokumenten zugrunde liegende Schema wird vorausgesetzt, dass es in XML-Schema-Notation [W3C04a] vorliegt. Es wird weiterhin einschränkend festgelegt, dass das XML-Schema in der Normalform XSDNF (**X**ML **S**chema **D**ocument **N**ormal **F**orm) [Tie03] vorliegen muss, weil die Normalform eine wichtige Grundlage für die später folgende Abbildung zwischen Schema und Dokument ist. Eine kurze Erläuterung zu der Normalform ist in Abschnitt 5.4.1 gegeben.

5.1 Einführung

Aufgabe dieser Arbeit ist die Erstellung einer Anfragesprache zur Manipulation von XML-Schemata. Zu deren Umsetzung wurden existierende Ansätze betrachtet und verglichen, was unter anderem die Möglichkeiten der Manipulation betrifft, die Anzahl und den Umfang der erforderlichen Operationen und wie die Auswirkungen dieser Operationen in den Dokumenten aussehen. Der nachfolgend vorgestellte Sprachentwurf umfasst nur Operationen zur Änderung des XML-Baumes, deswegen wird fortan von XSEL als einer Änderungssprache gesprochen.

Eine weitere Anforderung an den Sprachentwurf ist der Einsatz der XML-Syntax. Das bedeutet, dass die Anfragenkonstrukte ein XML-Dialekt sind. Dadurch müssen neben dem ohnehin eingesetzten XML-Parser keine zusätzlichen Parser eingesetzt werden, um die Anfragen zu bearbeiten. Einzig der Einsatz spezieller Module zur Anfragenausführung ist notwendig. Ein genauer Überblick über eine konkrete Implementation der Module ist im Kapitel 6 gegeben. Diese Module werden zusammenfassend als Anfrageprozessor bezeichnet.

5.1.1 Wiederverwendbarkeit

Die Anfragesprache XSEL ist in ihrem Entwurf so allgemein gehalten, dass sie im Rahmen der Evolutionsdurchführung eine Wiederverwendbarkeit besitzt. Der Entwurf sieht Anfragen vor, die an beliebige XML-Dokumente gestellt werden können und so die einzelnen Dokumente manipulieren und transformieren. Dadurch wird die Anfragesprache von der Semantik der Daten unabhängig. Die einzelnen Anfragen müssen also nicht wissen, ob sie auf ein Schema oder ein beliebiges XML-Dokument angewendet werden.

Die einzelnen Operationen im Sprachumfang sind so angelegt, dass sie in ihrer Syntax unabhängig vom darunter liegenden Datenmodell interpretiert werden können. Dies bedeutet, dass der Anfrageprozessor die Anfragen interpretiert und die in den Anfragen geforderten Änderungen auf das Datenmodell umsetzt.

In der konkreten Umsetzung eines Anfrageprozessors, vorgestellt in Kapitel 6, wird die Sprache auf die DOM-Repräsentation des Dokumentbaumes angewendet, der mit Hilfe der Operationen im DOM bearbeitet und verändert wird. Die einzelnen Anfragen werden somit auf die entsprechenden DOM-Operationen abgebildet.

Unter dem Aspekt der Wiederverwendbarkeit ist nun Folgendes zu verstehen. Nachdem die Anfragen an ein Schema gestellt wurden, welches dadurch verändert und evolviert wurde, werden zur Adaption der Daten erneut XSEL-Anfragen generiert. Dementsprechend ist keine andere Sprache zur Dokumentanpassung notwendig, da XSEL allgemein auf dem XML-Format arbeitet. Der Anfrageprozessor generiert nach erfolgter Umsetzung auf dem Schema eine Liste an neuen XSEL-Anfragen. Indem diese Anfragen an die vom Schema definierten Dokumente gestellt werden, erfolgt die notwendige Adaption der Dokumente.

Einzig bei der Anpassung von eventuell vorliegenden XQuery-Anfragen muss die Anpassung auf anderem Weg geschehen, da XQuery eine eigenständige Notation besitzt. Zwar existiert parallel zu der eigenständigen Syntax für XQuery auch eine XML-Syntax [W3C03], XQueryX genannt, aber sie findet selten Einsatz.

5.2 Operationen

Im folgenden Abschnitt werden die Operationen vorgestellt, die im Umfang von XSEL definiert sind. Es wird erklärt, wie die Operationen syntaktisch als Anfragen aussehen und wie die Funktionsweise und die Effekte auf den Dokumenten aussehen. Zusammenfassend sind mit der Funktionsweise auch die Bedingungen gemeint, die beschreiben, wann die Anfragen ausgeführt oder nicht ausgeführt werden.

5.2.1 Überblick

Der Operationsumfang ist so ausgelegt, dass mit Hilfe der Operationen Baum-Strukturen manipuliert werden können. Genauer gesagt, arbeiten die Operationen auf den Knoten der Bäume. Die nachfolgende Auflistung zeigt die definierten Operationen im Überblick:

- **add** : fügt neue Knoten hinzu
- **insert_before** : fügt neue Knoten vor einem bestimmten Knoten ein
- **insert_after** : fügt neue Knoten nach einem bestimmten Knoten ein
- **move** : verschiebt Knoten im Baum
- **move_before** : verschiebt Knoten vor einen bestimmten Knoten im Baum
- **move_after** : verschiebt Knoten hinter einen bestimmten Knoten im Baum
- **replace** : ersetzt Knoten durch neue
- **delete** : löscht Knoten aus dem Baum
- **rename** : benennt einen Knoten um
- **change** : ändert den Wert eines Knotens

Der Name der Operation gibt bereits an, welche Änderung durchgeführt werden soll. Es müssen aber noch weitere Informationen bereitgestellt werden, um die Operationen ausführen zu können. Da die Anfragen speziell auf den Knoten des Baumes arbeiten, unabhängig davon, ob diese Knoten nun Elemente, Attribute, Text oder von einem anderen Typ sind, muss eine Möglichkeit gegeben sein, genauer zu spezifizieren, welcher Knoten manipuliert werden soll. Dies geschieht über den Wert des `select`-Attributes der Anfragen. Als gültige Werte werden nur XPath-Ausdrücke zugelassen, mit denen man die zu ändernden Knoten selektieren kann.

Neben der Selektionsangabe sind bei einigen Operationen zusätzliche Angaben notwendig. Bei der Operation *rename* muss zum Beispiel neben der Knotenspezifizierung ein neuer Name für den Knoten angegeben werden. Bei der Operation *delete* sind solche zusätzlichen Angaben nicht notwendig.

Zusammenfassend wird festgelegt, dass alle Operationen die Wohlgeformtheit und die Validität des Schemas nicht verletzen dürfen. Das bedeutet, dass die Operationen nur dann ausgeführt werden, wenn sie die Bedingungen an die Struktur von XML-Schema oder allgemein von XML-Dokumenten nicht verletzten und wenn sie nicht gegen die Spezifikationen von XML-Schema verstoßen. Das geänderte Schema muss nach erfolgtem Evolutionsschritt immer noch gültig und valide bezüglich seines Schemas[1] sein. Der Grund für diese Festlegung ist die Anforderung der Abgeschlossenheit[2] an XML-Anfragesprachen. Da davon ausgegangen wird, dass ein gültiges und valides XML-Dokument als Eingabe für eine Anfrage vorliegt und die

[1] die Definition von XML-Schema ist durch ein XML-Schema und eine DTD festgelegt: http://www.w3.org/2001/XMLSchema.xsd und http://www.w3.org/2001/XMLSchema.dtd

[2] siehe 3.1

Abgeschlossenheit besagt, dass das Ergebnis einer Anfrage als Eingabe einer neuen Anfrage dienen kann, muss nach Ausführung einer Anfrage ein Dokument gültig sein muss bzw. valide bezüglich eines Schemas.

Zeichen	Entität	ASCII-Entität
<	<	<
>	>	&#62;
&	&amp;	&
“	&quot;	&#34;
‘	&apos;	&#39;

Tabelle 5.1: vordefinierte Entitäten (predefined entities)

5.2.2 add-Operation

Mit Hilfe der *add*-Operation wird an einen beliebigen Knoten ein beliebiges XML-Fragment entweder als neuer Kindknoten, als Unterbaum oder als ein Attribut angehängt. Die Angabe des XML-Fragmentes ist dabei als Zeichenkette vorgesehen. Damit die Anfrage als solches weiterhin wohlgeformt und korrekt ist, müssen in der Zeichenkette spezielle Zeichen ersetzt werden und als deren Entität auftreten. Diese Entitäten sind die so genannten vordefinierten Zeichen-Entitäten (engl: *predefined character entities*). In Tabelle 5.1 ist eine Übersicht über diese Entitäten und die Zeichen, die sie repräsentieren, aufgeführt. Für eine genauere Beschreibung zur Thematik von Entitäten in XML siehe [W3C04e].

In der Umsetzung als Anfrage sieht die *add*-Operation syntaktisch so aus, wie es in Beispiel 5.1 gezeigt wird.

Beispiel 5.1 (Syntax der add-Operation)

```
<add select="..." content="..." />
```

Das `select`-Attribut gibt, wie schon erwähnt, den Knoten an, der selektiert und anschließend so manipuliert wird, dass der Wert des `content`-Attributes als Kindknoten oder als Attribut hinzugefügt wird. Die Selektion geschieht mit Hilfe eines XPath-Ausdrucks. An alle Knoten, die durch den Ausdruck selektiert werden, wird dann der Wert vom `content`-Attribut angehängt. Ein Beispiel, wie der Wert des `select`-Attributes und wie der hinzuzufügende Inhalt, also der Wert des `content`-Attributes, aussehen kann, ist in Beispiel 5.2 gezeigt.

Beispiel 5.2 (add-Operation zum Einfügen eines Elementes)

```
<add
    select="//xs:complexType[@name='kontakt']/xs:sequence"
    content="&lt;xs:element
            name=&quot;Nachname&quot;
            type=&quot;xs:string&quot;/&gt;"
/>
```

Es werden zunächst alle Elemente mit dem Namen `xs:complexType` selektiert, die irgendwo unterhalb des Wurzelknotens auftreten und die ein Attribut `name` aufweisen, welches den Wert `kontakt` hat. Anschließend werden aus dieser Menge alle Knoten selektiert, die den Namen `xs:sequence` haben und ein Kind von `xs:complexType` sind. Dann wird an jeden dieser Knoten ein neues Element als Kindknoten angehängt. Das neue Element hat den Namen `xs:element` und besitzt zwei Attribute, ein Attribut `name` mit den Wert `Nachname` und ein Attribut `type` mit dem Wert `xs:string`.

Abbildung 5.1 zeigt, wie sich die Baumstruktur eines konkreten Beispiels ändert. Es handelt sich um einen Ausschnitt des Schemas aus Anhang B, das als fortführendes Beispiel dient.

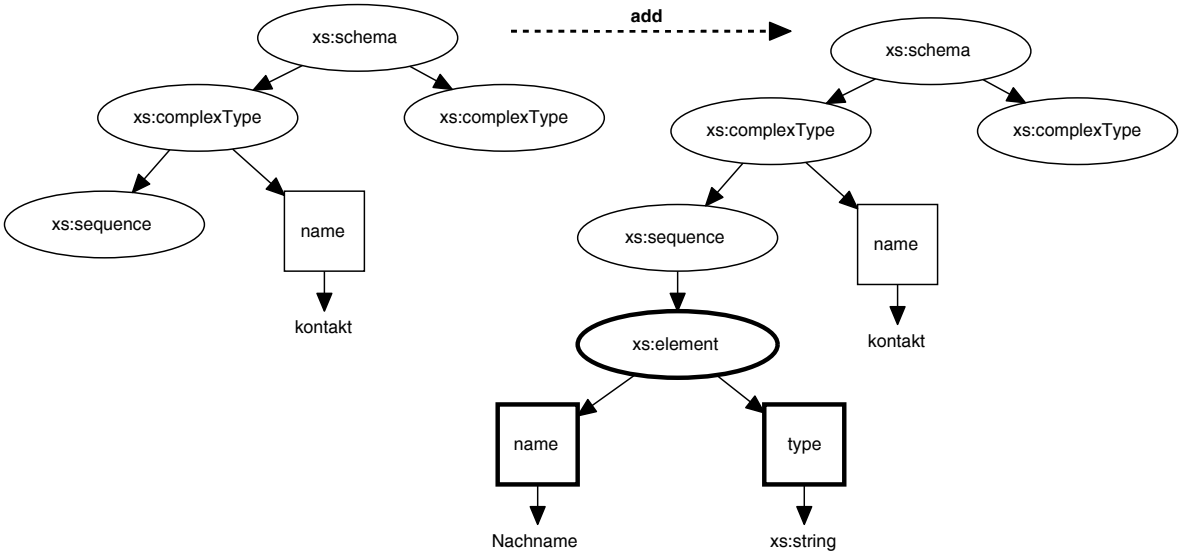

Abbildung 5.1: Beispiel add-Operation für das Einfügen eines Elementes

Im Beispiel wurde ein einer Elementdeklaration entsprechendes XML-Fragment eingefügt. Will man nun zu einem Element ein Attribut hinzufügen, dann muss dieses Element selektiert werden und der Wert des `content`-Attributes der Anfrage muss folgendem Muster entsprechen: `attributname=attributwert`. Auf diese Weise wird an die selektierten Elemente ein neues Attribut mit dem Namen `attributname` und dem Wert `attributwert` angefügt. Beispiel 5.3 zeigt eine Anfrage, welche im Schema an alle Knoten mit dem Namen `xs:element` das Attribut `maxOccurs` anhängt.

Beispiel 5.3 (add-Operation zum Einfügen eines Attributes)

```
<add
    select="//xs:element"
    content="maxOccurs=5"
/>
```

Ein dritter Fall liegt vor, wenn der Wert des `content`-Attributes weder mit der Entität `<` beginnt, noch exakt ein Gleichheitszeichen besitzt. In dem Fall wird der Wert als Textknoten eingefügt. Möchte man textuellen Inhalt in der Form `foo=bar` einfügen, dann muss die Entität des Gleichheitszeichen aufgeführt werden.

Bedingungen für die Ausführung

Es wird davon ausgegangen, dass das zu evolvierende Schema in Normalform vorliegt. Aus diesem Grund werden *add*-Operationen auch zurückgewiesen, wenn sie die Definition der Normalform verletzten. Das gilt ebenso für alle anderen Operationen. Als Beispiel wäre das Hinzufügen einer Elementdeklaration als Kind des Schema-Knotens zu nennen. Die Operation würde nicht ausgeführt werden, wenn schon eine Elementdeklaration als Kindknoten des Schemaknotens existiert, da die Normalform nur die Existenz einer globalen Elementdeklaration erlaubt.

Beim Hinzufügen von Attributen wird die Anfrage zurückgewiesen, sofern die selektierten Knoten keine Elemente sind, da nur Elemente über Attribute verfügen können. Anfragen, die Attribute einfügen, werden gleichfalls zurückgewiesen, sofern schon ein Attribut mit einem solchen Namen, wie in der Anfrage angegeben, existiert.

5.2.3 insert_before-Operation

Mit Hilfe der *insert_before*-Operation lässt sich ein XML-Fragment im Dokument an bestimmter Stelle einfügen. Die gewünschte Position wird durch das `select`-Attribut und das `before`-Attribut der Anfrage spezifiziert. Das `content`-Attribut der Anfrage genügt dabei den Bedingungen, wie schon bei der *add*-Operation erklärt, und ist auch genauso aufgebaut.

Nun wird durch das `select`-Attribut eine Knotenmenge selektiert, deren Elemente als Referenzknoten dienen. Durch das `before`-Attribut wird dann ein Kindknoten der Referenzknoten näher bestimmt und vor diesen Kindknoten wird der Wert des `content`-Attributes eingefügt. Dabei kann die *insert_before*-Operation nur zum Einfügen von Elementen oder von textuellem Inhalt genutzt werden. Ist der Wert des `content`-Attributes kein XML-Fragment, dann wird der Wert als Textknoten vor dem spezifizierten Knoten eingefügt.

Beispiel 5.4 zeigt eine Übersicht über den Aufbau der *insert_before*-Anfrage. Eine konkrete Anfrage ist in Beispiel 5.5 gegeben und die Abbildung 5.2 zeigt deren Auswirkung im Dokumentbaum.

Beispiel 5.4 (Syntax der insert_before-Operation)

```
<insert_before select="..." before="..." content="..." />
```

Beispiel 5.5 (insert_before-Operation)

```
<insert_before
    select="/xs:schema/xs:complexType[@name='firma']/xs:sequence"
    before="xs:element[@name='Telefon']"
    content="&lt;xs:element
            name=&quot;Adresse&quot;
            type=&quot;adresse&quot;
            minOccurs=&quot;0&quot;/&gt;"
/>
```

Da mittels der *insert_before*-Operation eine Reihenfolge erhalten bleibt, ist es nicht notwendig, dass das Einfügen von Attributen unterstützt wird, da Attribute keiner Reihenfolge

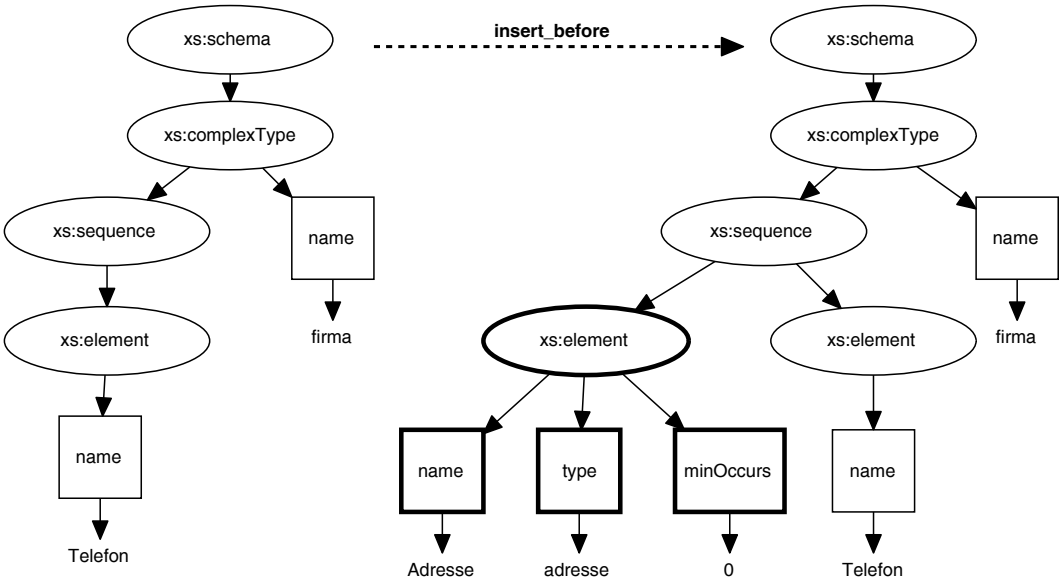

Abbildung 5.2: Beispiel insert_before-Operation für das Einfügen eines Elementes

bedürfen, sondern unsortiert vorliegen. Attribute werden nur über den Namen angesprochen, weswegen auch die Forderung nach eindeutigen Attributnamen für jedes Element existiert. Eine Reihenfolgeabhängigkeit bei Elementen liegt jedoch vor und hat auch konkrete Auswirkungen auf Instanzdokumente, wenn z.B. in einem XML-Schema ein komplexer Typ als Sequenz `sequence` von Elementen definiert wird. Dann muss in Instanzdokumenten jedes der in dieser Sequenz deklarierten Elemente genau einmal auftreten und dieses dann exakt in der Reihenfolge der Deklarationen im Schema.

Der mögliche Inhalt des `before`-Attributes wird so erweitert, dass ein Nullwert als Attributwert erlaubt ist. Der Nullwert entspricht dabei dem leeren Inhalt eines Attributes. Wenn nun dieser Nullwert auftritt, dann wird der Wert von `content` als letztes Kind der selektierten Knoten angehängt.

Bedingungen für die Ausführung

Die *insert_before*-Operation wird nicht ausgeführt, wenn im `select`-Attribut nur Attributknoten selektiert worden sind. Die Operation wird auch dann nicht ausgeführt, wenn der im `before`-Attribut spezifizierte Knoten entweder nicht eindeutig ist, kein Kindelement der Knoten aus dem `select`-Attribut oder selber ein Attribut ist.

5.2.4 insert_after-Operation

Die *insert_after*-Operation verhält sich wie die *insert_before*-Operation, nur dass sie statt eines `before`-Attributes über ein `after`-Attribut verfügt, mit dem angegeben wird, hinter welches Element man den Wert des `content`-Attributes einfügen möchte.

Für die Selektion und den einzufügenden Inhalt gelten die gleichen Bedingungen, wie schon bei der *insert_before*-Operation beschrieben. Wenn nun das `after`-Attribut keinen Wert, also

einen Nullwert, aufweist, dann wird der Inhalt des `content`-Attributes als erstes Kind der selektierten Knoten eingefügt.

Beispiel 5.6 zeigt die Syntax der *insert_after*-Operation und das Beispiel 5.7 und Abbildung 5.3 zeigen eine kurze Anwendung der *insert_after*-Operation und der Auswirkung auf die Baumstruktur eines Dokumentes.

Beispiel 5.6 (Syntax der insert_after-Operation)

```
<insert_after select="..." after="..." content="..." />
```

Beispiel 5.7 (insert_after-Operation)

```
<insert_after
    select="//xs:complexType[@name='firma']/xs:sequence"
    after="xs:element[@name='Telefon']"
    content="&lt;xs:element
            name=&quot;e-mail&quot;
            type=&quot;email&quot;/&gt;"
/>
```

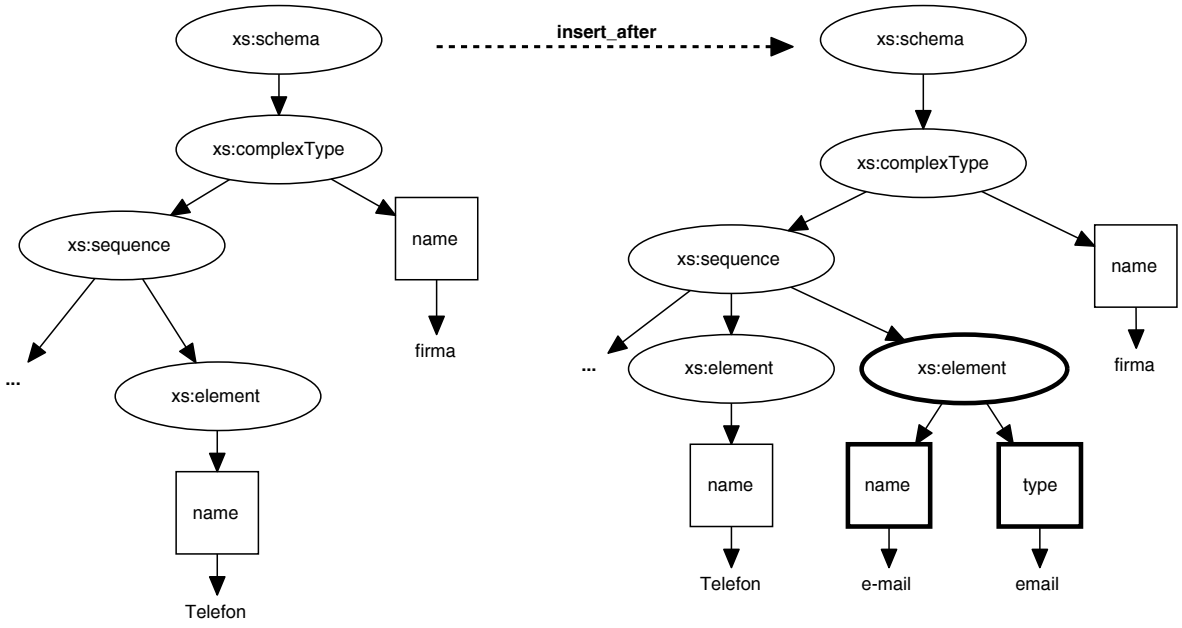

Abbildung 5.3: Beispiel insert_after-Operation für das Einfügen eines Elementes

Bedingungen für die Ausführung

Aufgrund der ähnlichen Funktionsweise der *insert_after*-Operation im Vergleich zu der *insert_before*-Operation gelten die selben möglichen Fehlerfälle und die gleichen Bedingungen für die Ausführung der Operation.

5.2.5 move-Operation

Im Großen und Ganzen verhält sich die *move*-Operation wie die *add*-Operation, nur dass sie nichts in den Baum einfügt, sondern einen schon im Baum bestehenden Teil an eine andere Position verschiebt. Deswegen muss zusätzlich spezifiziert werden, welcher Teil aus dem Dokumentbaum verschoben werden soll. Weil nichts eingefügt wird, besitzt die *move*-Operation kein `content`-Attribut

Somit sieht die Syntax der *move*-Operation wie folgt aus:

Beispiel 5.8 (Syntax der move-Operation)

```
<move select="..." to="..." />
```

Der Ausdruck im `select`-Attribut gibt an, welcher Knoten inklusive dem darunter befindlichen Teilbaum verschoben werden soll.

Das `to`-Attribut definiert nun, an welchen Knoten der selektierte angehängt werden soll. Dabei wird das Anhängen genauso umgesetzt, wie schon bei der *add*-Operation beschrieben. Wurde also ein Element oder Textknoten selektiert, dann wird es einfach an die Liste der Kindknoten des durch das `to`-Attribut spezifizierten Knotens angehängt. Die Reihenfolge bleibt dabei unbeachtet. Wurde hingegen ein Attributknoten selektiert, so wird das Attribut als neues Attribut des Zielknotens eingefügt. Anschließend wird der selektierte Knoten gelöscht, um die *move*-Operation abzuschließen.

Beispiel 5.9 (move-Operation)

```
<move
    select="//xs:complexType[@name='adresse']//
            xs:element[@name='Land']"
    to="//xs:complexType[@name='firma']/xs:sequence"
/>
```

Beispiel 5.9 und Abbildung 5.4 zeigen eine mögliche Anfrage, in dem das Element `xs:element` mit dem Attribut `name`, dessen Wert `Land` ist, aus dem komplexen Typen `adresse` in die Sequenz des komplexen Typen `firma` verschoben wird.

Bedingungen für die Ausführung

Die Operation *move* wird nicht ausgeführt, wenn durch das `select` und `to`-Attribut nicht jeweils genau ein Knoten selektiert wurde oder die Knotentypen zueinander nicht kompatibel sind.

Diese Kompatibilität ergibt sich aus den Definitionen zur Wohlgeformtheit von XML. Das bedeutet, dass nur Attribut-, Element- und Textknoten an Elementknoten angehängt werden können. Ein selektiertes Element kann nicht zu einem Attribut verschoben werden, da Attribute keine Kindknoten besitzen dürfen. Es kann ebenfalls kein Attribut zu einem Attribut verschoben werden, da Attribute selber keine Attribute besitzen dürfen.

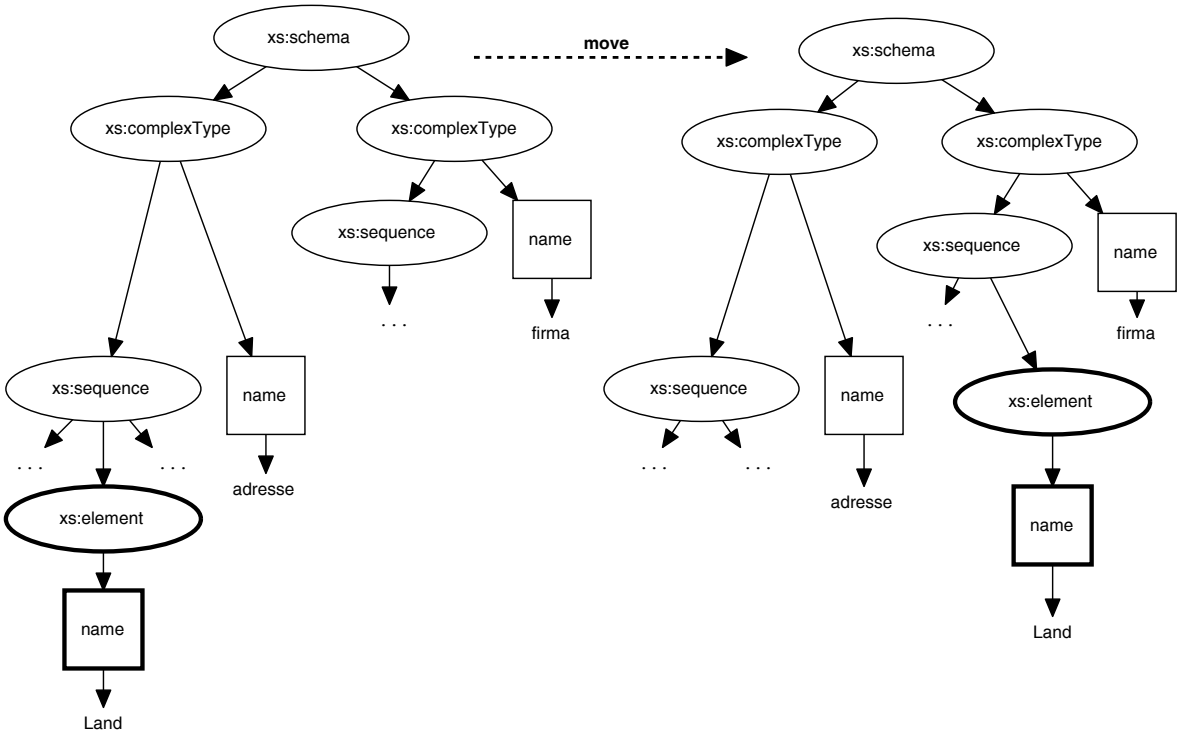

Abbildung 5.4: Beispiel move-Operation

5.2.6 move_before-Operation

Die *move_before*-Operation verhält sich ähnlich der *insert_before*-Operation, nur dass jetzt durch das `select`-Attribut spezifiziert wird, welcher Knoten inklusive darunter befindlichem Teilbaum verschoben werden soll. Das `before`-Attribut gibt an, vor welchen Knoten der zu verschiebende eingefügt werden soll.

Der spezifizierte Knoten wird an den Vater des Zielknotens angehängt. Die Reihenfolge wird so gewählt, dass er der unmittelbar linke Geschwisterknoten des Knotens aus dem `before`-Attribut ist. Beispiel 5.10 zeigt die Syntax der *move_before*-Operation.

Beispiel 5.10 (Syntax der move_before-Operation)

```
<move_before select="..." before="..." />
```

Dabei darf der Wert des `before`-Attributes nicht leer sein, da ansonsten nicht festgestellt werden kann, wohin der selektierte Knoten verschoben werden soll. Um einen Knoten als letztes Kind eines anderen Knotens einfügen kann, muss man die *move*- oder *move_after*-Operation einsetzen, welche im nächsten Abschnitt erläutert wird.

Beispiel 5.11 (move_before-Operation)

```
<move_before
    select="//xs:complexType[@name='adresse']/xs:sequence/
            xs:element[@name='Land']"
    before="//xs:complexType[@name='firma']/xs:sequence/
        xs:element[@name='Webseite']"
/>
```

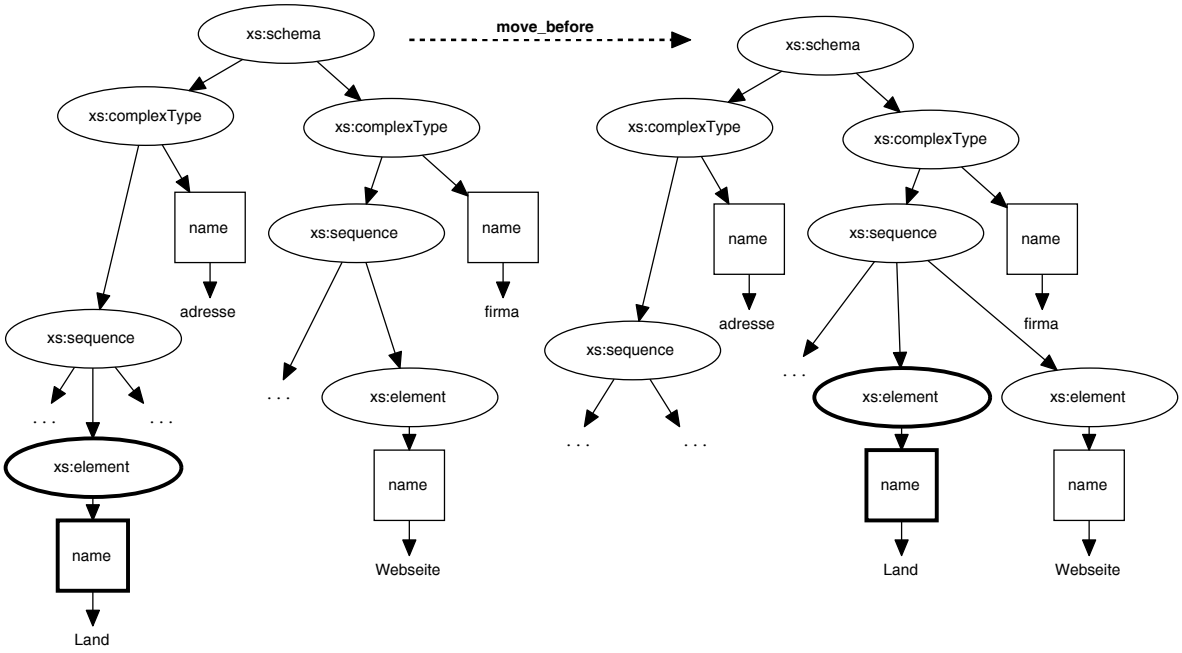

Abbildung 5.5: Beispiel move_before-Operation

Eine denkbare konkrete Anfrage zur *move_before*-Operation ist im Beispiel 5.11 gegeben und Abbildung 5.5 zeigt die Auswirkung in der Baumstruktur. In dem Beispiel wird eine Elementdeklaration, die des Elementes `Land`, aus der Sequenz des komplexen Typen `adresse` in den komplexen Typen `firma` verschoben und genauer gesagt, direkt vor die Elementdeklaration des Elementes `Webseite`.

Bedingungen für die Ausführung

Die *move_before*-Operation wird zurückgewiesen und nicht ausgeführt, wenn die selektierten Knoten aus dem `select`- und `before`-Attribut nicht eindeutig sind, also mehrere Knoten oder kein Knoten selektiert wurden.

Nicht ausgeführt wird sie ebenfalls bei Verstößen gegen die Kompatibilität der Knotentypen im Dokumentbaum, was schon bei der *move*-Operation erklärt wurde, wenn z.B. ein Attribut vor ein anderes Attribut verschoben werden soll oder wenn ein Element verschoben werden soll und der Zielknoten ein Attribut ist, dann wird die Operation zurückgewiesen.

5.2.7 move_after-Operation

Die *move_after*-Operation verläuft analog der *move_before*-Operation, nur dass sie ein `after`-Attribut aufweist statt eines `before`-Attributes, mit dessen Hilfe man spezifiziert, hinter welchen Knoten man einen aus dem Dokumentbaum selektierten Knoten verschieben möchte, wie die Syntax der *move_after*-Operation nachfolgend zeigt:

Beispiel 5.12 (Syntax der move_after-Operation)

```
<move_after select="..." after="..." />
```

Eine Anwendung der Operation ist im Beispiel 5.13 gegeben, die grafische Darstellung zu dem Beispiel ist in Abbildung 5.6 aufgeführt.

Beispiel 5.13 (move_after-Operation)

```
<move_after
    select="//xs:complexType[@name='adresse']/xs:sequence/
            xs:element[@name='Land']"
    after="//xs:complexType[@name='firma']/xs:sequence/
        xs:element[@name='Webseite']"
/>
```

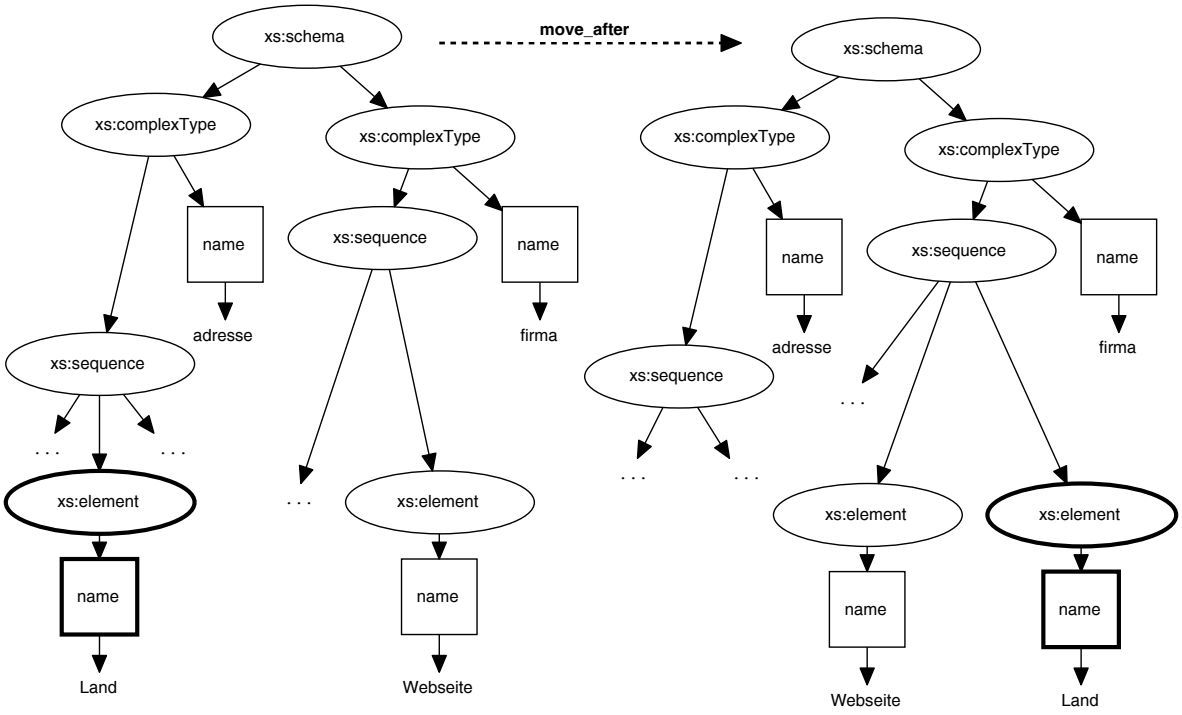

Abbildung 5.6: Beispiel move_after-Operation

Bedingungen für die Ausführung

Die Bedingungen, welche für die Operation eingehalten werden müssen, damit diese erfolgreich ausgeführt wird, sind dieselben wie schon für die *move_before*-Operation.

5.2.8 delete-Operation

Mit Hilfe der *delete*-Operation lassen sich Teile aus dem Dokumentbaum entfernen. Dabei wird durch das `select`-Attribut spezifiziert, welcher Knoten gelöscht werden soll. Alle Knoten, die dabei selektiert wurden und die unter ihnen befindlichen Teilbäume werden aus dem Dokument entfernt. Einschränkungen, was die Selektion betrifft, gibt es dabei nicht. Alle selektierten Knoten und ihre Kinder und Kindeskinder werden gelöscht, sofern die Validität des Dokumentes nicht verletzt wird.

Syntaktisch besitzt die *delete*-Operation nur ein `select`-Attribut, wie in Beispiel 5.14 gezeigt ist. Wie bei allen anderen Operationen ist der erlaubte Wert des `select`-Attributes ein XPath-Ausdruck.

Beispiel 5.15 und Abbildung 5.7 zeigen nun ein kurzes Beispiel, in dem aus dem komplexen Typ `firma` und seiner Sequenz die Elementdeklaration des Elementes `Webseite` gelöscht wird. Wenn nun im `select`-Attribut die Wurzel des Dokumentbaumes selektiert wird, dann wird diese gelöscht und damit alle darunter liegenden Knoten. Somit hätte diese Anweisung das leere Dokument als Ergebnis.

Beispiel 5.14 (Syntax der delete-Operation)

```
<delete select="..."/>
```

Beispiel 5.15 (delete-Operation)

```
<delete
    select="//xs:complexType[@name='firma']/xs:sequence/
            xs:element[@name='Webseite']"
/>
```

Bedingungen für die Ausführung

Einschränkungen gibt es bei der *delete*-Operation prinzipiell nicht.

Da über die Ausdrücke im `select`-Attribut immer Knoten selektiert werden und diese dann komplett gelöscht werden, kann so die Wohlgeformtheit der XML-Dokumente nicht verletzt werden. Es ist aber notwendig, dass die Validität bezüglich eines Schemas, also seine Gültigkeit, nach der Anwendung der Anfragen noch gegeben ist. Aus diesem Grund wird in einer konkreten Implementation[3] ein Unterschied gemacht, ob die Anfrage gegen ein Schema oder ein beliebiges Dokument gestellt wird. Der Anfrageprozessor muss deshalb darauf achten, dass aus einem Schema nur Teile, welche nicht weiter benötigt werden, gelöscht werden. Dies ist sinnvoll, da ansonsten die Regeln eines XML-Schemas verletzt würden. Bei der Ausführung der Operation wird vorher geprüft, wenn z.B. eine Typdefinition gelöscht werden soll, ob

[3] siehe Kapitel 6

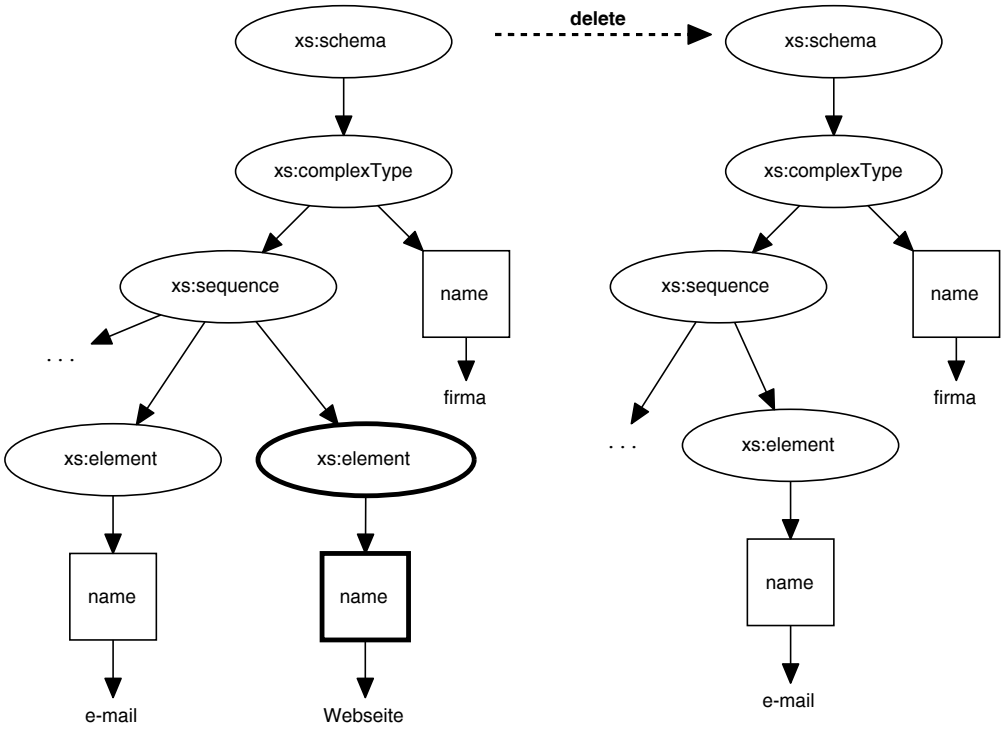

Abbildung 5.7: grafisches Beispiel: delete-Operation

dieser Typ benötigt wird, ob also eine Element- oder Attributdeklaration diesen Typen als Wert ihres `type`-Attributes besitzt. Ebenso muss geprüft werden, wenn ganze Element- oder Attributdeklarationen gelöscht werden sollen, ob andere Deklarationen auf diese referenzieren. Eine Bedingung ist jedoch, dass das XML-Schema in Normalform vorliegt, in der Referenzen auf Gruppen oder Deklarationen nicht erlaubt sind. Die Möglichkeiten für Schlüssel (`key` und `keyref`) und `IDREF`-Attribute müssen weiterhin beachtet werden.

5.2.9 replace-Operation

Wenn im Dokumentbaum ein selektierter Knoten ersetzt werden muss, dann kommt die *replace*-Operation zur Anwendung. Diese selektiert über ihr `select`-Attribut eine Menge an Knoten und ersetzt sie durch den Wert ihres `content`-Attributes. Dabei muss die Kompatibilität der Knotentypen wieder berücksichtigt werden, was im Abschnitt *Bedingungen für die Ausführung* erläutert wird.

Beispiel 5.16 (Syntax der replace-Operation)

```
<replace select="..." content="..." />
```

Für das `content`-Attribut gelten bezüglich der erlaubten Werte die gleichen Bedingungen wie bei der *add*-Operation, in Abschnitt 5.2.2 auf Seite 44 erklärt. Möchte man also ein Element ersetzen, muss der Inhalt ein gültiges XML-Fragment sein. Wenn man hingegen ein Attribut ersetzen will, dann muss der Inhalt des `content`-Attributes dem Muster `name=wert` folgen und

die selektierten Knoten vom Typ Attribut sein. Alle anderen Werte werden als Text interpretiert und werden als Textknoten die selektierten ersetzen. Es muss das Gleichheitszeichen in seiner Entität[4] auftreten, sofern man einen Textknoten mit dem Wert `foo=bar` als Ersetzung haben möchte, da der Wert des `content`-Attributes sonst als Attributersetzung interpretiert werden würde.

Das Beispiel 5.17 und die Abbildung 5.8 zeigen, wie ein Element durch einen Textknoten ersetzt wird. Da Textknoten in einem Schema in Normalform nicht erlaubt sind, wurde das Beispiel aus einem möglichen Instanzdokument des Schemas aus Anhang B entnommen.

Beispiel 5.17 (replace-Operation, Ersetzung durch einen Textknoten)

```
<replace
    select="/Kontakt/Vorname"
    content="ein neuer Textknoten anstelle eines alten Namens"
/>
```

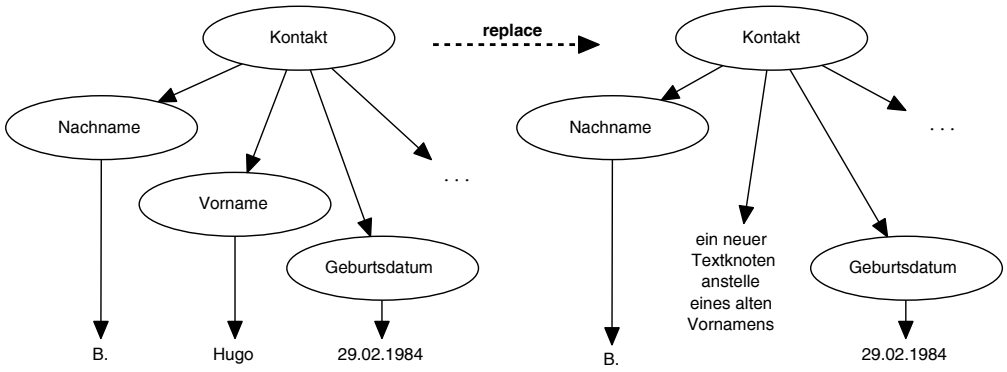

Abbildung 5.8: grafisches Beispiel: replace-Operation

Das Beispiel 5.18 zeigt hingegen, wie eine Elementdeklaration in einem Schema ersetzt wird. Dabei wird ein Element mit dem Namen `xs:element` durch ein anderes Element mit demselben Namen ersetzt, das aber über andere Attribute verfügt.

Beispiel 5.18 (replace-Operation, Ersetzung durch eine Elementdeklaration)

```
<replace
    select="//xs:complexType[@name='kontakt']/xs:sequence/
            xs:element[name='Nachname']"
    content="&lt;xs:element
            name=&quot;Name&quot;
            type=&quot;xs:string&quot;
            minOccurs=&quot;1&quot;
            nilable=&quot;false&quot;/&gt;"
/>
```

Aus diesem Beispiel wird auch ersichtlich, dass die *replace*-Operation bei leerem `content`-Attribut den gleichen Effekt wie die *delete*-Operation aufweist.

[4] siehe Tabelle 5.1 auf Seite 44

Bedingungen für die Ausführung

Einschränkung bei der Benutzung der Operation liegen dann vor, wenn die Typen der selektierten Knoten und die Typen der zu ersetzenden Knoten nicht kompatibel zueinander sind, so z.B. wenn ein Element- oder Textknoten durch ein Attribut ersetzt werden soll oder ein Attribut durch einen Element- oder einen Textknoten ausgetauscht werden soll. Dann wird die Anfrage zurückgewiesen und die Operation nicht ausgeführt. Ein Textknoten kann allerdings einen Elementknoten ersetzen und umgekehrt.

5.2.10 rename-Operation

Die *rename*-Operation dient dazu, Elemente oder Attribute umzubenennen. Über das `select`-Attribut werden die Knoten selektiert, die umbenannt werden sollen. Der neuen Name wird über das `name`-Attribut spezifiziert. Dabei sind Umbenennungen nur von Elementen und Attributen erlaubt, für jegliche andere Art von Knoten kann die Operation nicht ausgeführt werden, da diese über keinen Namen verfügen.

Die Syntax und eine Anwendung der *rename*-Operation sind in Beispiel 5.19 und 5.20 aufgeführt. Im Beispiel wird der Gruppentyp `xs:sequence` eines komplexen Typen mit Namen `adresse` in `xs:all` geändert wird, was einer Umbenennung des Kindknotens `xs:sequence` entspricht.

Beispiel 5.19 (Syntax der rename-Operation)

```
<rename select="..." name="..." />
```

Beispiel 5.20 (rename-Operation)

```
<rename
    select="//xs:complexType[@name='adresse']/xs:sequence"
    name="xs:all"
/>
```

Bedingungen für die Ausführung

Die *rename*-Operation wird zurückgewiesen, wenn der neue Name nicht den Konventionen von Attribut- und Elementnamen folgt. Er muss vom Typ `NMTOKEN` sein, wie er im Vorschlag des W3C zu XML und der DTD in [W3C04e] definiert wurde. Dieser Typ ist im Vorschlag zu XML-Schema [W3C04a] übernommen worden und beschreibt die einzuhaltenden Namenskonventionen.

Ein weiterer Grund für die Nichtausführung der *rename*-Operation ist gegeben, wenn durch den Wert des `select`-Attributes keine Element- oder Attributknoten selektiert worden sind, sondern Knoten wie Textknoten, die keinen Namen besitzen und somit nicht umbenannt werden können.

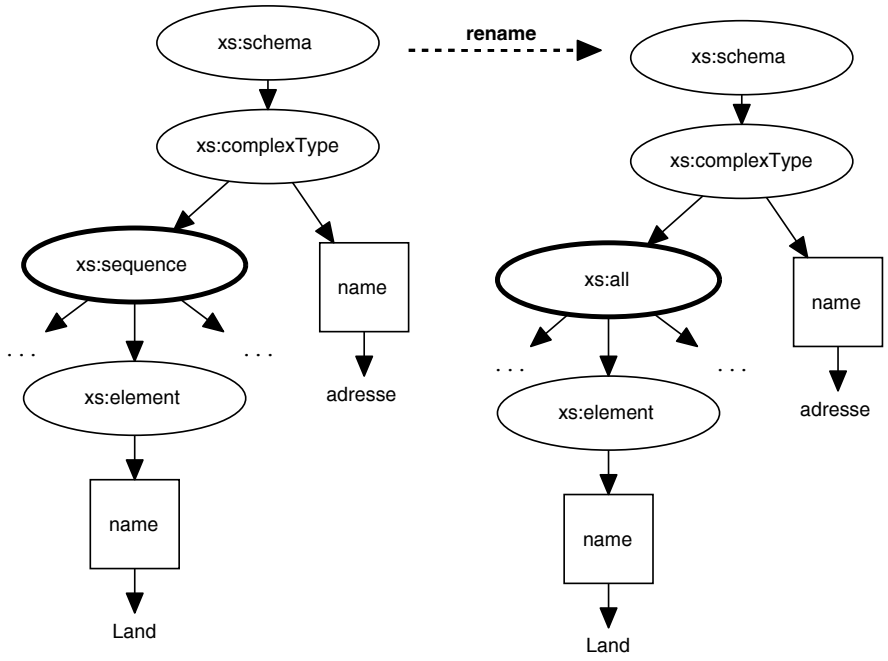

Abbildung 5.9: grafisches Beispiel: rename-Operation

5.2.11 change-Operation

Ähnlich der *rename*-Operation für die Namensänderung, dient die *change*-Operation zum Ändern des Wertes eines Elementes oder eines Attributes. Mit Hilfe des `select`-Attributes wird der Knoten selektiert, dessen Wert geändert werden soll.

Zusätzlich verfügt die *change*-Operation noch über ein `value`-Attribut, das als Attributwert den neuen Wert des selektierten Knotens besitzt. Der Wert des Attributes entweder der Wert eines primitiven Datentypen, wie z.B. eine ganze Zahl, eine reelle Zahl, ein Boole'scher Wert oder eine einfache Zeichenkette, wenn der Wert von einem Attribut geändert werden soll, da Attribute nur von einfachem Typ sind, die in XML-Schema durch `simpleType` beschrieben werden oder durch die Basisdatentypen repräsentiert sind.

Der Wert des `value`-Attributes kann aber auch die Zeichenkettenrepräsentation eines XML-Fragmentes sein, wenn unter anderem der Wert eines Elementes geändert werden soll. Dabei ist der Wert eines Elementes die Gesamtheit seiner Kinder und Kindeskinder. Attribute eines Elementes werden als Eigenschaften aufgefasst und gehören deswegen nicht zum Wert eines Elementes.

Beispiel 5.21 zeigt die Syntax der *change*-Operation. Die Beispiele 5.22 und 5.23 geben die Anwendung der *change*-Operation wieder. Im ersten Beispiel wird der Wert eines Elementes ersetzt, grafisch dargestellt in Abbildung 5.10, und im zweiten Beispiel wird der Wert eines Attributes neu gesetzt.

Beispiel 5.21 (Syntax der change-Operation)

```
<change select="..." value="..." />
```

Beispiel 5.22 (change-Operation zur Ersetzung eines Elementwertes)

```
<change
    select="//xs:simpleType[@name='telefon']/xs:restriction"
    value="&lt;xs:pattern value=&quot;+\d{2,2}\d{3,6}/ \d{4,8}&quot;/&gt;"
/>
```

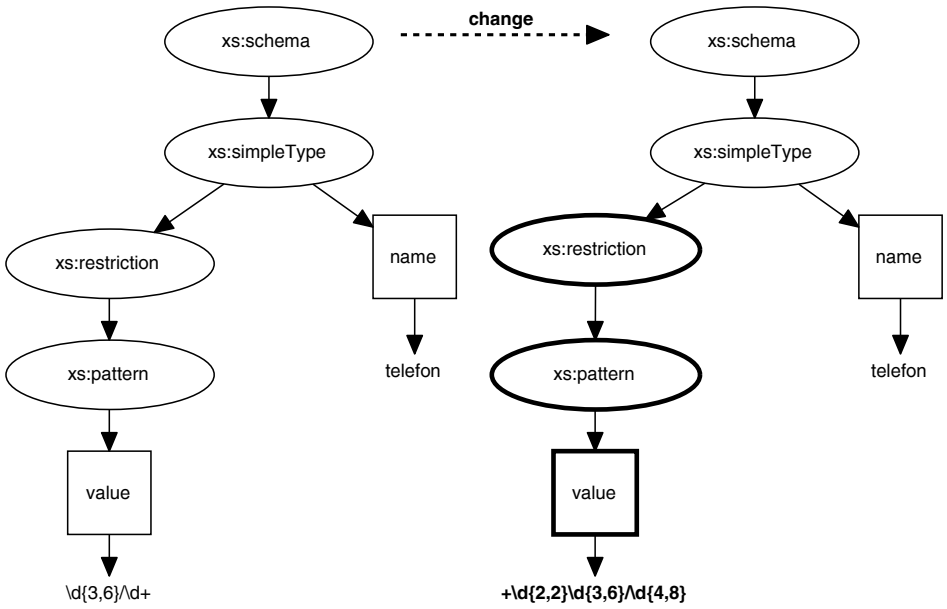

Abbildung 5.10: grafisches Beispiel: change-Operation

Beispiel 5.23 (change-Operation zur Änderung eines Attributwertes)

```
<change
    select="//xs:simpleType[@name='email']/xs:restriction/
            xs:pattern/attribute::value"
    value="[\w.-_]*@[\w.-_]*.[\w]{2,3}"
/>
```

Dabei arbeitet die Operation ähnlich der *replace*-Operation. Es wird jedoch nicht der gesamte Knoten ersetzt, sondern nur sein Wert ausgetauscht. Das bedeutet, dass der Name eines Attributes oder Elementes gleich bleibt. Beim Ändern des Wertes von einem Element bleiben außerdem alle seine Attribute unverändert. Hingegen werden bei der *replace*-Operation der Knoten mit all seinen möglichen Attributen und dem gesamten unter ihm befindlichen Teilbaum ersetzt wird.

Bedingungen für die Ausführung

Einschränkungen existieren für diese Operation nicht, da in XML jeder Datentyp und jeder Wert als Zeichenkette betrachtet wird. Tritt also der Fall auf, dass durch den Wert des `select`-Attributes Attribute selektiert worden sind und der Wert des `value`-Attributes ein gültiges

XML-Fragment darstellt, dann wird die Operation dennoch ausgeführt und der Inhalt wird als Zeichenkette interpretiert und als neuer Attributwert eingetragen. Wären hingegen Elemente selektiert worden, würde der Inhalt als XML-Fragment interpretiert werden und als neue Kindknoten der selektierten Elemente deren alte Kindknoten ersetzen.

Ist der Wert des `value`-Attributes nun eine Zahl oder eine einfache Zeichenkette, die keine Zeichenkettenrepräsentation eines XML-Fragmentes ist, dann wird der Wert entweder als Textknoten die Kinder und Kindeskinder des selektierten Knotens ersetzen oder als neuer Attributwert, sofern Attribute selektiert worden sind, eingesetzt.

Unabhängig vom Wert des `value`-Attributes wird dieser immer als Text interpretiert, sofern Textknoten im `select`-Attribut selektiert wurden, da der Inhalt der Textknoten deren Wert ist und somit durch den Wert des `value`-Attributes ersetzt. Ähnlich verhält es sich bei anderen Knotentypen, deren Inhalt auch ihren Wert darstellen wie z.B. bei Kommentarknoten.

5.2.12 Übersicht über die Operationen

Die Tabelle 5.2 zeigt in einer Übersicht alle Operationen noch einmal zusammengefasst. In der Spalte *selektierte Knoten* wird angegeben, wie die durch das `select`-Attribut selektierte Knotenmenge aussehen muss, ob nur ein Knoten in der Menge vorkommen darf oder ob mehrere erlaubt sind. Dabei steht eine 1, wenn nur ein Knoten erlaubt ist, ein n, wenn mehrere Knoten selektiert werden dürfen. Unter dem Begriff Zusatzattribut werden die `before`- und `after`-Attribute zusammengefasst und angegeben, wie die durch sie selektierte Knotenmenge aussehen muss, ob nur ein Knoten selektiert werden darf (1) oder mehrere (n) Knoten möglich sind.

Name	**Anzahl selektierte** Knoten `select`(`before`/`after`)		**Besonderheiten**
add	n		
insert_before	n	before: (1)	Attribute werden nicht verarbeitet (eingefügt)
insert_after	n	after: (1)	Attribute werden nicht verarbeitet (eingefügt)
move	1		
move_before	1	before: (1)	Attribute werden nicht verarbeitet (verschoben)
move_after	1	after: (1)	Attribute werden nicht verarbeitet (verschoben)
delete	n		
replace	n		
rename	n		
change	n		Attribute sind Eigenschaften von Elementen und gehören nicht zu derem Wert

Tabelle 5.2: Operationen in XSEL

In der letzten Spalte sind eventuelle Besonderheiten aufgeführt. So können mit den Operationen, die über ein `before`- oder `after`-Attribut verfügen, keine Attribute behandelt werden,

da diese Operationen die Reihenfolge beachten, welche bei Attributen nicht gegeben ist.

5.3 weitere Konzepte

Ein weiteres Konzept ist die Möglichkeit zur Gruppierung von Anfragen, wie sie in dem Entwurf zu XSEL vorliegt. Sinnvoll ist dies vor allem dann, wenn mehrere Anfragen in einer Datei gespeichert und zusammen zur Ausführung gebracht werden sollen, oder wenn die Überprüfung der Validität, welche nach jeder Anfrage stattfindet, soweit umgangen werden soll, dass nichtvalide Zwischenzustände eines Schemas zugelassen werden. Nach Abarbeitung aller Anfragen einer Gruppierung muss die Validität wieder gegeben sein. Im Entwurf sind zwei Arten von Gruppierungen vorgesehen:

- Queries, als lose Gruppierung einzelner Anfragen
- Transaktion, als Kapselung einzelner Anfragen

Die beiden Formen der Gruppierung sind im Schema in Anhang A bereits integriert.

5.3.1 Gruppierung (Queries)

Der Name *Queries* für die lose Gruppierung leitet sich aus dem Plural des englischen Wortes für Anfrage (engl: *Query*) ab. Dahinter verbirgt sich eine lose Zusammenfassung mehrerer Anfragen, damit diese unter anderem in einer Datei abgelegt werden können. Da die Anfragen in XML-Syntax vorliegen, müssen diese Dateien vom her Inhalt valide XML-Dokumente darstellen, welche beim Parsen gegen ein Schema, welches in Anhang A aufgeführt ist, validiert werden können. Aus diesem Grund werden die einzelnen Anfragen als Kindelemente eines Elementes `queries` aufgefasst.

Durch diese lose Gruppierung bleiben die Bedingungen unverändert, die die einzelnen Anfragen und die Operationen, welche sie repräsentieren, erfüllen müssen. Es wird also bei Ausführung einer einzelnen Anfrage aus der Gruppierung die Validität weiterhin überprüft und der Bearbeitungsvorgang notfalls abgebrochen, falls diese nicht gegeben ist. Der Nutzer hat auch keine Möglichkeit, die Effekte einer ganzen Gruppierung rückgängig zu machen. Somit würden bei Ausführung einer Gruppe mit fünf Anfragen und bei Zurücksetzen der letzten Anfrage die Effekte der ersten vier im Schema erhalten bleiben, sofern die Validität zu keinem Zeitpunkt verletzt wurde.

Beispiel 5.24 (Syntax der losen Gruppierung *Queries*)

```
<queries>
    <insert_before select="..." before="..." content="..."/>
    <rename select="..." name="..."/>
    ...
    <transaction>
        ...
    </transaction>
    ...
</queries>
```

5.3.2 Transaktionen

Ein weiteres Konzept sind *Transaktionen*. Sind sie jedoch nicht so definiert wie die Transaktionen im Datenbankbereich, welche den Eigenschaften des Konzepts ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability) genügen.

Transaktionen dienen im Rahmen des Entwurfes dem Zusammenfassen von mehreren Anfragen, ähnlich den Queries. Im Gegensatz zu den Queries wird bei den Transaktionen die Kontrolle der Validität nach Ausführung einer einzelnen Anfrage nicht durchgeführt. Diese Überprüfung wird erst am Ende der gesamten Transaktion gemacht. Ebenso ist die Zurücksetzung des Schemas bei Verletzung der Validität nicht für jede einzelne Anfrage durchzuführen, sondern das Zurücksetzen geschieht für die gesamte Transaktion. Dadurch hat der Nutzer die Möglichkeit, die Effekte einer gesamten Transaktion rückgängig zu machen wie bei einzelnen Operationen. Damit ist die Transaktion weniger eine Zusammenfassung mehrerer Anfragen als vielmehr eine Kapselung, da eine Transaktion als eine atomare Anfrage betrachtet wird, welche auch dieselben Eigenschaften wie eine atomare Operation besitzt. Betrachtet man eine Anfrage als einen Evolutionsschritt auf einem Schema, der das Schema von einem Zustand in einen neuen überführt, dann sind Queries eine Gruppierung mehrerer Evolutionsschritte, während eine Transaktion einen einzelnen Evolutionsschritt darstellt.

Im Entwurf sind Mechanismen für Sperren zur Vermeidung von Seiteneffekten bei gleichzeitiger Anwendung mehrerer Transaktionen nicht vorgesehen, da davon ausgegangen wird, dass nur eine Transaktion zur gleichen Zeit auf dem Dokument arbeitet und Sperren daher nicht notwendig sind.

Beispiel 5.25 (Syntax der Gruppierung *Transaktion*)

```
<transaction>
    <add select="..." content="..."/>
    <delete select="..."/>
    ...
</transaction>
```

Aus den Beispielen 5.24 und 5.25 erkennt man, dass eine lose Gruppierung alle Arten von Operationen und Transaktionen umfassen kann. Eine Transaktion hingegen darf nur atomare Operationen umfassen und keine anderen Transaktionen, da der Entwurf eine Schachtelung von Transaktionen nicht vorsieht. Einerseits ist dies nicht notwendig, da die Operationen aus einer inneren Transaktion in die äußere übernommen werden können, so dass man eine einzelne Transaktion erhält. Bei verschachtelten Transaktionen müsste man bei den inneren Transaktionen die Validitätskontrolle vollständig abschalten, da sie als Eingabe und Ausgabe nicht valide Zwischenergebnisse liefern können. Dadurch muss man wieder unterscheiden, ob es sich um eine innere oder äußere Transaktion handelt und nur wenn die gesamte Transaktion ausgeführt wurde, wird die Validität kontrolliert und notfalls alle Effekte der ausgeführten Transaktionen rückgängig gemacht.

5.4 Abbildung Schema-Dokument

In diesem Abschnitt wird gezeigt, wie eine eindeutige Abbildung vom Schema in mögliche Instanzdokumente aussehen kann. Eine solche Abbildung ist notwendig, da nach erfolgten Evolutionsschritten die Instanzdokumente angepasst werden müssen, wenn diese nicht mehr valide bezüglich des neuen Schemas sind.

Hilfsmittel für diese Abbildung sind gewisse Bedingungen, so genannte *Constraints*. Mit Hilfe dieser Constraints werden im Schema alle Attribut- und Elementdeklarationen und deren Typen beschrieben. Bei der Adaption der Daten dienen diese dazu, auf genaue Teilbäume in der Instanz zu zeigen, damit die Möglichkeiten der notwendigen Änderungen lokal begrenzt bleiben. Somit werden diese Constraints dazu benutzt, das Instanzdokument inkrementell zu validieren. Notwendig wird dieser Schritt, da nach der Evolution auf der Schemaebene die Dokumente beim Parsen nicht auf Validität geprüft werden können, da die Validität bezüglich des geänderten Schemas nicht garantiert werden kann.

5.4.1 Abbildung

Das Schema liegt in XML-Schema-Notation vor und in der Normalform XSDNF ([Tie03]). Die Normalform schreibt vor, dass alle Typdefinitionen global erfolgen müssen. Lokal oder anonym definierte Typen sind nicht zulässig. Dadurch muss jeder nutzerdefinierte Typ einen Namen besitzen, der für die Typen global eindeutig sein muss. Jedem deklarierten Element und Attribut muss ein Typ zugewiesen werden, welcher entweder ein in XML-Schema integrierter Basisdatentyp ist oder ein nutzerdefinierter Typ. Referenzen auf Deklarationen oder Gruppendefinitionen sind in der Normalform nicht erlaubt und Rekursionen müssen eliminiert worden sein. Dadurch darf kein Element ein Nachfolgerelement besitzen, welches den gleichen Namen und den gleichen Typ besitzt. Wäre dies der Fall, dann würde eine Rekursion vorliegen. Elemente werden darüber hinaus nicht global deklariert, mit Ausnahme des Wurzelelementes der Instanzen. Dadurch können Elemente und Attribute nur in den Typdefinitionen deklariert werden. Aufgrund dieser Voraussetzungen kann man nun eine eindeutige Zuordnung der Typen zu den Elementen und Attributen finden.

Mathematisch exakt betrachtet, liegt keine Abbildung vor, sondern eine Kongruenz aus dem Schema in die Dokumente, da eine Deklaration auf mehrere Knoten in der Instanz zeigen kann[5] und nicht alle Deklarationen eine Entsprechung in der Instanz haben, wenn die Elemente oder Attribute als optional deklariert wurden. Diese Kongruenz wird fortan als Abbildung bezeichnet.

Weil die Eindeutigkeit der Namen für Elemente und Attribute[6] nicht verlangt wird, können mehrere Deklarationen auf dieselben Instanzknoten verweisen. Um dies zu verhindern wird der Kontext einer Deklaration eingeführt, welcher neben dem Namen auch Typinformationen enthält, so dass diese bei der Abbildung mit betrachtet werden und so nicht mehrere Dekla-

[5] wenn Elemente mehrfach auftreten müssen oder wenn gleiche Elemente in verschiedenen Teilbäumen der Instanz auftreten

[6] es können mehrere Attribute im Schema mit dem gleichen Namen deklariert werden, sie dürfen lediglich nicht einem Element zugeordnet werden

rationen auf einen Instanzknoten verweisen können. Die Eigenschaften des Kontextes werden in einem Constraint mit aufgenommen.

5.4.2 Kontext

Der Kontext einer Deklaration umfasst die Eigenschaften der Deklaration an sich und er umfasst die Informationen bezüglich des zugewiesenen Typs. Element- und Attributdeklarationen werden verallgemeinert als Deklaration zusammengefasst, da nur Elemente und Attribute deklariert werden, Typen hingegen definiert werden. Mit Definitionen werden alle definierten Typen, unabhängig ob einfacher oder komplexer Typ, zusammengefasst. Attribute und Elemente werden zusammenfassend als Knoten bezeichnet.

Eigenschaften der Deklaration

Die Eigenschaften einer Deklaration umfassen zuerst den Namen des deklarierten Knotens. Dieser ist als Wert des Attributes `name` in der Deklaration festgelegt. Zusätzlich wird im Kontext vermerkt, mit welcher Häufigkeit die Knoten in den Instanzen auftreten, was als Wert der `minOccurs`- und `maxOccurs`-Attribute bei Elementdeklarationen vorliegt. Bei Attributdeklarationen entspricht der Wert des `use`-Attributes dieser Angabe. Diese Attribute werden als Quantor bezeichnet. Mögliche Werte des `use`-Attributes der Deklaration sind `optional`, das deklarierte Attribut kann in der Instanz auftreten, `required`, das Attribut muss auftreten, und `fixed`, das Attribut muss mit dem festgelegten Wert in der Instanz auftreten.

Neben dem Quantor existiert noch die Festlegung, ob Knoten Werte aufweisen müssen. Diese Festlegung ist als Wert des `nillable`-Attributes in der Deklaration angegeben. Die Namen der Nachbarelemente werden ebenfalls mit in den Kontext aufgenommen, sofern es sich um den Kontext einer Elementdeklaration handelt, da bei Elementen eine Reihenfolgeabhängigkeit bestehen kann, wenn diese in einer Sequenz deklariert wurden.

Die folgende Auflistung gibt eine kurze Zusammenfassung der Eigenschaften einer Deklaration wieder:

- Name des deklarierten Knotens
- Wert des Quantors des deklarierten Knotens
- Wert des `nillable`-Attributes der Deklaration
- Nachbarelemente des deklarierten Elementes (Knoten der XPath[7]-Achsen `preceding-sibling` und `following-sibling`)
- Name des zugewiesenen Typs

Typinformationen

Die Typinformationen umfassen alle Eigenschaften des verwendeten Typs. An erster Stelle steht der Name des Typs und die Art, ob der Typ ein Basisdatentyp, ein einfacher oder ein

[7] siehe Abschnitt 3.2 Seite 10

komplexer Typ ist. Diese Unterscheidung wird gemacht, da jeder Typ andere Informationen enthält. Ein Basisdatentyp besitzt nur einen Namen und einen Wertbereich. Dagegen wird ein einfacher Typ weiter unterschieden, ob er eine Vereinigung anderer einfacher Typen ist, ob er einer Liste entspricht oder ob er durch eine Aufzählung aller gültigen Werte definiert wird. Ein einfacher Typ kann zusätzlich eine Einschränkung sein, in XML-Schema `restriction`, was einer Ableitung in den objektorientierten Sprachen entspricht. Diese Einschränkung kann auf einem Basisdatentyp oder auf einem anderen definierten einfachen Typ geschehen und kann auf verschiedene Arten, mit Hilfe so genannter Facetten, realisiert werden. Einerseits kann der Wertebereich eingeschränkt werden, indem die obere und untere Grenze oder die Länge von erlaubten Zeichenketten angegeben wird oder über so genannte Pattern, welche regulären Ausdrücken entsprechen.

Ein komplexer Typ kann neben einer Einschränkung auch eine Erweiterung sein, in XML-Schema `extension`. Die Definition kann auch eine Elementgruppe beinhalten, oder eine lose Deklaration von Attributen. All diese Informationen über Gruppentyp, der Anzahl und den Namen der deklarierten Knoten und von welchem Typ er erweiternd oder einschränkend erbt, muss im Kontext mit aufgenommen werden.

Abbildung 5.11 zeigt eine Übersicht über die Informationen des Kontextes einer Deklaration.

5.4.3 Constraint

Ein Constraint stellt eine Bedingung dar, die ein Knoten in einem Instanzdokument, also ein Element oder Attribut, erfüllen muss, damit er valide bezüglich seiner Deklaration im Schema ist. Da ein Instanzknoten genau einem Constraint genügen muss, dienen die Constraints dazu, die inkrementelle Validierung zu realisieren, da sie wie ein Zeiger auf die Knoten im Dokument verweisen. Um diese Eindeutigkeit der Zuordnung Instanzknoten zu einem Constraint zu wahren, umfasst ein Constraint die Informationen des Kontextes einer Deklaration.

Dadurch dient er einerseits als Lokalisierungshilfe, um die Knoten in den Instanzen, die möglicherweise an die Schemaevolution angepasst werden müssen, zu finden und er stellt alle Eigenschaften bereit, welche zu überprüfen sind. Abbildung 5.11 zeigt in einer Übersicht, welche Eigenschaften im Kontext zusammengefasst sind. Neben diesen Informationen beinhaltet ein Constraint zusätzlich noch Wissen über die Art des deklarierten Knotens, ob eine Attribut- oder Elementdeklaration zugrunde liegt und ob dieser Knoten während der Evolution auf dem Schema geändert wurde oder nicht.

Die Übersicht über die Eigenschaften ist als Baum dargestellt, da verschiedene Informationen in einem Kontext zusammengefasst sind, je nachdem, um welche Art Typ es sich handelt, oder je nachdem, wie die Ableitung bei einfachen Typen realisiert wurde. Dadurch entspricht ein Ast im Baum, also ein Pfad von der Wurzel bis zu einem Blatt gerade den Informationen, die im Kontext gespeichert sind.

Abbildung 5.12 zeigt, wie die einzelnen Deklarationen im Schema auf die Knoten eines Instanzdokumentes zeigen. Auf der linken Seite der Abbildung sind drei Constraints, die jeweils eine Deklaration repräsentieren, als Kreis dargestellt. Diese Constraints zeigen auf Elementknoten eines XML-Dokumentes, in der Abbildung rechts dargestellt. Man erkennt, dass ein Constraint auf mehrere Knoten zeigen kann (Constraint 1), auf genau ein Element

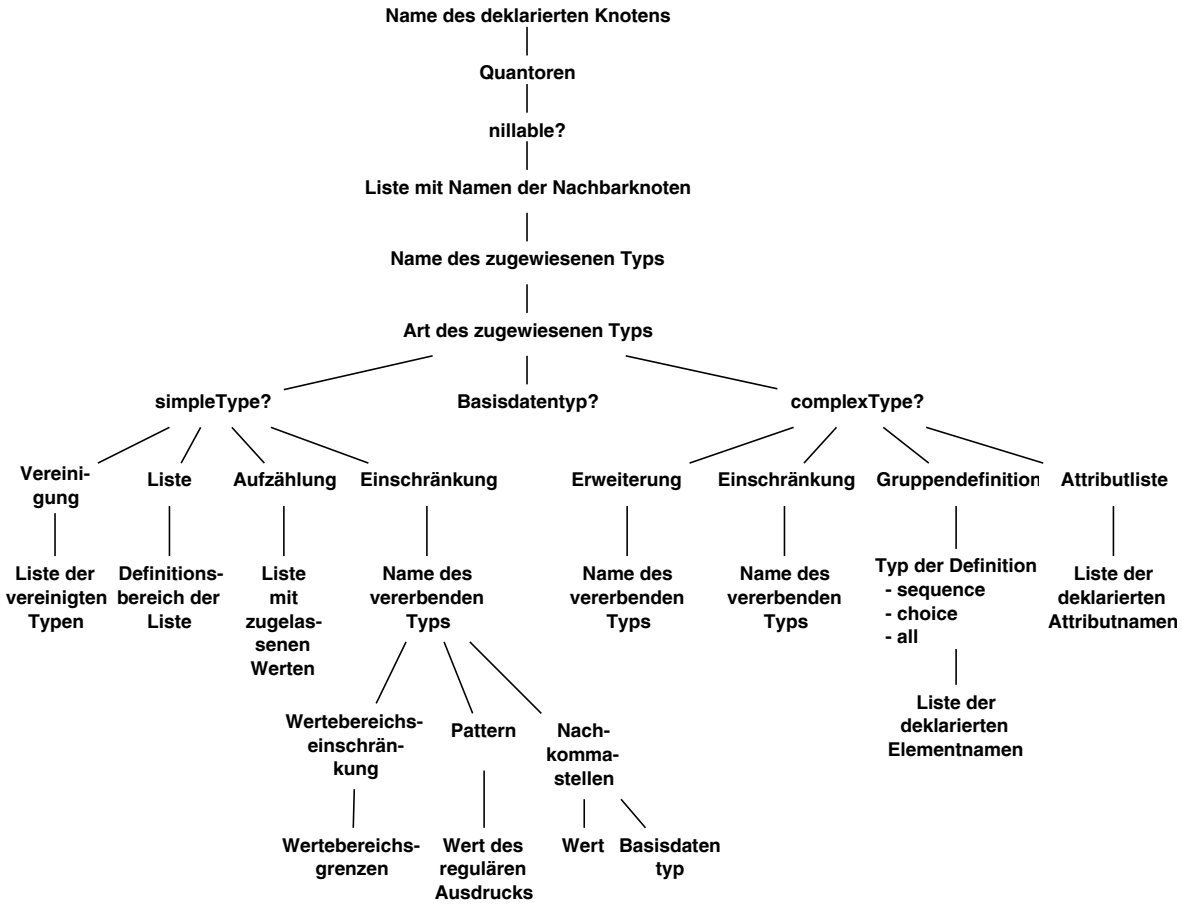

Abbildung 5.11: Eigenschaften des Kontextes einer Deklaration

oder der Constraint zeigt auf kein Element (Constraint 3).

5.5 Inkrementelle Validierung

Nachdem das Schema evolviert wurde, kann beim Laden eines Instanzdokumentes nicht auf Validität geprüft werden, da das Dokument bezüglich des alten Schemas valide war. Die Kontrolle der Validität wird deswegen über inkrementelle Validierung realisiert, die auf der Abbildung vom Schema in die Instanzdokumente und auf den konstruierten Constraints aufbaut. Die Grafik 5.12 zeigt schematisch eine solche mögliche Abbildung.

Im Prozess der inkrementellen Validierung werden zunächst alle Constraints überprüft, ob sie eine Deklaration repräsentieren, die während der Änderung des Schemas beeinflusst wurde. Das ist der Fall, wenn die Operationen auf dem Schema diese Deklarationen verändert haben, indem z.B. Quantoren modifiziert oder andere Typen zugewiesen wurden oder die zugewiesenen Typen variiert wurden. Eine andere Möglichkeit der Beeinflussung liegt vor, wenn die Position von Deklarationen innerhalb des Schemas geändert wurde, wie es bei den *move*-Operationen der Fall ist, oder wenn Deklarationen gelöscht wurden.

Anschließend werden die entsprechenden Knoten in den Instanzen gesucht und gegen die

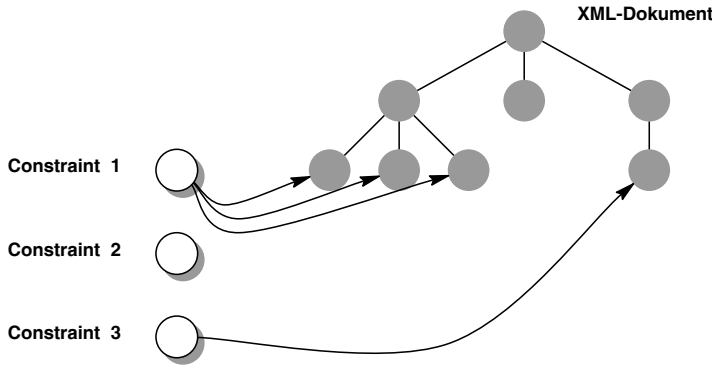

Abbildung 5.12: schematische Darstellung der Abbildung Schema-Dokument

Eigenschaften der Constraints verglichen. Wenn die Eigenschaften erfüllt sind, dann sind die gefundenen Instanzknoten valide bezüglich ihrer Deklaration im Schema. Ansonsten ist die Validität nicht gegeben und es müssen Änderungen in den Dokumenten vorgenommen werden.

Da durch diesen Vorgang nur die Teile der Instanzen kontrolliert werden, deren Deklarationen geändert wurden, spricht man von inkrementeller Validierung.

5.5.1 Anfragengenerierung zur Adaption der Daten

Ein weiteres Ziel der inkrementellen Validierung ist neben der Validitätskontrolle die Generierung von XSEL-Anfragen zur Adaption der Dokumente. Dieser Schritt geschieht während des Vergleiches mit den Eigenschaften des Constraints, da hier genau festgestellt werden kann, wodurch die Validität verletzt ist und so gezielt Anfragen generiert werden können, die die Validität wiederherstellen.

Der Prozess der inkrementellen Validierung und die einzelnen Vergleiche mit den Eigenschaften eines Constraints sind als Übersicht in 5.13 abgebildet. Diese Übersicht ist als Graph dargestellt, da je nachdem, wie die Vergleiche ausfallen, unterschiedlich fortgefahren wird. Dadurch stellt ein Weg im Graph vom Startknoten bis zu einem der Endknoten einen kompletten Durchlauf der Validitätskontrolle für einen Constraint und einen Knoten dar. Als Konsequenz dieses Vergleiches sind als Endknoten die XSEL-Operationen angegeben, die ausgeführt werden müssen, damit die Validität wieder hergestellt wird.

Es folgt nun eine genauere Beschreibung der einzelnen Vergleiche.

Anfänglich ist zu unterscheiden, ob zu einem Constraint entsprechende Knoten im Dokument gefunden wurden oder nicht. Wurden keine Knoten gefunden, wird der Constraint überprüft, ob durch die Quantoren festgelegt wurde, dass die Knoten nicht aufzutreten brauchen. Ist dies der Fall, muss keine Änderung vorgenommen werden. Müssen die durch den Constraint bestimmten Knoten aufgrund der Quantorenwerte in der Instanz vorkommen, sind aber dort nicht vorhanden, müssen deren Vorgängerknoten und Quantoren geprüft werden. Ist ein Vorgängerknoten optional und tritt in der Instanz nicht auf, können dessen Nachfolgerknoten ebenfalls nicht in der Instanz auftreten. Sofern die Vorgänger vorhanden sind und laut den Quantorenwerten der aktuell zu prüfende Knoten ebenfalls auftreten muss, dann müssen Standardwerte generiert werden und diese an passender Stelle für die fehlenden Kno-

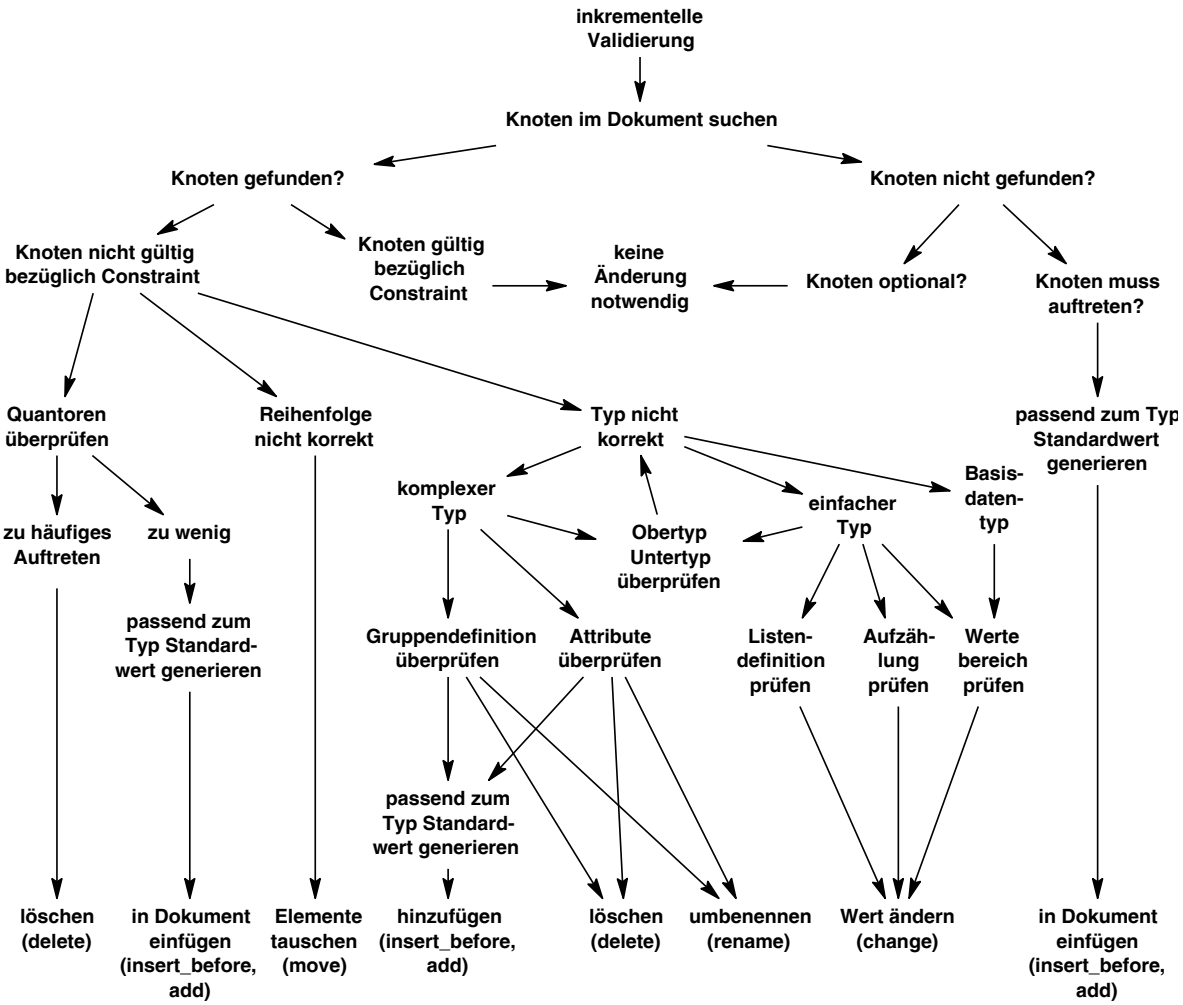

Abbildung 5.13: Übersicht über die inkrementelle Validierung

ten in den Baum eingefügt werden. Die Schwierigkeiten, die allgemein bei der Generierung von Standardwerten auftreten können, werden im Abschnitt 7.1.1 kurz erläutert.

Wurden in der Instanz Knoten gefunden, werden deren Eigenschaften mit denen des Constraints verglichen. Stimmen diese überein, so muss keine Anpassung stattfinden, ansonsten hängt es davon ab, welche Eigenschaft die Validität verletzt.

Differiert die Anzahl des Auftretens mit der durch die Quantoren vorgegebenen, müssen Knoten entweder gelöscht oder hinzugefügt werden. Bei Hinzufügen neuer Knoten werden für diese entsprechend ihres Typs Standardwerte generiert und dann in die Dokumentstruktur eingefügt.

Wenn die Reihenfolge von Elementen im Instanzdokument nicht mit der Reihenfolge der Deklarationen, sofern diese in einer Sequenz deklariert wurden, übereinstimmen, dann müssen entweder Elemente getauscht werden, oder, sofern Elementdeklarationen in der Sequenz gelöscht wurden, die betroffenen Instanzelemente aus dem Dokument entfernt werden. Bei gelöschten Deklarationen zählt die Gruppendefinition `all` als Sequenz, da dieser Gruppentyp

vorschreibt, dass alle deklarierten Elemente in einer Instanz vorkommen müssen, wenn auch in beliebiger Reihenfolge. Dadurch ist die Adaption der Daten gleich. Ein weiterer Fall tritt bei der Änderung der Namen der deklarierten Elemente auf, weil die Reihenfolge in den Instanzen nicht mehr gegeben ist. Dazu müssen in den Dokumenten die betroffenen Elemente umbenannt werden.

Zur Adaption der Daten kommt es auch, wenn sich der Typ der Instanzknoten zu dem in der Deklaration verwendeten unterscheidet. Dabei kommt es darauf an, von welcher Art der verwendete Typ ist. Bei einem Basisdatentyp kann nur die Definition des Wertebereiches verletzt sein. Bei nutzerdefinierten Typen werden die darin erfolgten Deklarationen kontrolliert, sofern es sich um einen komplexen Typ handelt. Darüber hinaus werden die Typen, von denen einschränkend oder erweiternd geerbt wurde, kontrolliert oder bei einfachen Typen wird die Definition des Wertbereiches kontrolliert.

Bei einfachen Typen findet dann eine Wertänderung statt, damit die Werte in den Dokumenten wieder dem Definitionsbereich entsprechen. Bei komplexen Typen, da diese weitere Deklarationen enthalten können, müssen notfalls Knoten in den Dokumenten gelöscht, hinzugefügt oder umbenannt werden. Wenn Knoten hinzugenommen werden müssen, wird für diese vorher ein Standardwert generiert.

Durch diese Vorgehensweise kann bei der Durchführung der inkrementellen Validierung gleich eine Anfragengenerierung stattfinden. Die generierten Anfragen werden dann auf das aktuelle validierte Dokument angewendet, so dass die Änderungen durch diese Anfragen die Validität bezüglich des zuvor evolvierten Schemas wieder herstellen.

5.6 Vergleich

Die Tabelle 3.2 (Abschnitt 3.6, Seite 26) wurde um eine Spalte für XSEL erweitert. Man

Kriterium	**XPath**	**DOM**	**XQuery**	**XUpdate**	**XSEL**
Deskriptivität	+	-	+	+	+
Adäquatheit	+	+	+	-	+
Optimierbarkeit	+	-	+	+	-
Orthogonalität	-	-	+	+	-
Abgeschlossenheit	+	+	+	+	-
Vollständigkeit	+	+	+	+	-
Datenextraktion	+	+	+	+	-
Datenmanipulation	-	+	-	+	+
Operationen auf dem Schema	-	-	-	-	+/-
Transaktionen	-	-	-	-	+/-

Tabelle 5.3: Vergleich der Ansätze zur XML-Verarbeitung und XSEL

kann aus der Tabelle erkennen, dass XSEL der Datenmanipulation und der Änderung von Schemata dient. Die Anforderungen, die speziell Anfragesprachen erfüllen sollen, sind nicht umgesetzt, da XSEL keine Anfragesprache, sondern eine Änderungssprache darstellt.

XSEL dient speziell zur Evolution des Schemas. Es wurden im Sprachumfang aber keine speziellen Operationen für das Schema wie Typkonstruktoren definiert, weil die Änderungssprache unabhängig davon, ob Schema oder Dokument, eingesetzt werden soll. Ebenso sind Transaktionen im Konzept inbegriffen, es wurden aber nicht alle Ansätze, wie sie z.B. bei Transaktionen im Datenbankbereich vorhanden sind, umgesetzt.

Kapitel 6

Realisierung eines Anfrageprozessors für XSEL

Im Rahmen der Diplomarbeit wurde die Implementation eines Anfrageprozessors vorgenommen, um die Umsetzung der Änderungssprache XSEL zu realisieren.

In diesem Kapitel wird eine Übersicht über den Aufbau und über die Funktionsweise gegeben. Anschließend wird eine Beschreibung zur Auswertung einer Anfrage aufgeführt und abschließend an einem Beispiel ein möglicher konkreter Anwendungsfall gezeigt.

6.1 Übersicht

Das Framework zu XSEL wurde in Java 1.4.2 implementiert. Als Parser für die XML-Dokumente und XML-Schemata wurde Xerces[1] verwendet. Dadurch werden die Dokumente, Schemata und die Anfragen, selber in XML-Syntax, auf ihre DOM-Repräsentation[2] abgebildet. Die Umsetzung der einzelnen Operationen erfolgt durch die Abbildung auf die entsprechenden DOM-Operationen zur Manipulation der Baumstruktur.

Nach dem Parsen der Schemata werden diese aufbereitet, indem für die Deklarationen Constraints angelegt werden, welche die Deklarationsinformationen und Typinformationen bereithalten, die für die inkrementelle Validierung benötigt werden. Anfragen werden über eine spezielle Klasse geladen und können danach auf das Schema angewendet werden. Anschließend werden die XML-Dokumente geladen und inkrementell validiert, wodurch auch eine Liste an XSEL-Anfragen generiert wird, die die Datenadaption durchführen.

6.2 Architektur

Der Aufbau des Systems umfasst verschiedene Komponenten, die das Parsen der Dokumente durchführen, die Validierung und Adaption umsetzen bis hin zu einer Ausgabe oder Visualisierung. In der Abbildung 6.1 ist die Architektur als Übersicht dargestellt.

Das **Parsermodul** realisiert das Laden oder Parsen der Dokumente und die Generierung der Dokumentobjekte, die anschließend durch andere Teile des Systems weiter verarbeitet

[1] http://xml.apache.org/xerces2-j
[2] siehe 3.3

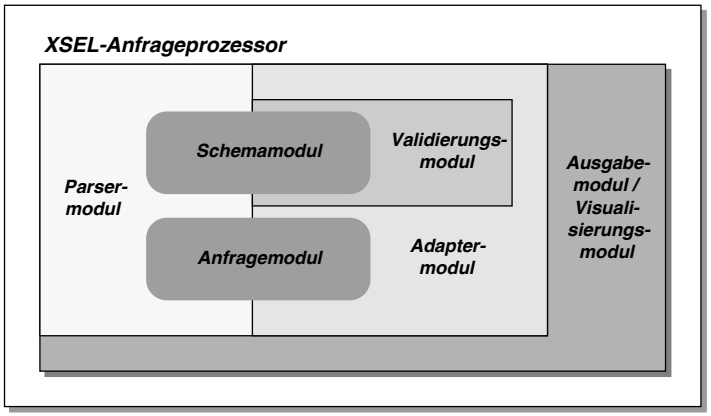

Abbildung 6.1: Framework zu XSEL

werden. XML-Schemata und die Anfragen beziehungsweise Anfragedateien stellen zwar valide XML-Dokumente dar, für deren Verarbeitung sind jedoch zwei selbstständige Module vorhanden, da sie gegenüber "normalen" XML-Dokumenten spezielle Eigenschaften besitzen.

Im **Schemamodul** wird neben dem Laden der Schemata eine Aufbereitung durchgeführt. In dieser Aufbereitung werden zu den einzelnen Deklarationen im Schema Constraint-Objekte angelegt und zu den Typdefinitionen werden Typinformations-Objekte generiert. Mit deren Hilfe wird im Validierungsmodul ein geladenes XML-Dokument auf Validität kontrolliert.

Das **Validierungsmodul** übernimmt den Vorgang der inkrementellen Validierung, wie er in Abschnitt 5.5 beschrieben wurde. Während dieses Prozesses werden die Knoten und deren Positionen im Dokument herausgefunden, die eventuell abgeändert werden müssen. Es wird dabei bestimmt, welche Eigenschaften gegen die Validität verstoßen. Mit Hilfe dieses Wissens werden dann die Anfragen zur Dokumentadaption generiert.

Das **Anfragemodul** umfasst das Laden der Anfragen. Es werden Anfragen-Objekte generiert, die gezielt auf ein Dokument angewendet werden können.

Kernstück des Frameworks ist neben dem Validierungsmodul das **Adaptermodul**. Darin werden nach erfolgter Validierung die Anfragen generiert, die durch das Anfragemodul auf den nachzuziehenden Dokumenten angewendet werden. Diese Anfragen werden mit Hilfe des Wissens über die durchgeführten Änderungen im Schema und den Positionen der zu ändernden Knoten im Dokument und deren Eigenschaften generiert. Diese Anfragen werden dann neben dem zu ändernden Dokument an das Anfragemodul übergeben, welches durch die Ausführung der Anfragen die Daten so adaptiert.

Für alle Ausgaben, seien es Debug-Ausgaben oder eine mögliche Visualisierung, ist das **Ausgabe-** oder **Visualisierungsmodul** zuständig. Auf das Modul haben alle anderen Module Zugriff, damit zu jedem Zeitpunkt der Verarbeitung eine Ausgabe möglich ist.

Im Abschnitt 6.3 ist mit der Abbildung 6.2 eine erweiterte Darstellung der Architekturübersicht aus Abbildung 6.1 gegeben. Die Abbildung zeigt die einzelnen Abläufe während der Verarbeitung einer Anfrage.

Aufgrund der Schwierigkeiten, die in Abschnitt 7.2 erläutert werden, ist im Framework kein Modul für die XQuery-Verarbeitung, die auch eine Anpassung dieser Anfragen im Zuge der

Evolution einschließt, vorgesehen. Im Informationssystem abgelegte XQuery-Anfragen werden lediglich durch das Parsermodul geparst und kontrolliert, ob sie auf Dokumente zugreifen, die sich während der Evolutionsumsetzung geändert haben. Ist dies der Fall, wird eine Meldung an den Nutzer generiert, dass die Anfragen möglicherweise angepasst werden müssen.

6.3 Ablauf eines Evolutionsschrittes

Der Begriff Evolutionsschritt wurde soweit konkretisiert, dass ein Evolutionsschritt durch eine Anfrage oder eine Transaktion repräsentiert wird.

6.3.1 Anfragenauswertung

Zur Umsetzung des Evolutionsschrittes wird zunächst die Anfrage geladen. Diese kann als Zeichenkette oder in einer Datei vorliegen. Das Laden geschieht über die `QueryParser`-Klasse. Diese erzeugt aus dem geladenen XML-Fragement, das eine Anfrage repräsentiert, ein `Query`-Objekt, dem anschließend ein Dokument übergeben wird. Durch Aufruf der `exec()`-Methode des `Query`-Objekts wird die Anfrage auf das übergebene Dokument angewendet und verändert dieses. Durch die Änderung des Dokumentes geschieht intern eine Neuberechnung von möglichen Indexstrukturen. Bei normalen XML-Dokumenten sind dies gewisse Indexknoten, die zu den einzelnen Dokumentknoten deren Eigenschaften und Position bezüglich der Baumhierarchie speichern. Ist das übergebene Dokument ein Schema, werden zusätzlich die Constraints im Schema aktualisiert und XPath-Ausdrücke generiert. Diese XPath-Ausdrücke, auf die Instanzen angewendet, selektieren gerade jene Knoten, die durch die hinter dem Constraint verborgene Deklaration definiert werden.

6.3.2 Constraintberechnung

Bei der Berechnung der Constraints werden zunächst die Deklarationen im Schema gesucht. Der Name des deklarierten Elementes oder Attributes bildet gleichzeitig den Namen des neuen Constraints. Danach wird untersucht, wie oft der deklarierte Knoten in Instanzdokumenten auftreten kann, was bei Elementdeklarationen durch die Attribute `minOccurs` und `maxOccurs` geschieht, bei Attributdeklarationen durch das `use`-Attribut. Anschließend wird der verwendete Typ überprüft. Hierfür wird ein Objekt vom Typ `TypeInfo` angelegt, sofern es sich um einen nutzerdefinierten Typ handelt. Wird ein Basisdatentyp verwendet, wird lediglich der Name des Typs im Constraint gespeichert. Die Klasse `TypeInfo` sammelt alle Informationen des in der Deklaration verwendeten Typs. Dazu zählen dessen Name, ob er komplex oder einfach ist, wie oft er in Deklarationen benutzt wird, ob er eine Einschränkung oder eine Erweiterung ist, wie diese zustande kommt, ob er eine Listendefinition beinhaltet, oder eine Aufzählung der erlaubten Werte. In der aktuellen Implementation der `TypeInfo`-Klasse werden jedoch nicht alle der in XML-Schema möglichen Typdefinition berücksichtigt. Die Listenbildung bei einfachen Typen und die Definition eines einfachen Typs durch Vereinigung zweier anderer einfacher Typen werden nicht unterstützt. Darüber hinaus ist die Behandlung der Vererbung bei komplexen Typen nicht implementiert.

Das `TypeInfo`-Objekt wird nun im `XSDocument`-Objekt, das ein Schema repräsentiert, gespeichert und eine Objektreferenz an das `Constraint`-Objekt übergeben.

Der XPath-Ausdruck zur Selektion in den Instanzen wird über den Aufstieg in der Hierarchie des Schemas generiert. Der Ausgangspunkt ist die Deklaration des Constraints, für den der XPath-Ausdruck konstruiert werden soll. Da das Schema in Normalform vorliegt, ist die Ausgangsdeklaration entweder die des Instanzwurzelelementes oder in einer Typdefinition gekapselt. Über den Namen des Typs, der eindeutig ist, gelangt man zu dem Element, dessen Deklaration den Typ verwendet. Diese Deklaration ist entweder die des Wurzelelementes oder ebenfalls in einer weiteren Typdefinition gekapselt. Dann wird die Deklaration gesucht, die diesen Typ verwendet usw. Da in der Normalform Referenzen und Rekursionen nicht erlaubt sind, endet dieser Prozess zwangsläufig bei der Deklaration des Instanzwurzelelementes. Auf diese Weise wird ein absoluter XPath-Ausdruck generiert, wobei der letzte Teil der Name des Ausgangsconstraints ist.

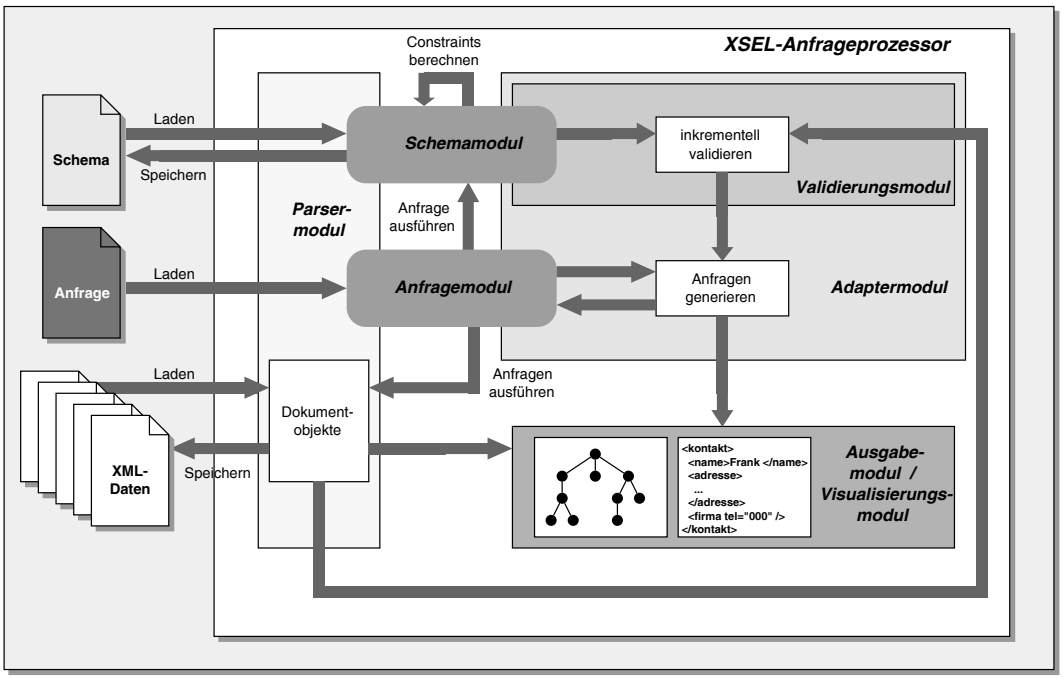

Abbildung 6.2: Funktionsablauf im XSEL-Framework

6.3.3 Dokumentadaption

Wenn während der Änderung des Schemas keine Fehler aufgetreten sind, erfolgt die Adaption der Daten. Fehler können dabei ungültige Anfragen sein oder das Schema ist nach der Änderung nicht mehr valide bzw. die Normalform ist nicht mehr gegeben. Anfragen sind dann ungültig, wenn die an sie gestellten Bedingungen nicht gelten, wie z.B. die Selektionsbedingungen und die Typkompatibilitäten der Knoten. Eine Beschreibung dieser Eigenschaften und Bedingungen ist in Kapitel 5.2 zu jeder Operation aufgeführt. Tritt hingegen ein Fehler auf, dann werden alle durchgeführten Änderungen zurückgesetzt und es wird eine Fehlermeldung

erzeugt.

Zur Umsetzung der Datenadaption werden das `Query`-Objekt und das `XSDocument`-Objekt an die `Adapter`-Klasse übergeben. Die Klasse hat Kenntnis, welche im System gespeicherten Dokumente durch das übergebene Schema definiert werden. Anschließend übernimmt die `Adapter`-Klasse das Laden der Dokumente und deren inkrementelle Validierung mit Hilfe der `Validator`-Klasse. Der Prozess der inkrementellen Validierung folgt dabei dem in Abbildung 5.13 auf Seite 67 gezeigten Graphen. Die `Validator`-Klasse fordert vom Schema-Objekt eine Liste der neuberechneten Constraints an. In dieser Liste sind alle Constraints aufgeführt, an deren Deklaration während der Evolution etwas geändert wurde. Solche Änderungen sind entweder Änderungen an der Deklaration selber, d.h. es wurden die Werte der Deklarationsattribute wie `name` oder `minOccurs` geändert, es sind neue Attribute hinzugekommen oder entfernt worden oder es wurde der verwendete Typ geändert.

Die `Validator`-Klasse selektiert nun die betreffenden Instanzknoten mit Hilfe des im Constraint gespeicherten XPath-Ausdrucks. Tritt während des folgenden Validierungsprozesses der selektierten Knoten ein Fehler auf, ist also ein Teil des Dokumentes nicht valide, werden neue Anfragen durch die `Adapter`-Klasse generiert. Sobald ein Fehler durch die `Validator`-Klasse gemeldet wird, befindet man sich in einem der Endzustände des Graphen aus Abbildung 5.13. Dadurch ist die Eigenschaft des Instanzknotens, welche nicht valide ist, genau spezifiziert und daraus ist ersichtlich, welche Operation zur Adaption genutzt werden muss. Nach Ablauf der Validierung erhält man so eine Liste von XSEL-Anfragen, die auf dem gerade validierten Dokument angewendet werden.

6.3.4 XQuery-Adaption

Bei der Anpassung der XQuery-Anfragen werden die im System abgelegten XQuery-Anfragen zunächst geladen und anschließend nach XPath-Ausdrücken durchsucht. Diese dienen zur Selektion von Knoten in den XML-Dokumenten. Wenn eine Anfrage Knoten eines geänderten Dokumentes verarbeitet, wird eine Meldung für den Nutzer generiert. In dieser Meldung wird lediglich darauf verwiesen, welche Anfrage geändert werden muss. Eine Anpassung muss dann vom Nutzer manuell durchgeführt werden. Dies ist dadurch begründet, dass neben syntaktischen Fehlern auch semantische auftreten können, die auf mehreren verschiedenen Wegen eliminiert werden können, was der Anfrageprozessor nicht automatisch entscheiden kann. Die genauen Schwierigkeiten, die auftreten können, sind in Abschnitt 7.2 aufgeführt.

6.4 Beispiel

In diesem Abschnitt wird die Durchführung eines Evolutionsschrittes an einem Beispiel erläutert. Das dem Beispiel zugrunde liegende Schema ist in Anhang B vollständig aufgeführt, da wegen der Übersichtlichkeit hier nur Ausschnitte betrachtet werden. Es wurde die Festlegung getroffen, dass die Namen von deklarierten Knoten prinzipiell groß und die Namen von Typen klein geschrieben werden.

Der Grund für die notwendige Evolution des Schemas im Beispiel ist ein ungenügender Entwurf. Das Schema weist einen “Mangel” auf. Zu einer Person ist zwar die Firma, bei

der er arbeitet, gespeichert, allerdings sieht das Schema keinen Namen für die Firma vor. Somit ist bei der Verarbeitung der Daten nicht ersichtlich, wo die Person arbeitet. Spätestens in Gebieten, wo viele Firmen in einer Straße und sogar unter der gleichen Hausnummer anzutreffen sind, wird sich dieser Mangel bemerkbar machen.

Es soll dem komplexen Typ `firma` deswegen ein Element `Name` hinzugefügt werden. Der Typ des neuen Elementes ist der Basisdatentyp `string`. Die Anfrage, die dies umsetzt, ist in Beispiel 6.1 aufgeführt. Weil der komplexe Typ eine Sequenz enthält, wurde als Operation die *insert_before*-Operation gewählt, die das neue Element an erster Stelle einfügt.

Beispiel 6.1 (Einfügen eines Elementes Name)

```
<insert_before
    select="/xs:schema/xs:complexType[@name='firma']/xs:sequence"
    before="xs:element[@name='Adresse']"
    content="&lt;xs:element
            name=&quot;Name&quot;
            type=&quot;xs:string&quot;/&gt;"
/>
```

Im Beispiel werden die Sequenz-Knoten aller komplexen Typen, die den Namen `firma` haben, selektiert. Durch die Angabe des Namens des komplexen Typs und weil das Schema in Normalform vorliegt, ist diese Selektion sogar eindeutig. Weiterhin wird spezifiziert, an welche Stelle genau das neue Element eingefügt werden soll, nämlich vor das Element mit dem Namen `Adresse`. Da der selektierte Knoten über ein solches Kindelement verfügt, welches den Namen `xs:element` hat und dessen Namensattribut den Wert `Adresse` besitzt, sind die Bedingungen für die Gültigkeit der Anfrage gegeben.

Nun soll der Wert des `content`-Attributes eingefügt werden. Dessen Wert beginnt und endet mit den Entitäten `<` und `>`, weshalb der Anfragenparser `QueryParser` versucht, ihn als XML-Fragment zu interpretieren und baut aus der Zeichenkette einen Baum bestehend aus einem Elementknoten und zwei Attributknoten auf (Abb. 6.3).

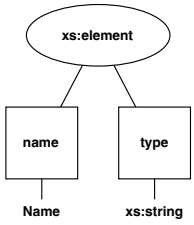

Abbildung 6.3: Beispiel XML-Fragement

Die Abbildung 6.4 zeigt, wie sich das Schema durch die Ausführung der Anfrage ändert. Es wird in der Abbildung lediglich ein Ausschnitt des betreffenden komplexen Typs gezeigt. Die einzelnen Knoten und Knotenbeziehungen des eingefügten Fragmentes sind auf der rechten Seite der Abbildung stärker eingezeichnet.

Nachdem das Schema geändert wurde, werden nun die Dokumente angepasst. Eine mögliche Instanz ist in Beispiel 6.2 gezeigt. Dieses Dokument wird nun geladen und inkrementell validiert.

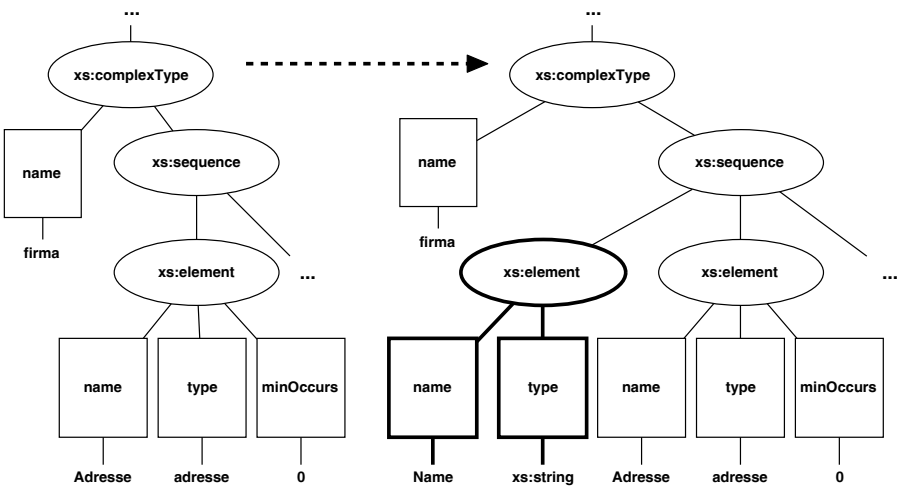

Abbildung 6.4: geändertes Schema

Beispiel 6.2 (Instanzdokument)

```
<?xml version="1.0" encoding="UTF-8"?>
<Kontakt  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="kontakt.xsd">
    <Nachname>B.</Nachname>
    <Vorname>Hugo</Vorname>
    <Geburtsdatum>06.03.1960</Geburtsdatum>
    <Adresse>
        <Strasse>Strasse 13a</Strasse>
        <PLZ>12345</PLZ>
        <Stadt KFZ="HRO" Bundesland="MV">Rostock</Stadt>
        <Land>Deutschland</Land>
    </Adresse>
    <Firma>
        <Telefon>012/3456789</Telefon>
        <e-mail>www@w3c</e-mail>
    </Firma>
    <Handy>0170/0000</Handy>
    <Telefon>0000/111222333</Telefon>
    <e-mail>w22@w22</e-mail>
</Kontakt>
```

Während der Neuberechnung der Constraints nach der Änderung im Schema ist nur der Constraint *Firma* geändert worden, da die Definition des Typs `firma` geändert wurde und der Typ nur in der Deklaration des Elementes `Firma` verwendet wird. Andere Constraints wurden nicht neuberechnet. Somit fordert die `Validator`-Klasse des Frameworks eine Liste mit nur einem Constraint an. Der XPath-Ausdruck, der im Constraint gespeichert ist, sieht wie folgt aus:

```
/Kontakt/Firma
```

Die Elementdeklaration zu `Firma`, die geändert wurde, kommt im Typ `kontakt` vor. Die Elementdeklaration, die diesen Typ verwendet, ist nur die Deklaration zu `Kontakt`. Da dieses schon das Wurzelelement darstellt, konnte in der Hierarchie nicht weiter aufgestiegen werden. Daraus ergibt sich der oben aufgeführte XPath-Ausdruck.

Die `Validator`-Klasse wendet den XPath-Ausdruck auf das Instanzdokument an und selektiert im Beispiel 6.2 nur einen einzelnen Knoten mit Namen `Firma`. Dieser wird nun auf seine Eigenschaften überprüft, ob diese mit denen im Constraint gespeicherten übereinstimmen. Diese Überprüfung erfolgt nach dem Muster aus Abbildung 5.13, Seite 67.

Da überhaupt ein Knoten durch den XPath-Ausdruck selektiert wurde, erfolgt die Überprüfung, ob die Häufigkeitsangaben korrekt sind. Im Schema wurden keine Angaben diesbezüglich gemacht, somit gelten die Standardwerte der `minOccurs` und `maxOccurs`-Attribute und das Element muss genau einmal auftreten. Diese Quantorenwerte stimmen also überein, weil nur ein Knoten durch die Anfrage selektiert wurde. Danach wird überprüft, ob die Nachbarelemente korrekt sind, was der Fall ist, da alle in der Sequenz zum Typ `kontakt` deklarierten Knoten in der Instanz in der richtigen Reihenfolge auftreten. Deswegen wird zum Schluss der Typ überprüft, ob dieser korrekt ist.

Es handelt sich dabei um einen komplexen Typ, der eine Sequenz, bestehend aus vier Elementen, definiert. Deswegen wird kontrolliert, ob alle Elemente auftreten. Im Beispiel ist dies nicht der Fall, da in der Instanz lediglich zwei der vier deklarierten Elemente auftauchen. Aus diesem Grund werden die Quantorenwerte der fehlenden Elemente überprüft. Das Element `Webseite` fehlt, aber das `minOccurs`-Attribut in der Deklaration hat den Wert 0. Das bedeutet, dass das Element nicht auftreten muss. Das zweite fehlende Element ist `Name`, das während der Evolution eingefügt wurde. Bei diesem Element wurden keine Angaben zu `minOccurs` und `maxOccurs` gemacht, deswegen wird für beide der Standardwert 1 angenommen. Das bedeutet, dass das Element `Name` auftreten muss, es aber im Beispiel nicht tut, weswegen ein Standardwert generiert und vor das Element `Adresse` eingefügt werden muss. Die genaue Angabe, vor welchen Knoten eingefügt werden soll, kann aus der XSEL-Anfrage abgeleitet werden. Da als Anfrage die *insert_before*-Operation benutzt wurde, wird ihr `before`-Attribut ausgewertet. Wäre hingegen die Operation *add* benutzt worden, dann würde zur Adaption ebenfalls die *add*-Operation genutzt.

Die Abbildung 6.5 zeigt noch mal den Pfad von der Knotenselektion bis zu dem Punkt, an dem die inkrementelle Validierung einen Fehler feststellt.

Die `Validator`-Klasse meldet diesen Fehler der `Adapter`-Klasse, die mit dem Wissen, dass die Operation auf Schema-Ebene *insert_before* war und dass in der Sequenz ein Element fehlt, einen Standardwert generiert. Der Typ des fehlenden Elementes ist `string` und der Wert des `nillable`-Attributes in der Deklaration ist `false`. Dadurch wird eine Zeichenkette generiert, die mindestens über ein Zeichen verfügt und diese wird dann als Kindknoten vom Typ Text an ein generiertes Element mit dem Namen `Name` eingefügt. Die bei der Generierung von Standardwerten auftretenden Besonderheiten sind in Abschnitt 7.1.1 dargelegt. Für dieses Beispiel wird angenommen, dass der generierte Standardwert dem Punkt "." entspricht.

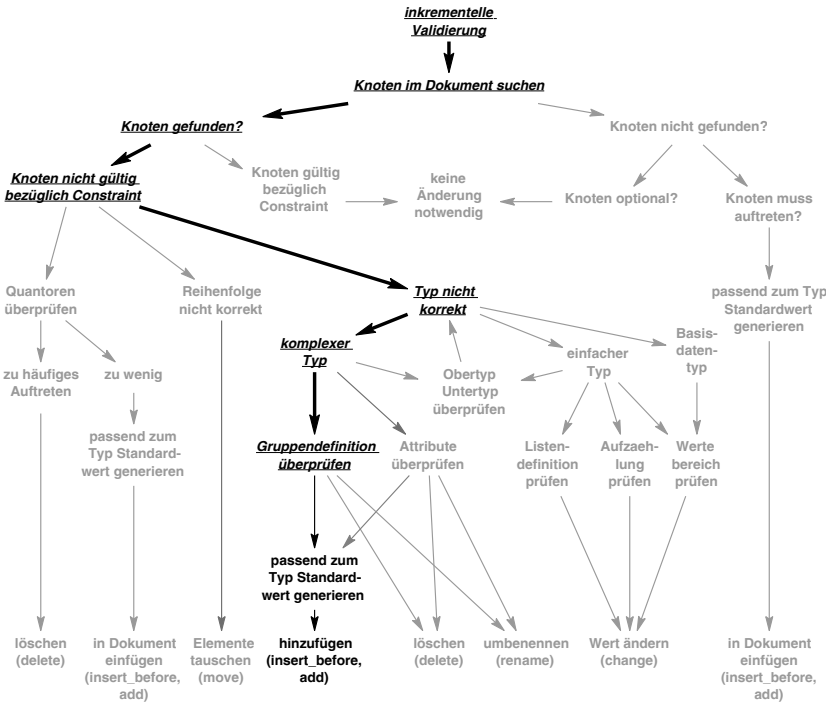

Abbildung 6.5: Ablauf bei der inkrementellen Validierung

Die von der `Adapter`-Klasse generierte Anfrage ist in Beispiel 6.3 aufgeführt.

Beispiel 6.3 (generierte Anfrage zur Adaption)

```
<insert_before
    select="/Kontakt/Firma"
    before="Adresse"
    content="&lt;Name&gt;.&lt;/Name&gt;"
/>
```

Zuerst wird der XPath-Ausdruck des Constraints zur Selektion genommen. Der Wert des `before`-Attributes der neuen Anfrage wird festgelegt durch die Auswertung des `before`-Attributes der Anfrage, die gegen das Schema gestellt wurde. Außerdem erfolgt eine erneute Kontrolle des Typs, damit sichergestellt werden kann, dass das Element `Adresse` in der Instanz vorkommt.

Der Wert des `content`-Attributes stellt dabei den generierten Standardwert für ein Element `Name` mit dem Typ `string` dar.

Nach beendeter inkrementeller Validierung des Instanzdokumentes umfasst die Liste der Anfragen nur die aus Beispiel 6.3, da keine weiteren Eigenschaften des Instanzelementes `Firma` verletzt sind. Die Liste wird dann zu einer losen Gruppierung zusammengefasst und an das Anfragemodul des XSEL-Frameworks übergeben. Dieses generiert ein `QueryCollection`-Objekt, dem das gerade validierte Dokument vom Typ `XMLDocument` übergeben wird. Nach Ausführung der losen Gruppierung, im Fall des Beispiels nur eine einzelne Anfrage, ist das Instanzdokument wieder valide. Beispiel 6.4 zeigt einen Ausschnitt mit dem angepassten Firmen-Element.

Beispiel 6.4 (adaptiertes Instanzdokument (Ausschnitt))

```
<?xml version="1.0" encoding="UTF-8"?>
<Kontakt  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="kontakt.xsd">
    ...
    <Firma>
        <Name>.</Name>
        <Telefon>012/3456789</Telefon>
        <e-mail>www@w3c</e-mail>
    </Firma>
    ...
</Kontakt>
```

Nach Adaption aller Dokumente müssen die Anfragen überprüft werden. Beispiel 6.5 zeigt eine kurze XQuery-Anfrage, die aus den Firmen-Elementen der Dokumente jeweils das erste Kind extrahiert und verarbeitet.

Beispiel 6.5 (anzupassende XQuery-Anfrage)

```
<Person>{
LET $a IN /Kontakt/Firma/child::*[1],
    $n IN /Kontakt/Nachname
    RETURN
        <Name>{$n/text()}</Name>
        <Firmenkontakt>{$a/text()}</Firmenkontakt>
}</Person>
```

Der Sinn dieser Anfrage besteht darin, die Nachnamen der Personen zu deren Firmenadressen zuzuordnen. Sofern keine Firmenadresse angegeben ist, soll die Telefonnummer der Firma erfasst werden. Weil dies aber durch die Selektion des ersten Kindknotens des Firmenelementes geschieht, kann nach Anpassung der Dokumente die Anfrage weiterhin ausgeführt werden. Das durch die Anfrage konstruierte Ergebnis ist jedoch falsch. Deshalb wird für den Nutzer eine Meldung generiert, dass die XQuery-Anfrage eventuell abzuändern ist, da sich der Typ des Elementes `Firma` geändert hat.

Aus dem Beispiel kann man ersehen, wie die Umsetzung eines Evolutionsschrittes aussieht. Dabei entspricht ein Evolutionsschritt einer atomaren Operation oder einer Transaktion.

Das Framework versucht die Evolution automatisch umzusetzen, inklusive der Datenadaption. Dabei können Besonderheiten, die in Kapitel 7 beschrieben werden, auftreten. In den Fällen, wo diese auftreten, kann eine automatische Adaption nur sehr schwer oder gar nicht mehr durchgeführt werden und es muss dem Nutzer gemeldet werden, wo eventuell Korrekturen durchzuführen sind. Die Anpassung der XQuery-Anfragen findet nur dahingehend statt, dass die Anfragen geladen werden und nach XPath-Ausdrücken gefiltert werden. Sofern diese Ausdrücke Knoten selektieren, die verändert wurden, oder die aufgrund der Datenadaption an andere Positionen im Dokument gerückt sind, wird dem Nutzer mitgeteilt, welche Anfragen er womöglich anzupassen hat. Nachdem alle XQuery-Anfragen so geprüft wurden und dem Nutzer mögliche Änderungen mitgeteilt wurden, ist ein Evolutionsschritt abgeschlossen.

Kapitel 7

Adaption der Daten und Anfragen

In diesem Kapitel wird ein Überblick über die Adaption gegeben. Es wird genauer darauf eingegangen, was zum Beispiel bei der Generierung von Standardwerten beachtet werden muss. Zusätzlich können noch Besonderheiten auftreten, deren Ursachen teilweise in den Definitionen von XML-Schema und XQuery liegen und die berücksichtigt werden müssen.

7.1 Dokumente

7.1.1 Generierung von Standardwerten

Je nachdem, wie die Änderungen auf dem Schema aussehen, müssen in manchen Anwendungsfällen Standardwerte in die Dokumente eingefügt werden. Dies ist der Fall, wenn bei einer Gruppendeklaration deren Typ von `choice` auf `sequence` geändert wird. Ursache ist die Festlegung, dass bei `choice` nur eines der deklarierten Elemente im Instanzdokument auftreten darf, bei der Sequenz `sequence` hingegen alle deklarierten Elemente in der Reihenfolge ihrer Deklaration im Schema. Somit ist vor der Änderung nur eines der deklarierten Elemente in den Instanzen vorhanden und für die darüber hinaus in der Gruppe deklarierten Elemente müssen Standardwerte in den Dokumenten eingefügt werden, damit die Instanz bezüglich des Schemas valide ist.

Zudem gestaltet sich die automatische Generierung von Standardwerten sehr verschieden, da in XML-Schema grundsätzlich drei Arten von Typen unterschieden werden, die Basisdatentypen, die einfachen Typen und die komplexen Typen.

Basisdatentyp

Zur Generierung von Standardwerten bezüglich der Basisdatentypen können bestimmte Werte vorgegeben werden. Zum Beispiel kann festgelegt werden, dass der Standardwert für Integer die Null (0) ist, für Datumstypen hingegen ein bestimmter festgelegter Tag wie der 01.01.1970 oder eine festgelegte Uhrzeit wie 0:00. Allerdings ist diese Festlegung nur ein Hilfsmittel und differiert je nach den vorliegenden Daten und Nutzeranforderungen.

Eine andere Möglichkeit ist das Einsetzen einer der Wertebereichsgrenzen des Typs als Standardwert. Es bleibt aber die Frage, wie die Grenzen aussehen oder wie diese definiert sind. Es wird eine Ordnung auf dem Wertebereich vorausgesetzt. Bei der Betrachtung von Zeichenketten, welche vom Typ `string` sind, stellt die leere Zeichenkette sicherlich eine untere

Grenze. Doch die obere Grenze ist theoretisch nicht gegeben, da unendlich lange Zeichenketten möglich wären. Dazu kommt eine gewisse Abhängigkeit vom verarbeitenden System, ob der Anfrageprozessor in Java oder C++, auf Windows oder Linux, auf einer 32Bit oder 64Bit Plattform entwickelt wurde, weil dadurch unter anderem Unterschiede in den Definitionen der Wertebereichsgrenzen von Zahlen auftreten können.

einfacher Typ (*simpleType*)

Am aufwendigsten ist die Generierung bei vorliegenden einfachen Typen, in XML-Schema als `simpleType` definiert. Diese einfachen Typen können einer Vereinigung verschiedener anderen einfacher Typen entsprechen oder einer Liste oder Aufzählung mit möglichen Werten. Liegt eine Aufzählung der erlaubten Werte vor, ist die Generierung einfach, da aus dieser Aufzählung ein Wert gewählt werden muss. Bei der Vereinigung von einfachen Typen hingegen muss man die einzelnen Typdefinitionen betrachten, welche wiederum eine Vereinigung anderer sein können. Zusätzlich können einfache Typen einer Einschränkung eines Basisdatentyps entsprechen oder eines zuvor definierten einfachen Typs. Es sind viele Möglichkeiten zur Einschränkung (`restriction`) in XML-Schema definiert, welche als Facetten bezeichnet werden. Eine Möglichkeit wäre die Angabe der Wertebereichsgrenzen (`minInclusive`, `maxInclusive`, `minExclusive`, `maxExclusive`), eine andere, aber schwerer handhabbarere Einschränkung wäre die Angabe eines regulären Ausdrucks, eines Patterns. Dieser muss gesondert analysiert werden und durch diese Analyse ein Wert konstruiert werden, welcher vom Pattern einerseits akzeptiert wird und andererseits als Standardwert dienen kann. Eine Festlegung, welcher von den akzeptierten Werten als Standardwert sinnvoll wäre, kann nicht automatisch geschehen, sondern muss wie bei den Standardwerten von Basisdatentypen vom Nutzer selber eingebracht werden.

Als Beispiel dient der Ausdruck $\backslash d\{5\}$ eines Pattern, wie er zur Definition für Postleitzahlen im Beispiel aus Anhang B eingesetzt wird. Die Frage ist nun, ob der Wert 00000 besser als Standardwert geeignet ist, als der Wert 11111, da beide Werte von dem angegebenen Ausdruck akzeptiert werden. Dadurch muss in einer Entscheidungsfindung der Nutzer mit einbezogen werden, da nur er nach seinen Anforderungen an die Daten entscheiden kann, welcher Wert sinnvoller und geeigneter ist. Das vorgestellte Framework versucht einen beliebigen Wert zu generieren, der durch das Pattern akzeptiert wird. Dies geschieht nach den Regeln, wie sie am Ende dieses Kapitels genannt werden. Zusätzlich meldet das System dem Nutzer, dass eine eventuelle Korrektur des Wertes nötig sein könnte.

komplexer Typ (*complexType*)

Bei Elementen mit komplexen Typen gestaltet sich die Generierung ganz unterschiedlich, da komplexe Typen die Attribute und Kindelemente eines Elementes deklarieren. Es ist deswegen denkbar, dass die Generierung über den Abstieg in der Hierarchie gelöst wird, da auf der untersten Ebene des Baumes die Elementdeklarationen immer einen einfachen Typ oder einen Basisdatentyp aufweisen oder nur über Attribute verfügen, die als Werte nur solche aus dem Wertebereich eines einfachen Typs oder Basisdatentyps besitzen dürfen.

Das bedeutet, dass für das Element, für das ein Standardwert generiert werden soll, XML-Literale produziert werden, die den Elementkonstruktoren in XQuery entsprechen. Anschließend wird für im Typ deklarierte Elemente dieselbe Prozedur durchgeführt und die so generierten XML-Elemente als Kindknoten an das zuvor generierte XML-Literal angehängt. Wenn die Typen der Kindknoten ebenfalls komplex sind, dann wird der Prozess erneut durchgeführt, bis man alle deklarierten Kinder und Kindeskinder generiert hat. Für die Attribute und die Knoten mit einfachen Typen oder Basisdatentypen werden dann die Standardwerte entsprechend hinzugefügt. Auf diesem Weg erhält man ein XML-Dokumentfragment, welches dann als Wert für das Element in der Instanz eingefügt wird. Abbildung 7.1 zeigt diesen iterativen Prozess in einer Übersicht.

Die Besonderheiten, die sich dabei ergeben, sind die gleichen wie bei der Generierung von Standardwerten für einfache und Basisdatentypen.

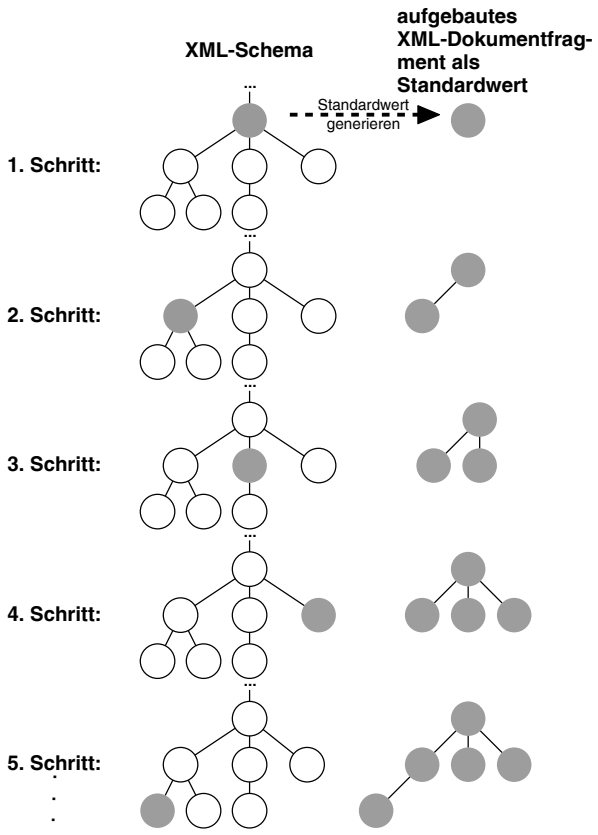

Abbildung 7.1: Generierung von Standardwerten bei komplexen Typen

7.2 XQuery-Anfragen

7.2.1 XPath-Ausdrücke

Die Schwierigkeit bei der Adaption von XQuery-Anfragen ist an erster Stelle die generelle Entscheidung, ob eine Anfrage geändert werden muss oder nicht. Um diese Entscheidung

treffen zu können, muss herausgefunden werden, ob die Anfragen auf Dokumentteile zugreifen, welche geändert worden sind.

Zur Selektion von Knotenmengen benutzt XQuery XPath-Ausdrücke. Bei der Anwendung der Ausdrücke auf Knotenmengen können sie diese über Prädikate. oder Bedingungen weiter einschränken, aber aufgrund der Navigation über Achsen diese Mengen auch erweitern. Als Beispiel dient die Selektion einer Knotenmenge, welche nur einen Knoten beinhaltet. Da XPath abgeschlossen ist, kann der selektierte Knoten als Kontextknoten dienen und damit als Eingabe für einen neuen XPath-Ausdruck. Von ihm ausgehend können dadurch alle seine Vorgänger, die Knoten auf seiner *ancestor*-Achse, selektiert werden. Je nach Dokument und anfänglich selektiertem Knoten sind in der neuen Ergebnismenge nun mehrere Knoten möglich, die wiederum Kontextknoten für erneute Auswertungen von XPath-Ausdrücken sein können.

Der Entwurf zu XQuery sieht Orthogonalität vor, somit sind alle XQuery-Konstrukte beliebig miteinander kombinierbar. Da ein XPath-Ausdruck ebenfalls ein XQuery-Konstrukt sein kann, können XPath-Ausdrücke an jeder Stelle in z.B. einem FLWOR-Ausdruck auftreten. Dies erschwert neben den oben genannten Eigenschaften von XPath zusätzlich die Entscheidung, ob eine Anfrage geändert werden soll oder nicht.

Wenn sich in der selektierten Knotenmenge ein Knoten befindet, welcher im Zuge der Dokumentadaption geändert wurde, genügt es nicht, nur diesen zu betrachten, sondern es müssen auch seine Vorfahren betrachtet werden, da es bei deren Änderungen einen direkten Einfluss auf den darunter liegenden Teilbaum gibt. Gleichermaßen muss der unter den selektierten Knoten befindliche Teibaum betrachtet werden, da in den XQuery-Konstrukten Teile davon ausgewertet werden können, welche aber zuvor nicht in einer Variablen gebunden wurden.

Eine Möglichkeit wäre das Filtern der Anfragen nach XPath-Ausdrücken und der Auswertung eines jeden einzelnen. Somit würde man nur die Knoten, welche wirklich durch die Anfrage verarbeitet werden, erhalten. Das Filtern ist allerdings schwierig, weil die XPath-Ausdrücke nicht speziell gekennzeichnet sind, anders als die Variablenbindungen in XQuery, welche mit einem Dollarzeichen (`$`) beginnen. Weil XPath-Ausdrücke Achsenbezeichner (siehe 3.2.2) beinhalten können, ist nach dem Filtern eine eindeutige Entscheidung darüber, ob die selektierten Knoten geändert wurden, nicht gegeben. Dies liegt daran, dass mittels der Achsenbezeichner von jedem Knoten zu jedem Knoten navigiert werden kann.

7.2.2 Semantik/Pragmatik der extrahierten Informationen

Die vollautomatische Anpassung von XQuery-Anfragen im Zuge der Abarbeitung eines Evolutionsschrittes ist nicht möglich, da neben vielen syntaktischen Fehlern, welche die Ausführbarkeit einer Anfrage verhindern, auch semantische Fehler auftreten können. Dies bedeutet, dass die Anfragen korrekt und ausführbar sind und auch Ergebnisse liefern. Diese sind syntaktisch richtig, jedoch sind die Informationen in diesen Ergebnissen semantisch nicht korrekt. Hinzu kommen Mehrdeutigkeiten bei den Möglichkeiten für eine Änderung der Anfragen. In Beispiel 7.1 ist eine XQuery-Anfrage gezeigt, bei der Fehler bezüglich den extrahierten Informationen auftreten können und Mehrdeutigkeiten in der Anpassung existieren.

Beispiel 7.1 (XQuery-Anfrage)

```
<Personen>{
FOR $a IN /Kontakt/Adresse,
    $n IN /Kontakt/Nachname/text()
    FOR $t in $a/following-sibling::*[1]
    RETURN <{$n}>
        <Adresse>{$a/text()}</Adresse>
        <Telefon>{$t/text()}</Telefon>
    </{$n}>
}</Personen>
```

In dem Beispiel wird aus einem Dokument, welches Kontaktinformationen repräsentiert, neben dem Namen der Person auch deren Adresse selektiert sowie der unmittelbar rechte Nachbar des Adresselementes. Die Selektion des Nachbarn geschieht über die Geschwisterachse `following-siblings` durch den Ausdruck `$a/following-sibling::*[1]`. Wird nun nach dem Adresselement ein neues Element eingefügt, dann liefert die Anfrage weiterhin Ergebnisse, diese sind aber in den extrahierten Informationen falsch. Es existieren bei der Änderung der Anfrage zwei Möglichkeiten, damit auch die extrahierten Informationen wieder korrekt sind. Einerseits kann man den XPath-Ausdruck von `$a/following-sibling::*[1]` zu `$a/following-sibling::*[2]` abändern, da nach dem Einfügen die vorher extrahierten Informationen nicht mehr im direkten rechten Nachbarelement, sondern im zweiten rechten Nachbarelement vorhanden sind. Andererseits kann man die `RETURN`-Klausel dahingehend abändern, dass nicht mehr ein Element `Telefon` erzeugt wird, sondern ein anderes Element mit dem Namen des in der Instanz neu eingefügten Elementes.

Die Entscheidung darüber, wie die XQuery-Abfrage geändert werden soll, kann nicht automatisch geschehen, sondern der Nutzer muss festlegen, welche Informationen er nach erfolgter Evolution aus den Dokumenten extrahieren möchte. Eine automatische Anpassung kann nur versuchen, die syntaktische Korrektheit sicherzustellen, jedoch nicht, dass die extrahierten Informationen weiterhin den Nutzeranforderungen genügen.

In der Implementierung des Frameworks wurde versucht, den Prozess der Adaption automatisch durchzuführen. Es wurde dafür eine Liste an Standardwerten für die Basisdatentypen im System definiert, damit bei der Generierung von Standardwerten der Nutzer so selten wie möglich korrigieren muss. Diese Liste kann der Nutzer frei definieren und so seinen Anforderungen anpassen. Weiterhin lässt sich die Generierung von komplexen Typen nahezu vollständig automatisieren, da komplexe Typen nur Beziehungen modellieren. Die einzelnen Elemente oder Attribute besitzen entweder wiederum einen komplexen oder einfachen Typen oder sie verfügen über einen Basisdatentyp, so dass die auftretenden Besonderheiten schon bei den Basisdatentypen behandelt wurden.

Bei der Generierung von Werten für einfache Typen wird der Prozess dadurch automatisiert, dass bei Einschränkungen der Wertebereichsgrenzen, diese Schranken als Standardwert dienen. Ein Beispiel sind die Angaben über `minInclusive` und `maxInclusive` bei der Definition des einfachen Typs. Sofern ein Wert für diesen Typ generiert werden soll, wird der Wert

von `minInclusive` genommen. Bei der Definition über eine Aufzählung wird immer der erste genannte Wert als Standardwert genommen. Bei der Einschränkung über ein Pattern wird dieses gesondert geparst und daraus ein Wert konstruiert. Für die einzeln benötigten Zeichen für die Ausdrücke $\backslash w$, $\backslash d$ und $\backslash s$ ist ebenfalls eine Liste im System abgelegt, die angibt, welches Zeichen z.B. den Ausdruck $\backslash d$ repräsentieren soll, da $\backslash d$ eine ganze Zahl zwischen 0 und 9 akzeptiert.

Über diese Hilfslisten und der in Abbildung 7.1 aufgezeigten Vorgehensweise bei komplexen Typen wird bei der Anpassung der Daten ein hoher Grad an Automatisierung erreicht.

Kapitel 8

Schlussbetrachtung

8.1 Zusammenfassung

Das Ziel dieser Arbeit war die Definition einer Änderungssprache für XML-Schema (XSEL). Neben der Definition der Sprache stand auch die Beschreibung und Umsetzung der Adaption der Daten und Anfragen im Vordergrund. Die prototypische Implementierung eines Anfrageprozessors zur Verarbeitung der XSEL-Anfragen wurde mit Java und unter Zuhilfenahme von Xerces[1] als XML-Parser realisiert.

Der Sprachumfang von XSEL umfasst zehn Operationen, die dem Einfügen, Löschen und Abändern von Teilen eines XML-Dokumentes dienen. Die allgemeine Syntax der Anfragen entspricht der XML-Syntax. Der Vorteil ist, dass dadurch keine zusätzlichen Parser benötigt werden. Neben diesen Operationen besitzt der Nutzer die Möglichkeit zur Gruppierung von Anfragen. Diese Anfragen und Gruppen von Anfragen werden an Schemata gestellt und durch deren Auswertung und Anwendung wird das Schema manipuliert.

Das System lädt nach dem Ändern des Schemas die Dokumente, die durch das Schema definiert werden und unterzieht sie der inkrementellen Validierung. Während des Prozesses der inkrementellen Validierung kommen so genannte Constraints zum Einsatz, die alle Informationen einer Deklaration bereitstellen. Diese Constraints dienen als Zeiger und weisen auf die entsprechenden Elemente und Attribute im Dokument, deren Deklaration sie repräsentieren. Somit werden nur die Knoten, deren Deklarationen im Schema geändert worden sind, auf Validität überprüft. Die einzelnen Eigenschaften der Constraints werden dabei nacheinander beurteilt und geschaut, ob sie erfüllt sind oder nicht.

Das Prinzip der inkrementellen Validierung ist ein Schwerpunkt dieser Arbeit, da diese nicht nur zur Validitätskontrolle der Dokumente dient, sondern es gleichzeitig erlaubt, gezielt Anfragen zur Dokumentadaption zu generieren. Das Prinzip ist damit so angelegt, dass eine Beschreibung der zu überprüfenden Eigenschaften vorliegt und dass durch deren Auswertung nicht nur eine Aussage getroffen werden kann, ob ein Knoten valide oder nicht ist, sondern man gleichzeitig die Möglichkeit hat, genau die Eigenschaften zu nennen, die die Validität nicht erfüllen. Somit ist es möglich, gezielt und automatisch Anfragen zu generieren, die diese Eigenschaften ändern. Das System generiert also während der Validitätskontrolle eine Liste an XSEL-Anfragen, welche nach dem Laden und Validieren der Dokumente gegen sie

[1] http://xml.apache.org/xerces2-j

gestellt werden. Durch die Auswertung dieser generierten Anfragen wird die Adaption der Daten vorgenommen. Aus diesem Grund wurden im Sprachentwurf zu XSEL die Operationen allgemein gehalten und keine speziellen Typ- oder Deklarationskonstruktoren definiert, so dass die Anfragen allgemein auf XML-Dokumenten arbeiten können, ohne eine Unterscheidung zwischen XML-Dokument und Schema vornehmen zu müssen. Das System verfügt über das Wissen, ob aktuell ein Schema oder ein Dokument verarbeitet wird.

Für die Anpassung der im System abgelegten XQuery-Anfragen werden Meldungen generiert, die dem Nutzer sagen, ob eine Anfrage überarbeitet werden muss oder nicht. Eine automatische Anpassung der Anfragen ist nicht möglich, da es einerseits Mehrdeutigkeiten bei den Änderungsmöglichkeiten gibt und andererseits das System nicht entscheiden kann, ob Anfragen syntaktisch zwar richtig sind, semantisch aber falsche Ergebnisse liefern.

8.2 Ausblick und weitere Ideen

Betreffend des Sprachumfangs ist eine Erweiterung um reine Anfrageoperationen denkbar, so dass XSEL mehr einer XML-Anfragesprache gleicht. Auch die Möglichkeit der Verschachtelung von Anfragen ist denkbar, so dass die Forderung nach Orthogonalität erfüllt werden kann. Darüber hinaus kann das in 5.3.2 eingeführte Transaktionskonzept um die fehlenden Konzepte erweitert werden. Es könnten Sperrmechanismen unterschiedlicher Granularität auf den Dokumenten definiert und im Framework integriert werden. Derzeit wird davon ausgegangen, dass der Anfrageprozessor als alleiniger Prozess auf die Schemata und Dokumente zugreift und dass während der Änderungen und der Datenadaption keine weiteren Schreib- oder Lesezugriffe stattfinden. Aus diesem Grund sind Sperrmechanismen nicht notwendig, da Seiteneffekte wie *Dirty Read*, *Phantom Read* oder *Lost Update*[2] nicht auftreten können, da gleichzeitige Transaktionen nicht vorkommen. Sollte das Framework aber in einem Informationssystem eingesetzt werden, kann nicht mehr davon ausgegangen werden, dass keine anderen Lese- und Schreibzugriffe auf die Dokumente während eines Evolutionsschrittes stattfinden. Spätestens dann wären Konzepte aus dem Datenbankbereich wie Sperren, Serialisierungsmechanismen oder Scheduler und ein umfangreicheres Transaktionskonzept sinnvoll.

Es wurde Wert auf eine Automatisierung der Datenadaption gelegt und die generierten Anfragen werden aus den schemaändernden Anfragen und den Eigenschaften der Deklarationen abgeleitet. In Kapitel 4 wurden darüber hinaus Ansätze vorgestellt, die bei einer Adaption auch die Informationen und Informationskapizitäten der Dokumente betrachten. Es wäre denkbar, diese Ansätze zu vereinen, damit die Evolutionsanforderung eines minimalen Datenverlustes besser erfüllt werden kann. So kann in den Prozess der Anfragengenerierung ein Informationsmaß integriert werden, so dass darüber eine bessere Kontrolle über die generierten Anfragen und die generierten Standardwerte erreicht werden kann. Über ein Informationsmaß kann die Wertgenerierung dahingehend verbessert werden, dass sinnvollere Werte erzeugt werden, da über ein solches Informationsmaß auch semantische Informationen in den Prozess einfließen können, so dass nicht nur die syntaktische Korrektheit gewährleistet wird.

[2] siehe [HS99] S.412*ff*

Es ist auch denkbar, dass der Umfang und die Möglichkeiten in XQuery sinnvoll eingeschränkt werden und dass eine bestimmte Form festgelegt wird, wie es in dieser Arbeit für XML-Schema angenommen wurde. Dadurch könnte man viele Schwierigkeiten, die eine automatische Adaption nicht erlauben, beseitigen. So könnte man festlegen, dass XPath-Ausdrücke nur in den Klauseln `FOR` und `LET` auftreten dürfen, wo deren Ergebnisse an Variablen gebunden werden. Dadurch würde sich der Aufwand des Filterns, wie er in Abschnitt 7.2 angesprochen wurde, erheblich reduzieren. Es wäre auch gewährleistet, dass nur die den Variablen zugewiesenen Knoten in der XQuery-Anfrage verarbeitet werden, was eine Entscheidung darüber, ob eine Anfrage geändert werden muss oder nicht, vereinfacht. Ob so eine Einschränkung sinnvoll und machbar ist, ohne die Mächtigkeit von XQuery zu schmälern, müsste geprüft werden. Eine Einschränkung für XPath, dass keine Achsenbezeichner verwendet werden dürfen ist möglich, es müsste aber geschaut werden, ob diese Einschränkung auch sinnvoll ist. Dadurch könnte man die Entscheidung über das Abändern der Anfrage weiter konkretisieren, da dann in den XPath-Ausdrücken nur entlang den Eltern-Kind-Beziehungen navigiert werden würde.

Um dem Nutzer mehr Möglichkeiten für Schemaänderungen zu geben, wäre es denkbar, dass die in Abschnitt 4.5 vorgestellten Grundprinzipien von SERF mit XSEL vereint werden. Man könnte ausgewählte XSEL-Anfragen als Grundprimitive verwenden. Durch die Verallgemeinerung von zusammengesetzten Anfragen, wie sie mit den Transaktionen schon vorgeschlagen wurden, könnte ebenso eine Optimierung einhergehen, die die Anfragen in Bezug auf verschiedene Prozessmetriken wie Ausführdauer und Speicherbedarf optimiert. Ein weiterer Punkt wäre die Eliminierung von sich gegenseitig aufhebenden Operationen oder von Operationen ohne Einfluss und eine mögliche Neuordnung der Anfragenreihenfolge in den Transaktionen, was weitere Optimierungen bezüglich der Effizienz nachsichführen könnte.

Anhang A

Schema für XSEL-Anfragen

Nachfolgend ist das XML-Schema für die einzelnen Anfragen von XSEL abgebildet.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified" attributeFormDefault="unqualified">
   <xs:complexType name="add" mixed="false">
      <xs:attribute name="select" type="xs:string" use="required"/>
      <xs:attribute name="content" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="insert_before" mixed="false">
      <xs:attribute name="select" type="xs:string" use="required"/>
      <xs:attribute name="content" type="xs:string" use="required"/>
      <xs:attribute name="before" type="xs:string" use="required" nillable="true"/>
   </xs:complexType>
   <xs:complexType name="insert_after" mixed="false">
      <xs:attribute name="select" type="xs:string" use="required"/>
      <xs:attribute name="content" type="xs:string" use="required"/>
      <xs:attribute name="after" type="xs:string" use="required" nillable="true"/>
   </xs:complexType>
      <xs:complexType name="delete" mixed="false">
      <xs:attribute name="select" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="change" mixed="false">
      <xs:attribute name="select" type="xs:string" use="required"/>
      <xs:attribute name="value" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="rename" mixed="false">
      <xs:attribute name="select" type="xs:string" use="required"/>
      <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="replace" mixed="false">
      <xs:attribute name="select" type="xs:string" use="required"/>
      <xs:attribute name="content" type="xs:string" use="required" nillable="true"/>
   </xs:complexType>
   <xs:complexType name="move_after" mixed="false">
      <xs:attribute name="select" type="xs:string" use="required"/>
      <xs:attribute name="after" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="move_before" mixed="false">
      <xs:attribute name="select" type="xs:string" use="required"/>
```

```
   <xs:attribute name="before" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="move" mixed="false">
   <xs:attribute name="select" type="xs:string" use="required"/>
   <xs:attribute name="to" type="xs:string" use="required"/>
</xs:complexType>

<xs:element name="add" type="add"/>
<xs:element name="insert_before" type="insert_before"/>
<xs:element name="insert_after" type="insert_after"/>
<xs:element name="move" type="move"/>
<xs:element name="move_before" type="move_before"/>
<xs:element name="move_after" type="move_after"/>
<xs:element name="replace" type="replace"/>
<xs:element name="rename" type="rename"/>
<xs:element name="change" type="change"/>
<xs:element name="delete" type="delete"/>

<xs:element name="transaction">
<xs:complexType>
   <xs:choice maxOccurs="unbounded">
      <xs:element ref="add"/>
      <xs:element ref="insert_before"/>
      <xs:element ref="insert_after"/>
      <xs:element ref="move"/>
      <xs:element ref="move_before"/>
      <xs:element ref="move_after"/>
      <xs:element ref="replace"/>
      <xs:element ref="rename"/>
      <xs:element ref="change"/>
      <xs:element ref="delete"/>
   </xs:choice>
</xs:complexType>
</xs:element>

<xs:element name="queries">
   <xs:complexType>
      <xs:choice maxOccurs="unbounded">
         <xs:element ref="transaction"/>

         <xs:element ref="add"/>
         <xs:element ref="insert_before"/>
         <xs:element ref="insert_after"/>
         <xs:element ref="move"/>
         <xs:element ref="move_before"/>
         <xs:element ref="move_after"/>
         <xs:element ref="replace"/>
         <xs:element ref="rename"/>
         <xs:element ref="change"/>
         <xs:element ref="delete"/>
      </xs:choice>
   </xs:complexType>
```

```
   </xs:element>

   <xs:group name="query">
      <xs:choice>
         <xs:element ref="queries"/>
         <xs:element ref="transaction"/>

         <xs:element ref="add"/>
         <xs:element ref="insert_before"/>
         <xs:element ref="insert_after"/>
         <xs:element ref="move"/>
         <xs:element ref="move_before"/>
         <xs:element ref="move_after"/>
         <xs:element ref="replace"/>
         <xs:element ref="rename"/>
         <xs:element ref="change"/>
         <xs:element ref="delete"/>
      </xs:choice>
   </xs:group>
</xs:schema>
```

Anhang B

XML-Schema Beispiel Kontakt

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="qualified">
    <xs:complexType name="firma">
        <xs:sequence>
            <xs:element name="Adresse" type="adresse" minOccurs="0"/>
            <xs:element name="Telefon" type="telefon"/>
            <xs:element name="e-mail" type="email"/>
            <xs:element name="Webseite" type="webseite" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="stadt" mixed="true">
        <xs:attribute name="KFZ" type="kfz"/>
        <xs:attribute name="Bundesland" type="xs:string"/>
    </xs:complexType>
    <xs:simpleType name="kfz">
        <xs:restriction base="xs:string">
            <xs:pattern value="[A-Z]{1,4}"/>
        </xs:restriction>
    </xs:simpleType>
    <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="stadt"/>
            <xs:element name="Land" 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="webseite">
        <xs:restriction base="xs:string">
            <xs:pattern value="http\://[\w.-_/]*"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="datum">
        <xs:restriction base="xs:string">
            <xs:pattern value="[0123][0-9].[01][0-9].\d{4}"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:complexType name="kontakt">
        <xs:sequence>
            <xs:element name="Nachname" type="xs:string"/>
            <xs:element name="Vorname" type="xs:string"/>
            <xs:element name="Geburtsdatum" type="datum"/>
            <xs:element name="Adresse" type="adresse" maxOccurs="2"/>
            <xs:element name="Firma" type="firma"/>
            <xs:element name="Handy" type="telefon" nillable="true" maxOccurs="unbounded"/>
            <xs:element name="Telefon" type="telefon" nillable="true"/>
            <xs:element name="e-mail" type="email"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="Kontakt" type="kontakt"/>
</xs:schema>
```

Abkürzungsverzeichnis

ACID	**A**tomicity, **C**onsistency, **I**solation, **D**urability	S. 61
CLOB	**C**haracter **L**arge **Ob**ject	S. 29
DOM	**D**ocument **O**bject **M**odel	S. 16
DTD	**D**ocument **T**ype **D**efinition	S. 1
ER	**E**ntity **R**elationship	S. 36
FLWOR	**F**or **L**et **W**here **O**rder by **R**eturn	S. 7
FLWR	**F**or **L**et **W**here **R**eturn	S. 7
NaN	**N**ot **a** **N**umber	S. 15
ODMG	**O**bject **D**ata **M**anagement **G**roup	S. 21
OODB	**o**bjekt**o**rientierte **D**aten**b**ank	S. 35
OODBS	**o**bjekt**o**rientiertes **D**aten**b**ank-**S**ystem	S. 34
OQL	**O**bject **Q**uery **L**anguage	S. 21
RTTI	**R**un**t**ime **T**ype **I**dentification	S. 19
SAX	**S**imple **A**PI for **X**ML	S. 17
SERF	**S**chemaevolution **E**xtensible **R**e-usable **F**lexible Framework	S. 35
SFW	**S**elect-**F**rom-**W**here-Klausel	S. 21
SQL	**S**tructured **Q**uery **L**anguage	S. 21
UML	**U**nified **M**odelling **L**anguage	S. 19
URI	**U**niform **R**esource **I**dentifier	S. 15
URL	**U**niform **R**esource **L**ocator	S. 11
XEM	**X**ML **E**volution **M**anagement	S. 29
XML	e**X**tensible **M**arkup **L**anguage	S. 1
XPath	**X**ML **Path** Language	S. 7
XQL	**X**ML **Q**uery **L**anguage	S. 21
XSDNF	**X**ML **S**chema **D**ocument **N**ormal **F**orm	S. 41
XSEL	**X**ML-**S**chema **E**volution **L**anguage	S. 41
XSL	e**X**tensible **S**tylesheet **L**anguage	S. 10
XSLT	e**X**tensible **S**tylesheet **L**anguage **T**ransformation	S. 5

Index

Beispiele

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AMN+01] Noga Alon, Tavo Mil, Frank Neven, Dan Suciu, and Victor Vianu. XML with Data Values: Typechecking Revisited. PODS, USA, 2001. Association for Computing Machinery.

[Bac00] Mike Bach. *XSL und XPATH.* Addison-Wesley, 2000.

[Bou03] Hassina Bounif. *Predictive Approach for Database Schema Evolution.* Swiss Federal Institute of Technology, 2003.

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

[CJR98a] Kajal T. Claypool, Jing Jin, and Elke A. Rundensteiner. *OQL-SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework.* Worcester Polythechnic Institute, Juli 1998.

[CJR98b] Kajal T. Claypool, Jing Jin, and Elke A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-Usable and Flexible Framework. In *7th International Conference on Information and Knowledge Management*, pages 314–321, Bethesda, Maryland, 1998.

[CNR99] Kajal T. Claypool, Chandrakant Natarajan, and Elke A. Rundensteiner. Optimizatizing the Performance of Schema Evolution Sequences, 1999.

[Coo03] Sergey V. Coox. *Axiomatization of the Evolution of XML Database Schemas.* St. Petersburg State University, 2003.

[CR99] Kajal T. Claypool and Elke A. Rundensteiner. Flexible Database Transformations: The SERF Approach. *IEEE Data Engineering Bulletin*, 22(1):19–24, 1999.

[CR03] Kajal T. Claypool and Elke A. Rundensteiner. *Sangam: A Transformation Modeling Franework*, Oktober 2003.

[CS03] Sergey V. Coox and Andrey A. Simanovsky. *Regular Expressions in XML Schema Evolution.* St. Petersburg State University, 2003.

[Fau01] Rasmus Faust. *Prolog-basierte Modellierung von Format-Evolution für semistrukturierte Daten.* Universität Rostock, Diplomarbeit, 2001.

[Ges03] Michael Gesmann. *Manipulation von XML-Dokumenten in Tamino*. Software AG, Darmstadt, 2003.

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

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

[HS99] Andreas Heuer and Gunter Saake. *Datenbanken: Implementierungstechniken*. mitp, 1999.

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

[JLST03] H.V. Jagadish, Laks V.S. Lakshmanan, Divesh Srivastava, and Keith Thompson. *TAX: A Tree Algebra for XML*, 2003.

[KM02] Meike Klettke and Holger Meyer. *XML und Datenbanken*. dpunkt, 2002.

[Kra01] Diane K. Kramer. *XEM: XML Evolution Management*. Worcester Polytechnic Institute, Master Arbeit, Mai 2001.

[KSR02] Bintou Kane, Hong Su, and Elke A. Rundensteiner. Consistently Updating XML Documents using Incremental Constraint Check Queries. WIDM, USA, 2002. Association for Computing Machinery.

[LN03] D. Luciv and B. Novikov. *Semistructered Database Scheme Evolution and Refactoring*. St. Petersburg State University, 2003.

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

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

[PV00] Yannis Papakonstantinou and Victor Vianu. *DTD Inference for Views of XML Data*. University of California, San Diego, 2000.

[PV03] Yannis Papakonstantinou and Victor Vianu. *Incremental Validation of XML Documents*. University of California, San Diego, 2003.

[RCC+02] Elke A. Rundensteiner, Li Chen, Kajal Claypool, Diane K. Kramer, and Hong Su. *XEM: Managing the Evolution of XML Documents*. Worcester Polytechnic Institute, 2002.

[Rei92] Raymond Reiter. Formalizing database evolution in the situation calculus. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 600–609, ICOT, Japan, 1992. Association for Computing Machinery.

[RKS02] Elke A. Rundensteiner, Diane K. Kramer, and Hong Su. *XEM: XML Evolution Management*. Worcester Polytechnic Institute, Januar 2002.

[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.

[Sim04] Andrey A. Simanovsky. *Evolution of Schema of XML-documents stroed in a Reational Database*. St. Petersburg State University, 2004.

[SS] Saeed Salehi and Magnus Steinby. *Tree Algebras*. Turku Centre for Computer Science.

[Tie03] Tobias Tiedt. *Normalform für XML-Schema*. Universität Rostock, Studienarbeit, 2003.

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

[W3C99a] James Clark/ W3C. *XSL Transformations (XSLT) Version 1.0*. W3C, http://www.w3.org/TR/xslt, November 1999.

[W3C99b] James Clark, Steve DeRose / W3C. *XML Path Language (XPath) Version 1.0*. W3C, http://www.w3.org/TR/xpath, November 1999.

[W3C00] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, Steve Byrne / W3C. *Document Object Model Level 2 Core*. W3C, http://www.w3.org/DOM/DOMTR, November 2000.

[W3C01] Eve Maler, Steve DeRose, David Orchard/ W3C. *XML Linking Language (XLink) Version 1.0*. W3C, http://www.w3.org/TR/xlink/, Juni 2001.

[W3C03] Ashok Malhotra, Jim Melton, Jonathan Robie, Micheal Rys / W3C. *XML Syntax for XQuery 1.0 (XQueryX)*. W3C, http://www.w3.org/TR/xqueryx, Dezember 2003.

[W3C04a] David C. Fallside, Henry S. Thompson, Paul V. Biron et al. / W3C. *XML Schema*. W3C, http://www.w3.org/TR/2004/REC-xmlschema-0-20041028 http://www.w3.org/TR/2004/REC-xmlschema-01-20041028 http://www.w3.org/TR/2004/REC-xmlschema-02-20041028, Oktober 2004.

[W3C04b] John Cowan, Richard Tobin / W3C. *XML Information Set (Second Edition)*. W3C, http://www.w3.org/TR/xml-infoset/, Februar 2004.

[W3C04c] Jonathan Marsh, Ashok Malhotra, Norman Walsh, Mary Fernández, Marton Nagy / W3C. *XQuery 1.0 and XPath 2.0 Data Model*. W3C, http://www.w3.org/TR/2004/WD-xpath-datamodel-20041029, Oktober 2004.

[W3C04d] Jonathan Robie, Jérôme Siméon, Scott Boag, Mary F. Fernández, Anders Berglund, Don Chamberlin, Michael Kay / W3C. *XML Path Language (XPath)*

Version 2.0. W3C, http://www.w3.org/TR/2004/WD-xpath20-20041029, Oktober 2004.

[W3C04e] Tim Bray, Jean Paoli, Eve Maler, C.M. Sperberg-McQueen / W3C. *Extensible Markup Language (XML) 1.0 (Third Edition)*. W3C, http://www.w3.org/TR/REC-xml, Februar 2004.

[W3C05] Mary F. Fernández, Daniela Florescu, Don Chamberlin, Jérôme Siméon, Jonathan Robie, Scott Boag / W3C. *XQuery 1.0: An XML Query Language*. W3C, http://www.w3.org/TR/xquery/, Oktober Februar 2005.

[XML00a] Working Draft der XML:DB-Initiative / XMLDB.org. *Requirements for XML Update Language*. XML:DB, http://xmldb-org.sourceforge.net/xupdate/xupdate-req.html, November 2000.

[XML00b] Working Draft der XML:DB-Initiative / XMLDB.org. *XUpdate - XML Update Language*. XML:DB, http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html, September 2000.

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

Selbstständigkeitserklärung

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

Rostock, den 06.03.2005

Thesen

1. Daten werden nicht mehr nur in relationalen oder objektorientierten Datenbanksystemen gespeichert, sondern zunehmend auch langfristig im XML-Format. Dadurch ist es wünschenswert, die Möglichkeiten der etablierten Datenbanksysteme auch im XML-Bereich zur Verfügung stehen zu haben.

2. Bei einer langfristigen Speicherung von Daten und der Datenorganisation ist Evolution nicht zu vermeiden. Somit ist es zwingend notwendig, bei einer Datenorganistion mittels XML die Prinzipien der Evolution aus dem Datenbankbereich zu adaptieren.

3. Die Evolution umfasst nicht nur die Änderung des Schemas, sondern schließt auch die Adaption der davon betroffenden Daten und möglicher Anfragen auf diesen Daten mit ein. Ein Framework zur Umsetzung der Evolution muss also nicht nur das Schema manipulieren, sondern auch die Daten und Anfragen wenn möglich automatisch anpassen.

4. Die Beschreibung der Schemaänderung und der Adaption der Daten durch eine Sprache ermöglicht eine einfachere Umsetzung der Evolution, da die Sprache vom Sprachumfang allgemeiner gehalten werden kann und das Anfragemodul in einem Framework zur Umsetzung der Änderungen nicht zwischen Schema und Dokument unterscheiden muss. Dadurch konnte das Framework "schlanker", übersichtlicher und fehlerunanfälliger gestaltet werden.

5. Die inkrementelle Validierung dient dazu, die Kontrolle der Validität effizienter zu gestalten, als es z.B. durch Validierung durch Neuparsen der Dokumente, also *Validierung from scratch*, möglich ist, da bei der inkrementellen Validierung nur die durch Schemaevolution betroffenden Teile eines Dokumentes überprüft werden müssen.

6. Die inkrementelle Validierung hat nicht nur den Vorteil, dass darüber herausgefunden wird, welche Teile im Dokument abgeändert werden müssen, sondern durch die Kontrolle der einzelnen Eigenschaften von Dokumentknoten kann zusätzlich bestimmt werden, was genau wie geändert werden muss. Dadurch erreicht man eine automatische Generierung von Anfragen zur automatischen Adaption der Dokumente.

7. Im Gegensatz zu einer automatischen Anpassung der Dokumente ist eine automatische Anpassung von XQuery-Anfragen nur sehr schwer zu realisieren, da Mehrdeutigkeiten bei der Änderung und semantische Fehler auftreten können.